Python Programlama Dili

Python 3.3.2

Uyarı

Bu belgeler sıklıkla güncellenmektedir. Eğer herhangi bir makalenin içeriğinde eksiklik görürseniz, daha sonra gelip içeriği tekrar kontrol edin.

Not

Python programlama dili ile ilgili her türlü sorunuzu ve istihza.com'daki makalelere ilişkin yorumlarınızı istihza.com/forum adresinde dile getirebilirsiniz.

Not

Tamamlandığında, Python3'le ilgili çalışma sorularına istihza.com/sorular adresinden ulaşabilirsiniz.

Not

Eğer Python'ın 3.x serisi yerine 2.x serisi ile çalışmak istiyorsanız http://belgeler.istihza.com/py2/ adresini ziyaret edebilirsiniz.

Not

Bu belgelere katkıda bulunanların listesini görmek için *Katkıda Bulunanlar* adlı sayfayı inceleyebilirsiniz.

- Python Hakkında
  - Python Nedir?
  - Neden Programlama Öğrenmek İsteyeyim?
  - Neden Python?
  - Python Nasıl Okunur?
  - Platform Desteği
  - Farklı Python Sürümleri
  - Hangi Seriyi Öğrenmeliyim?
  - Bu Kitap Hakkında
  - Nereden Yardım Alabilirim?

- Kullanım Koşulları
- Temel Komut Satırı Bilgisi
  - Komut Satırına Nasıl Ulaşırız?
  - Hangi Dizin Altındayım?
  - Dizin İçeriğini Listelemek
  - Dizin Değiştirme İşlemleri
  - Çevre Değişkenleri
  - Dizin Adı Tamamlama
  - Dizin Ayraçları
  - Sembolik Bağlar
  - Çalıştırma Yetkisi
  - Dosya kopyalama, Taşıma ve Silme
    - cp
    - rm
    - mv
- YOL (*PATH*) Kavramı
  - YOL Nedir?
  - YOL'a Dizin Eklemek
    - Windows
    - GNU/Linux
- Python Nasıl Kurulur?
  - GNU/Linux Kullanıcıları
    - Kurulu Python Sürümü
    - Paket Deposundan Kurulum
    - Kaynaktan Kurulum
      - root Hakları İle Kurulum
      - Yetkisiz Kullanıcı Olarak Kurulum
  - Windows Kullanıcıları
- Python Nasıl Çalıştırılır?
  - GNU/Linux Kullanıcıları
    - Kurulu Python3'ü Kullananlar
    - Python3'ü Depodan Kuranlar
    - Python3'ü root Olarak Derleyenler
      - Çok Önemli Bir Uyarı
    - Python3'ü Ev Dizinine Kuranlar
    - Farklı Sürümleri Birlikte Kullanmak
  - Windows Kullanıcıları
    - Farklı Sürümleri Birlikte Kullanmak
  - Hangi Komut Hangi Sürümü Çalıştırıyor?
  - Sistem Komut Satırı ve Python Komut Satırı
- Etkileşimli Python
  - Python'ın Etkileşimli Kabuğu
  - Etkileşimli Kabukta İlk Adımlar
    - Karakter Dizilerine Giriş
    - Sayılara Giriş

        - Değişkenler
            - Değişken Adı Belirleme Kuralları
            - Uygulama Örnekleri
            - Değişkenlere Dair Bazı İpuçları
                - Aynı Değere Sahip Değişkenler Tanımlama
                - Değişkenlerin Değerini Takas Etme
    - Etkileşimli Kabuğun Hafızası
- print() Fonksiyonu
    - Nedir, Ne İşe Yarar?
    - Nasıl Kullanılır?
    - Bir Fonksiyon Olarak print()
    - print() Fonksiyonunun Parametreleri
        - sep
        - end
        - file
        - flush
    - Birkaç Pratik Bilgi
        - Yıldızlı Parametreler
        - sys.stdout'u Kalıcı Olarak Değiştirmek
- Kaçış Dizileri
    - Ters Taksim (\)
    - Satır Başı (\n)
    - Sekme (\t)
    - Zil Sesi (\a)
    - Aynı Satır Başı (\r)
    - Düşey Sekme (\v)
    - Geri Silme (\b)
    - Küçük Unicode (\u)
    - Büyük Unicode (\U)
    - Uzun Ad (\N)
    - Onaltılı Karakter (\x)
    - Etkisizleştirme (r)
    - Sayfa Başı (\f)
    - Kaçış Dizilerine Toplu Bakış
- Temel Program Kaydetme ve Çalıştırma Mantığı
    - GNU/Linux
    - Windows
- Program Çalıştırmada Alternatif Yöntemler
    - GNU/Linux
        - Programları Sadece İsmi ile Çalıştırmak
        - Programları Çift Tıklayarak Çalıştırmak
    - Windows
        - Programları Komut Satırından Çalıştırmak
        - Programları Sadece İsmi ile Çalıştırmak
        - Programları Çift Tıklayarak Çalıştırmak

        - Programları .EXE Haline Dönüştürmek
            - cx_Freeze
            - cx_Freeze Çalışmıyor!
            - cx_Freeze Nasıl Kullanılır?
- Çalışma Ortamı Tavsiyesi
    - Windows Kullanıcıları
    - GNU/Linux Kullanıcıları
    - Metin Düzenleyici Ayarları
    - MS-DOS Komut Satırı Ayarları
    - Program Örnekleri
- Yorum ve Açıklama Cümleleri
    - Yorum İşareti
    - Yorum İşaretinin Farklı Kullanımları
        - Etkisizleştirme Amaçlı
        - Süsleme Amaçlı
- Kullanıcıyla Veri Alışverişi
    - input() Fonksiyonu
    - Tip Dönüşümleri
        - int()
        - str()
        - float()
        - complex()
    - eval() ve exec() Fonksiyonları
    - format() Metodu
- Koşullu Durumlar
    - Koşul Deyimleri
        - if
        - elif
        - else
    - Örnek Uygulama
- İşleçler
    - Aritmetik İşleçler
    - Karşılaştırma İşleçleri
    - Bool İşleçleri
    - Değer Atama İşleçleri
    - Aitlik İşleçleri
    - Kimlik İşleçleri
    - Uygulama Örnekleri
        - Basit Bir Hesap Makinesi
        - Sürüme Göre İşlem Yapan Program
- Döngüler (Loops)
    - while Döngüsü
    - for Döngüsü
    - İlgili Araçlar
        - range Fonksiyonu

        - pass Deyimi
        - break Deyimi
        - continue Deyimi
    - Örnek Uygulamalar
        - Karakter Dizilerinin İçeriğini Karşılaştırma
        - Dosyaların İçeriğini Karşılaştırma
        - Karakter Dizisindeki Karakterleri Sayma
        - Dosya içindeki Karakterleri Sayma
- Hata Yakalama
    - Hata Türleri
    - try... except...
    - try... except... as...
    - try... except... else...
    - try... except... finally...
    - raise
    - Bütün Hataları Yakalamak
    - Örnek Uygulama
- Karakter Dizileri
    - Karakter Dizilerinin Öğelerine Erişmek
    - Karakter Dizilerini Dilimlemek
    - Karakter Dizilerini Ters Çevirmek
    - Karakter Dizilerini Alfabe Sırasına Dizmek
    - Karakter Dizileri Üzerinde Değişiklik Yapmak
    - Üç Önemli Fonksiyon
        - dir()
        - enumerate()
        - help()
    - Notlar
- Karakter Dizilerinin Metotları
    - replace()
    - split(), rsplit(), splitlines()
    - lower()
    - upper()
    - islower(), isupper()
    - endswith()
    - startswith()
- Karakter Dizilerinin Metotları (Devamı)
    - capitalize()
    - title()
    - swapcase()
    - casefold()
    - strip(), lstrip(), rstrip()
    - join()
    - count()
    - index(), rindex()

    - find, rfind()
    - center()
    - rjust(), ljust()
    - zfill()
    - partition(), rpartition()
    - encode()
    - expandtabs()
- Karakter Dizilerinin Metotları (Devamı)
    - str.maketrans(), translate()
    - isalpha()
    - isdigit()
    - isalnum()
    - isdecimal()
    - isidentifier()
    - isnumeric()
    - isspace()
    - isprintable()
- Karakter Dizilerini Biçimlendirmek
    - % İşareti ile Biçimlendirme (Eski Yöntem)
        - Biçimlendirme Karakterleri
            - s
            - d
            - i
            - o
            - x
            - X
            - f
            - c
    - format() Metodu ile Biçimlendirme (Yeni Yöntem)
        - Biçimlendirme Karakterleri
            - s
            - c
            - d
            - o
            - x
            - X
            - b
            - f
            - ,
- Listeler ve Demetler
    - Listeler
        - Liste Tanımlamak
        - list() Fonksiyonu
        - Listelerin Öğelerine Erişmek
        - Listelerin Öğelerini Değiştirmek

        - Listeye Öğe Eklemek
        - Listeleri Birleştirmek
        - Listeden Öğe Çıkarmak
        - Listeleri Silmek
        - Listeleri Kopyalamak
        - Liste Üreteçleri (List Comprehensions)
        - Örnek Program: X.O.X Oyunu
    - Demetler
        - Demet Tanımlamak
        - Tek Öğeli bir Demet Tanımlamak
        - Demetlerin Öğelerine Erişmek
        - Demetlerle Listelerin Birbirinden Farkı
        - Demetlerin Kullanım Alanı
- Listelerin ve Demetlerin Metotları
    - Listelerin Metotları
        - append()
        - extend()
        - insert()
        - remove()
        - reverse()
        - pop()
        - sort()
        - index()
        - count()
    - Demetlerin Metotları
        - index()
        - count()
- Sayma Sistemleri
    - Onlu Sayma Sistemi
    - Sekizli Sayma Sistemi
    - On Altılı Sayma Sistemi
    - İkili Sayma Sistemi
    - Sayma Sistemlerini Birbirine Dönüştürme
        - Fonksiyon Kullanarak
            - bin()
            - hex()
            - oct()
            - int()
        - Biçimlendirme Yoluyla
            - b
            - x
            - o
    - Sayma Sistemlerinin Birbirlerine Karşı Avantajları
- Sayılar
    - Sayıların Metotları

        - Tam Sayıların Metotları
            - bit_length()
        - Kayan Noktalı Sayıların Metotları
            - as_integer_ratio()
            - is_integer()
        - Karmaşık Sayıların Metotları
            - imag
            - real
    - Aritmetik Fonksiyonlar
        - abs()
        - divmod()
        - max()
        - min()
        - sum()
- Temel Dosya İşlemleri
    - Dosya Oluşturmak
    - Dosyaya Yazmak
    - Dosya Okumak
    - Dosyaları Otomatik Kapatma
    - Dosyayı İleri-Geri Sarmak
    - Dosyalarda Değişiklik Yapmak
        - Dosyaların Sonunda Değişiklik Yapmak
        - Dosyaların Başında Değişiklik Yapmak
        - Dosyaların Ortasında Değişiklik Yapmak
    - Dosyaya Erişme Kipleri
- Dosyaların Metot ve Nitelikleri
    - closed() Metodu
    - readable() Metodu
    - writable() Metodu
    - truncate() Metodu
    - mode Niteliği
    - name Niteliği
    - encoding Niteliği
- İkili (*Binary*) Dosyalar
    - İkili Dosyalarla Örnekler
        - PDF Dosyalarından Bilgi Alma
        - Resim Dosyalarının Türünü Tespit Etme
            - JPEG
            - PNG
            - GIF
            - TIFF
            - BMP
- Basit bir İletişim Modeli
    - 8 Bitlik bir Sistem
    - Hata Kontrolü

    - Karakterlerin Temsili
- Karakter Kodlama (*Character Encoding*)
    - Giriş
    - ASCII
        - 7 Bitlik bir Sistem
        - Genişletilmiş ASCII
        - 1 Karakter == 1 Bayt
    - UNICODE
        - Sınırsız Bitlik bir Sistem
        - UTF-8 Kod Çözücüsü
        - 1 Karakter != 1 Bayt
        - Eksik Karakterler ve encode Metodu
        - Dosyalar ve Karakter Kodlama
            - encoding
            - errors
    - Konu ile ilgili Fonksiyonlar
        - repr()
        - ascii()
        - ord()
        - chr()
- Baytlar (Bytes) ve Bayt Dizileri (Bytearrays)
    - Giriş
    - Eskisi ve Yenisi
    - Bayt Tanımlamak
    - bytes() Fonksiyonu
    - Baytların Metotları
        - decode
        - fromhex
    - Bayt Dizileri
    - Bayt Dizilerinin Metotları
- Sözlükler
    - Sözlük Tanımlamak
    - Sözlük Öğelerine Erişmek
    - Sözlüklerin Yapısı
    - Sözlüklere Öğe Eklemek
    - Sözlük Öğeleri Üzerinde Değişiklik Yapmak
    - Sözlük Üreteçleri (*Dictionary Comprehensions*)
- Sözlüklerin Metotları
    - keys()
    - values()
    - items()
    - get()
    - clear()
    - copy()
    - fromkeys()

    - pop()
    - popitem()
    - setdefault()
    - update()
- Kümeler ve Dondurulmuş Kümeler
    - Kümeler
        - Küme Oluşturmak
        - Kümelerin Yapısı
        - Küme Üreteçleri (*Set Comprehensions*)
        - Kümelerin Metotları
            - clear()
            - copy()
            - add()
            - difference()
            - difference_update()
            - discard()
            - remove()
            - intersection()
            - intersection_update()
            - isdisjoint()
            - issubset()
            - issuperset()
            - union()
            - update()
            - symmetric_difference()
            - symmetric_difference_update()
            - pop()
    - Dondurulmuş Kümeler (Frozenset)
- Fonksiyonlar
    - Fonksiyon Nedir ve Ne İşe Yarar?
    - Fonksiyon Tanımlamak ve Çağırmak
    - Fonksiyonların Yapısı
    - Fonksiyonlar Ne İşe Yarar?
    - Parametreler ve Argümanlar
        - Sıralı (veya İsimsiz) Parametreler
        - İsimli Parametreler
        - Varsayılan Değerli Parametreler
        - Rastgele Sayıda İsimsiz Parametre Belirleme
        - Rastgele Sayıda İsimli Parametre Belirleme
    - return Deyimi
    - Örnek bir Uygulama
    - Fonksiyonların Kapsamı ve global Deyimi
- Gömülü Fonksiyonlar
    - abs()
    - round()

- all()
- any()
- ascii()
- repr()
- bool()
- bin()
- bytes()
- bytearray()
- chr()
- list()
- set()
- tuple()
- frozenset()
- complex()
- float()
- int()
- str()
- dict()
- callable()
- ord()
- oct()
- hex()
- eval(), exec(), globals(), locals(), compile()
- copyright()
- credits()
- license()
- dir()
- divmod()
- enumerate()
- exit()
- help()
- id()
- input()
- format()
- filter()
- hash()
- isinstance()
- len()
- map()
- max()
- min()
- open()
- pow()
- print()
- quit()

    - range()
    - reversed()
    - sorted()
    - slice()
    - sum()
    - type()
    - zip()
    - vars()
- İleri Düzey Fonksiyonlar
    - Lambda Fonksiyonları
    - Özyinelemeli (*Recursive*) Fonksiyonlar
    - Bezeyiciler (*Decorators*)
    - Kapalı Fonksiyonlar (*Closures*)

başa dön

Python Hakkında

Eğer yaşamınızın bir döneminde herhangi bir programlama dili ile az veya çok ilgilendiyseniz, Python adını duymuş olabilirsiniz. Önceden bir programlama dili deneyiminiz hiç olmamışsa dahi, Python adının bir yerlerden kulağınıza çalınmış olma ihtimali bir hayli yüksek. Bu satırları okuyor olduğunuza göre, Python adını en az bir kez duymuş olduğunuzu ve bu *şeye* karşı içinizde hiç değilse bir merak uyandığını varsayabiliriz.

Peki, en kötü ihtimalle kulak dolgunluğunuz olduğunu varsaydığımız bu *şey* hakkında acaba neler biliyorsunuz?

İşte biz bu ilk bölümde, fazla teknik ayrıntıya kaçmadan, Python hakkında kısa kısa bilgiler vererek Python'ın ne olduğunu ve bununla neler yapabileceğinizi anlatmaya çalışacağız.

Python Nedir?

Tahmin edebileceğiniz gibi Python (C, C++, Perl, Ruby ve benzerleri gibi) bir programlama dilidir ve tıpkı öteki programlama dilleri gibi, önünüzde duran kara kutuya, yani bilgisayara hükmetmenizi sağlar.

Guido Van Rossum

Bu programlama dili Guido Van Rossum adlı Hollandalı bir programcı tarafından 90'lı yılların başında geliştirilmeye başlanmıştır. Çoğu insan, isminin Python olmasına aldanarak, bu programlama dilinin, adını piton yılanından aldığını düşünür. Ancak zannedildiğinin aksine bu programlama dilinin adı piton yılanından gelmez. Guido Van Rossum bu programlama dilini, *The Monty Python* adlı bir İngiliz komedi grubunun, *Monty Python's Flying Circus* adlı gösterisinden esinlenerek adlandırmıştır. Ancak her ne kadar gerçek böyle olsa da, Python programlama dilinin pek çok yerde bir yılan figürü ile temsil edilmesi neredeyse bir gelenek halini almıştır.

Dediğimiz gibi, Python bir programlama dilidir. Üstelik pek çok dile kıyasla öğrenmesi kolay bir programlama dilidir. Bu yüzden, eğer daha önce hiç programlama deneyiminiz olmamışsa, programlama maceranıza Python'la başlamayı tercih edebilirsiniz.

Neden Programlama Öğrenmek İsteyeyim?

Günlük yaşamınıza şöyle bir bakın. Gerek işyerinizde olsun, gerek evde bilgisayar başında olsun, belli işleri tekdüze bir şekilde tekrar ettiğinizi göreceksiniz. Mesela sürekli olarak yazılı belgelerle uğraşmanızı gerektiren bir işte çalışıyor olabilirsiniz. Belki de her gün onlarca belgeyi açıp bu belgelerde birtakım bilgiler arıyor, bu bilgileri düzeltiyor, yeniliyor veya siliyorsunuzdur. Bu işlemlerin ne kadar vakit alıcı ve sıkıcı olduğunu düşünün. Eğer bir programlama dili biliyor olsaydınız, bütün bu işlemleri sizin yerinize bu programlama dili hallediyor olabilirdi.

İşte Python programlama dili böyle bir durumda devreye girer. Her gün saatler boyunca uğraştığınız işlerinizi, yalnızca birkaç satır Python kodu yardımıyla birkaç saniye içinde tamamlayabilirsiniz.

Piton

Ya da şöyle bir durum düşünün: Çalıştığınız işyerinde *PDF* belgeleriyle bolca haşır neşir oluyor olabilirsiniz. Belki de yüzlerce sayfalık kaşeli ve imzalı belgeyi *PDF* haline getirmeniz gerekiyordur. Üstelik sizden bu belgeleri mümkün olduğunca tek belge halinde *PDF*'lemeniz isteniyor olabilir. Ama o yüzlerce sayfayı tarayıcıdan geçirirken işin tam ortasında bir aksilik oluyor, makine arızalanıyor ve belki de ister istemez belgeniz bölünüyordur.

İşte Python programlama dili böyle bir durumda da devreye girer. Eğer Python programlama dilini öğrenirseniz, internette saatlerce ücretsiz *PDF* birleştirme programı aramak veya profesyonel yazılımlara onlarca dolar para vermek yerine, belgelerinizi birleştirip işinizi görecek programı kendiniz yazabilirsiniz.

Örneğin bu satırların yazarı, okuduğunuz bu belgeleri *HTML* ve *PDF* biçimlerine dönüştürmek için Python programlama dilinden yararlanıyor. Mesela aşağıdaki gibi bir *RST* (reStructuredText) dosyası oluşturarak, Python ile yazılmış bir program yardımıyla belgeler.istihza.com/py3/index.html adresindeki *HTML* belgesini elde edebiliyor:

```
.. meta::
   :description: Python Programlama Dilinin 3.x Serisi için Türkçe Kaynak
   :keywords: Python, 3.x, belgelendirme, Türkçe, kaynak

.. include:: replacements.rpl

**************************
Python Programlama Dili
**************************

Python |py3|
************
```

```
.. warning:: Bu belgeler sıklıkla güncellenmektedir. Eğer herhangi bir makalenin
   içeriğinde eksiklik görürseniz, daha sonra gelip içeriği tekrar kontrol edin.

.. note:: Python programlama dili ile ilgili her türlü sorunuzu ve
   istihza.com'daki makalelere ilişkin yorumlarınızı `istihza.com/forum
   <http://www.istihza.com/forum>`_ adresinde dile getirebilirsiniz.

.. note:: Tamamlandığında, Python3'le ilgili çalışma sorularına
   `istihza.com/sorular <http://www.istihza.com/sorular>`_ adresinden
   ulaşabilirsiniz.

.. note:: Eğer Python'ın 3.x serisi yerine 2.x serisi ile çalışmak istiyorsanız
   `http://belgeler.istihza.com/py2/ <http://belgeler.istihza.com/py2/>`_
   adresini ziyaret edebilirsiniz.

.. note:: Bu belgelere katkıda bulunanların listesini görmek
   için :doc:`katkida_bulunanlar` adlı sayfayı inceleyebilirsiniz.

.. toctree::

   python_hakkinda
   temel_komut_satiri_bilgisi
   path
   kurulum
   calistirma
   etkilesimli_python
   print
```

Elbette Python'la yapabilecekleriniz yukarıda verdiğimiz basit örneklerle sınırlı değildir. Python'ı kullanarak masaüstü programlama, oyun programlama, taşınabilir cihaz programlama, web programlama ve ağ programlama gibi pek çok alanda çalışmalar yürütebilirsiniz.

Not

Eğer piyasada Python programlama dili ile yazılmış hangi yazılımların olduğunu merak ediyorsanız http://goo.gl/iTtaQ adresindeki listeyi inceleyebilirsiniz.

Neden Python?

Python programlarının en büyük özelliklerinden birisi, C ve C++ gibi dillerin aksine, derlenmeye gerek olmadan çalıştırılabilmeleridir. Python'da derleme işlemi ortadan kaldırıldığı için, bu dille

oldukça hızlı bir şekilde program geliştirilebilir.

Ayrıca Python programlama dilinin basit ve temiz söz dizimi, onu pek çok programcı tarafından tercih edilen bir dil haline getirmiştir. Python'ın söz diziminin temiz ve basit olması sayesinde hem program yazmak, hem de başkası tarafından yazılmış bir programı okumak, başka dillere kıyasla çok kolaydır.

Google ve Python

Python'ın yukarıda sayılan özellikleri sayesinde dünya çapında ün sahibi büyük kuruluşlar (Google, YouTube, Yahoo! gibi) bünyelerinde her zaman Python programcılarına ihtiyaç duyuyor. Mesela pek çok büyük şirketin Python bilen programcılara iş imkanı sağladığını, Python'ın baş geliştiricisi Guido Van Rossum'un 2005 ile 2012 yılları arasında Google'da çalıştığını, 2012 yılının sonlarına doğru ise Dropbox şirketine geçtiğini söylersek, bu programlama dilinin önemi ve geçerliliği herhalde daha belirgin bir şekilde ortaya çıkacaktır.

Python programlama dili ve bu dili hakkıyla bilenler sadece uluslararası şirketlerin ilgisini çekmekle kalmıyor. Python son zamanlarda Türkiye'deki kurum ve kuruluşların da dikkatini çekmeye başladı. Bu dil artık yavaş yavaş Türkiye'deki üniversitelerin müfredatında da kendine yer buluyor.

Sözün özü, pek çok farklı sebepten, başka bir programlama dilini değil de, Python programlama dilini öğrenmek istiyor olabilirsiniz.

Python Nasıl Okunur?

Python programlama dili üzerine bu kadar söz söyledik. Peki yabancı bir kelime olan *python*'ı nasıl telaffuz edeceğimizi biliyor muyuz?

Geliştiricisi Hollandalı olsa da *python* İngilizce bir kelimedir. Dolayısıyla bu kelimenin telaffuzunda İngilizcenin kuralları geçerli. Ancak bu kelimeyi hakkıyla telaffuz etmek, ana dili Türkçe olanlar için pek kolay değil. Çünkü bu kelime içinde, Türkçede yer almayan ve okunuşu peltek s'yi andıran [th] sesi var. İngilizce bilenler bu sesi *think* (düşünmek) kelimesinden hatırlayacaklardır. Ana dili Türkçe olanlar *think* kelimesini genellikle [tink] şeklinde telaffuz eder. Dolayısıyla *python* kelimesini de [paytın] şeklinde telaffuz edebilirsiniz.

Python kelimesini tamamen Türkçeleştirerek [piton] şeklinde telaffuz etmeyi yeğleyenler de var. Elbette siz de dilinizin döndüğü bir telaffuzu tercih etmekte özgürsünüz.

Bu arada, eğer *python* kelimesinin İngilizce telaffuzunu dinlemek istiyorsanız howjsay.com adresini ziyaret edebilir, Guido Van Rossum'un bu kelimeyi nasıl telaffuz ettiğini merak ediyorsanız da

http://goo.gl/bx9iJu adresindeki tanıtım videosunu izleyebilirsiniz.

Platform Desteği

Python programlama dili pek çok farklı işletim sistemi ve platform üzerinde çalışabilir. GNU/Linux, Windows, Mac OS X, BSD, Solaris, AIX, AROS, AS/400, BeOS, MorphOS, S60, iPOD, iPhone, Android ve adını dahi duymadığınız pek çok ortamda Python uygulamaları geliştirebilirsiniz. Ayrıca herhangi bir ortamda yazdığınız bir Python programı, üzerinde hiçbir değişiklik yapılmadan veya ufak değişikliklerle başka ortamlarda da çalıştırılabilir.

Biz bu belgelerde Python programlama dilini GNU/Linux ve Microsoft Windows işletim sistemi üzerinden anlatacağız. Ancak sıkı sıkıya bel bağlayacağımız özel bir GNU/Linux dağıtımı veya Windows sürümü yok. Bu yüzden, hangi GNU/Linux dağıtımını veya hangi Windows sürümünü kullanıyor olursanız olun, buradaki bilgiler yardımıyla Python programlama dilini öğrenebilir, öğrendiklerinizi kendi işletim sisteminize uyarlayabilirsiniz.

Not

Bu satırların yazarının, *Ubuntu*, *JoliOs*, *CentOs*, *Windows XP* ve (kısmen) *Windows 7* kurulu bilgisayarlara erişimi olduğu için, bu kitaptaki ekran görüntüleri genellikle bu işletim sistemlerinden alınmış olacaktır.

Not

http://www.istihza.com/wiki adresinde, Python'ın farklı işletim sistemlerinde kullanımına ilişkin bilgi bulabilirsiniz.

Farklı Python Sürümleri

Eğer daha önce Python programlama dili ile ilgili araştırma yaptıysanız, şu anda piyasada iki farklı Python serisinin olduğu dikkatinizi çekmiş olmalı. 30/10/2013 tarihi itibariyle piyasada olan en yeni Python sürümleri Python 2.7.5 ve Python 3.3.2'dir.

Eğer bir Python sürümü *2* sayısı ile başlıyorsa (mesela 2.7.5), o sürüm Python 2.x serisine aittir. Yok eğer bir Python sürümü *3* sayısı ile başlıyorsa (mesela 3.3.2), o sürüm Python 3.x serisine aittir.

Peki neden piyasada iki farklı Python sürümü var ve bu bizim için ne anlama geliyor?

Python programlama dili 1990 yılından bu yana geliştirilen bir dil. Bu süre içinde pek çok Python programı yazıldı ve insanların kullanımına sunuldu. Şu anda piyasadaki çoğu Python programı 2.x serisinden bir sürümle yazılmış durumda. 3.x serisi ise yeni yeni yaygınlık kazanıyor.

Not

Biz bu kitapta kolaylık olsun diye Python'ın 3.x serisini Python3; 2.x serisini ise Python2 olarak adlandıracağız.

Python3, Python2'ye göre hem çok daha güçlüdür, hem de Python2'nin hatalarından arındırılmıştır. Python3'teki büyük değişikliklerden ötürü, Python2 ile yazılmış bir program Python3 altında çalışmayacaktır. Aynı durum bunun tersi için de geçerlidir. Yani Python3 kullanarak yazdığınız bir program Python2 altında çalışmaz.

Dediğimiz gibi, piyasada Python2 ile yazılmış çok sayıda program var. İşte bu sebeple Python geliştiricileri uzun bir süre daha Python2'yi geliştirmeye devam edecek. Elbette geliştiriciler bir yandan da Python3 üzerinde çalışmayı ve bu yeni seriyi geliştirmeyi sürdürecek.

Farklı Python serilerinin var olmasından ötürü, Python ile program yazarken hangi seriye ait sürümlerden birini kullandığınızı bilmeniz, yazacağınız programın kaderi açısından büyük önem taşır.

Not

Farklı Python sürümleri hakkında daha ayrıntılı bilgi: http://goo.gl/tLPKQ

Python2 ile Python3 arasındaki temel farklar: http://goo.gl/DFHJ0

Hangi Seriyi Öğrenmeliyim?

Dediğimiz gibi, şu anda piyasada iki farklı Python serisi var: Python3 ve Python2. Peki acaba hangi seriye ait bir sürümü öğrenmek daha uygun olur?

Eğer Python programlama diline yeni başlıyorsanız Python3'ü öğrenmeniz daha doğru olacaktır. Ama eğer önceden bir Python bilginiz varsa ve bir Python projesi üzerinde çalışmayı planlıyorsanız, hangi sürümü öğrenmeniz gerektiği, projede kullanacağınız ek kütüphanelerin durumuna bağlıdır. Zira şu anda piyasada bulunan bütün Python modülleri/programları henüz Python3'e aktarılmış değil.

Eğer projenizde kullanmayı planladığınız ek kütüphaneler halihazırda Python3'e aktarılmışsa Python3'ü öğrenebilirsiniz. Ancak eğer bu kütüphanelerin henüz Python3 sürümü çıkmamışsa sizin de Python2 ile devam etmeniz daha uygun olabilir. Ama her halükarda Python3'ün bu dilin geleceği olduğunu ve günün birinde Python2'nin tamamen tedavülden kalkacağını da aklınızın bir köşesinde bulundurun.

Eğer Python3 yerine Python2'yi öğrenmeye karar verdiyseniz veya kullandığınız kütüphanelerin henüz Python3'e aktarılmamış olması nedeniyle şimdilik Python2 ile devam etmeniz gerekiyorsa internet sitemizde Python2'ye dair bilgiler de bulabilirsiniz. [istihza.com/py2/icindekiler_python.html]

Bu Kitap Hakkında

Bizim bu kitaptaki amacımız, herhangi bir sebeple Python'a ilgi duyan, bu programlama dilini öğrenmek isteyen kişilere bu dili olabildiğince hızlı, ayrıntılı ve kolay bir şekilde öğretmektir. Bu kitaptan yararlanabilmek için herhangi bir programlama dilini biliyor olmanıza gerek yok. Eğer internete girip e.postalarınızı okuyabilecek kadar bilgisayar ve internet bilgisine sahipseniz, bu kitap yardımıyla Python programlama dilini de öğrenebilirsiniz.

Bu kitapta, Python programlama diline ilişkin konular birbirini takip edecek şekilde ve birbiriyle bağlantılı olarak anlatılmıştır. Dolayısıyla bu kitaptan en yüksek verimi alabilmek için, konuları en baştan en sona kadar **sıra atlamadan** takip etmeniz gerekiyor.

Bu kitapta Python programlama diline ilişkin konular olabildiğince ayrıntılı bir şekilde anlatılmıştır. Yani bu kitapta epey uzun metinlerle karşılaşacaksınız. Eğer uzun metinleri okumakta zorlanıyorsanız veya bu tür metinleri sıkıcı buluyorsanız bu kitap size göre olmayabilir.

Eğer bu kitapta herhangi bir hata bulursanız lütfen istihza [at] yahoo [nokta] com adresine yazarak beni bu hatadan haberdar edin. Bu kitabın amacı, kitabı okuyanlara Python programlama dilini doğru ve ayrıntılı bir şekilde öğretmek olduğu kadar, bu programlama dilini öğretirken düzgün ve anlaşılır bir Türkçe de kullanmaktır. Bu bakımdan, kitapta bulabileceğiniz kod hatalarıyla birlikte, kitaptaki anlatım, yazım ve noktalama hatalarını da bana iletmeniz, kitabın amacına ulaşmasını kolaylaştıracaktır.

Nereden Yardım Alabilirim?

Bu kitapta Python programlama diline ilişkin konuları olabildiğince temiz ve anlaşılır bir dille anlatmaya çalıştık. Ancak yine de bazı konular zihninizde tam olarak yer etmeyebilir. Üstelik kimi zaman, bir konuyu daha iyi anlayabilmek ya da bir sorunun üstesinden gelebilmek için bilen birinin yardımına da ihtiyaç duyabilirsiniz. İşte böyle durumlarda istihza.com/forum adresine uğrayarak başka Python programcılarından yardım isteyebilirsiniz.

Forum alanı hem bilgi edinmek, hem de bildiklerinizi paylaşmak için oldukça elverişli bir ortamdır. Foruma ilk girişiniz muhtemelen yardım istemek için olacaktır. Ama ilerleyen zamanlarda Python bilginiz arttıkça bir de bakacaksınız ki yardım ararken yardım eder duruma gelmişsiniz. İşte forum; kendinizdeki değişimi görmek, bilgi düzeyinizin artışını takip etmek ve hatta yeni şeyler öğrenmek için bulunmaz bir fırsattır.

Kullanım Koşulları

Bu kitaptaki bilgiler, istihza.com'un öteki kısımları gibi, Creative Commons lisansı altındadır. Bu lisansa göre bu kitaptaki bütün bilgilerden herkes ücretsiz olarak yararlanabilir. Eğer isterseniz burada gördüğünüz belgelerin çıktısını alabilir, tanıdık-tanımadık herkesle gönül rahatlığıyla paylaşabilirsiniz. Ancak bu belgeleri başka bir yerde kullanacaksanız, http://www.istihza.com adresini kaynak olarak göstermeli ve bu belgeleri kesinlikle satmamalısınız. Arzu ederseniz belgeleri

çoğaltıp **ücretsiz olarak** dağıtabilirsiniz.

başa dön

Temel Komut Satırı Bilgisi

Biraz sonra Python programlama diliyle ilgili ilk çalışmalarımızı yapmaya başlayacağız. Ama herhangi bir programlama faaliyetine girişmeden önce mutlaka biliyor olmamız gereken bazı şeyler var. Örneğin, programlama öğrenmek isteyen bir kişi her şeyden önce, program yazmayı tasarladığı işletim sisteminde komut satırının nasıl kullanılacağına dair temel bilgileri edinmiş olmalıdır.

Bu temel bilgilere halihazırda sahip olup olmadığınızı anlamak için aşağıdaki önermelerin sizin açınızdan doğru olup olmadığını kendi kendinize sorgulayabilirsiniz:

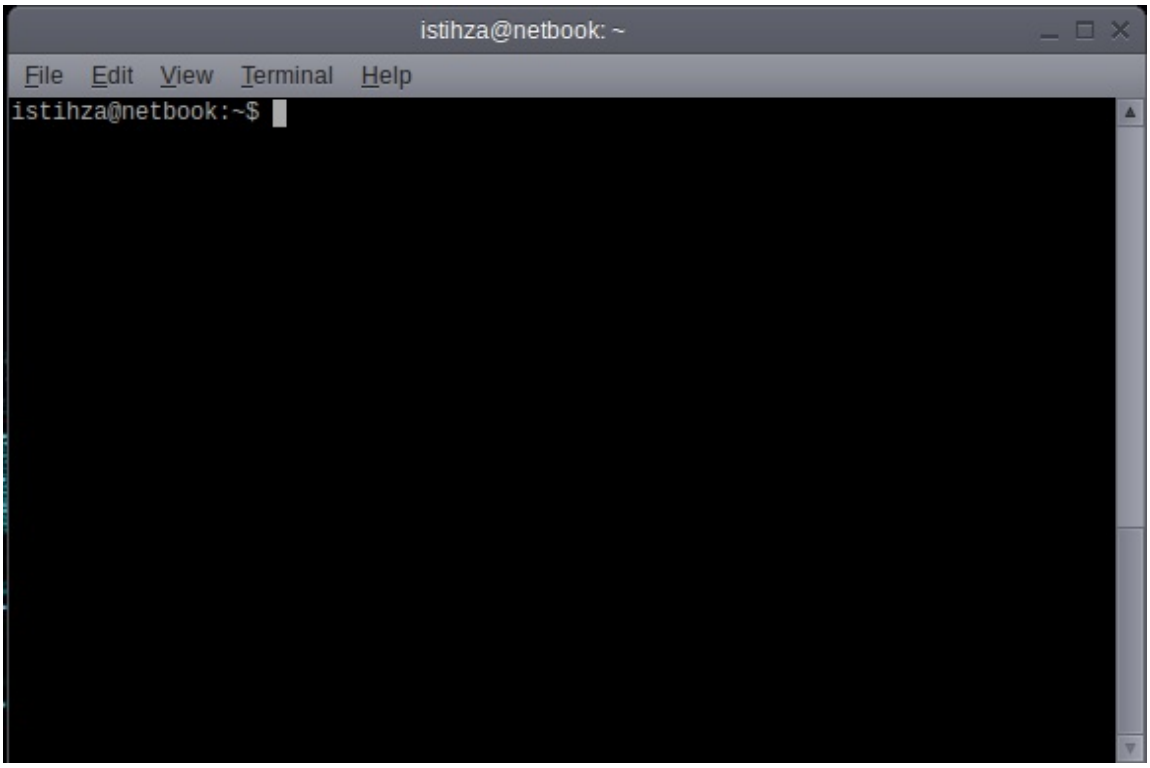
1. Kullandığım işletim sisteminde komut satırına nasıl ulaşacağımı biliyorum.
2. `dir` veya `ls` komutlarının ne işe yaradığını biliyorum.
3. `cd` komutunun nasıl kullanılacağını biliyorum.
4. Komut satırını açtıktan sonra, o anda hangi dizin altında bulunduğumu anlayabilirim.
5. `echo $HOME` veya `echo %USERPROFILE%` komutlarının ne işe yaradığını, bu komutlardaki `$HOME` ve `%USERPROFILE%` ifadelerinin ne anlama geldiğini gayet iyi biliyorum.
6. Herhangi bir dosyayı veya dizini, sembolik bağ ile başka bir konuma bağlayabilirim.
7. *permission denied* ifadesinin ne anlama geldiğini biliyorum. Bu hataya bakarak, sorunun nereden kaynaklandığını tahmin edebilir, çözümün ne olduğunu kestirebilirim.
8. Bir dosyaya çalıştırma yetkisi verebilirim.
9. Masaüstünde bulunan bir dosyayı, komut satırını kullanarak */usr/bin* dizini içine kopyalayabilirim, taşıyabilirim veya bu dosyayı silebilirim.
10. Bir dosya silmek ile dizin silmenin birbirinden farklı şeyler olduğunu ve birbirinden farklı işlemler uygulamak gerektiğini biliyorum.

Eğer yukarıdaki önermelerin tamamına 'doğru' cevabını verebiliyorsanız bu bölümü hızlıca gözden geçirip yolunuza devam edebilirsiniz. Ama eğer komut satırı konusunda bilginiz yoksa veya yetersizse, üstelik yukarıdaki önermelere de olumlu cevap veremiyorsanız bu bölümü iyice sindirmeden lütfen bir sonraki bölüme geçmeyin. Bu bölümü hakkıyla tamamlamanız kitaptan sağlıklı bir şekilde yararlanabilmeniz açısından büyük önem taşıyor. Zira bütün programlama dillerinde olduğu gibi, Python'da da komut satırı en büyük yardımcımız olacak. Bu kitapta Python programlama dilinin temellerini komut satırı üzerinde çalışarak öğreneceğiz. Özellikle yazacağımız ilk programlar komut satırında çalışacak. O yüzden, hiç değilse bu kitaptan yararlanabilmek için, kullandığınız işletim sisteminin komut satırına aşina olmalı, yukarıdaki önermelerin sizin için doğru olduğundan emin olmalısınız. Ayrıca sırf bu kitaptan yararlanabilmek için değil, bütün programcılık maceranız boyunca pek çok farklı sebepten, komut satırı üzerinde çalışmaya ihtiyaç duyacaksınız. İşte bundan ötürü, en azından bu kitabı rahatlıkla takip etmenizi sağlayacak kadar komut satırı bilgisini bu bölümde edinmenizi sağlamaya çalışacağız.

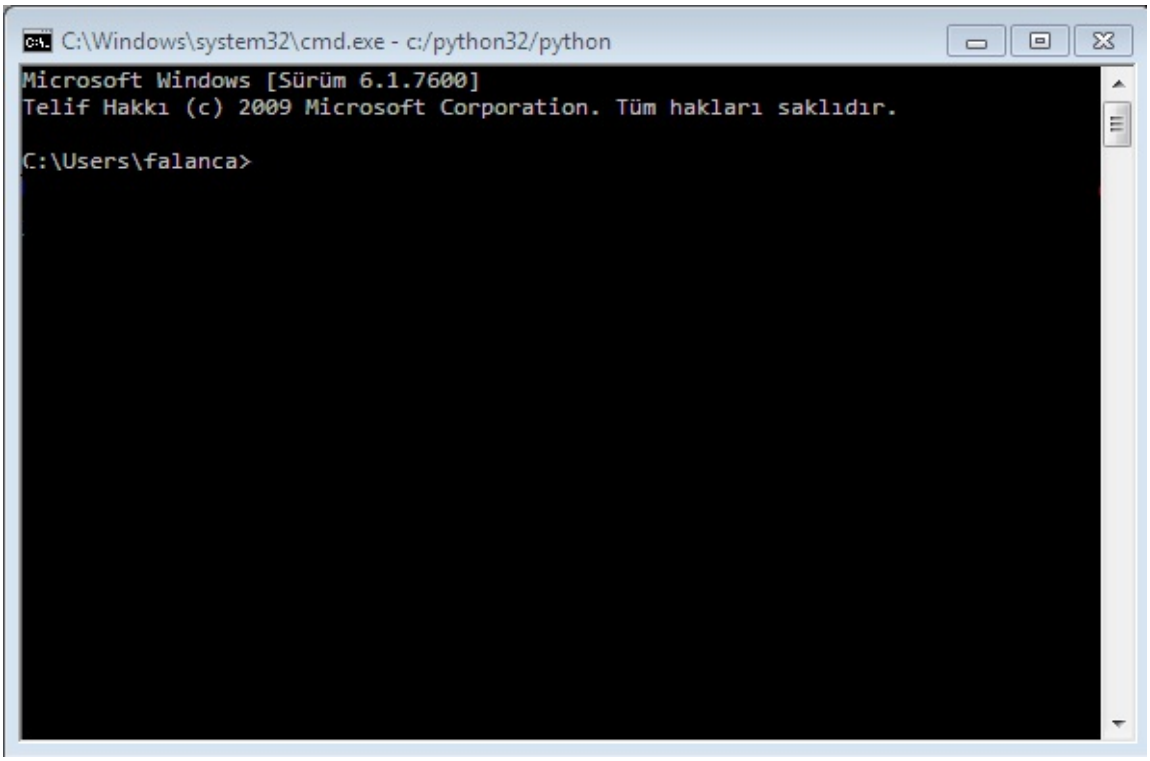
Komut Satırına Nasıl Ulaşırız?

Elbette komut satırı hakkında bilmemiz gereken ilk şey, kullandığımız işletim sisteminde komut satırına nasıl ulaşabileceğimizdir.

Eğer Ubuntu GNU/Linux dağıtımı üzerinde Unity masaüstü ortamını kullanıyorsanız sadece *Ctrl+Alt+T* tuşlarına basarak komut satırına ulaşabilirsiniz. Bu komutu verdiğinizde şuna benzer bir ekranla karşılaşacaksınız:



Windows 7 kullanıcıları ise *Başlat > Tüm Programlar > Donatılar > Komut İstemi* yolunu takip ederek komut satırına ulaşabilir. Bu işlemleri yapan Windows kullanıcıları şöyle bir ekranla karşılaşacak:



Windows kullanıcıları komut satırına ulaşmak için alternatif olarak şu yöntemi de kullanabilir:

1. Klavyenizdeki Windows logolu tuşa ve *R* tuşuna birlikte basarak 'çalıştır [*run*]'

penceresini açın.
2. Daha sonra, açılan pencereye cmd yazın ve *Enter* tuşuna basın.

Eğer Windows 7 veya Ubuntu dışında bir işletim sistemi kullanıyorsanız, kullandığınız işletim sisteminde komut satırına nasıl ulaşabileceğinizi öğrenmek için http://goo.gl/ZyjAU adresini ziyaret edebilirsiniz.

Böylece farklı işletim sistemlerinde komut satırına nasıl ulaşacağımızı öğrenmiş olduk. Ancak iyi bir programcı olabilmek için komut satırına ulaşabilmek yeterli değildir. Ulaştığımız bu komut satırını nasıl kullanacağımızı da bilmemiz gerekiyor. Örneğin komut satırı üzerinde dosyalarımızı nasıl listeleyeceğimizi ve nasıl dizin değiştirebileceğimizi de bilmeliyiz. Dediğimiz gibi, eğer bu işlemleri nasıl yapacağınızı zaten biliyorsanız bir sonraki başlığa geçebilirsiniz. Ama eğer bilginizden emin değilseniz okumaya devam edin.

Hangi Dizin Altındayım?

Şimdi biraz önce anlattığımız gibi, kullandığımız işletim sistemine uygun bir şekilde komut satırını başlatalım. Yukarıdaki ekran görüntülerinden de görebileceğiniz gibi, komut satırını başlattığımızda, siyah zemin üzerinde Windows sistemlerinde *C:\Users\falanca>*, GNU/Linux sistemlerinde ise *istihza@netbook:~$* gibi bir ibare ile karşılaşıyoruz. Burada elbette 'falanca', 'istihza' ve 'netbook' ifadeleri bilgisayar adına ve kullanıcı adınıza bağlı olarak farklı olacaktır. Örneğin kullanıcı adınız 'ahmet' ise, yukarıdaki ibare Windows'ta *C:\Users\ahmet* olacaktır. Aynı şekilde kullanıcı adınız 'mehmet', bilgisayar adınız da 'evbilgisayari' ise GNU/Linux'ta komut satırını başlattığınızda *mehmet@evbilgisayari:~$* gibi bir ibare ile karşılaşabilirsiniz. Ya da eğer kullandığınız Windows sürümü Türkçe, kullanıcı adınız da 'zeynep' ise *C:\Kullanıcılar\zeynep* ibaresini görebilirsiniz.

Bu ibareler, komut satırını başlattığımızda hangi dizinde bulunduğumuzu gösteriyor. Buradan anladığımıza göre, Windows'ta *C:\Users\falanca*, GNU/Linux'ta ise */home/istihza* dizini altındayız.

Windows'taki ekran görüntüsünden hangi dizin altında bulunduğumuz rahatlıkla anlaşılabiliyor, ama GNU/Linux'taki ekran görüntüsünde görünen *istihza@netbook:~$* ifadesine bakarak, */home/istihza* dizini altında bulunduğumuzu çıkarmak pek kolay olmayabilir. Eğer o anda hangi dizin altında bulunduğunuzdan emin olmak istiyorsanız, GNU/Linux'ta şu komutu verebilirsiniz:

```
pwd
```

Windows'ta ise aynı işlev için şu komutu kullanıyoruz:

```
cd
```

Windows'un komut satırında *C:\Users\falanca>* ibaresi, GNU/Linux'un komut satırında ise *istihza@netbook:~$* ibaresi görünürken, yukarıdaki komutlardan işletim sistemimize uygun olanı yazıp *Enter* düğmesine basarsak Windows'ta *C:\Users\falanca*, GNU/Linux'ta ise */home/istihza* gibi bir çıktı alırız.

pwd ve cd komutları, o anda hangi dizin altında bulunduğumuzu açık bir şekilde gösteriyor. Bu arada, daha önce de söylediğimiz gibi, sizin bilgisayarınızdaki kullanıcı adına bağlı olarak yukarıda gösterilenlerden farklı çıktılar alabilirsiniz.

Dizin İçeriğini Listelemek

pwd veya cd komutları yardımıyla o anda hangi dizin altında bulunduğumuzu öğrendik. Peki acaba o anda altında bulunduğumuz dizinde hangi dosya ve dizinler var? Bunu bilmeniz önemlidir, çünkü programlama maceranız boyunca, o anda içinde bulunduğunuz dizinde hangi dosyaların olduğunu tespit etmenizi gerektirecek durumlarla mutlaka karşılaşacaksınız. Örneğin bir dizin içindeki herhangi bir dosyayı komut satırı üzerinden açacaksanız, öncelikle o dosyanın dizin içinde yer alıp yer almadığından emin olmanız gerekebilir.

GNU/Linux'ta, o anda altında bulunduğumuz dizinin içeriğini listelemek için şu komuttan yararlanıyoruz (komut küçük L harfi ile başlıyor):

```
ls
```

GNU/Linux kullanıcıları, komut satırında *istihza@netbook:~$* ibaresi görünürken bu komutu verdiğinde şuna benzer bir çıktı alacaktır:

```
deneme.py Documents  makaleler.rst Pictures Templates
Desktop   Downloads  Music         Public   Videos
```

Bu çıktıda sekiz dizin, iki de dosya görüyoruz. GNU/Linux'ta dizin ve dosyalar komut ekranında farklı renklerle gösterilir. Bu sayede dizin ve dosyaları birbirinden rahatlıkla ayırt edebilirsiniz.

Windows komut satırında ise ls komutu yerine şu komutu kullanacağız:

```
dir
```

Windows'ta da komut satırında *C:\Users\falanca>* ibaresi görünürken bu komutu verirseniz şuna benzer bir çıktı alırsınız:

```
27.06.2011  12:31    <DIR>          Belgeler
17.11.2011  10:09    <DIR>          Desktop
27.09.2011  15:35               183 TESTLOG.log
28.09.2010  12:13               251 tkcon.hst
31.03.2011  14:49    <DIR>          Start Menu
02.11.2009  11:19    <DIR>          Sık Kullanılanlar

              16 Dosya     13.376.408 bayt
              22 Dizin  28.174.561.280 bayt boş
```

Gördüğünüz gibi, dir komutu da, tıpkı GNU/Linux'taki ls komutu gibi, o anda içinde bulunduğunuz

dizinde yer alan dosya ve klasörleri listeliyor.

Yukarıdaki çıktıda, sol tarafında '<DIR>' ibaresi taşıyan öğeler birer dizindir. Eğer sol tarafta herhangi bir ibare yoksa o öğe bir dosyadır. Örneğin yukarıdaki çıktıda *Belgeler*, *Desktop*, *Start Menu* ve *Sık Kullanılanlar* birer dizinken, *TESTLOG.log* ve *tkcon.hst* birer dosyadır.

Dizin Değiştirme İşlemleri

Böylece komut satırına nasıl ulaşacağımızı ve dizinlerin içeriğini nasıl listeleyeceğimizi öğrenmiş olduk. Bunun dışında, komut satırı ile ilgili olarak bilmemiz gereken önemli bir konu da dizinler arasında hareket edebilme kabiliyetidir. Yani eğer iyi bir programcı olmak istiyorsak, komut satırına ulaşıp hangi dizin altında bulunduğumuzu tespit edebilmenin yanısıra, komut satırını kullanarak o anda bulunduğumuz dizinden başka dizinlere geçiş de yapabiliyor olmalıyız.

Dizinler arasında geçiş yapabilmek için cd adlı bir komuttan yararlanacağız.

Not

Windows kullanıcıları bu komutun, GNU/Linux'taki pwd komutunun eşdeğeri olarak da kullanıldığını biliyor.

Bu komut hem GNU/Linux'ta hem de Windows'ta çalışır. Dolayısıyla bu komutu her iki işletim sisteminde de rahatlıkla kullanabilirsiniz.

Peki bu komutu nasıl kullanacağız?

GNU/Linux ve Windows'ta ls veya dir komutlarıyla elde ettiğiniz çıktılara tekrar bakın. Bu çıktılarda o dizinin içeriğinde yer alan dosya ve dizinleri görüyorsunuz. İşte bu çıktılarda görünen dizinlerin içine girebilmek için cd komutundan yararlanabilirsiniz. Diyelim ki biz o anda bulunduğumuz konumdan masaüstüne geçmek istiyoruz. Bildiğiniz gibi, bütün işletim sistemlerinde masaüstü dizini 'Desktop' adıyla gösteriliyor. Yukarıda verdiğimiz dir ve ls komutlarının çıktılarını kontrol edecek olursanız hem Windows'ta hem de GNU/Linux'ta 'Desktop' adlı dizinleri görebilirsiniz. Buna göre bütün işletim sistemlerinde şu komut yardımıyla masaüstüne ulaşabiliriz:

```
cd Desktop
```

Daha önce de söylediğimiz gibi cd komutu dizin değiştirmemizi sağlar. cd Desktop dediğimizde masaüstünün bulunduğu dizine gelmiş oluyoruz. Eğer biraz önce öğrendiğimiz dir veya ls komutlarını bu konumda verecek olursanız, masaüstünde bulunan dosyalarınızın listesini görürsünüz.

Not

Bazı Türkçe GNU/Linux dağıtımlarında masaüstüne ulaşmak için cd Desktop komutu yerine cd

Masaüstü komutunu vermeniz gerekebilir.

Yalnız burada dikkat etmemiz gereken önemli bir konu var. Yukarıda verdiğimiz cd Desktop komutunun bizi masaüstüne götürebilmesi için, bu komutu verdiğimiz sırada altında bulunduğumuz dizinin Windows'ta *C:\Users\falanca*, GNU/Linux'ta ise */home/istihza* olması gerekiyor. Neden? Çünkü cd komutu ile ulaşmaya çalıştığımız *Desktop* (Masaüstü) dizini Windows'ta *C:\Users\falanca* dizinin, GNU/Linux'ta ise */home/istihza* dizininin içinde bulunuyor. Dolayısıyla sadece cd Desktop yazarak masaüstüne ulaşabiliyoruz. Bunun ne demek olduğunu daha iyi kavramak için isterseniz birkaç deneme çalışması yapalım.

Biraz önce cd Desktop komutunu vererek masaüstüne gelmiştik. Yani şu anda Windows'ta *C:\Users\falanca\Desktop*, GNU/Linux'ta ise */home/istihza/Desktop* dizini içinde bulunuyoruz. Şimdi masaüstünün bulunduğu konumdayken şu komutu verelim:

```
cd ..
```

Bu komut bizi Windows'ta *C:\Users\falanca*, GNU/Linux'ta ise */home/istihza* dizinine geri götürecektir. Dikkat ederseniz yukarıdaki komut bizi bir üst dizine götürüyor. Bu komutu tekrar verirseniz yine bir üst dizine, yani Windows'ta *C:\Users*, GNU/Linux'ta ise */home* dizinine gidersiniz.

Şimdi ekranda *C:\Users* veya */home* ibaresi görünürken şu komutu vererek masaüstüne ulaşmaya çalışın:

```
cd Desktop
```

Gördüğünüz gibi, bu defa bu komut bizi masaüstüne götürmek yerine bir hata mesajı verdi. Çünkü o anda bulunduğumuz konumda artık *Desktop* bir alt dizin değil. Bu durumu teyit etmek için yine dir ve ls komutları yardımıyla dizin içeriğini kontrol edebilirsiniz. Gördüğünüz gibi, bu dizin içeriğini gösteren çıktıda *Desktop* adı görünmüyor. Eğer aldığınız çıktıda *Desktop* yoksa, elbette cd Desktop komutu sizi masaüstüne götürmeyecektir. Böyle bir durumda masaüstüne ulaşmak için Windows'ta şöyle bir komut vermemiz gerekir:

```
cd falanca\Desktop
```

GNU/Linux'ta ise:

```
cd istihza/Desktop
```

Buradaki mantığı kavradığınızı zannediyorum. Yukarıdaki komutları verirken bulunduğumuz konum Windows'ta *C:\Users*, GNU/Linux'ta ise */home*. Bu konum ile masaüstü arasında Windows'ta *falanca*, GNU/Linux'ta ise *istihza* klasörleri yer alıyor. Dolayısıyla bu konumdan masaüstüne ulaşabilmek için önce *falanca* klasörüne (GNU/Linux'ta *istihza* klasörüne), oradan da *Desktop* klasörüne ulaşmamız gerekiyor.

Alternatif olarak, bir dizine ulaşmak için o dizinin tam adresini de yazabiliriz. Örneğin GNU/Linux'ta o anda hangi konumda bulunursak bulunalım, şu komutla masaüstüne ulaşabiliriz:

```
cd /home/istihza/Desktop
```

Dikkat ederseniz, burada kök dizinden (*/*) itibaren *Desktop* dizinine kadar olan yolu eksiksiz bir biçimde yazdık.

Aynı işlev için Windows'ta şöyle bir komut kullanıyoruz:

```
cd C:\Users\falanca\Desktop
```

Burada da *C:* dizininden *Desktop* dizinine kadar olan yolu eksiksiz olarak yazdık. Böylece komut satırında o anda hangi konumda bulunduğumuzdan bağımsız olarak, masaüstünün bulunduğu konuma geçebilmiş olduk.

Çevre Değişkenleri

Bu noktaya gelinceye kadar komut satırına ilişkin epey şey öğrendiniz. O halde artık size şöyle bir soru sorabilirim:

Diyelim ki bir program yazdınız. Kullanıcılarınızın programınızı kolaylıkla çalıştırabilmesini sağlamak için de programınızın kısayolunu kullanıcılarınızın masaüstlerine yerleştirmek istiyorsunuz. Peki ama bu işlemi nasıl yapacaksınız?

Böyle bir şey yapabilmek için, kullanıcılarınızın masaüstüne giden yolu tespit edebilmeniz lazım. Ama burada şöyle bir problem var. Bildiğiniz gibi herkesin masaüstüne giden yol aynı değil. Bir bilgisayardaki masaüstünü bulabilmek için, o bilgisayarı kullanan kişinin kullanıcı adını da biliyor olmanız lazım. Çünkü masaüstünün bulunduğu dizin kullanıcı adına bağlı olarak farklı olacaktır.

Mesela */home/istihza/Desktop* veya *C:\Users\selin\Desktop*

Hatta işletim sisteminin dilinin Türkçe veya İngilizce olmasına göre de masaüstünün yolu farklı olabilir. Mesela *C:\Kullanıcılar\sami\Desktop*...

Peki biz bunca farklılıkla karşı karşıyayken, masaüstüne giden yolu sağlıklı bir şekilde nasıl tespit edeceğiz?

İşte böyle bir durumda imdadımıza çevre değişkenleri (veya 'ortam değişkenleri') denen birtakım araçlar yetişecek. Peki nedir bu çevre değişkenleri denen şey?

Çevre değişkenleri, kullandığımız işletim sisteminde belli değerlerin atandığı birtakım isimlerdir. Bu tanım yeterince açık olmayabilir. O yüzden isterseniz çevre değişkenlerini bir örnekle anlatmaya çalışalım.

Mesela Windows komut satırında şu komutu verelim:

```
echo %USERPROFILE%
```

Bu komut şuna benzer bir çıktı verir:

```
C:\Users\falanca
```

GNU/Linux'ta aynı işlev için şu komutu kullanıyoruz:

```
echo $HOME
```

Bu da şuna benzer bir çıktı verir:

```
/home/istihza
```

İşte Windows'ta *%USERPROFILE%*, GNU/Linux'ta ise *$HOME* adlı bu değişkenlere teknik dilde 'çevre değişkeni' (*environment variable*) adı verilir.

Not

Burada gördüğümüz `echo` komutu herhangi bir değeri ekrana yazdırmamızı sağlayan bir sistem komutudur. Örneğin yukarıda *$HOME* ve *%USERPROFILE%* değişkenlerinin değerini ekrana basabilmek için `echo` komutundan yararlandık. Bu komut hem Windows'ta hem de GNU/Linux'ta aynı şekilde çalışır.

Gördüğünüz gibi Windows'ta *%USERPROFILE%*, GNU/Linux'ta ise *$HOME* adlı çevre değişkeni, kullanıcı dizininin adını içinde saklıyor. Dolayısıyla bu komut Ahmet adlı kişinin bilgisayarında çalıştırılırsa farklı, Ayşe adlı kişinin bilgisayarında çalıştırılırsa farklı bir çıktı verecektir. Bu değişkenleri kullanarak, masaüstüne giden yolu çok rahat bir şekilde tespit edebilirsiniz:

[**Windows**]

```
echo %USERPROFILE%\Desktop
```

[**GNU/Linux**]

```
echo $HOME/Desktop
```

Gördüğünüz gibi, yukarıdaki değişkenleri kullandığımızda, işletim sistemimiz bu değişkenlerin değerini otomatik olarak yerine koyabiliyor.

Bu sayede, bir program yazdığınızda, programınızı kullanan kişinin masaüstünün tam adresini tespit edip, programınızın kısayolunu, programınızı kuran kişinin masaüstüne otomatik olarak atabilirsiniz.

Tahmin edebileceğiniz gibi, bu çevre değişkenlerini kullanarak, dizinler arasında dolaşma işlemlerinizi de kolaylaştırabilirsiniz:

Bildiğiniz gibi, dizinler arasında dolaşmak için cd adlı bir komuttan yararlanıyoruz. Mesela masaüstüne ulaşmak için cd Desktop gibi bir komut kullanıyoruz. Ancak bu komutla masaüstüne ulaşabilmemiz için, masaüstünün o anda bulunduğumuz dizin konumuna göre bir alt dizin olması gerekiyor. Şimdiye kadar yaptığımız örnekler, masaüstünün bir alt dizin olmadığı durumlarda masaüstüne cd komutu ile ulaşmanın sıkıntılı bir iş olduğunu bize gösterdi. Ama neyse ki işletim sistemleri bize çevre değişkenleri gibi bir kolaylık sunuyor. Bu değişkenleri kullanarak, hangi konumda olursak olalım rahatlıkla masaüstüne ulaşabiliriz:

[**Windows**]

```
cd %USERPROFILE%\Desktop
```

[**GNU/Linux**]

```
cd $HOME/Desktop
```

Bunun dışında, GNU/Linux işletim sistemlerinde ~ işareti de kullanıcı dizininin adını tutar. Örneğin, kullandığınız GNU/Linux işletim sisteminin komut satırında şu komutu verin:

```
echo ~
```

Bu komut şuna benzer bir çıktı verecektir:

```
/home/istihza
```

Dolayısıyla GNU/Linux'ta şu komutu hangi konum altından verirseniz verin masaüstüne ulaşabilirsiniz:

```
cd ~/Desktop
```

Böylece 'çevre değişkeni' kavramını da anlatmış ve bu esnada birkaç önemli çevre değişkenini incelemiş olduk. Elbette GNU/Linux ve Windows'taki çevre değişkenleri yukarıda gösterdiklerimizden ibaret değildir. Sisteme ve o sistemi kullanan kişiye ilişkin pek çok önemli değeri içinde barındıran başka çevre değişkenleri de bulunur. Kullandığınız işletim sisteminde hangi çevre değişkenlerinin bulunduğunu öğrenmek için şu komutu verebilirsiniz:

```
set
```

Bu komut hem Windows'ta hem de GNU/Linux'ta aynı amaca hizmet eder. Yani çevre değişkenlerinin ve bu değişkenlere ait değerlerin ekrana basılmasını sağlar. Çıktı biçimi şöyledir:

```
ÇEVRE_DEĞİŞKENİ=değer
```

Bu listede gördüğünüz çevre değişkenlerini Windows'ta;

```
echo %ÇEVRE_DEĞİŞKENİ%
```

veya GNU/Linux'ta;

```
echo $ÇEVRE_DEĞİŞKENİ
```

formülüne göre kullanabilirsiniz.

Bu değişkenlerin bazılarını ilerleyen derslerde biz göstereceğiz, geri kalanlarını ise programlama maceranız sırasında yeri geldikçe kendiniz öğreneceksiniz. Biz şimdilik bu konuyu bir kenara bırakıp, komut satırının kullanımına ilişkin önemli başka bir konudan söz edelim.

Dizin Adı Tamamlama

Bir önceki bölümde gördüğünüz gibi, çevre değişkenlerini kullanarak dizin bulma ve dizinler arasında dolaşma işlemlerimizi kolaylaştırabiliyoruz. İşletim sistemlerinin dizinlere ilişkin bize sunduğu bir başka kolaylık da *Tab* (Sekme) tuşu ile ilgilidir.

Sözünü ettiğimiz bu kolaylığın ne olduğu anlamak için, komut satırında */home/istihza* veya *C:\Users\falanca* ifadesi görünürken, cd yazıp bir boşluk bıraktıktan sonra iki kez klavyedeki *Tab* tuşuna basın. Bu işlem, bir alt dizinde yer alan bütün dizinleri listeleyecektir:

```
.adobe/          Dropbox/         .local/          .shotwell/
.aptitude/       .fontconfig/     .macromedia/     Templates/
.cache/          .gconf/          .mission-control/ .themes/
.compiz-1/       .gegl-0.0/       .mozilla/        .thumbnails/
.config/         .gimp-2.6/       .mplayer/        .thunderbird/
.dbus/           .gnome2/         Music/           Ubuntu One/
Desktop/         .gstreamer-0.10/ Pictures/        Videos/
Documents/       .gvfs/           .pki/
Downloads/       .icons/          Public/
.dropbox/        lang/            .pulse/
```

Not

Yukarıdaki çıktı Ubuntu işletim sisteminden. Sizin kullandığınız işletim sisteminin çıktısı elbette daha farklı olacaktır.

Not

Burada başında . işareti olanlar gizli, olmayanlar ise görünür dizinlerdir.

Şimdi aynı konumda cd D yazıp yine iki kez *Tab* tuşuna basın. Şöyle bir çıktı alacaksınız:

```
Desktop/   Documents/ Downloads/ Dropbox/
```

Gördüğünüz gibi, adı ‘D’ harfi ile başlayan dizinleri listeledik. *Tab* tuşunun, bu kod tamamlama özelliği sayesinde, ulaşmak istediğiniz dizin adının tamamını yazmak zorunda kalmadan, sadece ilk birkaç harfini yazıp *Tab* tuşuna basarak o dizine rahatlıkla ulaşabilirsiniz. Örneğin `cd De` yazıp iki kez *Tab* tuşuna basarak (eğer bir alt dizinde ilk harfleri ‘De’ olan başka bir dizin yoksa) doğrudan *Desktop* dizinine ulaşabilirsiniz.

Yukarıda anlattığımız dizin tamamlama özelliği GNU/Linux’ta geçerlidir. Bu anlattıklarımız Windows sistemlerinde bazı farklılıklar gösterir. Örneğin Windows’ta `cd` yazıp bir boşluk bıraktıktan sonra *Tab* tuşuna bir kez bastığımızda alfabe sırasına göre adı önce gelen dizin adı tamamlanır.

Kullandığınız işletim sisteminde dizin adı tamamlama özelliğinin nasıl çalıştığını deneme yanılma yöntemiyle rahatlıkla keşfedebilirsiniz.

Bu arada, elbette *Tab* tuşuna basıldığında yalnızca dizin adları tamamlanmaz. Aynı zamanda bu şekilde dosya adlarını da tamamlayabilirsiniz. `cd` komutu yalnızca dizin adlarıyla birlikte kullanılabildiği için bu komuttan sonra *Tab* tuşuna basıldığında yalnızca dizin adları tamamlanacaktır. Ama mesela eğer `echo` komutunu yazdıktan sonra *Tab* tuşuna basarsanız dizinlerle birlikte dosyaların da tamamlandığını görürsünüz.

Dizin Ayraçları

Yukarıda verdiğimiz örneklerde belki dikkatinizi çekmiştir. GNU/Linux’ta dizin ayracı olarak / (düz taksim) işaretini, Windows’ta ise \ (ters taksim) işaretini kullanıyoruz.

[Windows]

```
cd %USERPROFILE%\Desktop
```

[GNU/Linux]

```
cd $HOME/Desktop
```

Çoğu durumda düz taksim işareti (/) hem Windows’ta hem de GNU/Linux işletim sisteminde çalışacaktır. Ancak ters taksim işareti (\) yalnızca Windows’ta çalışır. Bu işareti GNU/Linux dizinlerini ayırmak için kullanamayız. Dolayısıyla yazdığınız programların birden fazla platform üzerinde çalışmasını istiyorsanız, her iki işletim sistemi için de ters taksim yerine düz taksim işaretini kullanmanız daha mantıklı olacaktır.

Ters ve düz taksim işaretlerinden hangisini kullandığınız, dosya-dizin adı tamamlama işlemleri için *Tab* düğmesine basarken önemlidir sadece.

Yani eğer Windows’ta dizinleri ayırmak için düz taksim işaretini kullanırsanız, dosya-dizin tamamlama özelliğinden yararlanamazsınız. Mesela o anda bulunduğumuz dizin altında *falanca* adlı bir klasörün altındaki *filanca* adlı başka bir klasöre ulaşmaya çalıştığımızı varsayalım:

```
cd falanca\fil
```

yazıp *Tab* düğmesine basarsak, Windows *filanca* dizinini bizim için tamamlayacaktır. Ama eğer yukarıdaki komutu ters taksim ile değil de düz taksim ile yazarsak bu tamamlama işlemi gerçekleşmez:

```
cd falanca/fil
```

filanca dizinine ulaşmak için dizinin yolunu tam olarak yazmamız gerekir:

```
cd falanca/filanca
```

Ama dediğimiz gibi, çoğu durumda her iki işletim sistemi için de düz taksim işaretini kullanmak çok daha makul bir yoldur.

Sembolik Bağlar

Buraya kadar hem Windows'ta hem de GNU/Linux'ta komut satırının nasıl kullanılacağından söz ettik. Sizden beklentimiz buraya kadar anlatılanları iyice sindirmeden yola devam etmemenizdi.

Eğer siz bir Windows kullanıcısı iseniz buraya kadar anlatılan konuları bilmeniz bu kitabı takip edebilmeniz açısından yeterli olacaktır. Ama eğer siz bir GNU/Linux kullanıcısı iseniz komut satırına ilişkin biraz daha fazla şey biliyor olmanız gerekiyor. O yüzden buradan itibaren yalnızca GNU/Linux kullanıcılarını ilgilendiren konulardan söz edeceğiz. Dolayısıyla Windows kullanıcıları eğer isterlerse okumayı burada kesip bir sonraki konuya geçebilir. Ama, 'fazla bilginin göz çıkarmayacağı' düsturundan hareketle, ben Windows kullanıcılarına konunun geri kalanını da GNU/Linux kullanıcıları ile beraber okumalarını tavsiye ederim...

İlk olarak sembolik bağları ele alacağız.

Sembolik bağ; bir dosyaya veya dizine bağlantı içeren özel bir dosya türüdür. GNU/Linux işletim sistemlerinde dosyalar ve dizinler birbirlerine sembolik bağ ile bağlanabilir. Peki bir insan neden sembolik bağ oluşturmak ister?

Bunun pek çok farklı sebebi olabilir. Örneğin bazen bir dosyanın adını değiştirmeden, o dosyaya başka bir isimle erişmeniz gereken durumlarla karşılaşabilirsiniz. Mesela orijinal dosyanın adı çok uzunsa, siz sistemde herhangi bir soruna yol açmamak için o dosyanın adını değiştirmeden, o dosyaya daha kısa bir isimle erişmek istiyor olabilirsiniz. Böyle durumlarda yapılabilecek en mantıklı iş o dosyaya bir sembolik bağ vermek olacaktır. Örneğin *$HOME* dizini altında *çok_uzun_bir_dosya_adı* adlı bir dosyanız olduğunu düşünün. Sizin amacınız bu dosyaya mesela sadece *dosya* gibi bir adla erişebilmek. Elbette isterseniz dosyanın adını değiştirebilirsiniz, ama bu durumda eğer o dosyayı kullanan başka uygulamalar varsa, isim değişikliği durumunda o uygulamalar da artık çalışmaz hale gelecektir. İşte böyle bir durumu önlemek için, dosyayı yeniden adlandırmak yerine o dosyaya bir sembolik bağ vermeyi tercih edebilirsiniz. Bunun için `ln` adlı bir komuttan yararlanacağız:

```
ln -s $HOME/çok_uzun_bir_dosya_adı $HOME/dosya
```

Böylece *$HOME* dizini altındaki *çok_uzun_bir_dosya_adı* adlı dosyaya, *$HOME* dizini altında *dosya* adlı bir sembolik bağ vermiş olduk. Burada ln komutunu *-s* parametresi ile kullandığımıza dikkat edin.

Sembolik bağ oluşturmanın gerekli olduğu bir başka durum da şudur: Mesela bilgisayarınıza kurduğunuz bir program, bir kütüphanenin belli bir konumda olmasını şart koşuyor olabilir. Örneğin Mozilla Firefox adlı program, flash kütüphanesini */usr/lib/mozilla/plugins/* dizini altında arar. Eğer sistemde başka bir konumda zaten *libflashplayer.so* adlı kütüphane mevcutsa, siz basit bir sembolik bağ aracılığıyla bu kütüphaneye */usr/lib/mozilla/plugins/* dizini altından bir bağlantı verebilirsiniz. Mesela *libflashplayer.so* adlı kütüphanenin */usr/lib/plugins* dizini altında bulunduğunu varsayalım:

```
ln -s /usr/lib/plugins/libflashplayer.so /usr/lib/mozilla/plugins/libflashplayer.so
```

Böylece hem flash kütüphanesini Firefox'a başarıyla tanıtmış, hem de aynı dosyayı iki farklı yere kopyalamak zorunda kalmamış olursunuz.

Elbette, eğer sembolik bağ oluşturacağınız dizin ev dizininiz dışında ise bağ oluşturma işlemi sırasında *root* haklarına sahip olmanız gerekir. Yani mesela Ubuntu'da yukarıdaki komutu şu şekilde vermeniz gerekir:

```
sudo ln -s /usr/lib/plugins/libflashplayer.so /usr/lib/mozilla/plugins/libflashplayer.so
```

Bu arada sembolik bağlarla ilgili önemli bir bilgi daha verelim: Eğer orijinal dosyada bir değişiklik yapılırsa bu değişiklik bağ dosyasını da etkiler. Böylece ana dosyada bir değişiklik yaptığınızda aynı değişikliği bir de bağ dosyasında yapmak zahmetine girmezsiniz.

Eğer oluşturduğunuz bir sembolik bağı kaldırmak isterseniz, bağ dosyasını silmeniz yeterli olacaktır. Yalnız burada bağ dosyası yerine yanlışlıkla ana dosyayı silmemeye dikkat etmelisiniz.

Son olarak, sembolik bağlarla ilgili bilmemiz gereken önemli bir konu da şu: Bağlayacağımız dosyaların bulunduğu dizin adlarını yazarken, bu dizin adlarını kök dizinden itibaren eksiksiz olarak yazmamız gerekiyor. Aksi halde oluşan bağ dosyasının içi boş olacak ve hiçbir işe yaramayacaktır.

Yukarıda verdiğimiz bilgiler GNU/Linux işletim sistemleri için geçerlidir. Benzer bir özellik Windows işletim sistemleri için de geçerlidir. Ancak bu kitap açısından Windows'ta sembolik bağ oluşturma işlemi bizi ilgilendirmiyor. Biz burada sembolik bağları yalnızca GNU/Linux işletim sisteminin söz konusu olduğu durumlarda kullanacağız. Eğer Windows'ta sembolik bağların nasıl oluşturulacağıyla ilgileniyorsanız mklink ve fsutil komutları hakkında bir araştırma yapmanızı tavsiye ederim.

Çalıştırma Yetkisi

Bildiğiniz gibi, GNU/Linux işletim sistemlerinde kullanıcılar dosya ve dizinler üzerinde bazı haklara

sahiptir veya değildir. Örneğin normal bir kullanıcı kendi *$HOME* dizini içindeki dosya ve dizinler üzerinde hem okuma hem de yazma yetkisine sahiptir. *$HOME* dizini dışındaki dosya ve dizinleri ise yalnızca okuyabilir. Mesela normal bir kullanıcı */usr/bin* dizini içindeki dosyalar üzerinde herhangi bir değişiklik yapamaz.

Aynı şekilde, bir kullanıcı öntanımlı olarak bir dosyayı çalıştırma yetkisine de sahip değildir. Yani o dosya bir program da olsa, o dosyayı çalıştırabilmek için öncelikle o dosya üzerinde çalıştırma yetkisine sahip olmanız gerekir. Ya da başka bir deyişle, bir program dosyasının çalıştırılabilmesi için o dosyanın 'çalıştırılabilir' olarak işaretlenmiş olması gerekir. Çalıştırma yetkiniz olmayan bir program dosyasını çalıştırmaya teşebbüs etmeniz durumunda *permission denied* [izin verilmedi] hatası alırsınız.

Dilerseniz bununla ilgili basit bir örnek verelim. Şimdi masaüstünde *deneme* adlı bir dosya oluşturun ve bu dosyanın içini açarak şu satırları harfi harfine yazın:

```
#!/bin/sh
echo "merhaba"
```

Çoğu GNU/Linux kullanıcısının bileceği gibi, bu basit bir kabuk betiğidir.

Not

Kabuk hakkında bilgi için bkz. http://belgeler.gen.tr/bashref/

Bu kabuk betiğini çalıştırmak için komut satırında masaüstüne gelerek şu komutu verin:

```
./deneme
```

Bu komut şu hatayı verecektir:

```
-bash: ./deneme: Permission denied
```

İşte bu hatanın nedeni, betiğimiz üzerinde çalıştırma yetkisine sahip olmamamızdır. Dosya izinleriyle ilgili bu hatayı almamak için iki seçeneğiniz var. Birinci seçenek olarak, dosyaya sağ tıklayıp dosya özelliklerinden bu dosyayı 'çalıştırılabilir' olarak işaretleyebilirsiniz. Ama bu işlemin grafik arayüz üzerinden nasıl yapılacağı, GNU/Linux dağıtımlarında farklılıklar gösterebilir. O yüzden bu işi yapmanın en kestirme ve kesin yöntemi chmod adlı bir komuttan yararlanmaktır. Bu komut yardımıyla betiğimiz üzerinde çalıştırma yetkisine sahip olabiliriz:

```
chmod +x deneme
```

Yukarıdaki komutu şu şekilde de verebilirsiniz:

```
chmod a+x deneme
```

Bu komutu bu şekilde verdiğinizde, *deneme* adlı dosya için, sistemdeki herkese çalıştırma yetkisi vermiş oluyorsunuz. Yani işletim sisteminizde oturum açan herkes *deneme* adlı bu programı çalıştırabilecektir. Eğer yukarıdaki komutu şu şekilde verirseniz dosyayı yalnızca dosyanın sahibi, yani siz, çalıştırabilirsiniz:

```
chmod u+x deneme
```

Buradaki *u* parametresi İngilizcedeki *user* (kullanıcı) kelimesinin kısaltmasıdır. Tahmin edebileceğiniz gibi, bir önceki komutta gördüğümüz *a* parametresi ise *all* (herkes, hepsi) kelimesinin...

Bu komutlardan sonra artık `./deneme` komutu başarıyla çalışacaktır. Eğer bir dosyadan çalıştırma yetkisini geri almak isterseniz şu komutu verebilirsiniz:

```
chmod -x deneme
```

Burada + işareti yerine - işaretini kullandığımıza dikkat edin.

Bu arada, *./deneme* komutunda gördüğünüz *./* işaretleri, o anda içinde bulunduğunuz dizinde yer alan *deneme* adlı bir dosyayı çalıştırmak istediğinizi gösterir. Eğer bu komutu yalnızca *deneme* olarak verirseniz, işletim sisteminiz *deneme* adlı dosyayı o anda içinde bulunduğunuz dizinde değil, `echo $PATH` adlı özel bir komutun çıktısında görünen dizinler içinde arayacak ve eğer program dosyasını o dizinler içinde bulamazsa da size şöyle bir hata mesajı gösterecektir:

```
-bash: deneme: command not found
```

Eğer `echo $PATH` komutunun çıktısında görünen dizinler içinde tesadüfen *deneme* adlı başka bir dosya varsa, yukarıdaki komutu *./* işaretleri olmadan verdiğinizde kendi yazdığınız program dosyası değil, o dosya çalışacaktır. O yüzden, o anda içinde bulunduğumuz dizinde yer alan program dosyalarını çalıştırırken *./* işaretlerini kullanmaya dikkat ediyoruz.

İlerleyen derslerde `echo $PATH` komutunun anlam ve öneminden söz edeceğiz. Biz şimdilik bu konuyu bir kenara bırakıp başka bir konuya geçelim.

Dosya kopyalama, Taşıma ve Silme

Windows işletim sistemlerinde bir kullanıcı, sistemdeki her konuma istediği dosyayı kopyalayabilir, taşıyabilir veya bu dosyayı silebilir. Ancak GNU/Linux dağıtımlarında yetkisiz bir kullanıcı ancak ev dizini içindeki dosyalara müdahale edebilir. Bir dosyayı, örneğin *$HOME* dizininden */usr/bin* dizini içine kopyalamak isterse *root* haklarını alması gerekir.

GNU/Linux dağıtımlarında bu tür işlemleri dosya yöneticileri aracılığıyla grafik olarak yapmanın bazı yöntemleri olsa da en kestirme yol kopyalama, taşıma ve silme gibi işlemleri komut satırı üzerinden halletmektir. Özellikle ev dizini dışındaki dosyalara müdahale ederken komut satırını kullanmak işlerinizi epey hızlandıracaktır. İşte biz de işlerinizi kolayca yapmanızı sağlamak için bu

bölümde GNU/Linux işletim sistemlerinde dosya kopyalama, taşıma ve silme işlemlerinin komut satırı aracılığıyla nasıl yapılacağını inceleyeceğiz.

cp

GNU/Linux dağıtımlarında bir dosyayı başka bir konuma kopyalamak için cp adlı bir komuttan yararlanıyoruz. Bu komut şöyle kullanılıyor:

```
cp özgün_konumdaki_dosya hedef_konumdaki_dosya
```

Örneğin masaüstünde bulunan *deneme.txt* adlı bir dosyayı */usr/bin* dizinine aynı adla kopyalamak için şöyle bir komut yazıyoruz:

```
cp $HOME/Desktop/deneme.txt /usr/bin/
```

Gördüğünüz gibi, eğer dosyayı bulunduğu konumdan farklı bir konuma aynı adla kopyalayacaksak dosyanın adını tekrar yazmamıza gerek yok. Ama eğer *deneme.txt* adlı bu dosyayı */usr/bin* dizinine farklı bir adla kopyalamak isterseniz yukarıdaki komutu şöyle yazabilirsiniz:

```
cp $HOME/Desktop/deneme.txt /usr/bin/dosyanın_yeni_adı
```

Elbette, daha önce de söylediğimiz gibi, */usr/bin* dizinine bir dosya kopyalayabilmek için *root* haklarına sahip olmalısınız. Dolayısıyla yukarıdaki komutları, başlarına *sudo* getirerek (veya önce *su -* komutuyla *root* haklarını alarak) yazmalısınız:

```
sudo cp $HOME/Desktop/deneme.txt /usr/bin/deneme.txt
```

Eğer bir dizinin tamamını başka bir konuma kopyalamak istiyorsanız, yukarıdaki komutları *-rf* parametresi ile çalıştırmanız gerekir. Yani:

```
cp -rf özgün_dizin hedef_dizin_adresi
```

veya:

```
sudo cp -rf özgün_dizin hedef_dizin_adresi
```

rm

GNU/Linux dağıtımlarında bir dosyayı silmek için rm adlı bir komuttan yararlanıyoruz. Örneğin masaüstünde bulunan *deneme.txt* adlı dosyayı silmek için şu komutu veriyoruz:

```
rm $HOME/Desktop/deneme.txt
```

Elbette eğer sileceğimiz dosya ev dizini dışındaysa bu komutu *root* yetkileri ile çalıştırmamız gerekir:

```
sudo rm /usr/bin/deneme.txt
```

rm komutu yukarıdaki şekilde ancak dosyaları silmek için kullanılabilir. Eğer silmek istediğiniz şey bir dosya değil de dizin ise o zaman rm komutunu -rf parametresi ile birlikte çalıştırmamız gerekir:

```
rm -rf $HOME/silinecek_dizin
```

Eğer silinecek dizin ev dizini dışındaysa *root* haklarını almayı unutmuyoruz:

```
sudo rm -rf /usr/share/silinecek_dizin
```

mv

GNU/Linux dağıtımlarında bir dosyayı bir yerden başka bir yere taşımak için mv adlı bir komuttan yararlanacağız:

```
mv özgün_konumdaki_dosya hedef_konumdaki_dosya
```

Örneğin masaüstünde bulunan *deneme.txt* adlı bir dosyayı */usr/bin* dizinine taşımak için şöyle bir komut yazıyoruz:

```
sudo mv $HOME/Desktop/deneme.txt /usr/bin/
```

Bu komut masaüstündeki *deneme.txt* dosyasını silip */usr/bin/* dizini altında *deneme.txt* adlı başka bir dosya oluşturacak, yani dosyayı masaüstünden */usr/bin* dizinine taşımış olacaktır.

Eğer *deneme.txt* dosyasını farklı bir adla taşımak isterseniz yukarıdaki komutu şöyle yazabilirsiniz:

```
sudo mv $HOME/Desktop/deneme.txt /usr/bin/yeni_ad
```

Yukarıdaki örneklerden de gördüğünüz gibi, mv komutu bir dosyanın konumunu değiştirmek için kullanılıyor. Ama isterseniz aynı komutu bir dosyanın adını değiştirmek için de kullanabilirsiniz:

```
mv $HOME/Desktop/deneme.txt $HOME/Desktop/yeni_ad
```

Bu komut masaüstünüzdeki *deneme.txt* dosyasının adını *yeni_ad* olarak değiştirecektir...

Böylece çok önemli bir konuyu geride bırakmış olduk. Buraya gelene kadar komut satırına ilişkin epey şey öğrendiniz. Böylece en azından bu kitabı takip etmenizi sağlayacak temel komut satırı bilgilerini edinmiş oldunuz. Bu kitaptan yararlanabilmek için komut satırına ilişkin olarak yukarıda anlattıklarımızı bilmeniz yeterli olacaktır. Ancak eğer yetkin bir programcı olmak istiyorsanız, programlama faaliyetlerinizi yürüttüğünüz işletim sisteminin komut satırını çok iyi tanımanızı tavsiye ederim.

başa dön

YOL (*PATH*) Kavramı

Geçen bölümde, komut satırı hakkında bilmemiz gereken temel şeyleri öğrendik. Ama işimiz henüz bitmedi. Programcılık faaliyetlerimizi sağlıklı ve sağlam bir zemin üzerinde yürütebilmemiz için öğrenmemiz gereken bir kavram daha var. O kavramın adı *PATH*, yani YOL. İşte biz bu bölümde, programcılık maceramız açısından bir hayli önemli bir kavram olan bu YOL (*PATH*) kavramından söz edeceğiz. Peki nedir bu YOL denen şey?

YOL Nedir?

Bir programın veya dosyanın yolu, kabaca o programın veya dosyanın içinde yer aldığı dizindir. Örneğin GNU/Linux işletim sistemlerinde önemli bir sistem dosyası olan *fstab* dosyasının yolu */etc/fstab*'dır. Windows işletim sisteminde ise, öntanımlı internet tarayıcısı olan Internet Explorer adlı programın yolu *C:\Program Files\Internet Explorer\iexplore.exe*'dir. Bu örneklerden de anlaşılacağı gibi, bir dosyanın ya da programın yolu, basitçe o dosyanın ya da programın bilgisayardaki tam adresidir.

Yol kelimesinin bir de daha özel bir anlamı bulunur. Bilgisayar dilinde, çalıştırılabilir dosyaların (*executables*) içinde yer aldığı dizinlerin adlarını tutan özel bir çevre değişkeni vardır. İşte bu çevre değişkenine *PATH* (YOL) adı verilir ve bu anlamda kullanıldığında yol kelimesi genellikle büyük harfle yazılır. Gelin isterseniz bu tanımı biraz daha açalım.

İşletim sistemleri, bir programı komut satırından ismiyle çağırdığımızda söz konusu programı çalıştırabilmek için bazı özel dizinlerin içine bakar. İlgili programın çalıştırılabilir dosyası eğer bu özel dizinler içindeyse, işletim sistemi bu dosyayı bulur ve çalıştırılmasını sağlar. Şimdi bu konuyu daha iyi anlayabilmek için birkaç deneme yapalım.

GNU/Linux sistemimizde hemen bir konsol ekranı açıp şu komutu veriyoruz:

```
echo $PATH
```

Bu komutun çıktısı şuna benzer bir şey olacaktır:

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

Windows işletim sisteminde ise aynı işlev için komut satırında şu komutu veriyoruz:

```
echo %PATH%
```

Bu da şuna benzer bir çıktı verecektir:

```
C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;
C:\Windows\System32\WindowsPowerShell\v1.0\;
```

İşte bu çıktı bize YOL değişkeni (İngilizcede *PATH variable*) dediğimiz şeyi gösteriyor. İşletim sistemimiz, bir programı çağırmak istediğimizde, yani komut satırında programın adını yazıp *Enter* tuşuna bastığımızda öncelikle yukarıda adı verilen dizinlerin içini kontrol edecektir. Eğer çağırdığımız programın çalıştırılabilir dosyası bu dizinlerden herhangi birinin içinde ise o programı ismiyle çağırabiliyoruz.

GNU/Linux işletim sisteminde bir programın YOL'unun ne olduğunu bulmak için which adlı bir sistem komutundan yararlanabiliriz. Örneğin Gedit programının YOL'unu bulmak için şu komutu verelim:

```
which gedit
```

Bu komut */usr/bin/gedit* çıktısını verecektir. Hemen yukarıda echo $PATH komutunun çıktısını kontrol ediyoruz ve görüyoruz ki */usr/bin/* dizini YOL değişkenleri arasında var. Dolayısıyla, Gedit programı YOL üstündedir, diyoruz. Zaten Gedit programının YOL üstünde olması sayesinde, konsolda sadece gedit komutunu vererek Gedit programını çalıştırabiliyoruz.

Bu arada elbette which gedit komutunun düzgün çıktı verebilmesi için Gedit adlı programın sisteminizde kurulu olması gerekiyor. Bu program genellikle GNOME ve UNITY masaüstü ortamlarının kurulu olduğu GNU/Linux dağıtımlarında bulunur. Eğer KDE adlı masaüstü ortamını kullanıyorsanız Gedit adlı program sizde kurulu olmayabilir. Eğer öyleyse siz Gedit yerine, sisteminizde kurulu olduğundan emin olduğunuz başka bir programın adını verebilirsiniz. Mesela kwrite veya kate.

Gelelim Windows'a...

Windows'ta ise GNU/Linux'taki gibi çalıştırılabilir dosyaların yolunu bulmamızı sağlayan *which* benzeri bir program kurulu olarak gelmez. Ama eğer isterseniz pankaj-k.net/archives/upload/which.cmd adresinden indireceğiniz *which.cmd* adlı betiği *C:\WINDOWS* dizini altına kopyalayarak benzer etkiyi elde edebilirsiniz.

Bu betiği indirip *C:\WINDOWS* dizini altına *which.cmd* adıyla kaydettiğinizi varsayarak komut satırında şöyle bir deneme yapabiliriz:

```
which notepad.exe
```

Bu komut şöyle bir çıktı verir:

```
Found in PATH: C:\WINDOWS\system32\notepad.exe
```

Demek ki *notepad.exe*'nin YOL'u buymuş. Hemen yukarıdaki echo %PATH% çıktısını kontrol ediyoruz ve görüyoruz ki *notepad.exe*'yi barındıran *C:\WINDOWS\system32* dizini YOL değişkeni içinde yer alıyor. O halde bu programı doğrudan ismiyle çağırabiliriz. Yani komut satırında doğrudan notepad.exe komutunu verirsek Notepad programı çalışmaya başlayacaktır. Bir de şunu deneyelim:

```
which iexplore.exe

Argument "iexplore.exe" not found in PATH
```

Demek ki *iexplore.exe* YOL üstünde değilmiş. O yüzden bu programı komut satırından doğrudan ismiyle çalıştıramıyoruz:

```
iexplore.exe 'iexplore.exe' iç ya da dış komut, çalıştırılabilir program ya
da toplu iş dosyası olarak tanınmıyor.
```

YOL üzerinde bulunmayan bir programı çalıştırmak için o programın tam yol adresini kendimiz yazmalıyız. Program YOL üstünde bulunmadığı için işletim sistemimiz bu programın çalıştırılabilir dosyasının nerede olduğunu bulamıyor. Bu yüzden programın nerede olduğunu işletim sistemimize bizim söylememiz gerekiyor. Mesela *iexplore.exe* YOL üstünde olmadığı için işletim sistemimizin bu programı sırf ismiyle çalıştırması mümkün değil. İşletim sistemimizin bu programı bulabilmesi için, bu programın tam yolunu elle bizim girmemiz gerekiyor:

```
"C:\Program Files\Internet Explorer\iexplore.exe"
```

Yukarıdaki komutu verirken tırnak işaretlerini koymayı unutmuyoruz. Ayrıca kullandığımız tırnak işaretlerinin çift tırnak olduğuna da dikkat edin. Windows işletim sistemi açısından tek ve çift tırnak işaretleri birbirinden farklıdır. Yukarıdaki komutu tırnaksız veya tek tırnak işaretleriyle çalıştırmayı denerseniz, 'Program Files' ve 'Internet Explorer' ifadeleri içinde geçen boşluk karakterleri nedeniyle Windows size bir hata mesajı gösterecektir.

Aynı şekilde GNU/Linux işletim sistemlerinde */etc* dizini de YOL üzerinde olmadığı için mesela bu dizinde yer alan *fstab* dosyasını doğrudan adıyla çağıramayız. Bu yüzden, bu dosyayı açmamız gereken durumlarda, dosyanın bulunduğu konumun tam adresini yazmamız lazım. Örneğin */etc/fstab* adlı dosyayı Gedit programıyla açmayı deneyelim:

```
gedit fstab
```

Bu komut, o anda bulunduğunuz dizin içinde *fstab* adlı boş bir dosya oluşturacaktır. Bizim açmaya çalıştığımız asıl *fstab* dosyası YOL üzerinde bulunmadığı için bu dosyayı sadece adını kullanarak açamayız. Bu dosyayı açabilmek için dosyanın konumunu eksiksiz bir biçimde yazmamız lazım:

```
gedit /etc/fstab
```

Bütün bu anlattıklarımız, özellikle GNU/Linux işletim sistemlerine kurmaya çalıştığınız bazı programların neden çalışmadığını açıklıyor olmalı. Eğer bir programı çalıştırmak için ismini komut satırında yazıyorsanız, ama bu komut o programı çalıştırmıyorsa ve siz de bu komutun doğru olduğuna eminseniz, muhtemelen o programın çalıştırılabilir dosyasını barındıran dizin YOL üzerinde değildir. Böyle bir durumda yapmanız gereken bazı işlemler var. Gelin bu işlemlerin neler olduğunu görelim.

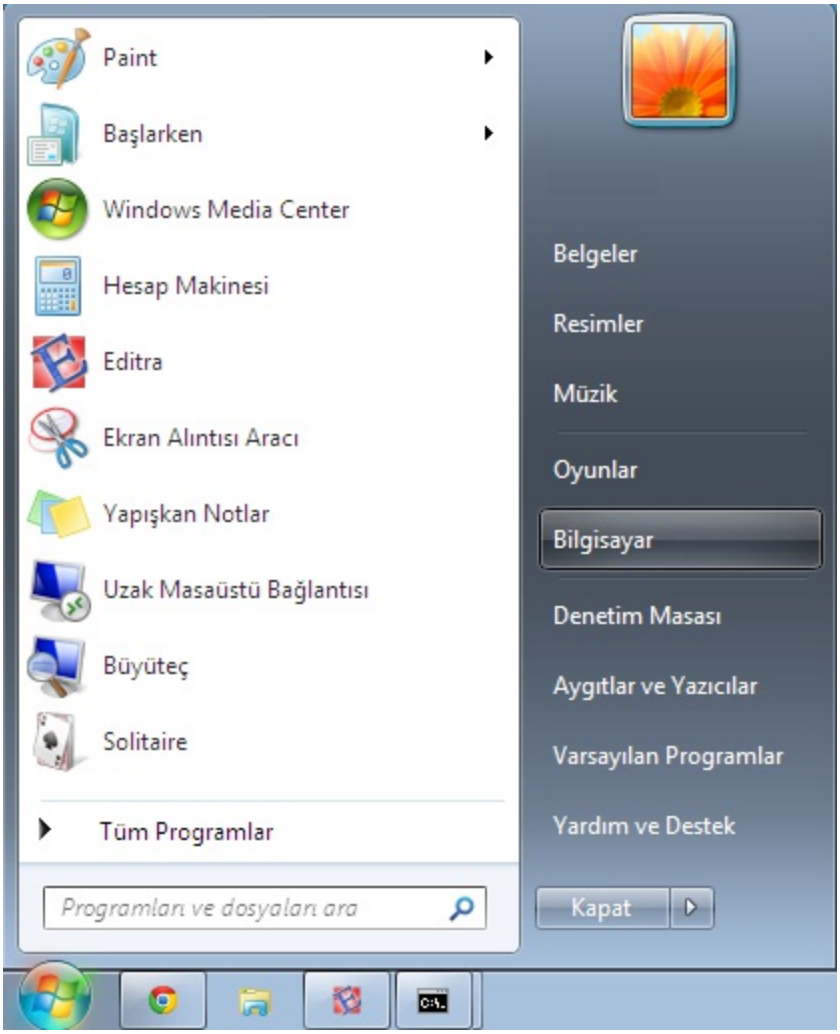
YOL'a Dizin Eklemek

Dediğimiz gibi, eğer bir program dizini YOL değişkenini oluşturan dizinler arasında yer almıyorsa, o programı komut satırından ismiyle çalıştıramayız. Böyle bir durumda o programı bizim elle YOL'a eklememiz gerekir. Peki bu işlemi nasıl yapacağız?

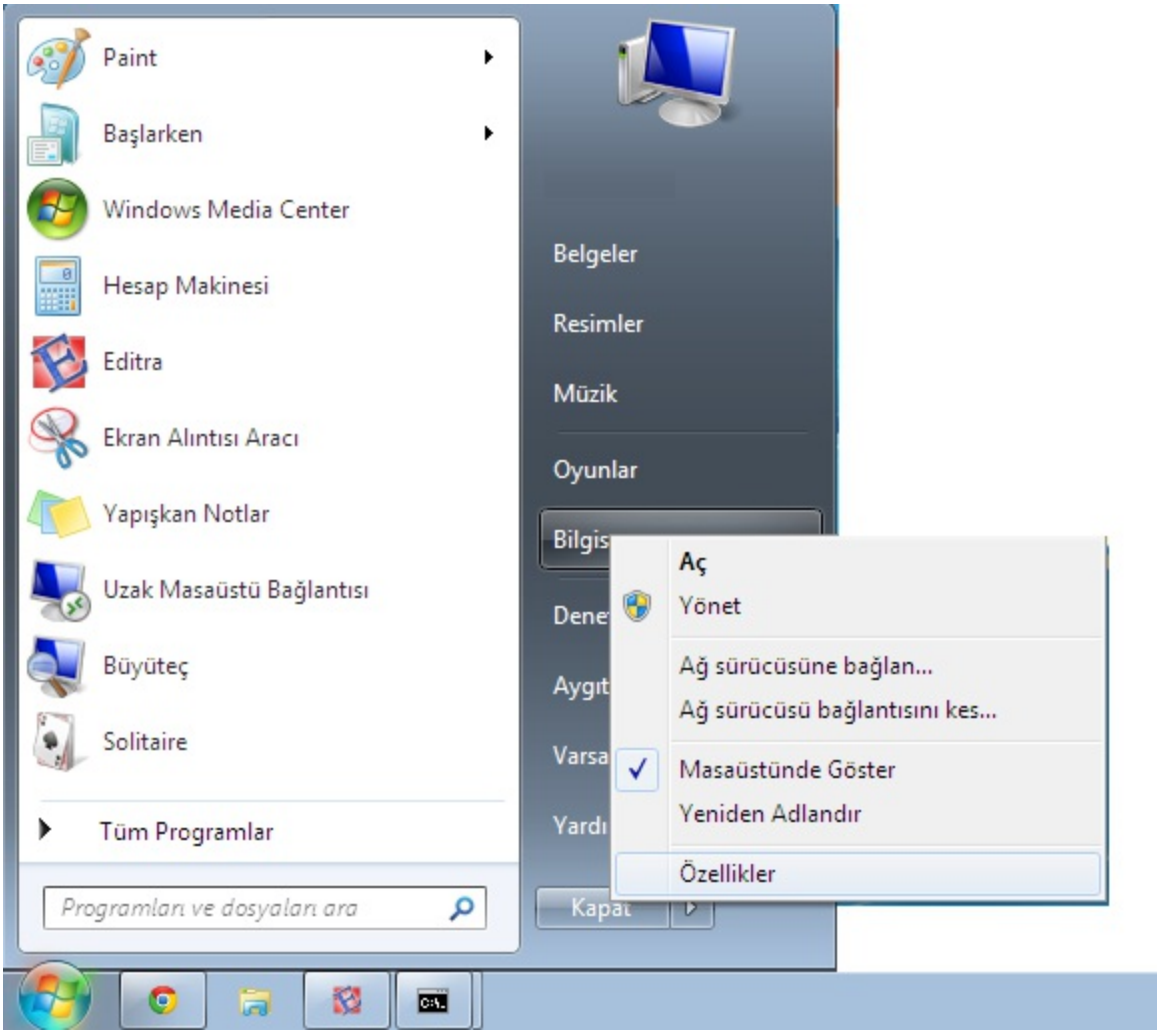
Öncelikle Windows'tan başlayalım...

Windows

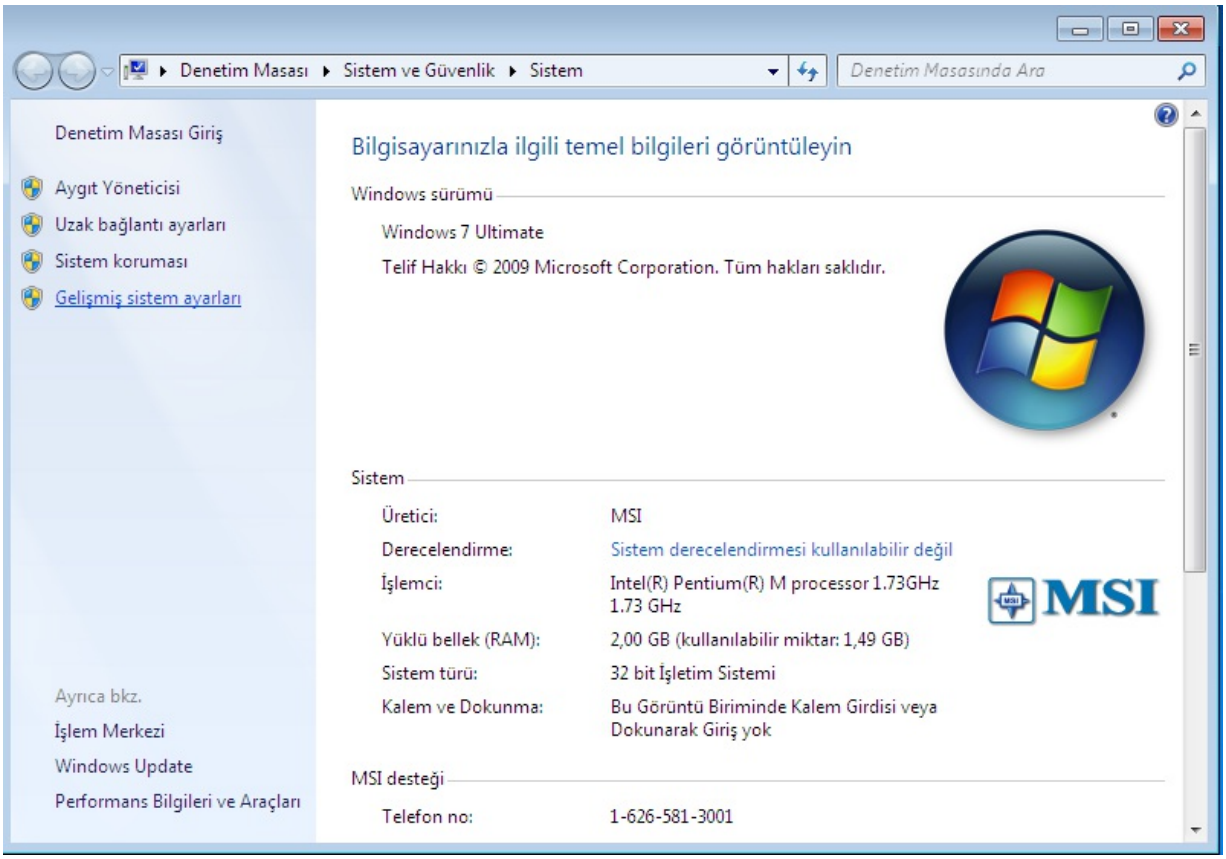
Windows'ta herhangi bir programı YOL'a eklemek için ilk olarak Başlat simgesine tıklıyoruz:



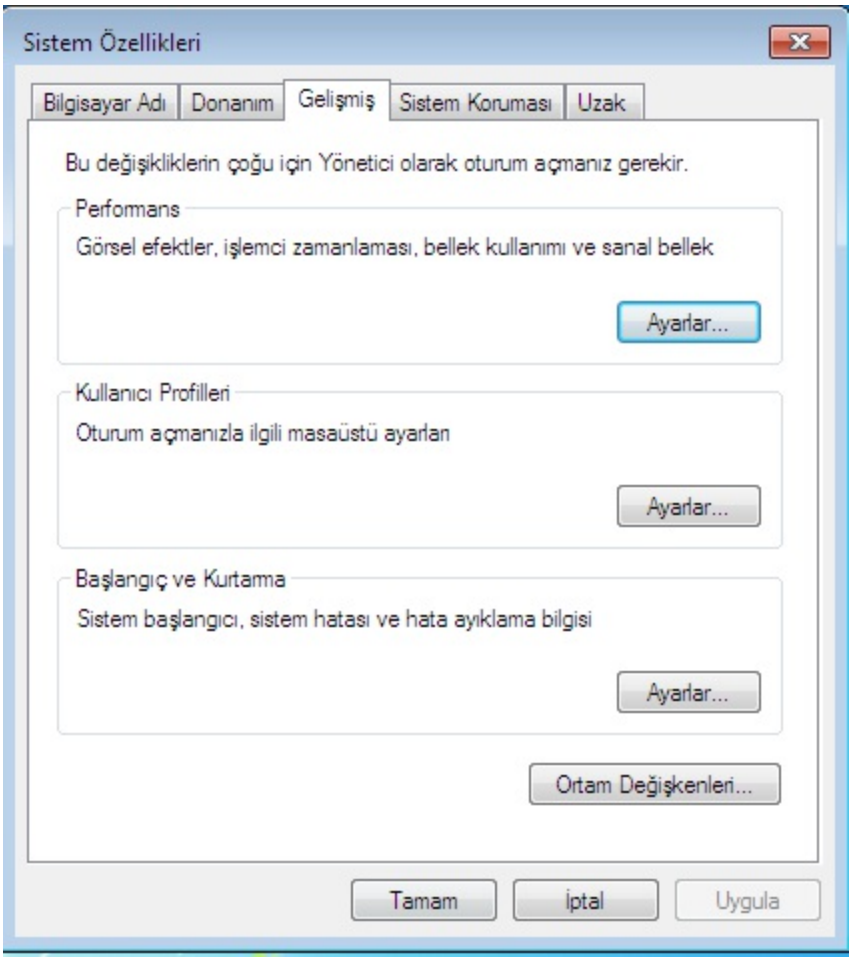
Burada menünün sağ tarafındaki listede yer alan 'Bilgisayar' düğmesine sağ tıklıyoruz ve şöyle bir ekranla karşılaşıyoruz:



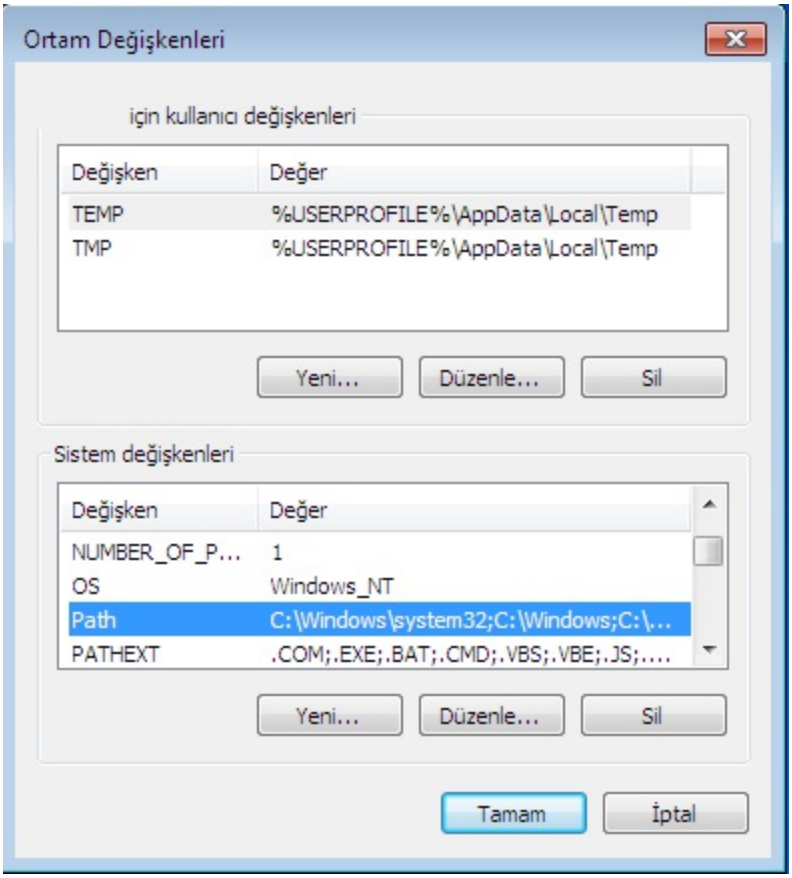
Karşımıza çıkan menünün en altındaki ‘Özellikler’ satırına tıklıyoruz. Bu satıra tıkladığımızda şöyle bir pencereyle karşılaşmamız gerekiyor:



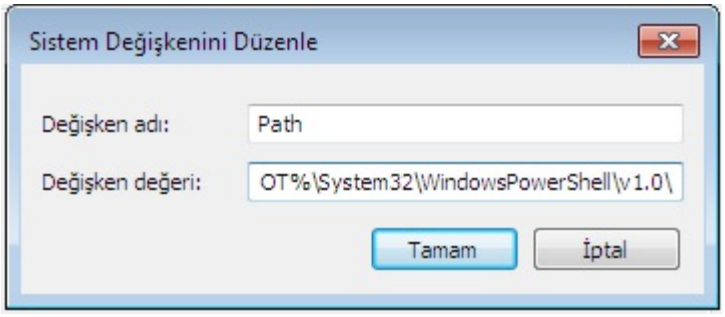
Burada, pencerenin sol tarafındaki seçenekler arasında ‘Gelişmiş Sistem Ayarları’ adlı bir satır göreceksiniz. O satıra tıklayın ve şu ekranı karşınıza alın:



Bu ekranda ‘Gelişmiş’ sekmesinin içinde ‘Ortam Değişkenleri’ adlı bir düğme göreceksiniz. Bu düğmeye tıkladığınızda ise karşınıza şöyle bir pencere açılacak:



Bu pencerede 'Sistem Değişkenleri' başlığı altında birtakım öğeleri barındıran bir liste göreceksiniz. O listede *Path* adlı girdiyi bulun. *Path* girdisine çift tıkladığınızda şöyle bir pencere ile karşılaşacaksınız:



Burada, 'Değişken Değeri' ifadesinin karşısında bir kutucuk ve bu kutucuğun içinde de birtakım dizin adları görüyorsunuz. İşte YOL'a eklemek istediğimiz programı barındıran dizini bu kutucuğa yazacağız.

Dikkat ederseniz bu kutucuğun içindeki değerler birbirinden *;* (noktalı virgül) işareti ile ayrılmış. Dolayısıyla biz de listenin en sonuna ekleyeceğimiz dizin adını *;* işaretini kullanarak önceki girdilerden ayıracağız.

Gelin isterseniz ufak bir deneme çalışması yapalım.

Şimdi yukarıda bahsettiğimiz kutucuğun içindeki listenin en sonuna şu ifadeyi ekleyin:

```
;C:\Program Files\Internet Explorer
```

Daha sonra *Tamam* düğmelerine basarak bütün pencereleri kapatın. Daha önce gösterdiğimiz şekilde komut satırına ulaşın ve orada şu komutu verin:

```
iexplore
```

Gördüğünüz gibi, artık bu komut bize bir hata mesajı vermek yerine, doğru bir şekilde Internet Explorer adlı programı çalıştırıyor. İşte bunun sebebi, bizim Internet Explorer programını barındıran dizini YOL'a eklemiş olmamızdır. Bu sayede biz iexplore komutunu verdiğimizde işletim sistemimiz ilgili programı nerede araması gerektiğini biliyor.

İsterseniz echo %PATH% komutunu verip *C:\Program Files\Internet Explorer* adlı dizinin gerçekten yola eklenip eklenmediğini kontrol edebilirsiniz.

Bu arada, YOL değişkeni üzerinde yaptığınız herhangi bir değişikliğin etkinleşebilmesi için açık olan bütün MS-DOS komut pencerelerini kapatmanız gerekir. Komut penceresini tekrar açtığınızda, yaptığınız değişiklikler etkinleşecektir.

Gelelim GNU/Linux kullanıcılarının durumuna...

GNU/Linux

Bildiğiniz gibi, GNU/Linux işletim sistemlerinde YOL dizinlerini listelemek için şöyle bir komut kullanıyoruz:

```
echo $PATH
```

Bu komutun çıktısının şuna benzer bir şey olacağını biliyoruz:

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

Gördüğünüz gibi, nasıl Windows işletim sistemlerinde YOL dizinleri birbirinden *;* işareti ile ayrılıyorsa, GNU/Linux dağıtımlarında da YOL dizinleri birbirinden *:* işareti ile ayrılıyor. Dolayısıyla GNU/Linux işletim sistemlerinde herhangi bir dizini YOL'a eklemek için şöyle bir komut kullanıyoruz:

```
PATH=$PATH:/yola/eklemek/istediğimiz/dizin
```

Bu komuttaki $PATH kısmı, YOL'a yeni dizini eklerken halihazırdaki dizinlerin de silinmemesini sağlıyor. Eğer yukarıdaki komutu şöyle verecek olursanız:

```
PATH=/yola/eklemek/istediğimiz/dizin
```

YOL üzerinde önceden varolan bütün dizinler silinip, tek girdi olarak yeni eklediğiniz dizin görünecektir. Böyle bir durumda da sisteminizde pek çok programın artık çalışmadığını görürsünüz. O yüzden bu komutu verirken dikkatli olun.

Bütün bu bilgiler ışığında, mesela *Desktop* (Masaüstü) dizinini YOL'a eklemek için şöyle bir komut yazabiliriz:

```
PATH=$PATH:$HOME/Desktop
```

Bu şekilde masaüstü dizinini YOL'a eklemiş olduk. Yeni eklediğimiz dizini, halihazırda YOL üzerinde bulunan dizinlerden *:* işareti ile ayırdığımıza dikkat edin.

İsterseniz `echo $PATH` komutuyla masaüstünüzün YOL üstünde görünüp görünmediğini kontrol edebilirsiniz. Bu sayede artık masaüstünde bulunan çalıştırılabilir dosyalar da kendi adlarıyla çağrılabilecekler. Ancak masaüstünü YOL'a eklediğinizde, masaüstünüz hep YOL üstünde kalmayacak, mevcut konsol oturumu kapatılınca her şey yine eski haline dönecektir. Eğer herhangi bir dizini kalıcı olarak YOL'a eklemek isterseniz, *.profile* (veya kullandığınız dağıtıma göre *.bash_profile* ya da *.bashrc*) dosyanızda değişiklik yapmanız gerekir. Mesela masaüstünü YOL'a eklemek için *$HOME/.profile* dosyanızın en sonuna şu satırı eklemelisiniz:

```
export PATH=$PATH:$HOME/Desktop
```

Ancak GNU/Linux'ta genel yaklaşımınız YOL'a kalıcı olarak dizin eklemek değil, ya bir dizini YOL'a geçici olarak eklemek, ya programınızı mevcut YOL dizinlerinden biri içine kopyalamak, ya da programımıza YOL dizinlerinden birisi içinden sembolik bağ vermek olmalıdır. Yani mesela masaüstünde bir programınız varsa, masaüstünü YOL'a eklemek yerine, programınızı YOL dizinlerinden biri içine (mesela */usr/bin*) kopyalamak ya da o konumda bir sembolik bağ oluşturmak daha akıllıca olacaktır.

Hatırlarsanız geçen bölümde 'çalıştırma yetkisi' konusundan söz ederken *deneme* adlı bir kabuk betiği yazmıştık. Masaüstüne kaydettiğimiz bu betiği çalıştırabilmek için de *./deneme* gibi bir komut kullanmıştık. Bu komutu verirken baştaki ./ işaretlerini kullanmazsak programımız hata veriyordu.

İşte bu bölümde öğrendiklerimiz sayesinde programımızın neden hata verip çalışmadığını artık daha iyi anlıyor olmalısınız. Dediğimiz gibi, *deneme* gibi bir komut verdiğimizde işletim sistemimiz `echo $PATH` çıktısında görünen YOL dizinlerinin içine bakıp, bu dizinlerin herhangi birinin içinde *deneme* adlı bir program olup olmadığını kontrol edecektir. Eğer siz kendiniz eklemediyseniz, işletim sisteminizde masaüstü öntanımlı olarak YOL'a ekli değildir. O yüzden *deneme* komutunu verdiğinizde işletim sisteminiz masaüstüne bakmaz. İşte işletim sisteminin *deneme* adlı programı başka yerde değil de masaüstünde aramasını sağlamak için komutun başına ./ işaretlerini yerleştiriyoruz...

Tıpkı daha önce öğrendiğimiz *%USERPROFILE%* (veya *$HOME*) adlı çevre değişkeni gibi, *%PATH%* (veya *$PATH*) de çok önemli çevre değişkenlerinden biri olduğu için YOL kavramıyla bundan sonra da yeterince haşır neşir olacağız. Ayrıca birkaç bölüm sonra gerçek Python programları yazmaya başladığımızda, şu anda belki de kafanızda muğlak kalmış bazı şeyler iyiden iyiye yerine oturmuş olacak. O yüzden şimdilik konuyu daha fazla uzatmadan burada noktalıyoruz ve önemli bir konuya daha geçiyoruz.

başa dön

Python Nasıl Kurulur?

Bu bölüme gelene kadar, herhangi bir programlama faaliyetine girişmeden önce halihazırda biliyor olmamız gereken her şeyi öğrendik. Artık Python ile program yazabilmemizin önünde tek bir engel kaldı. İşte biz bu bölümde Python ile program yazmamızın önündeki son engel olan 'kurulum'dan söz edeceğiz.

Dediğimiz gibi, Python ile program yazabilmemiz için bu programlama dilinin bilgisayarımızda kurulu olması gerekiyor. Bu programlama dilini kurmanızın gerekip gerekmediği, kullandığınız işletim sistemine bağlıdır. Biz burada hem GNU/Linux hem de Windows kullanıcılarının durumunu sırasıyla ve ayrı ayrı inceleyeceğiz. Dilerseniz öncelikle GNU/Linux kullanıcılarının durumuna bakalım:

Not

Bu kitap boyunca bazı konuların GNU/Linux ve Windows kullanıcıları için ayrı ayrı anlatıldığını göreceksiniz. Ancak konular bu şekilde ayrılmış da olsa, ben size her ikisini de okumanızı tavsiye ederim. Çünkü bu bölümlerde her iki kullanıcı grubunun da ilgisini çekebilecek bilgilere rastlayacaksınız. Ayrıca bu bölümler farklı kullanıcı gruplarına hitap ediyor olsa da, aslında bu bölümlerin birbirini tamamlayıcı nitelikte olduğunu göreceksiniz.

GNU/Linux Kullanıcıları

GNU/Linux dağıtımlarına Python programlama dilini kurarken bazı noktaları göz önünde bulundurmamız gerekiyor. İşte bu bölümde bu önemli noktaların neler olduğunu inceleyeceğiz.

Kurulu Python Sürümü

Hemen hemen bütün GNU/Linux dağıtımlarında Python programlama dili kurulu olarak gelir. Örneğin Ubuntu'da Python zaten kuruludur.

Ancak burada şöyle bir durum var:

Daha önce de belirttiğimiz gibi, şu anda piyasada iki farklı Python serisi bulunuyor. Bunlardan birinin Python'ın 2.x serisi, ötekinin ise 3.x serisi olduğunu biliyorsunuz.

30/10/2013 tarihi itibariyle, piyasadaki çoğu GNU/Linux dağıtımında sadece Python2 kuruludur. Sisteminizde kurulu olan Python sürümünü denetlemek için komut satırında öncelikle şu komutu vermeyi deneyin (büyük 'V' ile):

```
python -V
```

Eğer bu komuttan *Python 2.x.y* şeklinde bir çıktı alıyorsanız, yani x ve y'den önceki kısım 2 ile

başlıyorsa sisteminizde Python2 kuruludur.

Not

Eğer sisteminizde Python2 kurulu ise ve siz Python3 yerine Python2 ile çalışmak istiyorsanız istihza.com/py2/icindekiler_python.html adresini ziyaret edebilirsiniz.

Ancak python -V komutundan *Python 2.x.y* şeklinde bir çıktı almanız sisteminizde **sadece** Python2'nin kurulu olduğunu göstermez. Sisteminizde Python2 ile birlikte Python3 de halihazırda kurulu olabilir. Örneğin Ubuntu GNU/Linux'un **12.10** sürümünden itibaren hem Python2, hem de Python3 sistemde kurulu vaziyettedir.

Kullandığınız GNU/Linux dağıtımında durumun ne olduğunu denetlemek için yukarıdaki komutu bir de python3 -V şeklinde çalıştırmayı deneyebilirsiniz. Eğer bu komut size bir hata mesajı yerine bir sürüm numarası veriyorsa sisteminizde Python3 de kuruludur.

Sisteminizdeki Python sürümlerine ilişkin daha kesin bir rapor için ise şu komutu kullanabilirsiniz:

```
ls -g {,/usr{,/local}}/bin | grep python
```

Buradan aldığınız çıktıyı inceleyerek de sisteminizde birden fazla Python sürümünün kurulu olup olmadığını görebilirsiniz. [Bununla ilgili bir tartışma için bkz. http://goo.gl/RnRRc]

Eğer Python2 ile birlikte Python3 de kuruluysa yukarıdaki komut şuna benzer bir çıktı verir (çıktı, fazla yer kaplamaması için kırpılmıştır):

```
dh_python2
dh_python3
pdb2.7 -> ../lib/python2.7/pdb.py
pdb3.2 -> ../lib/python3.2/pdb.py
py3versions -> ../share/python3/py3versions.py
python -> python2.7
python2 -> python2.7
python2.7
python3 -> python3.2
python3.2 -> python3.2mu
python3.2mu
python3mu -> python3.2mu
pyversions -> ../share/python/pyversions.py
```

Çıktıda iki farklı Python sürümüne ait kayıtların olması sistemde iki farklı Python sürümünün kurulu olduğunu doğruluyor. Bu çıktıya göre, bu komutun verildiği sistemde Python'ın 2.7 ve 3.2 sürümleri zaten kurulu.

Eğer sisteminizde Python3 kuruluysa ve siz de kurulu olan Python3 sürümünden memnunsanız herhangi bir şey yapmanıza gerek yok. Farklı bir Python sürümü kurmaya çalışmadan yolunuza devam edebilirsiniz.

Paket Deposundan Kurulum

Sistemlerinde öntanımlı olarak herhangi bir Python3 sürümü kurulu olmayan veya sistemlerinde kurulu öntanımlı Python3 sürümünden memnun olmayan GNU/Linux kullanıcılarının, Python3'ü elde etmek için tercih edebileceği iki yol var: Birincisi ve benim size önereceğim yol, öncelikle kullandığınız dağıtımın paket yöneticisini kontrol etmenizdir. Python3 sisteminizde kurulu olmasa bile, dağıtımınızın depolarında bu sürüm paketlenmiş halde duruyor olabilir. O yüzden sisteminize uygun bir şekilde paket yöneticinizi açıp orada 'python' kelimesini kullanarak bir arama yapmanızı öneririm. Örneğin Ubuntu GNU/Linux dağıtımının paket depolarında Python3 var. Dolayısıyla Ubuntu kullanıcıları, eğer sistemlerinde zaten kurulu değilse (ki muhtemelen kuruludur), bu paketi Ubuntu Yazılım Merkezi aracılığıyla veya doğrudan şu komutla kurabilir:

```
sudo apt-get install python3
```

Bu komut, Python3'ü bütün bağımlılıkları ile beraber bilgisayarınıza kuracaktır.

Kaynaktan Kurulum

Peki ya kullandığınız dağıtımın depolarında Python3 yoksa veya depodaki Python3 sürümü eskiyse ve siz daha yeni bir Python3 sürümü kullanmak istiyorsanız ne yapacaksınız?

Eğer dağıtımınızın depolarında Python3 paketini bulamazsanız veya depodaki sürüm sizi tatmin etmiyorsa, Python3'ü kaynaktan derlemeniz gerekecektir. Python3'ü kaynaktan derlerken iki seçeneğiniz var: Python3'ü *root* hakları ile kurmak veya Python3'ü yetkisiz kullanıcı olarak kurmak. Normal şartlar altında eğer kullandığınız sistemde *root* haklarına sahipseniz Python3'ü yetkili kullanıcı olarak kurmanızı tavsiye ederim.

root Hakları İle Kurulum

Python'ı kurmadan önce sistemimizde bulunması gereken bazı programlar var. Aslında bu programlar olmadan da Python kurulabilir, ancak eğer bu programları kurmazsanız Python'ın bazı özelliklerinden yararlanamazsınız. Bu programlar şunlardır:

1. tcl8.5-dev
2. tk8.5-dev
3. zlib1g-dev
4. ncurses-dev
5. libreadline-dev
6. libdb-dev
7. libgdbm-dev
8. libzip-dev

9. libssl-dev
10. libsqlite3-dev
11. libbz2-dev

Bu programları, kullandığınız GNU/Linux dağıtımının paket yöneticisi aracılığıyla kurabilirsiniz. Yalnız paket adlarının ve gerekli paket sayısının dağıtımlar arasında farklılık gösterebileceğini unutmayın. Yukarıdaki liste Ubuntu için geçerlidir. Mesela yukarıda *tcl8.5-dev* olarak verdiğimiz paket adı başka bir dağıtımda sadece *tcl* veya *tcl8.5* olarak geçiyor olabilir ya da yukarıdaki paketlerin bazıları kullandığınız dağıtımda halihazırda kurulu olduğu için sizin daha az bağımlılık kurmanız gerekiyor olabilir.

Ubuntu'da yukarıdaki paketlerin hepsini şu komutla kurabilirsiniz:

```
sudo apt-get install tcl8.5-dev tk8.5-dev zlib1g-dev ncurses-dev libreadline-dev libdb-dev libgdbm-dev libzip-dev libssl-dev libsqlite3-dev libbz2-dev
```

Not

Farklı GNU/Linux dağıtımlarında, Python3'ü kaynaktan derleme işleminden önce halihazırda kurulu olması gereken paketlerin listesi için http://goo.gl/zfLpX adresindeki tabloyu inceleyebilirsiniz.

Yukarıdaki programları kurduktan sonra şu adresi ziyaret ediyoruz: python.org/download

Bu adreste, üzerinde 'Python 3.3.2 xzipped source tarball (for Linux, Unix or Mac OS X, better compression)' yazan bağlantıya tıklıyoruz. (Bu sayfada Python 3.3.2 yerine yanlışlıkla Python 2.7.5 bağlantısına tıklamamaya özen gösteriyoruz!)

İlgili *.tar.xz* dosyasını bilgisayarımıza indiriyoruz.

Daha sonra bu sıkıştırılmış dosyayı açıyoruz. Açılan klasörün içine girip, orada ilk olarak şu komutu veriyoruz:

```
./configure
```

Bu komut, Python programlama dilinin sisteminize kurulabilmesi için gereken hazırlık aşamalarını gerçekleştirir. Bu betiğin temel olarak yaptığı iş, sisteminizin Python programlama dilinin kurulmasına uygun olup olmadığını, derleme işlemi için gereken yazılımların sisteminizde kurulu olup olmadığını denetlemektir. Bu betik ayrıca, bir sonraki adımda gerçekleştireceğimiz inşa işleminin nasıl yürüyeceğini tarif eden *Makefile* adlı bir dosya da oluşturur.

Bu arada bu komutunun başındaki ./ işaretlerinin anlamını artık gayet iyi biliyorsunuz. Bu şekilde, o anda içinde bulunduğunuz dizinde yer alan *configure* adlı bir betiği çalıştırıyorsunuz. Eğer yalnızca `configure` komutu verirseniz, işletim sistemi bu betiği YOL dizinleri içinde arayacak ve bulamayacağı için de hata verecektir.

./configure komutu hatasız olarak tamamlandıktan sonra ikinci olarak şu komutu veriyoruz:

```
make
```

Burada aslında ./configure komutu ile oluşan *Makefile* adlı dosyayı *make* adlı bir program aracılığıyla çalıştırmış oluyoruz. make bir sistem komutudur. Bu komutu yukarıdaki gibi parametresiz olarak çalıştırdığımızda make komutu, o anda içinde bulunduğumuz dizinde bir *Makefile* dosyası arar ve eğer böyle bir dosya varsa onu çalıştırır. Eğer bir önceki adımda çalıştırdığımız ./configure komutu başarısız olduysa, dizinde bir *Makefile* dosyası oluşmayacağı için yukarıdaki make komutu da çalışmayacaktır. O yüzden derleme işlemi sırasında verdiğimiz komutların çıktılarını takip edip, bir sonraki aşamaya geçmeden önce komutun düzgün sonlanıp sonlanmadığından emin olmamız gerekiyor.

make komutunun yaptığı iş, Python programlama dilinin sisteminize kurulması esnasında sistemin çeşitli yerlerine kopyalanacak olan dosyaları inşa edip oluşturmaktır. Bu komutun tamamlanması, kullandığınız bilgisayarın kapasitesine bağlı olarak biraz uzun sürebilir.

Bazı sistemlerde make komutunun sonunda şöyle bir hata mesajıyla karşılaşabilirsiniz:

```
Undefined reference to '_PyFaulthandler_Init'
```

Eğer böyle bir hatayla karşılaşırsanız http://goo.gl/jzMIZ adresindeki önerimizi uygulayabilirsiniz.

make komutu tamamlandıktan sonra, komut çıktısının son satırlarına doğru şöyle bir uyarı mesajı da görebilirsiniz:

```
Python build finished, but the necessary bits to build these modules were
not found: [burada eksik olan modül veya modüllerin adları sıralanır]
```

Burada Python, sistemimizde bazı paketlerin eksik olduğu konusunda bizi uyarıyor. Uyarı mesajında bir veya daha fazla paketin eksik olduğunu görebilirsiniz. Eğer öyleyse, eksik olduğu bildirilen bütün paketleri kurmamız gerekiyor.

Gerekli paketi ya da paketleri kurduktan sonra make komutunu tekrar çalıştırıyoruz. Endişe etmeyin, make komutunu ikinci kez verdiğimizde komutun tamamlanması birincisi kadar uzun sürmez. Eğer bu komutu ikinci kez çalıştırdığınızda yukarıdaki uyarı mesajı kaybolduysa şu komutla yolunuza devam edebilirsiniz:

```
sudo make altinstall
```

Daha önce kaynaktan program derlemiş olan GNU/Linux kullanıcılarının eli, make komutundan sonra make install komutunu vermeye gitmiş olabilir. Ama burada bizim make install yerine make altinstall komutunu kullandığımıza dikkat edin. make altinstall komutu, Python kurulurken klasör ve dosyalara sürüm numarasının da eklenmesini sağlar. Böylece yeni kurduğunuz Python, sistemdeki eski

Python3 sürümünü silip üzerine yazmamış olur ve iki farklı sürüm yan yana varolabilir. Eğer make altinstall yerine make install komutunu verirseniz sisteminizde zaten varolan eski bir Python3 sürümüne ait dosya ve dizinlerin üzerine yazıp silerek o sürümü kullanılamaz hale getirebilirsiniz. Bu da sistemde beklenmedik problemlerin ortaya çıkmasına yol açabilir. Bu önemli ayrıntıyı kesinlikle gözden kaçırmamalısınız.

Not

Python3'ün kaynaktan kurulumu ile ilgili bir tartışma için bkz.
http://www.istihza.com/forum/viewtopic.php?f=50&t=544

Derleme aşamalarının hiçbirinde herhangi bir hata mesajı almadıysanız kurulum başarıyla gerçekleşmiş ve sisteminize Python programlama dilinin 3.x sürümü kurulmuş demektir.

Yetkisiz Kullanıcı Olarak Kurulum

Elbette sudo make altinstall komutunu verip Python'ı kurabilmek için *root* haklarına sahip olmanız gerekiyor. Ama eğer kullandığınız sistemde bu haklara sahip değilseniz Python'ı bu şekilde kuramazsınız. Kısıtlı haklara sahip olduğunuz bir sistemde Python'ı ancak kendi ev dizininize ($HOME) kurabilirsiniz.

Eğer Python'ı yetkisiz kullanıcı olarak kuracaksanız, öncelikle yukarıda bahsettiğimiz Python bağımlılıklarının sisteminizde kurulu olup olmadığını kontrol etmeniz lazım. Kullandığınız sistemde herhangi bir Python sürümü halihazırda kuruluysa, bu bağımlılıklar da muhtemelen zaten kuruludur. Ama değilse, bunları kurması için ya sistem yöneticisine ricada bulunacaksınız, ya da bu bağımlılıkları da tek tek kendi ev dizininize kuracaksınız. Eğer sistem yöneticisini bu bağımlılıkları kurmaya ikna edemezseniz, internet üzerinden bulabileceğiniz bilgiler yardımıyla bu bağımlılıkları tek tek elle kendiniz kurabilirsiniz. Ancak bu işlemin epey zaman alacağını ve süreç sırasında pek çok başka bağımlılıkla da karşılacağınızı söyleyebilirim. O yüzden ne yapıp edip sistem yöneticisini bağımlılıkları kurmaya ikna etmenizi tavsiye ederim... Tabii sistem yöneticisini bu bağımlılıkları kurmaya ikna edebilirseniz, istediğiniz Python sürümünü de kurmaya ikna edebileceğinizi düşünebiliriz! Ama biz burada sizin Python'ı kendinizin kuracağını varsayarak yolumuza devam edelim.

Python'ı yetkisiz olarak kurmak, *root* haklarıyla kurmaya çok benzer. Aralarında yalnızca bir-iki ufak fark vardır. Mesela Python'ı yetkisiz kullanıcı olarak kurarken, ./configure komutunu şu şekilde vermeniz gerekiyor:

```
./configure --prefix=$HOME/python
```

Python'ı *root* haklarıyla kurduğunuzda Python */usr* dizini altına kurulacaktır. Ancak siz yetkisiz kullanıcı olduğunuz için */usr* dizinine herhangi bir şey kuramazsınız. İşte bu yüzden, *configure* betiğine verdiğimiz *–prefix* parametresi yardımıyla Python'ı, yazma yetkimiz olan bir dizine

kuruyoruz. Mesela yukarıdaki komut Python'ın */usr* dizinine değil, ev dizininiz içinde *python* adlı bir klasöre kurulmasını sağlayacaktır. Elbette siz *python* yerine farklı bir dizin adı da belirleyebilirsiniz... Burada önemli olan nokta, *–prefix* parametresine vereceğiniz dizin adının, sizin yazmaya yetkili olduğunuz bir dizin olmasıdır.

Not

Ben bu kitapta sizin Python'ı *$HOME/python* dizinine kurduğunuzu varsayacağım.

Bu komutu çalıştırdıktan sonra `make` komutunu normal bir şekilde veriyoruz. Bunun ardından da `make altinstall` komutuyla Python'ı ev dizinimize kuruyoruz. Burada `make altinstall` komutunu `sudo`'suz kullandığımıza dikkat edin. Çünkü, dediğimiz gibi, siz yetkili kullanıcı olmadığınız için `sudo` komutunu kullanamazsınız.

Python'ı bu şekilde ev dizininiz altında bir klasöre kurduğunuzda Python ile ilgili bütün dosyaların bu klasör içinde yer aldığını göreceksiniz. Bu klasörü dikkatlice inceleyip neyin nerede olduğuna aşinalık kazanmaya çalışın. Eğer mümkünse *root* hakları ile kurulmuş bir Python sürümünü inceleyerek, dosyaların iki farklı kurulum türünde nerelere kopyalandığını karşılaştırın.

Böylece Python programlama dilini bilgisayarımıza nasıl kuracağımızı öğrenmiş olduk. Ama bu noktada bir uyarı yapmadan geçmeyelim: Python özellikle bazı GNU/Linux dağıtımlarında pek çok sistem aracıyla sıkı sıkıya bağlantılıdır. Yani Python, kullandığınız dağıtımın belkemiği durumunda olabilir. Bu yüzden Python'ı kaynaktan derlemek bazı riskler taşıyabilir. Eğer yukarıda anlatıldığı şekilde, kaynaktan Python derleyecekseniz, karşı karşıya olduğunuz risklerin farkında olmalısınız. Ayrıca GNU/Linux üzerinde kaynaktan program derlemek konusunda tecrübeli değilseniz ve eğer yukarıdaki açıklamalar size kafa karıştırıcı geliyorsa, mesela 'Ben bu komutları nereye yazacağım?' diye bir soru geçiyorsa aklınızdan, kesinlikle dağıtımınızla birlikte gelen Python sürümünü kullanmalısınız. Python sürümlerini başa baş takip ettiği için, ben size Ubuntu GNU/Linux'u denemenizi önerebilirim. Ubuntu'nun depolarında Python'ın en yeni sürümlerini rahatlıkla bulabilirsiniz. Ubuntu'nun resmi sitesine ubuntu.com adresinden, yerel Türkiye sitesine ise forum.ubuntu-tr.net adresinden ulaşabilirsiniz. Eğer şu anda kullandığınız GNU/Linux dağıtımından vazgeçmek istemiyorsanız, sabit diskinizden küçük bir bölüm ayırıp bu bölüme sadece Python çalışmalarınız için Ubuntu dağıtımını da kurmayı tercih edebilirsiniz.

Yalnız küçük bir uyarı daha yapalım. Kaynaktan kurulum ile ilgili bu söylediklerimizden, Python'ın GNU/Linux'a kesinlikle kaynaktan derlenerek kurulmaması gerektiği anlamı çıkmamalı. Yukarıdaki uyarıların amacı, kullanıcının Python'ı kaynaktan derlerken sadece biraz daha dikkatli olması gerektiğini hatırlatmaktır. Örneğin bu satırların yazarı, kullandığı Ubuntu sisteminde Python3'ü kaynaktan derleyerek kullanmayı tercih ediyor ve herhangi bir problem yaşamıyor.

Bu önemli uyarıları da yaptığımıza göre gönül rahatlığıyla yolumuza devam edebiliriz.

Kurduğumuz yeni Python'ı nasıl çalıştıracağımızı biraz sonra göreceğiz. Ama önce Windows kullanıcılarının Python3'ü nasıl kuracaklarına bakalım.

Windows Kullanıcıları

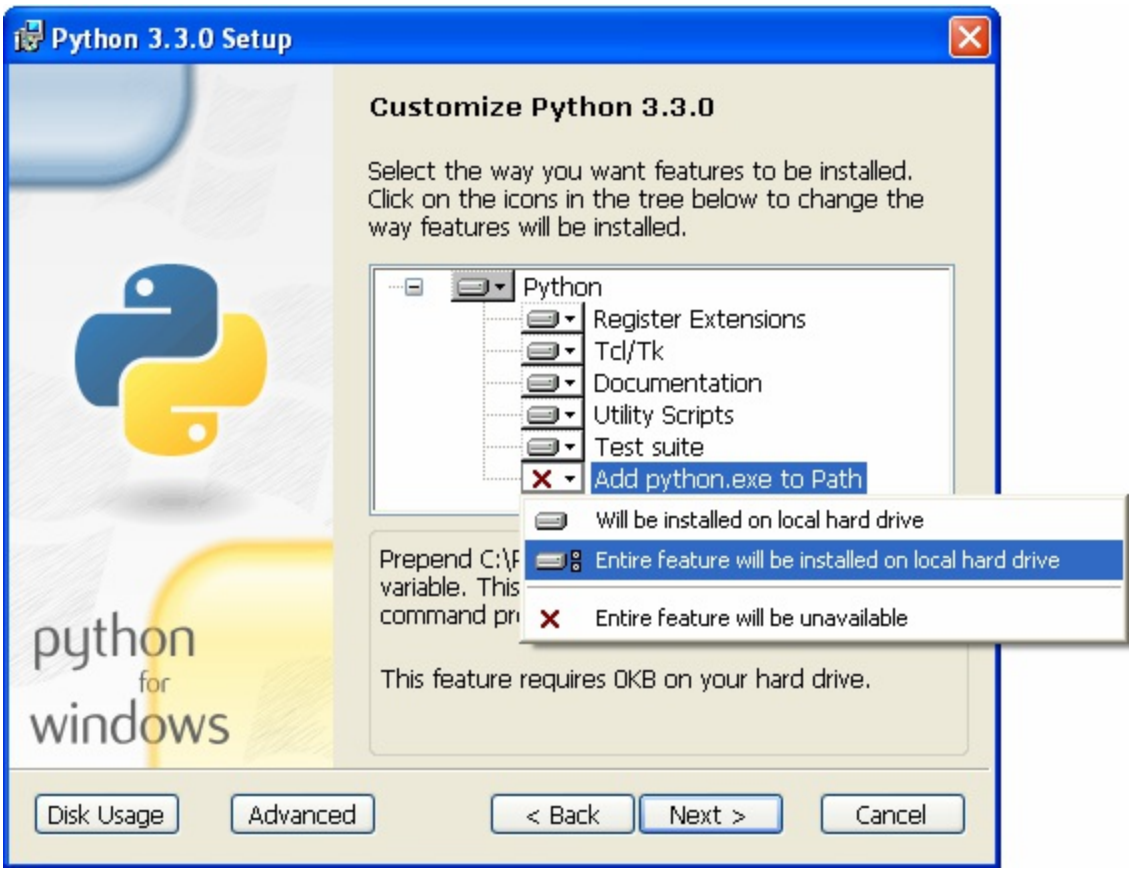
Windows sürümlerinin hiçbirinde Python kurulu olarak gelmez. O yüzden Windows 7 kullanıcıları, Python'ı sitesinden indirip kuracak. Bunun için şu adımları takip ediyoruz:

1. python.org/download adresini ziyaret ediyoruz.
2. Orada, üzerinde 'Python 3.3.2 Windows x86 MSI Installer (Windows binary – does not include source)' yazan bağlantıya tıklıyoruz.

   Not

   Eğer 64 bit mimariye sahip bir işletim sistemi kullanıyorsanız, üzerinde 'Windows X86-64 MSI Installer (3.3.2)' yazan bağlantıya tıklayın. Eğer hangi mimariyi kullandığınıza dair hiçbir fikriniz yoksa, 'Windows x86 MSI Installer (3.3.2)' bağlantısını tıklayabilirsiniz.

3. *MSI* uzantılı dosyayı bilgisayarımıza indiriyoruz.
4. İnen dosyaya çift tıklayıp normal bir şekilde kuruluma başlıyoruz.
5. Kurulum adımlarından birinde şöyle bir ekranla karşılaşacaksınız:



Burada *Add python.exe to Path* (python.exe'yi yola ekle) diye bir seçenek görüyorsunuz.

Tahmin edebileceğiniz gibi, bu seçenek Python programlama dilininin kurulu olduğu dizini YOL (*PATH*) dizinleri arasına ekleyerek, Python'ı kurulumdan sonra sadece adını kullanarak çalıştırabilmemizi sağlayacak.

6. Bu seçeneğin yanındaki küçük siyah oka tıklayarak, açılan menüden *Entire feature will be installed on local hard drive* girdisini seçiyoruz. Bundan sonra kuruluma normal bir şekilde devam edebiliriz.

7. Ben bu kitapta sizin Python'ı yukarıda gösterdiğim şekilde kurduğunuzu varsayacağım. Eğer siz farklı bir kurulum gerçekleştirdiyseniz kitaptaki bazı yönergeleri, kitapta gösterildiği şekilde çalıştıramayabilirsiniz. O yüzden, eğer ne yaptığınızdan emin değilseniz, Python'ı tıpkı burada anlatıldığı gibi kurmanızı ve kurulum sırasında öntanımlı ayarları değiştirmemenizi öneririm...

Windows'ta Python kurulumu bu kadar basittir. Artık bilgisayarımıza kurduğumuz Python programını nasıl çalıştıracağımızı görebiliriz.

başa dön

Python Nasıl Çalıştırılır?

Bir önceki bölümde, Python'ı farklı platformlara nasıl kuracağımızı bütün ayrıntılarıyla anlattık. Bu bölümde ise kurduğumuz bu Python programını hem GNU/Linux'ta hem de Windows'ta nasıl çalıştıracağımızı göreceğiz. Öncelikle GNU/Linux kullanıcılarının Python'ı nasıl çalıştıracağına bakalım.

GNU/Linux Kullanıcıları

Geçen bölümlerde gördüğünüz gibi, Python3'ü GNU/Linux sistemleri üzerine farklı şekillerde kurabiliyoruz. Bu bölümde, herbir kurulum türü için Python3'ün nasıl çalıştırılacağını ayrı ayrı inceleyeceğiz.

Kurulu Python3'ü Kullananlar

Eğer sisteminizde zaten Python3 kurulu ise komut satırında yalnızca şu komutu vererek Python3'ü başlatabilirsiniz:

```
python
```

Ancak daha önce de dediğimiz gibi, 30/10/2013 tarihi itibariyle pek çok GNU/Linux dağıtımında öntanımlı olarak Python2 kuruludur. Dolayısıyla python komutunu verdiğinizde çalışan sürüm muhtemelen Python2 olacaktır. Bu yüzden sistemimizde öntanımlı olarak hangi sürümün kurulu olduğuna ve python komutunun hangi sürümü başlattığına çok dikkat etmelisiniz.

Yine daha önce de söylediğimiz gibi, sisteminizde hem Python2 hem de Python3 zaten kurulu durumda olabilir. O yüzden yukarıdaki komutu bir de python3 şeklinde vermeyi deneyebilirsiniz.

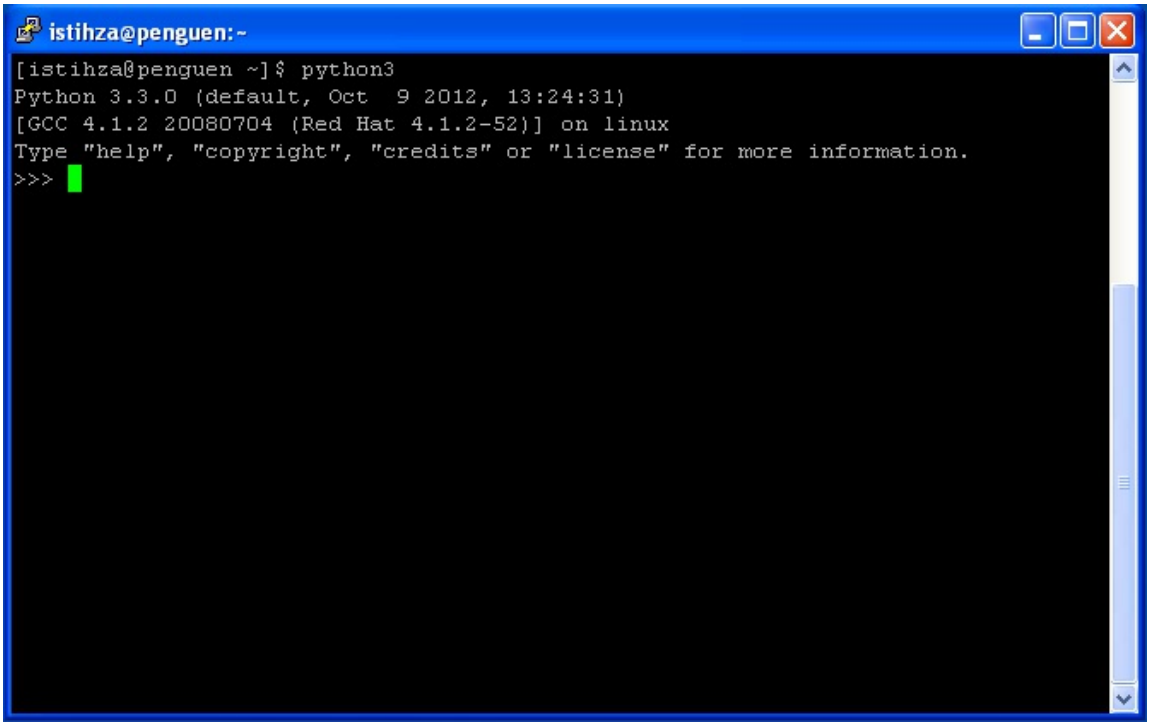
Örneğin Ubuntu GNU/Linux dağıtımının **12.10** sürümünden itibaren python komutu Python2'yi; python3 komutu ise Python3'ü çalıştırıyor.

Python3'ü Depodan Kuranlar

Dediğimiz gibi, 30/10/2013 tarihi itibariyle GNU/Linux dağıtımlarında öntanımlı Python sürümü ağırlıklı olarak Python2'dir. Dolayısıyla python komutu Python'ın 2.x sürümlerini çalıştırır. Bu durumdan ötürü, herhangi bir çakışmayı önlemek için GNU/Linux dağıtımları Python3 paketini farklı bir şekilde adlandırma yoluna gider. Şu anda piyasada bulunan dağıtımların ezici çoğunluğu Python3 paketini 'python3' şeklinde adlandırıyor. O yüzden GNU/Linux kullanıcıları, eğer paket yöneticilerini kullanarak Python kurulumu gerçekleştirmiş iseler, komut satırında şu komutu vererek Python3'ü başlatabilirler:

```
python3
```

Bu komutun ardından şuna benzer bir ekranla karşılaşmış olmalısınız:



Eğer yukarıdaki ekranı gördüyseniz Python'la programlama yapmaya hazırsınız demektir. Değilse, geriye dönüp işlerin nerede ters gittiğini bulmaya çalışabilirsiniz.

Bu aşamada işlerin nerede ters gitmiş olabileceğine dair birkaç ipucu verelim:

1. Python3 kurulurken paket yöneticinizin herhangi bir hata vermediğinden, programın sisteminize başarıyla kurulduğundan emin olun. Bunun için Python3'ün kurulu paketler listesinde görünüp görünmediğini denetleyebilirsiniz.
2. python3 komutunu doğru verdiğinize emin olun. Python programlama diline özellikle yeni başlayanların en sık yaptığı hatalardan biri *python* kelimesini yanlış yazmaktır. *Python* yerine yanlışlıkla *pyhton*, *pyton* veya *phyton* yazmış olabilirsiniz. Ayrıca python3 komutunun tamamen küçük harflerden oluştuğuna dikkat edin. *Python* ve *python* bilgisayar açısından aynı şeyler değildir.
3. Kullandığınız dağıtımın Python3 paketini adlandırma politikası yukarıda anlattığımızdan farklı olabilir. Yani sizin kullandığınız dağıtım, belki de Python3 paketini farklı bir şekilde adlandırmıştır. Eğer durum böyleyse, dağıtımınızın yardım kaynaklarını (wiki, forum, irc, yardım belgeleri, kullanıcı listeleri, vb.) kullanarak ya da istihza.com/forum adresinde sorarak Python3'ün nasıl çalıştırılacağını öğrenmeyi deneyebilirsiniz.

Gelelim Python3'ü kaynaktan derlemiş olanların durumuna...

Python3'ü root Olarak Derleyenler

Eğer Python3'ü önceki bölümlerde anlattığımız şekilde kaynaktan *root* hakları ile derlediyseniz python3 komutu çalışmayacaktır. Bunun yerine şu komutu kullanmanız gerekecek:

```
python3.3
```

Not

Kurduğunuz Python3 sürümünün *3.3* olduğunu varsayıyorum. Eğer farklı bir Python3 sürümü kurduysanız, elbette başlatıcı komut olarak o sürümün adını kullanmanız gerekecektir. Mesela: python3.0 veya python3.1. Bu arada python3.3 komutunda 33 sayısının rakamları arasında bir adet nokta işareti olduğunu gözden kaçırmıyoruz...

Tıpkı paket deposundan kurulumda olduğu gibi, eğer yukarıdaki komut Python'ı çalıştırmanızı sağlamıyorsa, kurulum esnasında bazı şeyler ters gitmiş olabilir. Örneğin kaynaktan kurulumun herhangi bir aşamasında bir hata almış olabilirsiniz ve bu da Python'ın kurulumunu engellemiş olabilir.

Gördüğünüz gibi, Python'ı kaynaktan derleyenler Python programlama dilini çalıştırabilmek için Python'ın tam sürüm adını belirtiyor. Dilerseniz bu şekilde çalışmaya devam edebilirsiniz. Bunun hiçbir sakıncası yok. Ancak ben size kolaylık açısından, */usr/bin/* dizini altına *py3* adında bir sembolik bağ yerleştirmenizi tavsiye ederim. Böylece sadece py3 komutunu vererek Python3'ü başlatabilirsiniz.

Peki bunu nasıl yapacağız?

Python kaynaktan derlendiğinde çalıştırılabilir dosya */usr/local/bin/* dizini içine *Python3.3* (veya kurduğunuz Python3 sürümüne bağlı olarak *Python3.0* ya da *Python3.1*) adıyla kopyalanır. Bu nedenle Python3'ü çalıştırabilmek için python3.3 komutunu kullanmamız gerekir. Python3'ü çalıştırabilmek için mesela sadece py3 gibi bir komut kullanmak istiyorsak yapacağımız tek şey */usr/local/bin/* dizini içindeki *python3.3* adlı dosyaya */usr/bin* dizini altından, *py3* adlı bir sembolik bağ oluşturmak olacaktır. Bunun için ln komutunu kullanmamız gerektiğini biliyorsunuz:

```
ln -s /usr/local/bin/python3.3 /usr/bin/py3
```

Tabii bu komutu yetkili kullanıcı olarak vermeniz gerektiğini söylememe herhalde gerek yoktur. Bu komutu verdikten sonra artık sadece py3 komutu ile Python programlama dilini başlatabilirsiniz.

Çok Önemli Bir Uyarı

Bir önceki adımda anlattığımız gibi Python3'ü resmi sitesinden indirip kendiniz derlediniz. Gayet güzel. Ancak bu noktada çok önemli bir konuya dikkatinizi çekmek isterim. En baştan beri söylediğimiz gibi, Python programlama dili GNU/Linux işletim sistemlerinde çok önemli bir yere sahiptir. Öyle ki bu programlama dili, kullandığınız dağıtımın belkemiği durumunda olabilir.

Örneğin Ubuntu GNU/Linux dağıtımında pek çok sistem aracı Python ile yazılmıştır. Bu yüzden, sistemdeki öntanımlı Python sürümünün ne olduğu ve dolayısıyla python komutunun hangi Python sürümünü çalıştırdığı çok önemlidir. Çünkü sisteminizdeki hayati bazı araçlar, python komutunun çalıştırdığı Python sürümüne bel bağlamış durumdadır. Dolayısıyla sizin bu python komutunun

çalıştırdığı Python sürümüne dokunmamanız gerekir.

Mesela eğer kullandığınız işletim sisteminde python komutu Python'ın 2.x sürümlerinden birini çalıştırıyorsa sembolik bağlar veya başka araçlar vasıtasıyla python komutunu Python'ın başka bir sürümüne bağlamayın. Bu şekilde bütün sistemi kullanılmaz hale getirirsiniz. Elbette eğer kurulum aşamasında tarif ettiğimiz gibi, Python3'ü make install yerine make altinstall komutu ile kurmaya özen gösterdiyseniz, sonradan oluşturduğunuz bağ dosyasını silip python komutunu yine sistemdeki öntanımlı sürüme bağlayabilirsiniz. Bu şekilde her şey yine eski haline döner. Ama eğer Python'ı make install komutuyla kurmanızdan ötürü sistemdeki öntanımlı Python sürümüne ait dosyaları kaybettiyseniz sizin için yapılacak fazla bir şey yok... Sistemi tekrar eski kararlı haline getirmek için kan, ter ve gözyaşı dökeceksiniz...

Aynı şekilde, kullandığınız dağıtımda python3 komutunun öntanımlı olarak belirli bir Python sürümünü başlatıp başlatmadığı da önemlidir. Yukarıda python komutu ile ilgili söylediklerimiz python3 ve buna benzer başka komutlar için de aynen geçerli.

Örneğin, Ubuntu GNU/Linux dağıtımında python komutu sistemde kurulu olan Python 2.x sürümünü; python3 komutu ise sistemde kurulu olan Python 3.x sürümünü çalıştırdığından, biz kendi kurduğumuz Python sürümleri için, sistemdeki sürümlerle çakışmayacak isimler seçtik. Mesela kendi kurduğumuz Python3 sürümünü çalıştırmak için py3 gibi bir komut tercih ettik.

İyi bir test olarak, Python programlama dilini kendiniz kaynaktan derlemeden önce şu komutun çıktısını iyice inceleyebilirsiniz:

```
ls -g {,/usr{,/local}}/bin | grep python
```

Bu komut iki farklı Python sürümünün kurulu olduğu sistemlerde şuna benzer bir çıktı verir (çıktı kırpılmıştır):

```
dh_python2
dh_python3
pdb2.7 -> ../lib/python2.7/pdb.py
pdb3.2 -> ../lib/python3.2/pdb.py
py3versions -> ../share/python3/py3versions.py
python -> python2.7
python2 -> python2.7
python2.7
python3 -> python3.2
python3.2 -> python3.2mu
python3.2mu
python3mu -> python3.2mu
pyversions -> ../share/python/pyversions.py
```

Yatık harflerle gösterdiğimiz kısımlara dikkat edin. Gördüğünüz gibi python ve python2 komutları

bu sistemde Python'ın 2.7 sürümünü çalıştırıyor. python3 komutu ise Python'ın 3.2 sürümünü... Dolayısıyla yukarıdaki çıktıyı aldığımız bir sistemde kendi kurduğumuz Python sürümlerine 'python', 'python2' veya 'python3' gibi isimler vermekten kaçınmalıyız.

Sözün özü, bir GNU/Linux kullanıcısı olarak sistemdeki öntanımlı hiçbir Python sürümünü silmemeli, öntanımlı sürüme ulaşan komutları değiştirmemelisiniz. Eğer mesela sisteminizde python3 komutu halihazırda bir Python sürümünü çalıştırıyorsa, siz yeni kurduğunuz Python sürümüne ulaşmak için öntanımlı adla çakışmayacak başka bir komut adı kullanın. Yani örneğin sisteminizde python3 komutu Python'nın *3.2* sürümünü çalıştırıyorsa, siz yeni kurduğunuz sürümü çalıştırmak için py3 gibi bir sembolik bağ oluşturun. Bırakın öntanımlı komut (python, python3 vb.) öntanımlı Python sürümünü çalıştırmaya devam etsin.

Asla unutmayın. Siz bir programcı adayı olarak, program yazacağınız işletim sistemini enine boyuna tanımakla yükümlüsünüz. Dolayısıyla işletim sisteminizi kararsız hale getirecek davranışları bilmeli, bu davranışlardan kaçınmalı, yanlış bir işlem yaptığınızda da nasıl geri döneceğinizi bilmelisiniz. Hele ki bir programı kaynaktan derlemeye karar vermişseniz...

Bu ciddi uyarıyı da yaptığımıza göre gönül rahatlığıyla yolumuza devam edebiliriz.

Python3'ü Ev Dizinine Kuranlar

Eğer Python3'ü kısıtlı kullanıcı hakları ile derleyip ev dizininize kurduysanız yukarıdaki komutlar Python'ı çalıştırmanızı sağlamayacaktır. Python3'ü ev dizinine kurmuş olan kullanıcılar Python3'ü çalıştırabilmek için, öncelikle komut satırı aracılığıyla Python3'ü kurdukları dizine, oradan da o dizin altındaki *bin/* klasörüne ulaşacak ve orada şu komutu verecek:

```
./python3.3
```

Diyelim ki Python3'ü *$HOME/python* adlı dizine kurdunuz. Önce şu komutla *$HOME/python/bin* adlı dizine ulaşıyoruz:

```
cd $HOME/python/bin
```

Ardından da şu komutu veriyoruz:

```
./python3.3
```

Not

Komutun başındaki *./* işaretinin ne işe yaradığını artık adınız gibi biliyorsunuz...

Not

Elbette ben burada kurduğunuz Python sürümünün 3.3 olduğunu varsaydım. Eğer farklı bir sürüm

kurduysanız yukarıdaki komutu ona göre yazmanız gerekiyor.

Eğer isterseniz bu şekilde çalışmaya devam edebilirsiniz. Ancak her defasında Python'ın kurulu olduğu dizin altına gelip orada ./python3.3 komutunu çalıştırmak bir süre sonra eziyete dönüşecektir. İşlerinizi kolaylaştırmak için şu işlemleri takip etmelisiniz:

1. ev dizininizin altında bulunan *.profile* (veya kullandığınız dağıtıma göre *.bash_profile* ya da *.bashrc*) adlı dosyayı açın.

2. Bu dosyanın en sonuna şuna benzer bir satır yerleştirerek Python'ı çalıştırmamızı sağlayan dosyanın bulunduğu dizini YOL'a ekleyin:

```
export PATH=$PATH:$HOME/python/bin/
```

3. *$HOME/python/bin/* satırı Python3'ün çalıştırılabilir dosyasının hangi dizin altında olduğunu gösteriyor. Ben burada Python3'ün çalıştırılabilir dosyasının *$HOME/python/bin* dizini içinde olduğunu varsaydım. O yüzden de *$HOME/python/bin/* gibi bir satır yazdım. Ama eğer Python3'ün çalıştırılabilir dosyası sizde farklı bir dizindeyse bu satırı ona göre yazmalısınız.

4. Kendi sisteminize uygun satırı dosyaya ekledikten sonra dosyayı kaydedip çıkın. Dosyada yaptığımız değişikliğin etkin hale gelebilmesi için şu komutu verin:

```
source .profile
```

Elbette eğer sizin sisteminizdeki dosyanın adı *.bash_profile* veya *.bashrc* ise yukarıdaki komutu ona göre değiştirmelisiniz.

5. Daha sonra *$HOME/python/bin/python3.3* adlı dosyaya *$HOME/python/bin/* dizini altından mesela *py3* gibi bir sembolik bağ verin:

```
ln -s $HOME/python/bin/python3.3 $HOME/python/bin/py3
```

6. Artık hangi konumda bulunursanız bulunun, şu komutu vererek Python3'ü başlatabilirsiniz:

```
py3
```

Burada da eğer yukarıdaki komut Python3'ü çalıştırmanızı sağlamıyorsa, bazı şeyleri eksik veya yanlış yapmış olabilirsiniz. Yardım almak için istihza.com/forum adresine uğrayabilirsiniz.

Python3'ü başarıyla kurup çalıştırabildiğinizi varsayarak yolumuza devam edelim.

Farklı Sürümleri Birlikte Kullanmak

Daha önce de dediğimiz gibi, şu anda piyasada iki farklı Python serisi bulunuyor: Python2 ve Python3. Çok uzun zamandan beri kullanımda olduğu için, Python2 Python3'e kıyasla daha yaygın.

Eğer hem Python2 ile yazılmış programları çalıştırmak, hem de Python3 ile geliştirme yapmak istiyorsanız, sisteminizde hem Python2'yi hem de Python3'ü aynı anda bulundurmayı tercih edebilirsiniz. Peki bunu nasıl yapacaksınız?

En başta da söylediğimiz gibi, hemen hemen bütün GNU/Linux dağıtımlarında Python2 kurulu olarak gelir. Dolayısıyla eğer sisteminize ek olarak Python3'ü de kurduysanız (kaynaktan veya paket deposundan), başka herhangi bir şey yapmanıza gerek yok. Yukarıda anlattığımız yönergeleri takip ettiyseniz, konsolda python komutu verdiğinizde Python2 çalışacak, python3 (veya py3) komutunu verdiğinizde ise Python3 çalışacaktır.

Ama eğer sisteminizde Python2 bile kurulu değilse, ki bu çok çok düşük bir ihtimaldir, Python2'yi paket yöneticiniz yardımıyla sisteminize kurabilirsiniz. Şu anda piyasada olup da paket deposunda Python bulundurmayan GNU/Linux dağıtımı pek azdır.

GNU/Linux'ta Python'ı nasıl çalıştıracağımızı ve farklı Python sürümlerini bir arada nasıl kullanacağımızı öğrendiğimize göre, Windows kullanıcılarının durumuna bakabiliriz.

Windows Kullanıcıları

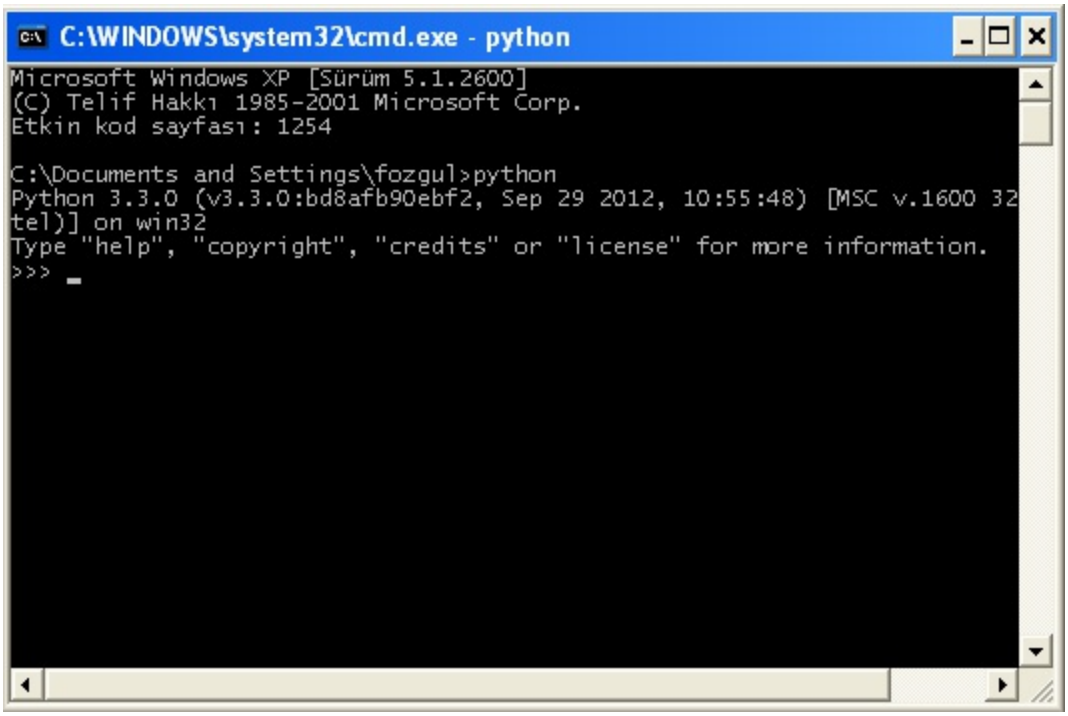
Windows kullanıcıları Python3'ü iki şekilde başlatabilir:

1. *Başlat > Tüm Programlar > Python3.3> Python (Command Line)* yolunu takip ederek.
2. Komut satırında python komutunu vererek.

Eğer birinci yolu tercih ederseniz, Python'ın size sunduğu komut satırına doğrudan ulaşmış olursunuz. Ancak Python komut satırına bu şekilde ulaştığınızda bazı kısıtlamalarla karşı karşıya kalırsınız. Doğrudan Python'ın komut satırına ulaşmak yerine önce MS-DOS komut satırına ulaşıp, oradan Python komut satırına ulaşmak özellikle ileride yapacağınız çalışmalar açısından çok daha mantıklı olacaktır. O yüzden komut satırına bu şekilde ulaşmak yerine ikinci seçeneği tercih etmenizi tavsiye ederim. Bunun için önceki bölümlerde gösterdiğimiz şekilde komut satırına ulaşın ve orada şu komutu çalıştırın:

```
python
```

Bu komutu verdiğinizde şuna benzer bir ekranla karşılaşacaksınız:



Eğer bu komut yukarıdakine bir benzer bir ekran yerine bir hata mesajı verdiyse kurulum sırasında bazı adımları eksik veya yanlış yapmış olabilirsiniz. Yukarıdaki komut çalışmıyorsa, muhtemelen kurulum sırasında bahsettiğimiz *Add python.exe to path* adımını yapmayı unutmuşsunuzdur. Eğer öyleyse, kurulum dosyasını tekrar çalıştırıp, ilgili adımı gerçekleştirmeniz veya Python'ı kendiniz YOL'a eklemeniz gerekiyor.

python komutunu başarıyla çalıştırabildiğinizi varsayarak yolumuza devam edelim.

Farklı Sürümleri Birlikte Kullanmak

Daha önce de dediğimiz gibi, şu anda piyasada iki farklı Python serisi bulunuyor: Python2 ve Python3. Çok uzun zamandan beri kullanımda olduğu için, Python2 Python3'e kıyasla daha yaygın. Eğer hem Python2 ile yazılmış programları çalıştırmak, hem de Python3 ile geliştirme yapmak istiyorsanız, sisteminizde hem Python2'yi hem de Python3'ü aynı anda bulundurmayı tercih edebilirsiniz. Peki bunu nasıl yapacaksınız?

Windows'ta bu işlemi yapmak çok kolaydır. python.org/download adresine giderek farklı Python sürümlerini bilgisayarınıza indirebilir ve bunları bilgisayarınıza normal bir şekilde kurabilirsiniz. Bu şekilde sisteminize istediğiniz sayıda farklı Python sürümü kurabilirsiniz. Peki bu farklı sürümlere nasıl ulaşacaksınız?

Windows işletim sistemlerinde hangi Python sürümünü kurarsanız kurun, Python'ı çalıştırmanızı sağlayan dosyanın adı *python.exe* olacaktır. Eğer sisteminizde birden fazla Python sürümü varsa python komutu YOL'a ekli ilk Python sürümünü çalıştırır. Kurulum sırasında menüden *Add python.exe to Path* seçeneğini seçerek Python'ı otomatik olarak YOL'a eklediğinizde Python, 'C:Python33;' satırını YOL dizinlerinin en başına yerleştirir. Yani şöyle bir şey yapar:

C:\Python33;C:\WINDOWS\system32;C:\WINDOWS;

Eğer Python 3.3'ü kurmadan önce python komutu sizin sisteminizde farklı bir Python sürümünü başlatıyorsa, kurulumdan sonra o sürüm değil, yeni kurduğunuz bu Python 3.3 sürümü çalışmaya başlayacaktır.

Peki bu durumda bilgisayarımızda kurulu farklı Python sürümlerine nasıl ulaşacağız?

Python, bilgisayarınızdaki farklı Python sürümlerini çalıştırabilmemiz için bize özel bir program sunar. Bu programın adı 'py'.

Py adlı bu programı çalıştırmak için, daha önce gösterdiğimiz şekilde sistem komut satırına ulaşıyoruz ve orada şu komutu veriyoruz:

```
py
```

Bu komutu verdiğinizde (teorik olarak) sisteminize en son kurduğunuz Python sürümü çalışmaya başlayacaktır. Ancak bu her zaman böyle olmayabilir. Ya da aldığınız çıktı beklediğiniz gibi olmayabilir. O yüzden bu komutu verdiğinizde hangi sürümün başladığına dikkat edin.

Eğer sisteminizde birden fazla Python sürümü kurulu ise, bu betik yardımıyla istediğiniz sürümü başlatabilirsiniz. Mesela sisteminizde hem Python'ın 2.x sürümlerinden biri, hem de Python'ın 3.x sürümlerinden biri kurulu ise, şu komut yardımıyla Python 2.x'i başlatabilirsiniz:

```
py -2
```

Python 3.x'i başlatmak için ise şu komutu veriyoruz:

```
py -3
```

Eğer sisteminizde birden fazla Python2 veya birden fazla Python3 sürümü kurulu ise, ana ve alt sürüm numaralarını belirterek istediğiniz sürüme ulaşabilirsiniz:

```
py -2.6
```

```
py -2.7
```

```
py -3.2
```

```
py -3.3
```

Bu arada dikkat ettiyseniz, Python programlarını başlatabilmek için hem python hem de py komutunu kullanma imkanına sahibiz. Eğer sisteminizde tek bir Python sürümü kurulu ise, Python'ı başlatmak için python komutunu kullanmak isteyebilir, farklı sürümlerin bir arada bulunduğu durumlarda ise py ile bu farklı sürümlere tek tek erişmek isteyebilirsiniz.

Böylece Python'la ilgili en temel bilgileri edinmiş olduk. Bu bölümde öğrendiklerimiz sayesinde Python programlama dilini bilgisayarımıza kurabiliyor ve bu programlama dilini başarıyla

çalıştırabiliyoruz.

Hangi Komut Hangi Sürümü Çalıştırıyor?

Artık Python programlama dilinin bilgisayarımıza nasıl kurulacağını ve bu programlama dilinin nasıl çalıştırılacağını biliyoruz. Ancak konunun öneminden ötürü, tekrar vurgulayıp, cevabını bilip bilmediğinizden emin olmak istediğimiz bir soru var: Kullandığınız işletim sisteminde acaba hangi komut, hangi Python sürümünü çalıştırıyor?

Bu kitapta anlattığımız farklı yöntemleri takip ederek, Python programlama dilini bilgisayarınıza farklı şekillerde kurmuş olabilirsiniz. Örneğin Python programlama dilini, kullandığınız GNU/Linux dağıtımının paket yöneticisi aracılığıyla kurduysanız, Python'ı başlatmak için python3 komutunu kullanmanız gerekebilir. Aynı şekilde, eğer Python'ı Windows'a kurduysanız, bu programlama dilini çalıştırmak için python komutunu kullanıyor olabilirsiniz. Bütün bunlardan farklı olarak, eğer Python'ın kaynak kodlarını sitesinden indirip derlediyseniz, Python'ı çalıştırmak için kendi belirlediğiniz bambaşka bir adı da kullanıyor olabilirsiniz. Örneğin belki de Python'ı çalıştırmak için py3 gibi bir komut kullanıyorsunuzdur...

Python programlama dilini çalıştırmak için hangi komutu kullanıyor olursanız olun, lütfen bir sonraki konuya geçmeden önce kendi kendinize şu soruları sorun:

1. Kullandığım işletim sisteminde Python programı halihazırda kurulu mu?
2. Kullandığım işletim sisteminde toplam kaç farklı Python sürümü var?
3. python komutu bu Python sürümlerinden hangisini çalıştırıyor?
4. python3 komutu çalışıyor mu?
5. Eğer çalışıyorsa, bu komut Python sürümlerinden hangisini çalıştırıyor?
6. Kaynaktan derlediğim Python sürümünü çalıştırmak için hangi komutu kullanıyorum?

Biz bu kitapta şunları varsayacağız:

1. Kullandığınız işletim sisteminde Python'ın **2.x** sürümlerini python komutuyla çalıştırıyorsunuz.
2. Kullandığınız işletim sisteminde Python'ın **3.x** sürümlerini python3 komutuyla çalıştırıyorsunuz.

Bu kitaptan yararlanırken, bu varsayımları göz önünde bulundurmalı, eğer bunlardan farklı komutlar kullanıyorsanız, kodlarınızı ona göre ayarlamalısınız.

Sistem Komut Satırı ve Python Komut Satırı

Buraya kadar Python programlama dilini nasıl çalıştıracağımız konusundaki bütün bilgileri edindik. Ancak programlamaya yeni başlayanların çok sık yaptığı bir hata var: Sistem komut satırı ile Python komut satırını birbirine karıştırmak.

Asla unutmayın, kullandığınız işletim sisteminin komut satırı ile Python'ın komut satırı birbirinden

farklı iki ortamdır. Yani Windows'ta cmd, Ubuntu'da ise *Ctrl+Alt+T* ile ulaştığınız ortam sistem komut satırı iken, bu ortamı açıp python3 (veya python ya da py3) komutu vererek ulaştığınız ortam Python'ın komut satırıdır. Sistem komut satırında sistem komutları (mesela cd, ls, dir, pwd) verilirken, Python komut satırında, biraz sonra öğrenmeye başlayacağımız Python komutları verilir. Dolayısıyla python3 (veya python ya da py3) komutunu verdikten sonra ulaştığınız ortamda cd Desktop ve ls gibi sistem komutlarını kullanmaya çalışmanız sizi hüsrana uğratacaktır.

başa dön

Etkileşimli Python

Önceki bölümlerde Python programlama dili ve genel olarak programlamanın temelleri üzerine epey söz söyledik. Dolayısıyla artık şu soruların cevabını rahatlıkla verebilecek durumdayız:

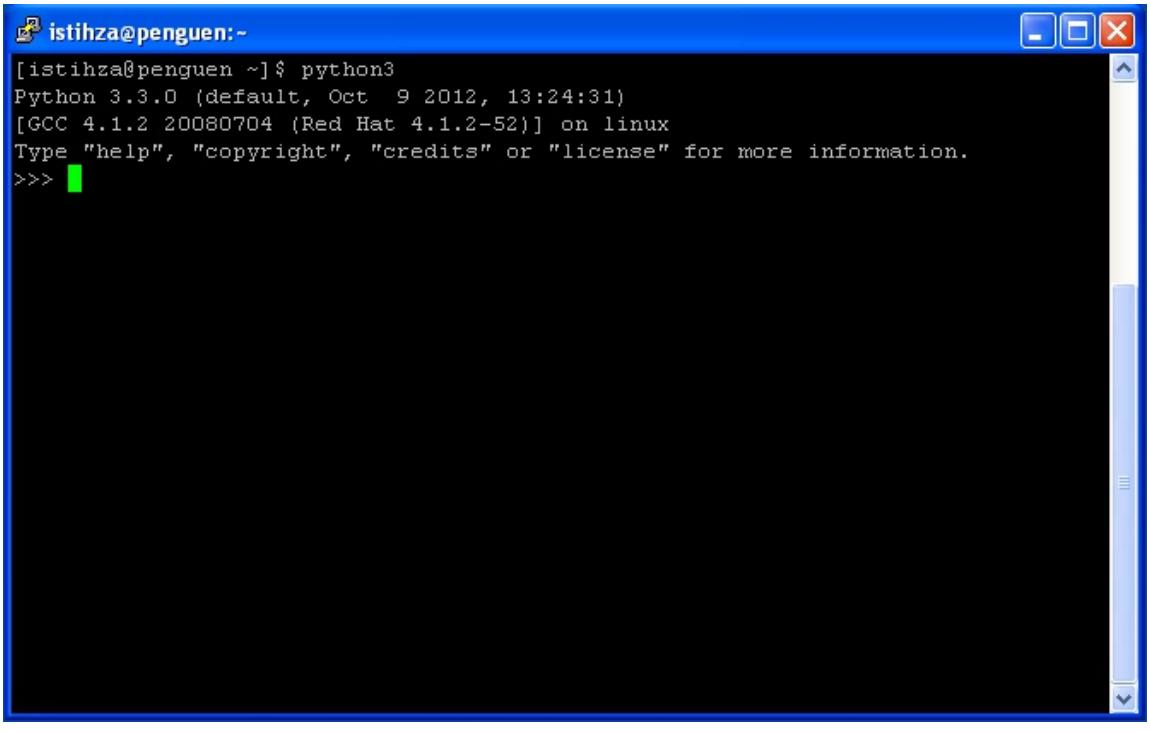
1. Python nasıl kurulur?
2. Python nasıl çalıştırılır?
3. YOL (*PATH*) nedir?
4. Bir dizin YOL'a nasıl eklenir?
5. Farklı işletim sistemlerinde komut satırına nasıl ulaşılır?
6. Komut satırı üzerinde dizinler arasında nasıl dolaşılır?
7. Komut satırında bir dizinin içeriği nasıl listelenir?
8. Sistem komut satırı ile Python komut satırı arasındaki fark nedir?

Eğer bu temel sorulara cevap verebiliyorsanız okumaya devam edin. Ama değilse, lütfen daha fazla ilerlemeden geri dönüp eksik bilgilerinizi tamamlayın.

Yukarıdaki soruların cevabını verebildiğinizi varsayarak yolumuza devam edelim...

Python'ın Etkileşimli Kabuğu

Dediğimiz gibi, şu ana kadar öğrendiklerimiz sayesinde Python programlama dilinin farklı sistemlere nasıl kurulacağını ve nasıl çalıştırılacağını biliyoruz. Dolayısıyla Python'ı bir önceki bölümde anlattığımız şekilde çalıştırdığımız zaman şuna benzer bir ekranla karşılaşacağımızın farkındayız:



Biz şimdiye kadar bu ekrana Python komut satırı demeyi tercih ettik. Dilerseniz bundan sonra da bu adı kullanmaya devam edebilirsiniz. Ancak teknik olarak bu ekrana etkileşimli kabuk (*interactive*

shell) adı verildiğini bilmemizde fayda var. Etkileşimli kabuk, bizim Python programlama dili ile ilişki kurabileceğimiz, yani onunla etkileşebileceğimiz bir üst katmandır. Etkileşimli kabuk, asıl programımız içinde kullanacağımız kodları deneme imkanı sunar bize. Burası bir nevi test alanı gibidir. Örneğin bir Python kodunun çalışıp çalışmadığını denemek veya nasıl çalıştığını, ne sonuç verdiğini görmek istediğimizde bu ekran son derece faydalı bir araç olarak karşımıza çıkar. Bu ortam, özellikle Python'a yeni başlayanların bu programlama diline aşinalık kazanmasını sağlaması açısından da bulunmaz bir araçtır. Biz de bu bölümde etkileşimli kabuk üzerinde bazı çalışmalar yaparak, Python'a alışma turları atacağız.

Bu arada, geçen bölümde söylediğimiz gibi, bu ortamın sistem komut satırı adını verdiğimiz ortamdan farklı olduğunu aklımızdan çıkarmıyoruz. O zaman da dediğimiz gibi, sistem komut satırında sistem komutları, Python komut satırında (yani etkileşimli kabukta) ise Python komutları verilir. Mesela `echo %PATH%`, `cd Desktop`, `dir` ve `ls` birer sistem komutudur. Eğer bu komutları etkileşimli kabukta vermeye kalkışırsanız, bunlar birer Python komutu olmadığı için, Python size bir hata mesajı gösterecektir. Mesela Python'ın etkileşimli kabuğunda `cd Desktop` komutunu verirseniz şöyle bir hata alırsınız:

```
>>> cd Desktop

  File "<stdin>", line 1
    cd Desktop
           ^
SyntaxError: invalid syntax
```

Çünkü `cd Desktop` bir Python komutu değildir. O yüzden bu komutu Python'ın etkileşimli kabuğunda veremeyiz. Bu komutu ancak ve ancak kullandığımız işletim sisteminin komut satırında verebiliriz.

Ne diyorduk? Etkileşimli kabuk bir veya birkaç satırlık kodları denemek/test etmek için gayet uygun bir araçtır. İsterseniz konuyu daha fazla lafa boğmayalım. Zira etkileşimli kabuğu kullandıkça bunun ne büyük bir nimet olduğunu siz de anlayacaksınız. Özellikle derlenerek çalıştırılan programlama dilleri ile uğraşmış olan arkadaşlarım, etkileşimli kabuğun gücünü gördüklerinde göz yaşlarına hakim olamayacaklar.

Farklı işletim sistemlerinde `py3`, `python3` veya `python` komutunu vererek Python'ın komut satırına nasıl erişebileceğimizi önceki derslerde ayrıntılı olarak anlatmıştık. Etkileşimli kabuğa ulaşmakta sıkıntı yaşıyorsanız eski konuları tekrar gözden geçirmenizi tavsiye ederim.

Etkileşimli kabuk üzerinde çalışmaya başlamadan önce dilerseniz önemli bir konuyu açıklığa kavuşturalım: Etkileşimli kabuğu başarıyla çalıştırdık. Peki bu kabuktan çıkmak istersek ne yapacağız? Elbette doğrudan pencere üzerindeki çarpı tuşuna basarak bu ortamı terk edebilirsiniz. Ancak bu işlemi kaba kuvvete başvurmadan yapmanın bir yolu olmalı, değil mi?

Etkileşimli kabuktan çıkmanın birkaç farklı yolu vardır:

1. Pencere üzerindeki çarpı düğmesine basmak (kaba kuvvet)

2. Önce *Ctrl+Z* tuşlarına, ardından da *Enter* tuşuna basmak (Windows)
3. *Ctrl+Z* tuşlarına basmak (GNU/Linux)
4. Önce *F6* tuşuna, ardından da *Enter* tuşuna basmak (Windows)
5. quit() yazıp *Enter* tuşuna basmak (Bütün işletim sistemleri)
6. import sys; sys.exit() komutunu vermek (Bütün işletim sistemleri)

Siz bu farklı yöntemler arasından, kolayınıza hangisi geliyorsa onu seçebilirsiniz. Bu satırların yazarı, Windows'ta 2 numaralı; GNU/Linux'ta ise 3 numaralı seçeneği tercih ediyor.

Etkileşimli Kabukta İlk Adımlar

Python'da etkileşimli kabuğu nasıl çalıştıracağımızı ve bu ortamı nasıl terk edeceğimizi öğrendiğimize göre artık etkileşimli kabuk aracılığıyla Python programlama dilinde ilk adımlarımızı atmaya başlayabiliriz.

Şimdi kendi sistemimize uygun bir şekilde etkileşimli kabuğu tekrar çalıştıralım. Etkileşimli kabuğu çalıştırdığımızda ekranda görünen >>> işareti Python'ın bizden komut almaya hazır olduğunu gösteriyor. Python kodlarımızı bu >>> işaretinden hemen sonra, **hiç boşluk bırakmadan** yazacağız.

Buradaki 'hiç boşluk bırakmadan' kısmı önemli. Python'a yeni başlayanların en sık yaptığı hatalardan biri >>> işareti ile komut arasında boşluk bırakmalarıdır. Eğer bu şekilde boşluk bırakırsanız yazdığınız kod hata verecektir.

İsterseniz basit bir deneme yapalım. >>> işaretinden hemen sonra, hiç boşluk bırakmadan şu komutu yazalım:

```
>>> "Merhaba Zalim Dünya!"
```

Bu arada yukarıdaki kodlar içinde görünen >>> işaretini siz yazmayacaksınız. Bu işareti etkileşimli kabuğun görünümünü temsil etmek için yerleştirdik oraya. Siz *"Merhaba Zalim Dünya!"* satırını yazdıktan sonra doğruca *Enter* düğmesine basacaksınız.

Bu komutu yazıp *Enter* tuşuna bastığımızda şöyle bir çıktı almış olmalıyız:

```
'Merhaba Zalim Dünya!'
```

Böylece yarım yamalak da olsa ilk Python programımızı yazmış olduk...

Muhtemelen bu kod, içinizde en ufak bir heyecan dahi uyandırmamıştır. Hatta böyle bir kod yazmak size anlamsız bile gelmiş olabilir. Ama aslında şu küçücük kod parçası bile bize Python programlama dili hakkında çok önemli ipuçları veriyor. Gelin isterseniz bu tek satırlık kodu biraz inceleyelim...

Karakter Dizilerine Giriş

Dediğimiz gibi, yukarıda yazdığımız küçücük kod parçası sizi heyecanlandırmamış olabilir, ama

aslında bu kod Python programlama dili ve bu dilin yapısı hakkında çok önemli bilgileri içinde barındırıyor.

Teknik olarak söylemek gerekirse, yukarıda yazdığımız *"Merhaba Zalim Dünya!"* ifadesi bir karakter dizisidir. İngilizcede buna *string* adı verilir ve programlama açısından son derece önemli bir kavramdır bu. Kavramın adından da rahatlıkla anlayabileceğiniz gibi, bir veya daha fazla karakterden oluşan öğelere karakter dizisi (*string*) diyoruz.

Karakter dizileri bütün programcılık maceramız boyunca karşımıza çıkacak. O yüzden bu kavramı ne kadar erken öğrenirsek o kadar iyi.

Peki bir verinin karakter dizisi olup olmamasının bize ne faydası var? Yani yukarıdaki cümle karakter dizisi olmuş olmamış bize ne?

Python'da, o anda elinizde bulunan bir verinin hangi tipte olduğunu bilmek son derece önemlidir. Çünkü bir verinin ait olduğu tip, o veriyle neler yapıp neler yapamayacağınızı belirler. Python'da her veri tipinin belli başlı özellikleri vardır. Dolayısıyla, elimizdeki bir verinin tipini bilmezsek o veriyi programlarımızda etkin bir şekilde kullanamayız. İşte yukarıda örneğini verdiğimiz *"Merhaba Zalim Dünya!"* adlı karakter dizisi de bir veri tipidir. Python'da karakter dizileri dışında başka veri tipleri de bulunur. Biraz sonra başka veri tiplerini de inceleyeceğiz.

Dikkat ederseniz *"Merhaba Zalim Dünya!"* adlı karakter dizisini tırnak içinde gösterdik. Bu da çok önemli bir bilgidir. Eğer bu cümleyi tırnak içine almazsak programımız hata verecektir:

```
>>> Merhaba Zalim Dünya!

  File "<stdin>", line 1
    Merhaba Zalim Dünya!
               ^
SyntaxError: invalid syntax
```

Zaten tırnak işaretleri, karakter dizilerinin ayırt edici özelliğidir. Öyle ki, Python'da tırnak içinde gösterdiğiniz her şey bir karakter dizisidir. Bu tanıma göre her şey bir karakter dizisi olabilir. Örneğin şu bir karakter dizisidir:

```
>>> "a"
```

Gördüğünüz gibi, tırnak içinde gösterilen tek karakterlik bir öğe de Python'da karakter dizisi sınıfına giriyor.

Dedik ya, her şey bir karakter dizisi olabilir. Mesela şu, içi boş bir karakter dizisidir:

```
>>> ""
```

Şu da içinde bir adet boşluk karakteri barındıran bir karakter dizisi...

```
>>> " "
```

Bu ikisi arasındaki farka dikkat ediyoruz: Python'da 'boş karakter dizisi' ve 'bir adet boşluktan oluşan karakter dizisi' birbirlerinden farklı iki kavramdır. Adından da anlaşılacağı gibi, boş karakter dizileri içlerinde hiçbir karakter (başka bir deyişle 'öğe') barındırmayan karakter dizileridir. Bir (veya daha fazla) boşluktan oluşan karakter dizileri ise içlerinde boşluk karakteri barındıran karakter dizileridir. Yani bu karakter dizilerinden biri boş, öteki ise doludur. Ama neticede her ikisi de karakter dizisidir. Şu anda oldukça anlamsız bir konu üzerinde vakit kaybediyormuşuz hissine kapılmış olabilirsiniz, ama emin olun, Python programlama diline yeni başlayanların önemli tökezleme noktalarından biridir bu söylediğimiz şey...

Dilerseniz biz karakter dizilerine elimizin alışması için birkaç örnek verelim:

```
>>> "Elma"

'Elma'

>>> "Guido Van Rossum"

'Guido Van Rossum'

>>> "Python programlama dili"

'Python programlama dili'

>>> "ömnhbgfgh"

'ömnhbgfgh'

>>> "$5&"

'$5&'

>>> ""

''

>>> " "

' '

>>> " "

' '
```

Yukarıdaki örneklerin hepsi birer karakter dizisidir. Dikkat ettiyseniz yukarıdaki karakter dizilerinin hepsinin ortak özelliği tırnak içinde gösteriliyor olmasıdır. Dediğimiz gibi, tırnak işaretleri karakter dizilerinin ayırt edici özelliğidir.

Peki bir verinin karakter dizisi olup olmadığından nasıl emin olabilirsiniz?

Eğer herhangi bir verinin karakter dizisi olup olmadığı konusunda tereddütünüz varsa, type() adlı bir fonksiyondan yararlanarak o verinin tipini sorgulayabilirsiniz. Bu fonksiyonu şöyle kullanıyoruz:

```
>>> type("Elma")

<class 'str'>
```

Not

Bu 'fonksiyon' kelimesinin kafanızı karıştırmasına izin vermeyin. İlerde fonksiyonları oldukça ayrıntılı bir şekilde inceleyeceğimiz için, type() ifadesinin bir fonksiyon olduğunu bilmeniz şimdilik yeterli olacaktır. Üstelik fonksiyon konusunu ayrıntılı bir şekilde anlatma vakti geldiğinde siz fonksiyonlara dair pek çok şeyi zaten öğrenmiş olacaksınız.

Burada amacımız *"Elma"* adlı öğenin tipini denetlemek. Denetlenecek öğeyi type() fonksiyonunun parantezleri arasında belirttiğimize dikkat edin. (Fonksiyonların parantezleri içinde belirtilen değerlere teknik dilde parametre adı verilir.)

Yukarıdaki çıktıda bizi ilgilendiren kısım, sondaki 'str' ifadesi. Tahmin edebileceğiniz gibi, bu ifade *string* kelimesinin kısaltmasıdır. Bu kelimenin Türkçede karakter dizisi anlamına geldiğini söylemiştik. O halde yukarıdaki çıktıya bakarak, *"Elma"* öğesinin bir karakter dizisi olduğunu söyleyebiliyoruz.

type() fonksiyonu yardımıyla kendi kendinize bazı denemeler yaparak konuyu iyice sindirmenizi tavsiye ederim. Mesela *"½{656$#gfd"* ifadesinin hangi sınıfa girdiğini kontrol etmekle başlayabilirsiniz.

Peki karakter dizileri ile neler yapabiliriz? Şu anda Python bilgimiz kısıtlı olduğu için karakter dizileri ile çok fazla şey yapamayız, ama ilerde bilgimiz arttıkça, karakter dizileriyle sıkı fıkı olacağız.

Esasında, henüz bilgimiz kısıtlı da olsa karakter dizileriyle yine de ufak tefek bazı şeyler yapamayacak durumda değiliz. Mesela şu anki bilgilerimizi ve görür görmez size tanıdık gelecek bazı basit parçaları kullanarak, karakter dizilerini birbirleriyle birleştirebiliriz:

```
>>> "istihza" + ".com"

'istihza.com'
```

Burada + işaretini kullanarak karakter dizilerini nasıl birleştirebildiğimize dikkat edin. İki karakter dizisini + işareti ile birleştirdiğimizde karakter dizilerinin arasında boşluk olmadığına özellikle dikkatinizi çekmek isterim. Bu durumu şu örnekte daha net görebiliriz:

```
>>> "Fırat" + "Özgül"

'FıratÖzgül'
```

Gördüğünüz gibi, bu iki karakter dizisi, arada boşluk olmadan birbiriyle bitiştirildi. Araya boşluk eklemek için birkaç farklı yöntemden yararlanabilirsiniz:

```
>>> "Fırat" + " " + "Özgül"

'Fırat Özgül'
```

Burada iki karakter dizisi arasına bir adet boşluk karakteri yerleştirdik. Aynı etkiyi şu şekilde de elde edebilirsiniz:

```
>>> "Fırat" + " Özgül"
```

Burada da *Özgül* karakter dizisinin başına bir adet boşluk yerleştirerek istediğimiz çıktıyı elde ettik.

Bu arada, karakter dizilerini birleştirmek için mutlaka + işareti kullanmak zorunda değilsiniz. Siz + işaretini kullanmasanız da Python sizin karakter dizilerini birleştirmek istediğinizi anlayacak kadar zekidir:

```
>>> "www" "." "google" "." "com"

'www.google.com'
```

Ancak gördüğünüz gibi, + işaretini kullandığınızda kodlarınız daha okunaklı oluyor.

+ işareti dışında karakter dizileri ile birlikte * (çarpı) işaretini de kullanabiliriz. O zaman şöyle bir etki elde ederiz:

```
>>> "w" * 3

'www'

>>> "yavaş " * 2

'yavaş yavaş '

>>> "-" * 10
```

```
'----------'

>>> "uzak" + " " * 5 + "çok uzak..."

'uzak çok uzak...'
```

Gördüğünüz gibi, çok basit parçaları bir araya getirerek karmaşık çıktılar elde edebiliyoruz. Mesela son örnekte *"uzak"* adlı karakter dizisine önce *5* adet boşluk karakteri (" " * 5), ardından da *"çok uzak..."* adlı karakter dizisini ekleyerek istediğimiz çıktıyı aldık.

Burada + ve * adlı iki yeni araç görüyoruz. Bunlar aslında sayılarla birlikte kullanılan birer aritmetik işleçtir. Normalde + işleci toplama işlemleri için, * işleci ise çarpma işlemleri için kullanılır. Ama yukarıdaki örneklerde, + işaretinin 'birleştirme'; * işaretinin ise 'tekrarlama' anlamından ötürü bu iki işleci bazı durumlarda karakter dizileri ile birlikte de kullanabiliyoruz. Bunların dışında bir de - (eksi) ve / (bölü) işleçleri bulunur. Ancak bu işaretleri karakter dizileri ile birlikte kullanamıyoruz.

Karakter dizilerini sonraki bir bölümde bütün ayrıntılarıyla inceleyeceğiz. O yüzden şimdilik bu konuya bir ara verelim.

Sayılara Giriş

Dedik ki, Python'da birtakım veri tipleri bulunur ve karakter dizileri de bu veri tiplerinden yalnızca biridir. Python'da karakter dizileri dışında başka veri tiplerinin de bulunduğunu söylemiştik hatırlarsanız. İşte veri tipi olarak karakter dizilerinin dışında, biraz önce aritmetik işleçler vesilesiyle sözünü ettiğimiz, bir de 'sayı' (*number*) adlı bir veri tipi vardır.

Herhalde sayıların ne anlama geldiğini tarif etmeye gerek yok. Bunlar bildiğimiz sayılardır. Mesela:

```
>>> 23

23

>>> 4567

4567

>>> 2.3

2.3

>>> (10+2j)

(10+2j)
```

Python'da sayıların farklı alt türleri bulunur. Mesela tamsayılar, kayan noktalı sayılar, karmaşık

sayılar...

Yukarıdaki örnekler arasında geçen *23* ve *4567* birer tamsayıdır. İngilizcede bu tür sayılara *integer* adı verilir.

2.3 ise bir kayan noktalı sayıdır (*floating point number* veya kısaca *float*). Bu arada kayan noktalı sayılarda basamak ayracı olarak virgül değil, nokta işareti kullandığımıza dikkat edin.

En sonda gördüğümüz *10+2j* sayısı ise bir karmaşık sayıdır (*complex*). Ancak eğer matematikle yoğun bir şekilde uğraşmıyorsanız karmaşık sayılar pek karşınıza çıkmaz.

Sayıları temel olarak öğrendiğimize göre etkileşimli kabuğu basit bir hesap makinesi niyetine kullanabiliriz:

```
>>> 5 + 2

7

>>> 25 * 25

625

>>> 5 / 2

2.5

>>> 10 - 3

7
```

Yukarıdaki örneklerde kullandığımız aritmetik işleçlerden biraz önce bahsetmiştik. O yüzden bunlara yabancılık çektiğinizi zannetmiyorum. Ama biz yine de bu işleçleri ve görevlerini şöylece sıralayalım:

İşleç	Görevi
+	toplama
-	çıkarma
*	çarpma
/	bölme

Yukarıdaki örneklerde bir şey dikkatinizi çekmiş olmalı: Karakter dizilerini tanımlarken tırnak işaretlerini kullandık. Ancak sayılarda tırnak işareti yok. Daha önce de dediğimiz gibi, tırnak işaretleri karakter dizilerinin ayırt edici özelliğidir. Python'da tırnak içinde gösterdiğiniz her şey bir karakter dizisidir. Mesela şu örneklere bakalım:

```
>>> 34657

34657
```

Bu bir sayıdır. Peki ya şu?

```
>>> "34657"

'34657'
```

Bu ise bir karakter dizisidir. Dilerseniz biraz önce öğrendiğimiz type() fonksiyonu yardımıyla bu verilerin tipini sorgulayalım:

```
>>> type(34657)

<class 'int'>
```

Buradaki 'int' ifadesi İngilizce "*integer*", yani tamsayı kelimesinin kısaltmasıdır. Demek ki *34657* sayısı bir tamsayı imiş. Bir de şuna bakalım:

```
>>> type("34657")

<class 'str'>
```

Gördüğünüz gibi, *34657* sayısını tırnak içine aldığımızda bu sayı artık sayı olma özelliğini yitiriyor ve bir karakter dizisi oluyor. Şu anda bu çok önemsiz bir ayrıntıymış gibi gelebilir size, ama aslında son derece önemli bir konudur bu. Bu durumun etkilerini şu örneklerde görebilirsiniz:

```
>>> 23 + 65

88
```

Burada normal bir şekilde iki sayıyı birbiriyle topladık.

Bir de şuna bakın:

```
>>> "23" + "65"

'2365'
```

Burada ise Python iki karakter dizisini yan yana yazmakla yetindi; yani bunları birleştirdi. Python açısından *"23"* ve *23* birbirinden farklıdır. *"23"* bir karakter dizisi iken, *23* bir sayıdır. Aynı şey *"65"* ve *65* için de geçerlidir. Yani Python açısından *"65"* ile *"Merhaba Zalim Dünya!"* arasında hiç bir fark yoktur. Bunların ikisi de karakter dizisi sınıfına girer. Ancak *65* ile *"65"* birbirinden farklıdır. *65* bir sayı iken, *"65"* bir karakter dizisidir.

Bu bilgi, özellikle aritmetik işlemlerde büyük önem taşır. Bunu dilerseniz şu örnekler üzerinde gösterelim:

```
>>> 45 + "45"

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Gördüğünüz gibi, yukarıdaki kodlar hata veriyor. Bunun sebebi bir sayı ile (*45*) karakter dizisini (*"45"*) birbiriyle toplamaya çalışmamızdır. Asla unutmayın, aritmetik işlemler ancak sayılar arasında yapılır. Karakter dizileri ile herhangi bir aritmetik işlem yapılamaz.

Bir de şuna bakalım:

```
>>> 45 + 45

90
```

Bu kodlar ise düzgün çalışır. Çünkü burada iki sayıyı aritmetik işleme soktuk ve başarılı olduk.

Son olarak şu örneği verelim:

```
>>> "45" + "45"

'4545'
```

Burada + işlecinin toplama anlamına gelmediğine dikkat edin. Bu işleç burada iki karakter dizisini birleştirme görevi üstleniyor. Yani yukarıdaki örneğin şu örnekten hiçbir farkı yoktur:

```
>>> "istihza." + "com"

'istihza.com'
```

Bu iki örnekte de yaptığımız şey karakter dizilerini birbiriyle birleştirmektir.

Gördüğünüz gibi, + işlecinin sağındaki ve solundaki değerler birer karakter dizisi ise bu işleç bu iki değeri birbiriyle birleştiriyor. Ama eğer bu değerler birer sayı ise + işleci bu değerleri birbiriyle aritmetik olarak topluyor.

* işleci de + işlecine benzer bir iş yapar. Yani eğer * işleci bir sayı ve bir karakter dizisi ile karşılaşırsa, o karakter dizisini, verilen sayı kadar tekrarlar. Örneğin:

```
>>> "w" * 3

'www'
```

Burada * işleci bir karakter dizisi (*"w"*) ve bir sayı (*3*) arasında işlem yaptığı için, karakter dizisini, ilgili sayı kadar tekrarlıyor. Yani *"w"* karakter dizisini *3* kez tekrarlıyor.

Bir de şuna bakalım:

```
>>> 25 * 3

75
```

Burada ise * işleci iki adet sayı arasında işlem yaptığı için bu değerleri birbiriyle aritmetik olarak çarpıyor ve *75* değerini elde etmemizi sağlıyor.

Gördüğünüz gibi, o anda elimizde bulunan verilerin tipini bilmek gerçekten de büyük önem taşıyor. Çünkü eğer elimizdeki verilerin tipini bilmezsek nasıl sonuçlar elde edeceğimizi de kestiremeyiz.

Böylece karakter dizileri ile sayılar arasındaki farkı öğrenmiş olduk. Bu bilgiler size önemsizmiş gibi gelebilir, ama aslında karakter dizileri ile sayılar arasındaki farkı anlamak, Python programlama dilinin önemli bir bölümünü öğrenmiş olmak demektir. İleride yazacağınız en karmaşık programlarda bile, bazen programınızın çalışmamasının (veya daha kötüsü yanlış çalışmasının) nedeninin karakter dizileri ile sayıları birbirine karıştırmanız olduğunu göreceksiniz. O yüzden burada öğrendiğiniz hiçbir bilgi kırıntısını baştan savmamanızı (ve sabırsızlık ya da acelecilik etmemenizi) tavsiye ederim.

Değişkenler

Şimdi şöyle bir durum düşünün: Diyelim ki sisteme kayıt için kullanıcı adı ve parola belirlenmesini isteyen bir program yazıyorsunuz. Yazacağınız bu programda, belirlenebilecek kullanıcı adı ve parolanın toplam uzunluğu *40* karakteri geçmeyecek.

Bu programı yazarken ilk aşamada yapmanız gereken şey, kullanıcının belirlediği kullanıcı adı ve parolanın uzunluğunu tek tek denetlemek olmalı.

Mesela kullanıcı şöyle bir kullanıcı adı belirlemiş olsun:

```
firat_ozgul_1980
```

Kullanıcının belirlediği parola ise şu olsun:

```
rT%65#$hGfUY56123
```

İşte bizim öncelikle kullanıcıdan gelen bu verilerin teker teker uzunluğunu biliyor olmamız lazım, ki bu verilerin toplam *40* karakter sınırını aşıp aşmadığını denetleyebilelim.

Peki bu verilerin uzunluğunu nasıl ölçeceğiz? Elbette bunun için verilerdeki harfleri elle tek tek saymayacağız. Bunun yerine, Python programlama dilinin bize sunduğu bir aracı kullanacağız. Peki nedir bu araç?

Hatırlarsanız birkaç sayfa önce type() adlı bir fonksiyondan söz etmiştik. Bu fonksiyonun görevi bir verinin hangi tipte olduğunu bize bildirmekti. İşte tıpkı type() gibi, Python'da len() adlı başka bir fonksiyon daha bulunur. Bu fonksiyonun görevi ise karakter dizilerinin (ve ileride göreceğimiz gibi, başka veri tiplerinin) uzunluğunu ölçmektir. Yani bu fonksiyonu kullanarak bir karakter dizisinin toplam kaç karakterden oluştuğunu öğrenebiliriz.

Biz henüz kullanıcıdan nasıl veri alacağımızı bilmiyoruz. Ama şimdilik şunu söyleyebiliriz: Python'da kullanıcıdan herhangi bir veri aldığımızda, bu veri bize bir karakter dizisi olarak gelecektir. Yani kullanıcıdan yukarıdaki kullanıcı adı ve parolayı aldığımızı varsayarsak, bu veriler bize şu şekilde gelir:

```
"firat_ozgul_1980"
```

ve:

```
"rT%65#$hGfUY56123"
```

Gördüğünüz gibi, elde ettiğimiz veriler tırnak içinde yer alıyor. Yani bunlar birer karakter dizisi. Şimdi gelin yukarıda bahsettiğimiz len() fonksiyonunu kullanarak bu karakter dizilerinin uzunluğunu ölçelim.

Dediğimiz gibi, len() de tıpkı type() gibi bir fonksiyondur. Dolayısıyla len() fonksiyonunun kullanımı type() fonksiyonunun kullanımına çok benzer. Nasıl type() fonksiyonu bize, kendisine verdiğimiz parametrelerin **tipini** söylüyorsa, len() fonksiyonu da kendisine verdiğimiz parametrelerin **uzunluğunu** söyler.

Dikkatlice bakın:

```
>>> len("firat_ozgul_1980")

16

>>> len("rT%65#$hGfUY56123")

17
```

Demek ki *"firat_ozgul_1980"* adlı karakter dizisinde *16*; *"rT%65#$hGfUY56123"* adlı karakter dizisinde ise *17* karakter varmış. Bizim istediğimiz şey bu iki değerin toplam uzunluğunun *40* karakteri aşmaması. Bunu denetlemek için yapmamız gereken şey bu iki değerin uzunluğunu birbiriyle toplamak olmalı. Yani:

```
>>> len("firat_ozgul_1980") + len("rT%65#$hGfUY56123")
```

Buradan alacağımız sonuç *33* olacaktır. Demek ki kullanıcı *40* karakter limitini aşmamış. O halde programımız bu kullanıcı adı ve parolayı kabul edebilir...

Bu arada, belki farkettiniz, belki de farketmediniz, ama burada da çok önemli bir durumla karşı karşıyayız. Gördüğünüz gibi len() fonksiyonu bize sayı değerli bir veri gönderiyor. Gelin isterseniz bunu teyit edelim:

```
>>> type(len("firat_ozgul_1980"))

<class 'int'>
```

len() fonksiyonunun bize sayı değerli bir veri göndermesi sayesinde bu fonksiyondan elde ettiğimiz değerleri birbiriyle toplayabiliyoruz:

```
>>> len("firat_ozgul_1980") + len("rT%65#$hGfUY56123")

33
```

Eğer len() fonksiyonu bize sayı değil de mesela karakter dizisi verseydi, bu fonksiyondan elde ettiğimiz değerleri yukarıdaki gibi doğrudan birbiriyle aritmetik olarak toplayamazdık. Öyle bir durumda, bu iki veriyi birbiriyle toplamaya çalıştığımızda, + işleci *16* ve *17* değerlerini birbiriyle toplamak yerine bu değerleri birbiriyle birleştirerek bize *'1617'* gibi bir sonuç verecekti.

Her zaman söylediğimiz gibi, Python'da veri tipi kavramını çok iyi anlamak ve o anda elimizde bulunan bir verinin hangi tipte olduğunu bilmek çok önemlidir. Aksi halde programlarımızda hata yapmamız kaçınılmazdır.

Eğer yukarıda anlattığımız şeyleri kafa karıştırıcı bulduysanız hiç endişe etmeyin. Birkaç bölüm sonra input() adlı bir fonksiyondan bahsettiğimizde şimdi söylediğimiz şeyleri çok daha net anlayacaksınız.

Biraz sonra len() fonksiyonundan bahsetmeye devam edeceğiz, ama isterseniz ondan önce çok önemli bir konuya değinelim.

Biraz önce şöyle bir örnek vermiştik:

```
>>> len("firat_ozgul_1980")

16

>>> len("rT%65#$hGfUY56123")

17

>>> len("firat_ozgul_1980") + len("rT%65#$hGfUY56123")
```

Bu kodlar, istediğimiz şeyi gayet güzel yerine getiriyor. Ama sizce de yukarıdaki kodlarda çok rahatsız edici bir durum yok mu?

Dikkat ederseniz, yukarıdaki örneklerde kullandığımız verileri, program içinde her ihtiyaç duyduğumuzda tekrar tekrar yazdık. Böylece aynı program içinde iki kez *"firat_ozgul_1980"*; iki kez de *"rT%65#$hGfUY56123"* yazmak zorunda kaldık. Halbuki bu verileri programlarımızın içinde her ihtiyaç duyduğumuzda tekrar tekrar yazmak yerine bir değişkene atasak ve gerektiğinde o değişkeni kullansak çok daha iyi olmaz mı? Herhalde olur...

Peki nedir bu değişken dediğimiz şey?

Python'da bir program içinde değerlere verilen isimlere değişken denir. Hemen bir örnek verelim:

```
>>> n = 5
```

Burada *5* sayısını bir değişkene atadık. Değişkenimiz ise *n*. Ayrıca *5* sayısını bir değişkene atamak için = işaretinden yararlandığımıza da çok dikkat edin. Buradan, = işaretinin Python programlama dilinde değer atama işlemleri için kullanıldığı sonucunu çıkarıyoruz.

`n = 5` gibi bir komut yardımıyla *5* değerini *n* adlı değişkene atamamız sayesinde artık ne zaman *5* sayısına ihtiyaç duysak bu *n* değişkenini çağırmamız yeterli olacaktır:

```
>>> n

5

>>> n * 10

50

>>> n / 2

2.5
```

Gördüğünüz gibi, *5* değerini bir değişkene atadıktan sonra, bu *5* değerini kullanmamız gereken yerlerde sadece değişkenin adını kullandığımızda değişkenin değerini Python otomatik olarak yerine koyabiliyor. Yani `n = 5` komutuyla *n* adlı bir değişken tanımladıktan sonra, artık ne zaman *5* sayısına ihtiyaç duysak *n* değişkenini çağırmamız yeterli olacaktır. Python o *5* değerini otomatik olarak yerine koyar.

Şimdi de *pi* adlı başka bir değişken tanımlayalım:

```
>>> pi = 3.14
```

Bu *pi* değişkeninin değeri ile *n* değişkeninin değerini toplayalım:

```
>>> pi + n
```

```
8.14
```

Gördüğünüz gibi, değerleri her defasında tekrar yazmak yerine bunları bir değişkene atayıp, gereken yerde bu değişkeni kullanmak çok daha pratik bir yöntem.

Aynı şeyi programımız için de yapabiliriz:

```
>>> kullanıcı_adı = "firat_ozgul_1980"
>>> parola = "rT%65#$hGfUY56123"
```

= işaretini kullanarak ilgili değerlere artık birer ad verdiğimiz, yani bu değerleri birer değişkene atadığımız için, bu değerleri kullanmamız gereken yerlerde değerlerin kendisini uzun uzun yazmak yerine, belirlediğimiz değişken adlarını kullanabiliriz. Mesela:

```
>>> len(kullanıcı_adı)

16

>>> len(parola)

17

>>> len(kullanıcı_adı) + len(parola)

33

>>> k_adı_uzunluğu = len(kullanıcı_adı)
>>> type(k_adı_uzunluğu)

<class 'int'>
```

Gördüğünüz gibi, değişken kullanımı işlerimizi bir hayli kolaylaştırıyor.

Değişken Adı Belirleme Kuralları

Python programlama dilinde, değişken adı olarak belirleyebileceğimiz kelime sayısı neredeyse sınırsızdır. Yani hemen hemen her kelimeyi değişken adı olarak kullanabiliriz. Ama yine de değişken adı belirlerken dikkat etmemiz gereken bazı kurallar var. Bu kuralların bazıları zorunluluk, bazıları ise yalnızca tavsiye niteliğindedir.

Şimdi bu kuralları tek tek inceleyelim:

1. Değişken adları bir sayı ile başlayamaz. Yani şu kullanım yanlıştır:

```
>>> 3_kilo_elma = "5 TL"
```

2. Değişken adları aritmetik işleçlerle başlayamaz. Yani şu kullanım yanlıştır:

```
>>> +değer = 4568
```

3. Değişken adları ya bir alfabe harfiyle ya da _ işaretiyle başlamalıdır:

```
>>> _değer = 4568
>>> değer = 4568
```

4. Değişken adları içinde Türkçe karakterler kullanabilirsiniz. Ancak ileride beklenmedik uyum sorunları çıkması ihtimaline karşı değişken adlarında Türkçe karakter kullanmaktan kaçınmak isteyebilirsiniz.

5. Aşağıdaki kelimeleri değişken adı olarak kullanamazsınız:

```
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class',
'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for',
'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not',
'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

Bunlar Python'da özel anlam ifade eden kelimelerdir. Etkileşimli kabuk zaten bu kelimeleri değişken adı olarak kullanmanıza izin vermez. Örneğin:

```
>>> elif = "hoş kız"

  File "<stdin>", line 1
    elif = "hoş kız"
       ^
SyntaxError: invalid syntax

>>> as = "kare"

  File "<stdin>", line 1
    as = "kare"
     ^
SyntaxError: invalid syntax

>>> False = 45

  File "<stdin>", line 1
SyntaxError: assignment to keyword
```

Ama ilerde göreceğimiz gibi, programlarınızı bir dosyaya yazarken bu kelimeleri değişken adı olarak kullanmaya çalışırsanız programınız tespit etmesi çok güç hatalar üretecektir.

Bu arada elbette yukarıdaki listeyi bir çırpıda ezberlemeniz beklenmiyor sizden. Python programlama dilini öğrendikçe özel kelimeleri bir bakışta tanıyabilecek duruma geleceksiniz. Ayrıca eğer isterseniz şu komutları vererek, istediğiniz her an yukarıdaki listeye ulaşabilirsiniz:

```
>>> import keyword
>>> keyword.kwlist

['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class',
'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for',
'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not',
'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

Size bir soru: Acaba bu listede kaç tane kelime var?

Bu soru karşısında listedeki kelimeleri tek tek elle saymaya kalkışan arkadaşlarıma teessüflerimi iletiyorum... Bu tür işler için hangi aracı kullanabileceğimizi artık çok iyi biliyor olmalısınız:

```
>>> len(keyword.kwlist)

33
```

Bu kodları şöyle yazabileceğimizi de biliyorsunuz:

```
>>> yasaklı_kelimeler = keyword.kwlist
>>> len(yasaklı_kelimeler)

33
```

Bu arada, yukarıdaki kodların bir kısmını henüz anlayamamış olabilirsiniz. Hiç endişe etmeyin. Yukarıdaki kodları vermemizin sebebi değişken adı olarak kullanılamayacak kelimelere kısa yoldan nasıl ulaşabileceğinizi gösterebilmek içindir. Bir-iki bölüm sonra burada yazdığımız kodları rahatlıkla anlayabilecek düzeye geleceksiniz.

Yukarıda verdiğimiz kodların çıktısından anladığımıza göre, toplam *33* tane kelime varmış değişken adı belirlerken kullanmaktan kaçınmamız gereken...

6. Yukarıdaki kelimeler dışında, Python programlama diline ait fonksiyon ve benzeri araçların adlarını da değişken adı olarak kullanmamalısınız. Örneğin yazdığınız programlarda değişkenlerinize *type* veya *len* adı vermeyin. Çünkü 'type' ve 'len' Python'a ait iki önemli fonksiyonun adıdır. Eğer mesela bir değişkene *type* adını verirseniz, o programda artık type() fonksiyonunu kullanamazsınız:

```
>>> type = 3456
```

Bu örnekte *type* adında bir değişken tanımladık. Şimdi mesela *"elma"* kelimesinin tipini denetlemek için type() fonksiyonunu kullanmaya çalışalım:

```
>>> type("elma")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

Gördüğünüz gibi, artık type() fonksiyonu çalışmıyor. Çünkü siz 'type' kelimesini bir değişken adı olarak kullanarak, type() fonksiyonunu kullanılamaz hale getirdiniz.

Bu durumdan kurtulmak için etkileşimli kabuğu kapatıp tekrar açabilirsiniz. Ya da eğer etkileşimli kabuğu kapatmak istemiyorsanız şu komut yardımıyla *type* değişkenini ortadan kaldırmayı da tercih edebilirsiniz:

```
>>> del type
```

Böylece, (tahmin edebileceğiniz gibi *delete* (silmek) kelimesinin kısaltması olan) del komutuyla *type* değişkenini silmiş oldunuz. Artık 'type' kelimesi yine type() fonksiyonunu çağıracak:

```
>>> type("elma")

<class 'str'>
```

7. Değişken adlarını belirlerken, değişkeni oluşturan kelimeler arasında boşluk bırakılamaz. Yani şu kullanım yanlıştır:

```
>>> kullanıcı adı = "istihza"
```

Yukarıdaki değişkeni şu şekilde tanımlayabiliriz:

```
>>> kullanıcı_adı = "istihza"
```

Ya da şöyle:

```
>>> kullanıcıAdı = "istihza"
```

8. Değişken adları belirlerken, değişken adının, değişkenin değerini olabildiğince betimlemesine dikkat etmemiz kodlarımızın okunaklılığını artıracaktır. Örneğin:

```
>>> personel_sayısı = 45
```

Yukarıdaki, tanımladığı değere uygun bir değişken adıdır. Şu ise kurallara uygun bir değişken adı olsa da yeterince betimleyici değildir:

```
>>> sayı = 45
```

9. Değişken adları ne çok kısa, ne de çok uzun olmalıdır. Mesela şu değişken adı, kodları okuyan

kişiye, değişken değerinin anlamı konusunda pek fikir vermez:

```
>>> a = 345542353
```

Şu değişken adı ise gereksiz yere uzundur:

```
>>> türkiye_büyük_millet_meclisi_milletvekili_sayısı = 550
```

Değişken adlarının uzunluğunu makul seviyede tutmak esastır:

```
>>> tbmm_mv_sayısı = 550
```

Yukarıda verdiğimiz bütün bu örnekler bize, Python'da değişkenlerin, değerlere atanmış adlardan ibaret olduğunu gösteriyor. Değişkenler, yazdığımız programlarda bize çok büyük kolaylık sağlar. Mesela *123432456322* gibi bir sayıyı ya da *"Türkiye Cumhuriyeti Çalışma ve Sosyal Güvenlik Bakanlığı"* gibi bir karakter dizisini gerektiği her yerde tek tek elle yazmak yerine, bunları birer değişkene atayarak, gerektiğinde sadece bu değişken adını kullanmak çok daha mantıklı bir iştir.

Ayrıca zaten ileride kullanıcıdan veri almaya başladığınızda, aldığınız bu verileri, yazdığınız programda kullanabilmek için mutlaka bir değişkene atamanız gerekecek. O yüzden Python'daki değişken kavramını şimdiden iyi tanıyıp anlamakta büyük fayda var.

Uygulama Örnekleri

Gelin isterseniz yukarıda verdiğimiz bilgileri pekiştirmek için birkaç ufak alıştırma yapalım, alıştırma yaparken de sizi yine Python programlama diline ilişkin çok önemli bazı yeni bilgilerle tanıştıralım.

Diyelim ki aylık yol masrafımızı hesaplayan bir program yazmak istiyoruz. Elimizdeki verilerin şunlar olduğunu varsayalım:

1. Cumartesi-Pazar günleri çalışmıyoruz.
2. Dolayısıyla ayda *22* gün çalışıyoruz.
3. Evden işe gitmek için kullandığımız vasıtanın ücreti *1.5* TL
4. İşten eve dönmek için kullandığımız vasıtanın ücreti *1.4* TL

Aylık yol masrafımızı hesaplayabilmek için gidiş ve dönüş ücretlerini toplayıp, bunları çalıştığımız gün sayısıyla çarpmamız yeterli olacaktır. Elimizdeki bu bilgilere göre aylık yol masrafımızı hesaplamak için şöyle bir formül üretebiliriz:

```
aylık yol masrafı = çalışılan gün sayısı x (işe gidiş ücreti + işten dönüş ücreti)
```

Dilerseniz hemen bunu bir Python programı haline getirelim:

```
>>> 22 * (1.5 + 1.4)
```

```
63.8
```

Demek ki bir ayda *63.8* TL'lik bir yol masrafımız varmış.

Bu arada, yukarıdaki örnekte bir şey dikkatinizi çekmiş olmalı. Aritmetik işlemi yaparken bazı sayıları parantez içine aldık. Python'da aritmetik işlemler yapılırken alıştığımız matematik kuralları geçerlidir. Yani mesela aynı anda bölme, çıkarma, toplama ve çarpma işlemleri yapılacaksa işlem öncelik sırası önce bölme ve çarpma, sonra toplama ve çıkarma şeklinde olacaktır. Elbette siz parantezler yardımıyla bu işlem sırasını değiştirebilirsiniz.

Bu anlattıklarımıza göre, eğer yukarıda yol masrafını hesaplayan programda parantezleri kullanmazsak, işlem öncelik kuralları gereğince Python önce *22* ile *1.5*'i çarpıp, çıkan sonucu *1.4* ile toplayacağı için elde ettiğimiz sonuç yanlış çıkacaktır. Bizim burada doğru sonuç alabilmemiz için önce *1.5* ile *1.4*'ü toplamamız, çıkan sonucu da *22* ile çarpmamız gerekiyor. Bu sıralamayı da parantezler yardımıyla elde ediyoruz.

Yine dikkat ederseniz, yukarıdaki örnek programda aslında çok verimsiz bir yol izledik. Gördüğünüz gibi, bu programda bütün değerleri tek tek elle kendimiz giriyoruz. Örneğin çalışılan gün sayısına karşılık gelen *22* değerini başka bir yerde daha kullanmak istesek aynı sayıyı tekrar elle doğrudan kendimiz girmek zorundayız. Mesela yılda kaç gün çalıştığımızı hesaplayalım:

```
>>> 22 * 12

264
```

Gördüğünüz gibi, burada da *22* sayısına ihtiyaç duyduk. Aslında değerleri bu şekilde her defasında tekrar tekrar elle girmek hem hata yapma riskini artırdığı, hem de bize fazladan iş çıkardığı için tercih edilmeyen bir yöntemdir. Bunun yerine, *22* sayısına bir isim verip, gereken yerlerde bu ismi kullanmak daha mantıklı olacaktır. Yani tıpkı kullanıcı ve parola örneğinde olduğu gibi, burada da verileri öncelikle bir değişkene atamak çok daha akıllıca bir iştir:

```
>>> gün = 22
>>> gidiş_ücreti = 1.5
>>> dönüş_ücreti = 1.4
>>> gün * (gidiş_ücreti + dönüş_ücreti)

63.8
```

Bütün değerleri birer değişkene atadığımız için, artık bu değişkenleri istediğimiz yerde kullanabiliriz. Mesela yılda toplam kaç gün çalıştığımızı bulmak istersek, ilgili değeri elle yazmak yerine, yukarıda tanımladığımız *gün* değişkenini kullanabiliriz:

```
>>> gün * 12

264
```

İlerleyen zamanda aylık çalışılan gün sayısı değişirse sadece *gün* değişkeninin değerini değiştirmemiz yeterli olacaktır:

```
>>> gün = 23
>>> gün * (gidiş_ücreti + dönüş_ücreti)

66.7

>>> gün * 12

276
```

Eğer bu şekilde değişken atamak yerine, değerleri gerektiği her yerde elle yazsaydık, bu değerlerde herhangi bir değişiklik yapmamız gerektiğinde program içinde geçen ilgili bütün değerleri bulup tek tek değiştirmemiz gerekecekti:

```
>>> 23 * (1.6 + 1.5)

71.3

>>> 23 * 12

276
```

Değişken kavramı şu anda gözünüze pek anlamlı görünmemiş olabilir. Ama programlarımızı ilerde dosyaya kaydettiğimiz zaman bu değişkenler çok daha kullanışlı araçlar olarak karşımıza çıkacaktır.

Dilerseniz bir örnek daha yaparak yukarıdaki bilgilerin kafamıza iyice yerleşmesiniz sağlayalım. Mesela bir dairenin alanını (yaklaşık olarak) hesaplayan bir program yazalım.

Öncelikle *çap* adlı bir değişken tanımlayarak dairenin çapını belirleyelim:

```
>>> çap = 16
```

Bu değeri kullanarak dairemizin yarıçapını hesaplayabiliriz. Bunun için *çap* değişkeninin değerinin yarısını almamız yeterli olacaktır:

```
>>> yarıçap = çap / 2
```

pi sayısını *3.14159* olarak alalım.

```
>>> pi = 3.14159
```

Bir dairenin alan formülü $(pi)r^2$'dir:

```
>>> alan = pi * (yarıçap * yarıçap)
```

Son olarak *alan* değişkeninin değerini ekrana yazdırabiliriz:

```
>>> alan

201.06176
```

Böylece bir dairenin alanını yaklaşık olarak hesaplamış olduk. Dilerseniz programımızı bir de derli toplu olarak görelim:

```
>>> çap = 16
>>> yarıçap = çap / 2
>>> pi = 3.14159
>>> alan = pi * (yarıçap * yarıçap)
>>> alan

201.06176
```

Görüyorsunuz ya, değişkenler işimizi nasıl da kolaylaştırıyor. Eğer yukarıdaki programda değişken kullanmasaydık kodlarımız şöyle görünecekti:

```
>>> 3.14159 * ((16/2) * (16/2))

201.06176
```

Bu kodlar tek kullanımlıktır. Eğer yukarıdaki örnekte mesela dairenin çapını değiştirmeniz gerekirse, iki yerde elle değişiklik yapmanız gerekir. Ama değişkenleri kullandığımızda sadece *çap* değişkeninin değerini değiştirmeniz yeterli olacaktır. Ayrıca değişken kullanmadığınızda, ilgili değeri program boyunca aklınızda tutmanız gerekir. Örneğin *çap* değişkenini kullanmak yerine, gereken her yerde *16* değerini kullanacaksanız, bu *16* değerini sürekli aklınızda tutmanız lazım. Ama bu değeri en başta bir değişkene atarsanız, *16* değerini kullanmanız gereken yerlerde, akılda tutması daha kolay bir ifade olan *çap* ismini kullanabilirsiniz.

Bu arada yeri gelmişken sizi yeni bir işleçle daha tanıştıralım. Şimdiye kadar Python'da toplama (+), çıkarma (-), çarpma (*), bölme (/) ve değer atama (=) işleçlerini gördük. Ama yukarıda verdiğimiz son örnek, başka bir işleç daha öğrenmemizi gerektiriyor...

Yukarıdaki şu örneğe tekrar bakalım:

```
alan = pi * (yarıçap * yarıçap)
```

Burada *yarıçap* değişkeninin karesini alabilmek için bu değeri kendisiyle çarptık. Aslında gayet mantıklı ve makul bir yöntem. Kare bulmak için değeri kendisiyle çarpıyoruz. Eğer bir sayının küpünü bulmak isteseydik o sayıyı üç kez kendisiyle çarpacaktık:

```
>>> 3 * 3 * 3
```

```
27
```

Peki ya bir sayının mesela beşinci kuvvetini hesaplamak istersek ne yapacağız? O sayıyı beş kez kendisiyle mi çarpacağız? Bu ne kadar vasat bir yöntem, değil mi?

Elbette bir sayının herhangi bir kuvvetini hesaplamak için o sayıyı kendisiyle kuvvetince çarpmayacağız. Python'da bu tür 'kuvvet hesaplamaları' için ayrı bir işleç (ve fonksiyon) bulunur.

Öncelikle kuvvet hesaplarını yapmamızı sağlayan işleçten söz edelim.

Python'da ** adlı bir işleç bulunur. Bu işlecin görevi bir sayının kuvvetini hesaplamamızı sağlamaktır. Örneğin bir sayının *2.* kuvvetini, ya da başka bir deyişle karesini hesaplamak istersek şöyle bir kod yazabiliriz:

```
>>> 12 ** 2

144
```

Burada *12* sayısının *2.* kuvvetini, yani karesini hesapladık. Bu bilgiyi yukarıdaki formüle uygulayalım:

```
>>> alan = pi * (yarıçap ** 2)
```

Bu işleci herhangi bir sayının herhangi bir kuvvetini hesaplamak için kullanabiliriz elbette. Mesela *23* sayısının küpünü (yani *3.* kuvvetini) hesaplayalım:

```
>>> 23 ** 3

12167
```

Aynı işleçten, bir sayının karekökünü hesaplamak için de yararlanabilirsiniz. Neticede bir sayının *0.5*'inci kuvveti, o sayının karekökü'dür:

```
>>> 144 ** 0.5

12.0
```

Gördüğünüz gibi, kuvvet hesaplama işlemleri için bu işleç son derece kullanışlı bir araç vazifesi görüyor. Ama eğer istersek aynı iş için özel bir fonksiyondan da yararlanabiliriz. Bu fonksiyonun adı pow().

Peki bu fonksiyonu nasıl kullanacağız?

Daha önce öğrendiğimiz type() ve len() fonksiyonlarını nasıl kullanıyorsak pow() fonksiyonu da aynı şekilde kullanacağız.

type() ve len() fonksiyonlarını birtakım parametreler ile birlikte kullanıyorduk hatırlarsanız. Aynı şekilde pow() fonksiyonu da birtakım parametreler alır.

Daha önce öğrendiğimiz fonksiyonları tek bir parametre ile birlikte kullanmıştık. pow() fonksiyonu ise toplam üç farklı parametre alır. Ama genellikle bu fonksiyon yalnızca iki parametre ile kullanılır.

Bu fonksiyonu şöyle kullanıyoruz:

```
>>> pow(12, 2)

144

>>> pow(23, 3)

12167

>>> pow(144, 0.5)

12.0
```

Gördüğünüz gibi, pow() fonksiyonunun ilk parametresi asıl sayıyı, ikinci parametresi ise bu sayının hangi kuvvetini hesaplamak istediğimizi gösteriyor.

Bu arada, fonksiyonun parantezleri içinde belirttiğimiz parametreleri birbirinden virgül ile ayırdığımızı gözden kaçırmayın.

Dediğimiz gibi, pow() fonksiyonu, pek kullanılmayan üçüncü bir parametre daha alır. Bu fonksiyonun üçüncü parametresi şöyle kullanılır. Dikkatlice bakın:

```
>>> pow(16, 2, 2)

0
```

Bu komut şu anlama gelir:

> *16* sayısının *2*'nci kuvvetini hesapla ve çıkan sayıyı *2*'ye bölüp, bölme işleminden kalan sayıyı göster!

16 sayısının *2*. kuvveti *256* sayısıdır. *256* sayısını *2*'ye böldüğümüzde, bölme işleminin kalanı *0*'dır. Yani *256* sayısı *2*'ye tam bölünür...

Bir örnek daha verelim:

```
>>> pow(11, 3, 4)
```

```
3
```

Demek ki, *11* sayısının *3*. kuvveti olan *1331* sayısı *4*'e bölündüğünde, bölme işleminden kalan sayı *3* imiş...

Dediğimiz gibi, pow() fonksiyonu genellikle sadece iki parametre ile kullanılır. Üçüncü parametrenin kullanım alanı oldukça dardır.

Değişkenlere Dair Bazı İpuçları

Değişkenin ne demek olduğunu öğrendiğimize göre, değişkenlere dair bazı ufak ipuçları verebiliriz.

Aynı Değere Sahip Değişkenler Tanımlama

Şimdi size şöyle bir soru sormama izin verin: Acaba aynı değere sahip iki değişkeni nasıl tanımlayabiliriz? Yani mesela değeri *4* sayısı olan iki farklı değişkeni nasıl belirleyeceğiz?

Aklınıza şöyle bir çözüm gelmiş olabilir:

```
>>> a = 4
>>> b = 4
```

Böylece ikisi de *4* değerine sahip *a* ve *b* adlı iki farklı değişken tanımlamış olduk. Bu tamamen geçerli bir yöntemdir. Ancak Python'da bu işlemi yapmanın daha kolay bir yolu var. Bakalım:

```
>>> a = b = 4
```

Bu kodlar bir öncekiyle tamamen aynı işlevi görür. Yani her iki kod da *4* değerine sahip *a* ve *b* değişkenleri tanımlamamızı sağlar:

```
>>> a

4

>>> b

4
```

Bu bilgiyi kullanarak mesela bir yıl içindeki her bir ayın çektiği gün sayısını ay adlarına atayabilirsiniz:

```
>>> ocak = mart = mayıs = temmuz = ağustos = ekim = aralık = 31
>>> nisan = haziran = eylül = kasım = 30
>>> şubat = 28
```

Böylece bir çırpıda değeri *31* olan yedi adet değişken, değeri *30* olan dört adet değişken, değeri *28* olan bir adet değişken tanımlamış olduk. Bu değişkenlerin değerine nasıl ulaşacağınızı biliyorsunuz:

```
>>> ocak

31
>>> haziran

30

>>> şubat

28

>>> mayıs

31

>>> ekim

31

>>> eylül

30
```

Eğer Python'ın aynı anda birden fazla değişkene tek bir değer atama özelliği olmasaydı yukarıdaki kodları şöyle yazmamız gerekirdi:

```
>>> ocak = 31
>>> şubat = 28
>>> mart = 31
>>> nisan = 30
>>> mayıs = 31
>>> haziran = 30
>>> temmuz = 31
>>> ağustos = 31
>>> eylül = 30
>>> ekim = 31
>>> kasım = 30
>>> aralık = 31
```

Bu değişkenleri nasıl bir program içinde kullanacağınız tamamen sizin hayal gücünüze kalmış. Mesela bu değişkenleri kullanarak aylara göre doğalgaz faturasını hesaplayan bir program

yazabiliriz.

Hemen son gelen doğalgaz faturasını (örn. Mart ayı) elimize alıp inceliyoruz ve bu faturadan şu verileri elde ediyoruz:

Mart ayı doğalgaz faturasına göre sayaçtan ölçülen hacim *346* m^3. Demek ki bir ayda toplam *346* m^3 doğalgaz harcamışız.

Fatura tutarı *273.87* TL imiş. Yani *346* m^3 doğalgaz tüketmenin bedeli *273.87* TL. Buna göre değişkenlerimizi tanımlayalım:

```
>>> aylık_sarfiyat = 346
>>> fatura_tutarı = 273.87
```

Bu bilgiyi kullanarak doğalgazın birim fiyatını hesaplayabiliriz. Formülümüz şöyle olmalı:

```
>>> birim_fiyat = fatura_tutarı / aylık_sarfiyat

>>> birim_fiyat

0.7915317919075144
```

Demek ki doğalgazın m^3 fiyatı (vergilerle birlikte yaklaşık) *0.79* TL'ye karşılık geliyormuş.

Bu noktada günlük ortalama doğalgaz sarfiyatımızı da hesaplamamız gerekiyor:

```
>>> günlük_sarfiyat = aylık_sarfiyat / mart
>>> günlük_sarfiyat

11.161290322580646
```

Demek ki Mart ayında günlük ortalama *11* m^3 doğalgaz tüketmişiz.

Bütün bu bilgileri kullanarak Nisan ayında gelecek faturayı tahmin edebiliriz:

```
>>> nisan_faturası = birim_fiyat * günlük_sarfiyat * nisan
>>> nisan_faturası

265.03548387096777
```

Şubat ayı faturası ise şöyle olabilir:

```
>>> şubat_faturası = birim_fiyat * günlük_sarfiyat * şubat
>>> şubat_faturası
```

```
247.36645161290326
```

Burada farklı değişkenlerin değerini değiştirerek daha başka işlemler de yapabilirsiniz. Örneğin pratik olması açısından *günlük_sarfiyat* değişkeninin değerini *15* yaparak hesaplamalarınızı buna göre güncelleyebilirsiniz.

Gördüğünüz gibi, aynı anda birden fazla değişken tanımlayabilmek işlerimizi epey kolaylaştırıyor.

Değişkenlerle ilgili bir ipucu daha verelim...

Değişkenlerin Değerini Takas Etme

Diyelim ki, işyerinizdeki personelin unvanlarını tuttuğunuz bir veritabanı var elinizde. Bu veritabanında şuna benzer ilişkiler tanımlı:

```
>>> osman = "Araştırma Geliştirme Müdürü"
>>> mehmet = "Proje Sorumlusu"
```

İlerleyen zamanda işvereniniz sizden Osman ve Mehmet'in unvanlarını değiştirmenizi talep edebilir. Yani Osman'ı Proje Sorumlusu, Mehmet'i de Araştırma Geliştirme Müdürü yapmanızı isteyebilir sizden.

Patronunuzun bu isteğini Python'da çok rahat bir biçimde yerine getirebilirsiniz. Dikkatlice bakın:

```
>>> osman, mehmet = mehmet, osman
```

Böylece tek hamlede bu iki kişinin unvanlarını takas etmiş oldunuz. Gelin isterseniz değişkenlerin son durumuna bakalım:

```
>>> osman

'Proje Sorumlusu

>>> mehmet

'Araştırma Geliştirme Müdürü'
```

Gördüğünüz gibi, *osman* değişkeninin değerini *mehmet*'e; *mehmet* değişkeninin değerini ise *osman*'a başarıyla verebilmişiz.

Yukarıdaki yöntem Python'ın öteki diller üzerinde önemli bir üstünlüğüdür. Başka programlama dillerinde bu işlemi yapmak için geçici bir değişken tanımlamanız gerekir. Yani mesela:

```
>>> osman = "Araştırma Geliştirme Müdürü"
>>> mehmet = "Proje Sorumlusu"
```

Elimizdeki değerler bunlar. Biz şimdi Osman'ın değerini Mehmet'e; Mehmet'in değerini ise Osman'a aktaracağız. Bunun için öncelikle bir geçici değişken tanımlamalıyız:

```
>>> geçici = "Proje Sorumlusu"
```

Bu sayede *"Proje Sorumlusu"* değerini yedeklemiş olduk. Bu işlem sayesinde, takas sırasında bu değeri kaybetmeyeceğiz.

Şimdi Mehmet'in değerini Osman'a aktaralım:

```
>>> mehmet = osman
```

Şimdi elimizde iki tane Araştırma Geliştirme Müdürü olmuş oldu:

```
>>> mehmet

'Araştırma Geliştirme Müdürü'

>>> osman

'Araştırma Geliştirme Müdürü'
```

Gördüğünüz gibi, `mehmet = osman` kodunu kullanarak *mehmet* değişkeninin değerini *osman* değişkeninin değeriyle değiştirdiğimiz için *"Proje Sorumlusu"* değeri ortadan kayboldu. Ama biz önceden bu değeri *geçici* adlı değişkene atadığımız için bu değeri kaybetmemiş olduk. Şimdi Osman'a *geçici* değişkeni içinde tuttuğumuz *"Proje Sorumlusu"* değerini verebiliriz:

```
>>> osman = geçici
```

Böylece istediğimiz takas işlemini gerçekleştirmiş olduk. Son durumu kontrol edelim:

```
>>> osman

'Proje Sorumlusu

>>> mehmet

'Araştırma Geliştirme Müdürü'
```

Basit bir işlem için ne kadar büyük bir zaman kaybı, değil mi? Ama dediğimiz gibi, Python'da bu şekilde geçici bir değişken atamakla uğraşmamıza hiç gerek yok. Sadece şu formülü kullanarak değişkenlerin değerini takas edebiliriz:

```
a, b = b, a
```

Bu şekilde *a* değişkeninin değerini *b* değişkenine; *b* değişkeninin değerini ise *a* değerine vermiş

oluyoruz. Eğer bu işlemi geri alıp her şeyi eski haline döndürmek istersek, tahmin edebileceğiniz gibi yine aynı yöntemden yararlanabiliriz:

```
b, a = a, b
```

Böylece değişkenler konusunu da oldukça ayrıntılı bir şekilde incelemiş olduk. Ayrıca bu esnada len() ve pow() adlı iki yeni fonksiyon ile ** adlı bir işleç de öğrendik.

Hazır lafı geçmişken, len() fonksiyonunun bazı kısıtlamalarından söz edelim. Dediğimiz gibi, bu fonksiyonu kullanarak karakter dizileri içinde toplam kaç adet karakter bulunduğunu hesaplayabiliyoruz. Örneğin:

```
>>> kelime = "muvaffakiyet"
>>> len(kelime)

12
```

Yalnız bu len() fonksiyonunu sayıların uzunluğunu ölçmek için kullanamıyoruz:

```
>>> len(123456)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
```

Gördüğünüz gibi, len() fonksiyonu, şu ana kadar öğrendiğimiz veri tipleri arasında yalnızca karakter dizileri ile birlikte kullanılabiliyor. Bu fonksiyonu sayılarla birlikte kullanamıyoruz.

Bu bölümün başında, o anda elimizde bulunan bir verinin tipini bilmemizin çok önemli olduğunu ve Python'da bir verinin tipinin, o veri ile neler yapıp neler yapamayacağınızı belirlediğini söylediğimizi hatırlıyorsunuz, değil mi? İşte len() fonksiyonu bu duruma çok güzel bir örnektir.

len() fonksiyonu sayılarla birlikte kullanılamaz. Dolayısıyla eğer elinizdeki verinin bir sayı olduğunu bilmezseniz, bu sayıyı len() fonksiyonu ile birlikte kullanmaya çalışabilir ve bu şekilde programınızın hata vererek çökmesine yol açabilirsiniz.

Ayrıca daha önce de söylediğimiz gibi, len() fonksiyonunu doğru kullanabilmek için, bu fonksiyonun bize sayı değerli bir çıktı verdiğini de bilmemiz gerekir.

len() fonksiyonu ile ilgili bu durumu da bir kenara not ettikten sonra yolumuza kaldığımız yerden devam edelim.

Etkileşimli Kabuğun Hafızası

Bir önceki bölümde Python'ın etkileşimli kabuğunun nasıl kullanılacağına dair epey örnek verdik ve etkileşimli kabuk üzerinden Python'ın bazı temel araçlarına kısa bir giriş yaptık. Şimdi isterseniz yeri gelmişken Python'ın etkileşimli kabuğunun bir başka yeteneğinden daha söz edelim.

Etkileşimli kabukta _ adlı işaret (alt çizgi işareti), yapılan son işlemin veya girilen son öğenin değerini tutma işlevi görür. Yani:

```
>>> 2345 + 54355

56700
```

Eğer bu işlemin ardından _ komutunu verirsek şöyle bir çıktı alırız:

```
>>> _

56700
```

Gördüğünüz gibi, _ komutu son girilen öğeyi hafızasında tutuyor. Bu özellikten çeşitli şekillerde yararlanabilirsiniz:

```
>>> _ + 15

56715
```

Burada _ komutunun değeri bir önceki işlemin sonucu olan *56715* değeri olduğu için, _ komutuna *15* eklediğimizde *56715* değerini elde ediyoruz. _ komutunun değerini tekrar kontrol edelim:

```
>>> _

56715
```

Gördüğünüz gibi, _ komutunun değeri artık *56715* sayısıdır...

_ komutu yalnızca sayıları değil, karakter dizilerini de hafızasında tutabilir:

```
>>> "www"

'www'

>>> _

'www'

>>> _ + ".istihza.com"
```

Bu işaret öyle çok sık kullanılan bir araç değildir, ama zaman zaman işinizi epey kolaylaştırır. Yalnız, unutmamamız gereken şey, bu özelliğin sadece etkileşimli kabuk ortamında geçerli olmasıdır. _ komutunun etkileşimli kabuk ortamı dışında herhangi bir geçerliliği yoktur.

Aslında burada söylenecek daha çok şey var. Ama biz şimdilik bunları sonraki konulara bırakacağız. Zira bu bölümdeki amacımız size konuların her ayrıntısını vermekten ziyade, Python'a ısınmanızı sağlamaktır.

başa dön

print() Fonksiyonu

Geçen bölümde bir yandan Python'ın etkileşimli kabuğunu yakından tanıyıp bu vesileyle bazı önemli fonksiyon ve araçları öğrenirken, öbür yandan bu öğrendiklerimizi kullanarak örnek programlar yazdık. Gördüğünüz gibi, azıcık bir bilgiyle dahi az çok işe yarar programlar yazmak mümkün olabiliyor. Daha yararlı programlar yazabilmek için henüz öğrenmemiz gereken pek çok şey var. İşte bu bölümde, 'daha yararlı programlar yazmamızı' sağlayacak çok önemli bir araçtan söz edeceğiz. Öneminden dolayı ayrıntılı bir şekilde anlatacağımız bu aracın adı print() fonksiyonu.

Elbette bu bölümde sadece print() fonksiyonundan bahsetmeyeceğiz. Bu bölümde print() fonksiyonunun yanısıra Python'daki bazı önemli temel konuları da ele alacağız. Mesela bu bölümde Python'daki karakter dizilerine ve sayılara ilişkin çok önemli bilgiler vereceğiz. Ayrıca print() fonksiyonu vesilesiyle Python'daki 'fonksiyon' konusuna da sağlam bir giriş yapmış, bu kavram ile ilgili ilk bilgilerimizi almış olacağız. Sözün özü, bu bölüm bizim için, deyim yerindeyse, tam anlamıyla bir dönüm noktası olacak.

O halde isterseniz lafi daha fazla uzatmadan işe print() fonksiyonunun ne olduğu ve ne işe yaradığını anlatarak başlayalım.

Nedir, Ne İşe Yarar?

Şimdiye kadar etkileşimli kabukta gerek karakter dizilerini gerekse sayıları doğrudan ekrana yazdık. Yani şöyle bir şey yaptık:

```
>>> "Python programlama dili"

'Python programlama dili'

>>> 6567

6567
```

Etkileşimli kabuk da, ekrana yazdığımız bu karakter dizisi ve sayıyı doğrudan bize çıktı olarak verdi. Ancak ilerde Python kodlarımızı bir dosyaya kaydedip çalıştırdığımızda da göreceğiniz gibi, Python'ın ekrana çıktı verebilmesi için yukarıdaki kullanım yeterli değildir. Yani yukarıdaki kullanım yalnızca etkileşimli kabukta çalışır. Bu kodları bir dosyaya kaydedip çalıştırmak istediğimizde hiçbir çıktı alamayız. Python'da yazdığımız şeylerin ekrana çıktı olarak verilebilmesi için print() adlı özel bir fonksiyondan yararlanmamız gerekir.

O halde gelin bu print() fonksiyonunun ne işe yaradığını ve nasıl kullanıldığını anlamaya çalışalım:

print() de tıpkı daha önce gördüğümüz type(), len() ve pow() gibi bir fonksiyondur. Fonksiyonları ilerde daha ayrıntılı bir şekilde inceleyeceğimizi söylemiştik hatırlarsanız. O yüzden fonksiyon

kelimesine takılarak, burada anlattığımız şeylerin kafanızı karıştırmasına, moralinizi bozmasına izin vermeyin.

print() fonksiyonunun görevi ekrana çıktı vermemizi sağlamaktır. Hemen bununla ilgili bir örnek verelim:

```
>>> print("Python programlama dili")

Python programlama dili
```

Bildiğiniz gibi burada gördüğümüz *"Python programlama dili"* bir karakter dizisidir. İşte print() fonksiyonunun görevi bu karakter dizisini ekrana çıktı olarak vermektir. Peki bu karakter dizisini print() fonksiyonu olmadan yazdığımızda da ekrana çıktı vermiş olmuyor muyuz? Aslında olmuyoruz. Dediğimiz gibi, ilerde programlarımızı dosyalara kaydedip çalıştırdığımızda, başında print() olmayan ifadelerin çıktıda görünmediğine şahit olacaksınız.

Daha önce de dediğimiz gibi, etkileşimli kabuk bir test ortamı olması açısından rahat bir ortamdır. Bu sebeple bu ortamda ekrana çıktı verebilmek için print() fonksiyonunu kullanmak zorunda değilsiniz. Yani başında print() olsa da olmasa da etkileşimli kabuk ekrana yazdırmak istediğiniz şeyi yazdırır. Ama iyi bir alışkanlık olması açısından, ekrana herhangi bir şey yazdıracağınızda ben size print() fonksiyonunu kullanmanızı tavsiye ederim.

print() son derece güçlü bir fonksiyondur. Gelin isterseniz bu güçlü ve faydalı fonksiyonu derin derin incelemeye koyulalım.

Nasıl Kullanılır?

Yukarıda verdiğimiz örnekte ilk gözümüze çarpan şey, karakter dizisini print() fonksiyonunun parantezleri içine yazmış olmamızdır. Biz bir fonksiyonun parantezleri içinde belirtilen öğelere 'parametre' dendiğini geçen bölümde öğrenmiştik. Tıpkı öğrendiğimiz öteki fonksiyonlar gibi, print() fonksiyonu da birtakım parametreler alır.

Bu arada print() fonksiyonunun parantezini açıp parametreyi yazdıktan sonra, parantezi kapatmayı unutmuyoruz. Python programlama diline yeni başlayanların, hatta bazen deneyimli programcıların bile en sık yaptığı hatalardan biri açtıkları parantezi kapatmayı unutmalarıdır.

Elbette, eğer istersek burada doğrudan *"Python programlama dili"* adlı karakter dizisini kullanmak yerine, önce bu karakter dizisini bir değişkene atayıp, sonra da print() fonksiyonunun parantezleri içinde bu değişkeni kullanabiliriz. Yani:

```
>>> dil = "Python programlama dili"
>>> print(dil)

Python programlama dili
```

Bu arada, hem şimdi verdiğimiz, hem de daha önce yazdığımız örneklerde bir şey dikkatinizi çekmiş olmalı. Şimdiye kadar verdiğimiz örneklerde karakter dizilerini hep çift tırnakla gösterdik. Ama aslında tek seçeneğimiz çift tırnak değildir. Python bize üç farklı tırnak seçeneği sunar:

1. Tek tırnak (‘ ‘)
2. Çift tırnak (” ”)
3. Üç tırnak (“”” “”””)

Dolayısıyla yukarıdaki örneği üç farklı şekilde yazabiliriz:

```
>>> print('Python programlama dili')

Python programlama dili

>>> print("Python programlama dili")

Python programlama dili

>>> print("""Python programlama dili""")

Python programlama dili
```

Gördüğünüz gibi çıktılar arasında hiçbir fark yok.

Peki çıktılarda hiçbir fark yoksa neden üç farklı tırnak çeşidi var?

İsterseniz bu soruyu bir örnek üzerinden açıklamaya çalışalım. Diyelim ki ekrana şöyle bir çıktı vermek istiyoruz:

```
Python programlama dilinin adı "piton" yılanından gelmez
```

Eğer bu cümleyi çift tırnaklar içinde gösterirsek programımız hata verecektir:

```
>>> print("Python programlama dilinin adı "piton" yılanından gelmez")

File "<stdin>", line 1
  print("Python programlama dilinin adı "piton" yılanından gelmez")
                                         ^
SyntaxError: invalid syntax
```

Bunun sebebi, cümle içinde geçen ‘piton’ kelimesinin de çift tırnaklar içinde gösterilmiş olmasıdır. Cümlenin, yani karakter dizisinin kendisi de çift tırnak içinde gösterildiği için Python, karakter dizisini başlatan ve bitiren tırnakların hangisi olduğunu ayırt edemiyor. Yukarıdaki cümleyi en kolay şu şekilde ekrana yazdırabiliriz:

```
>>> print('Python programlama dilinin adı "piton" yılanından gelmez')
```

```
Python programlama dilinin adı "piton" yılanından gelmez
```

Burada karakter dizisini tek tırnak içine aldık. Karakter dizisi içinde geçen ‘piton’ kelimesi çift tırnak içinde olduğu için, karakter dizisini başlatıp bitiren tırnaklarla ‘piton’ kelimesindeki tırnakların birbirine karışması gibi bir durum söz konusu değildir.

Bir de şöyle bir örnek verelim: Diyelim ki aşağıdaki gibi bir çıktı elde etmek istiyoruz:

```
İstanbul'un 5 günlük hava durumu tahmini
```

Eğer bu karakter dizisini tek tırnak işaretleri içinde belirtirseniz Python size bir hata mesajı gösterecektir:

```
>>> print('İstanbul'un 5 günlük hava durumu tahmini')

File "<stdin>", line 1
  print('İstanbul'un 5 günlük hava durumu tahmini')
                 ^
SyntaxError: invalid syntax
```

Bu hatanın sebebi ‘İstanbul’un’ kelimesi içinde geçen kesme işaretidir. Tıpkı bir önceki örnekte olduğu gibi, Python karakter dizisini başlatan ve bitiren tırnakların hangisi olduğunu kestiremiyor. Python, karakter dizisinin en başındaki tek tırnak işaretinin ardından ‘İstanbul’un’ kelimesi içindeki kesme işaretini görünce karakter dizisinin burada sona erdiğini zannediyor. Ancak karakter dizisini soldan sağa doğru okumaya devam edince bir yerlerde bir terslik olduğunu düşünüyor ve bize bir hata mesajı göstermekten başka çaresi kalmıyor. Yukarıdaki karakter dizisini en kolay şöyle tanımlayabiliriz:

```
>>> print("İstanbul'un 5 günlük hava durumu tahmini")

İstanbul'un 5 günlük hava durumu tahmini
```

Burada da, karakter dizisi içinde geçen kesme işaretine takılmamak için karakter dizimizi çift tırnak işaretleri içine alıyoruz.

Yukarıdaki karakter dizilerini düzgün bir şekilde çıktı verebilmek için üç tırnak işaretlerinden de yararlanabiliriz:

```
>>> print("""Python programlama dilinin adı "piton" yılanından gelmez""")

Python programlama dilinin adı "piton" yılanından gelmez

>>> print("""İstanbul'un 5 günlük hava durumu tahmini""")
```

İstanbul'un 5 günlük hava durumu tahmini

Bütün bu örneklerden sonra kafanızda şöyle bir düşünce uyanmış olabilir:

> Görünüşe göre üç tırnak işaretiyle her türlü karakter dizisini hatasız bir şekilde ekrana çıktı olarak verebiliyoruz. O zaman ben en iyisi bütün karakter dizileri için üç tırnak işaretini kullanayım!

Elbette, eğer isterseniz **pek çok karakter dizisi için** üç tırnak işaretini kullanabilirsiniz. Ancak Python'da karakter dizileri tanımlanırken genellikle tek tırnak veya çift tırnak işaretleri kullanılır. Üç tırnak işaretlerinin asıl kullanım yeri ise farklıdır. Peki nedir bu üç tırnak işaretlerinin asıl kullanım yeri?

Üç tırnak işaretlerini her türlü karakter dizisiyle birlikte kullanabiliyor olsak da, bu tırnak tipi çoğunlukla sadece birden fazla satıra yayılmış karakter dizilerini tanımlamada kullanılır. Örneğin şöyle bir ekran çıktısı vermek istediğinizi düşünün:

```
[H]=========HARMAN========[-][o][x]
|                          |
|    Programa Hoşgeldiniz!       |
|          Sürüm 0.8             |
|   Devam etmek için herhangi    |
|      bir düğmeye basın.        |
|                          |
|==================================|
```

Böyle bir çıktı verebilmek için eğer tek veya çift tırnak kullanmaya kalkışırsanız epey eziyet çekersiniz. Bu tür bir çıktı vermenin en kolay yolu üç tırnakları kullanmaktır:

```
>>> print("""
... [H]=========HARMAN========[-][o][x]
... |                          |
... |    Programa Hoşgeldiniz!       |
... |          Sürüm 0.8             |
... |   Devam etmek için herhangi    |
... |      bir düğmeye basın.        |
... |                          |
... |==================================|
... """)
```

Burada bazı şeyler dikkatinizi çekmiş olmalı. Gördüğünüz gibi, üç tırnaklı yapı öteki tırnak tiplerine göre biraz farklı davranıyor. Şimdi şu örneğe bakın:

```
>>> print("""Game Over!
...
```

Buraya çok dikkatli bakın. Karakter dizisine üç tırnakla başladıktan sonra, kapanış tırnağını koymadan *Enter* tuşuna bastığımızda >>> işareti ... işaretine dönüştü. Python bu şekilde bize, 'yazmaya devam et!' demiş oluyor. Biz de buna uyarak yazmaya devam edelim:

```
>>> print("""Game Over!
... Insert Coin!""")

Game Over!
Insert Coin!
```

Kapanış tırnağı koyulmadan *Enter* tuşuna basıldığında >>> işaretinin ... işaretine dönüşmesi üç tırnağa özgü bir durumdur. Eğer aynı şeyi tek veya çift tırnaklarla yapmaya çalışırsanız programınız hata verir:

```
>>> print("Game Over!

File "<stdin>", line 1
  print("Game Over!
                ^
SyntaxError: EOL while scanning string literal
```

...veya:

```
>>> print('Game Over!

File "<stdin>", line 1
  print("Game Over!
                ^
SyntaxError: EOL while scanning string literal
```

Üç tırnak işaretlerinin tırnak kapanmadan *Enter* tuşuna basıldığında hata vermeme özelliği sayesinde, bu tırnak tipi özellikle birden fazla satıra yayılmış karakter dizilerinin gösterilmesi için birebirdir.

Gelin isterseniz üç tırnak kullanımına ilişkin bir örnek daha verelim:

```
>>> print("""Python programlama dili Guido Van Rossum
... adlı Hollandalı bir programcı tarafından 90'lı
... yılların başında geliştirilmeye başlanmıştır. Çoğu
... insan, isminin "Python" olmasına bakarak, bu programlama
... dilinin, adını piton yılanından aldığını düşünür.
... Ancak zannedildiğinin aksine bu programlama dilinin
... adı piton yılanından gelmez.""")

Python programlama dili Guido Van Rossum
adlı Hollandalı bir programcı tarafından 90'lı
```

```
yılların başında geliştirilmeye başlanmıştır. Çoğu
insan, isminin "Python" olmasına bakarak, bu programlama
dilinin, adını piton yılanından aldığını düşünür.
Ancak zannedildiğinin aksine bu programlama dilinin
dı piton yılanından gelmez.
```

Elbette eğer istersek bu metni önce bir değişkene atamayı da tercih edebiliriz:

```
>>> python_hakkinda = """Python programlama dili Guido Van Rossum
... adlı Hollandalı bir programcı tarafından 90'lı
... yılların başında geliştirilmeye başlanmıştır. Çoğu
... insan, isminin "Python" olmasına bakarak, bu programlama
... dilinin, adını piton yılanından aldığını düşünür.
... Ancak zannedildiğinin aksine bu programlama dilinin
... adı piton yılanından gelmez."""
>>> print(python_hakkinda)

Python programlama dili Guido Van Rossum
adlı Hollandalı bir programcı tarafından 90'lı
yılların başında geliştirilmeye başlanmıştır. Çoğu
insan, isminin "Python" olmasına bakarak, bu programlama
dilinin, adını piton yılanından aldığını düşünür.
Ancak zannedildiğinin aksine bu programlama dilinin
dı piton yılanından gelmez.
```

Siz yukarıdaki çıktıyı tek veya çift tırnak kullanarak nasıl ekrana yazdırabileceğinizi düşünedurun, biz önemli bir konuya geçiş yapalım!

Bir Fonksiyon Olarak print()

print() ifadesinin bir fonksiyon olduğunu söylemiştik hatırlarsanız. Dediğimiz gibi, fonksiyonlarla ilgili ayrıntılı açıklamaları ilerleyen derslerde vereceğiz. Ancak şimdi dilerseniz bundan sonra anlatacaklarımızı daha iyi kavrayabilmemiz için, fonksiyonlar hakkında bilmemiz gereken bazı temel şeyleri öğrenmeye çalışalım.

Gördüğünüz gibi, print() fonksiyonunu şöyle kullanıyoruz:

```
>>> print("Aramak istediğiniz kelimeyi yazın: ")
```

Burada print() bir fonksiyon, *"Aramak istediğiniz kelimeyi yazın:"* adlı karakter dizisi ise bu fonksiyonun parametresidir. Daha önce len() adlı başka bir fonksiyon daha öğrenmiştik hatırlarsanız. Onu da şöyle kullanıyorduk:

```
>>> len("elma")
```

Burada da len() bir fonksiyon, *"elma"* adlı karakter dizisi ise bu fonksiyonun parametresidir. Aslında biçim olarak print() ve len() fonksiyonlarının birbirinden hiçbir farkı olmadığını görüyorsunuz.

Daha önce söylediğimiz ve bu örneklerden de anladığımız gibi, bir fonksiyonun parantezleri içinde belirtilen öğelere parametre adı veriliyor. Mesela aşağıdaki örnekte print() fonksiyonunu tek bir parametre ile kullanıyoruz:

```
>>> print('En az 8 haneli bir parola belirleyin.')
```

print() fonksiyonu, tıpkı pow() fonksiyonu gibi, birden fazla parametre alabilir:

```
>>> print('Fırat', 'Özgül')

Fırat Özgül
```

Bu örnekte bizim için çıkarılacak çok dersler var. Bir defa burada print() fonksiyonunu iki farklı parametre ile birlikte kullandık. Bunlardan ilki *Fırat* adlı bir karakter dizisi, ikincisi ise *Özgül* adlı başka bir karakter dizisi. Python'ın bu iki karakter dizisini nasıl birleştirdiğine dikkat edin. print() fonksiyonu bu iki karakter dizisini çıktı olarak verirken aralarına da birer boşluk yerleştirdi. Ayrıca, geçen derste de vurguladığımız gibi, parametrelerin birbirinden virgül ile ayrıldığını da gözden kaçırmıyoruz.

Gelin bununla ilgili bir iki örnek daha verelim elimizin alışması için:

```
>>> print("Python", "Programlama", "Dili")

Python Programlama Dili

>>> print('Fırat', 'Özgül', 'Adana', 1980)

Fırat Özgül Adana 1980
```

Bu arada dikkatinizi önemli bir noktaya çekmek istiyorum. Yukarıdaki örneklerde bazen tek tırnak, bazen de çift tırnak kullandık. Daha önce de söylediğimiz gibi, hangi tırnak tipini kullandığımız önemli değildir. Python hangi tırnak tipini kullandığımızdan ziyade, tırnak kullanımında tutarlı olup olmadığımızla ilgilenir. Yani Python için önemli olan, karakter dizisini hangi tırnakla başlatmışsak, o tırnakla bitirmemizdir. Yani şu tip kullanımlar geçerli değildir:

```
>>> print("karakter dizisi')

>>> print('karakter dizisi")
```

Karakter dizisini tanımlamaya başlarken kullandığımız tırnak tipi ile karakter dizisini tanımlamayı bitirirken kullandığımız tırnak tipi birbirinden farklı olduğu için bu iki kullanım da hata verecektir.

print() Fonksiyonunun Parametreleri

Şimdiye kadar verdiğimiz örneklerde belki çok da belli olmuyordur, ama aslında print() fonksiyonu son derece güçlü bir araçtır. İşte şimdi biz bu fonksiyonun gücünü gözler önüne seren özelliklerini incelemeye başlayacağız. Bu bölümü dikkatle takip etmeniz, ilerde yapacağımız çalışmaları daha rahat anlayabilmeniz açısından büyük önem taşır.

sep

print() fonksiyonu ile ilgili olarak yukarıda verdiğimiz örnekleri incelediğimizde, bu fonksiyonun kendine özgü bir davranış şekli olduğunu görüyoruz. Mesela bir önceki bölümde verdiğimiz şu örneğe bakalım:

```
>>> print('Fırat', 'Özgül')

Fırat Özgül
```

Burada print() fonksiyonunu iki farklı parametre ile birlikte kullandık. Bu fonksiyon, kendisine verdiğimiz bu parametreleri belli bir düzene göre birbiriyle birleştirdi. Bu düzen gereğince print(), kendisine verilen parametreleri birleştirirken, parametreler arasına bir boşluk yerleştiriyor. Bunu daha net görmek için şöyle bir örnek daha verelim:

```
>>> print("Python", "PHP", "C++", "C", "Erlang")

Python PHP C++ C Erlang
```

Gördüğünüz gibi, print() fonksiyonu gerçekten de, kendisine verilen parametreleri birleştirirken, parametrelerin her biri arasına bir boşluk yerleştiriyor. Halbuki bu boşluğu biz talep etmedik! Python bize bu boşluğu eşantiyon olarak verdi. Çoğu durumda istediğimiz şey bu olacaktır, ama bazı durumlarda bu boşluğu istemeyebiliriz. Örneğin:

```
>>> print("http://", "www.", "istihza.", "com")

http:// www. istihza. com
```

Ya da boşluk karakteri yerine daha farklı bir karakter kullanmak istiyor da olabiliriz. Peki böyle bir durumda ne yapmamız gerekir?

İşte bu noktada bazı özel araçlardan yararlanarak print() fonksiyonunun öntanımlı davranış kalıpları üzerinde değişiklikler yapabiliriz.

Peki nedir print() fonksiyonunu özelleştirmemizi sağlayacak bu araçlar?

Hatırlarsanız, Python'da fonksiyonların parantezleri içindeki değerlere parametre adı verildiğini

söylemiştik. Mesela print() fonksiyonunu bir ya da daha fazla parametre ile birlikte kullanabileceğimizi biliyoruz:

```
>>> print("Mehmet", "Öz", "İstanbul", "Çamlıca", 156, "/", 45)

Mehmet Öz İstanbul Çamlıca 156 / 45
```

print() fonksiyonu içinde istediğimiz sayıda karakter dizisi ve/veya sayı değerli parametre kullanabiliriz.

Fonksiyonların bir de daha özel görünümlü parametreleri vardır. Mesela print() fonksiyonun *sep* adlı özel bir parametresi bulunur. Bu parametre print() fonksiyonunda görünmese bile her zaman oradadır. Yani diyelim ki şöyle bir kod yazdık:

```
>>> print("http://", "www.", "google.", "com")
```

Burada herhangi bir *sep* parametresi görmüyoruz. Ancak Python yukarıdaki kodu aslında şöyle algılar:

```
>>> print("http://", "www.", "google.", "com", sep=" ")
```

sep ifadesi, İngilizcede *separator* (ayırıcı, ayraç) kelimesinin kısaltmasıdır. Dolayısıyla print() fonksiyonundaki bu *sep* parametresi, ekrana basılacak öğeler arasına hangi karakterin yerleştirileceğini gösterir. Bu parametrenin öntanımlı değeri bir adet boşluk karakteridir (" "). Yani siz bu özel parametrenin değerini başka bir şeyle değiştirmezseniz, Python bu parametrenin değerini bir adet boşluk karakteri olarak alacak ve ekrana basılacak öğeleri birbirinden birer boşlukla ayıracaktır. Ancak eğer biz istersek bu *sep* parametresinin değerini değiştirebiliriz. Böylece Python, karakter dizilerini birleştirirken araya boşluk değil, bizim istediğimiz başka bir karakteri yerleştirebilir. Gelin şimdi bu parametrenin değerini nasıl değiştireceğimizi görelim:

```
>>> print("http://", "www.", "istihza.", "com", sep="")

http://www.istihza.com
```

Gördüğünüz gibi, karakter dizilerini başarıyla birleştirip, geçerli bir internet adresi elde ettik.

Burada yaptığımız şey aslında çok basit. Sadece *sep* parametresinin 'bir adet boşluk karakteri' olan öntanımlı değerini silip, yerine 'boş bir karakter dizisi' değerini yazdık. Bu iki kavramın birbirinden farklı olduğunu söylediğimizi hatırlıyorsunuz, değil mi?

Gelin bir örnek daha yapalım:

```
>>> print("T", "C", sep=".")

T.C
```

Burada Python'a şöyle bir emir vermiş olduk:

> *"T"* ve *"C"* karakter dizilerini birbiriyle birleştir! Bunu yaparken de bu karakter dizilerinin arasına nokta işareti yerleştir!

sep parametresinin öteki parametrelerden farkı her zaman ismiyle birlikte kullanılmasıdır. Zaten teknik olarak da bu tür parametrelere 'isimli parametreler' adı verilir. Örneğin:

```
>>> print("Adana", "Mersin", sep="-")

Adana-Mersin
```

Eğer burada *sep* parametresinin ismini belirtmeden, doğrudan parametrenin değerini yazarsak, bu değerin öteki parametrelerden hiçbir farkı kalmayacaktır:

```
>>> print("Adana", "Mersin", "-")

Adana Mersin -
```

Gelin isterseniz bu parametreyle ilgili bir örnek daha yapalım:

'Bir mumdur iki mumdur...' diye başlayan türküyü biliyorsunuzdur. Şimdi bu türküyü Python'la nasıl yazabileceğimizi görelim!

```
>>> print("bir", "iki", "üç", "dört", "on dört", sep="mumdur")

birmumdurikimumdurüçmumdurdörtmumduron dört
```

Burada bir terslik olduğu açık! Karakter dizileri birbirlerine sıkışık düzende birleştirildi. Bunların arasında birer boşluk olsa tabii daha iyi olurdu. Ancak biliyorsunuz *sep* parametresinin öntanımlı değerini silip, yerine *"mumdur"* değerini yerleştirdiğimiz için, Python'ın otomatik olarak yerleştirdiği boşluk karakteri kayboldu. Ama eğer istersek o boşluk karakterlerini kendimiz de ayarlayabiliriz:

```
>>> print("bir", "iki", "üç", "dört", "on dört", sep=" mumdur ")

bir mumdur iki mumdur üç mumdur dört mumdur on dört
```

Gördüğünüz gibi, *sep* parametresine verdiğimiz *"mumdur"* değerinin sağında ve solunda birer boşluk bırakarak sorunumuzu çözebildik. Bu sorunu çözmenin başka bir yolu daha var. Hatırlarsanız etkileşimli kabukta ilk örneklerimizi verirken karakter dizilerini birleştirmek için + işaretinden de yararlanabileceğimizi söylemiştik. Dolayısıyla *sep* parametresini şöyle de yazabiliriz:

```
>>> print("bir", "iki", "üç", "dört", "on dört", sep=" " + "mumdur" + " ")
```

Burada da, *"mumdur"* adlı karakter dizisinin başında ve sonunda birer boşluk bırakmak yerine,

gerekli boşlukları + işareti yardımıyla bu karakter dizisine birleştirdik. Hatta istersek + işlecini kullanmak zorunda olmadığımızı dahi biliyorsunuz:

```
>>> print("bir", "iki", "üç", "dört", "on dört", sep=" " "mumdur" " ")
```

Ama gördüğünüz gibi bir problemimiz daha var. Türkünün sözleri şu şekilde olmalıydı:

> bir mumdur iki mumdur üç mumdur dört mumdur on dört mumdur

Ama sondaki 'mumdur' kelimesi yukarıdaki çıktıda yok. Normal olan da bu aslında. *sep* parametresi, karakter dizilerinin **arasına** bir değer yerleştirir. Karakter dizilerinin son tarafıyla ilgilenmez. Bu iş için print() fonksiyonu başka bir parametreye sahiptir.

Bu arada, yukarıdaki örneklerde hep karakter dizilerini kullanmış olmamız sizi yanıltmasın. *sep* parametresi yalnızca karakter dizilerinin değil sayıların arasına da istediğiniz bir değerin yerleştirilmesini sağlayabilir. Mesela:

```
>>> print(1, 2, 3, 4, 5, sep="-")

1-2-3-4-5
```

Ancak *sep* parametresinin değeri bir karakter dizisi (veya 'hiçbir şey' anlamına gelen None) olmalıdır:

```
>>> print(1, 2, 3, 4, 5, sep=0)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sep must be None or a string, not int
```

Gördüğünüz gibi, *sep* parametresine bir sayı olan *0* değerini veremiyoruz.

print() fonksiyonunun *sep* parametresini bütün ayrıntılarıyla incelediğimize göre, bu fonksiyonun bir başka özel parametresinden söz edebiliriz.

end

Bir önceki bölümde şöyle bir laf etmiştik:

> print() fonksiyonun *sep* adlı özel bir parametresi bulunur. Bu parametre print() fonksiyonunda görünmese bile her zaman oradadır.

Aynı bu şekilde, print() fonksiyonunun *end* adlı özel bir parametresi daha bulunur. Tıpkı *sep* parametresi gibi, *end* parametresi de print() fonksiyonunda görünmese bile her zaman oradadır.

Bildiğiniz gibi, *sep* parametresi print() fonksiyonuna verilen parametreler birleştirilirken araya hangi karakterin gireceğini belirliyordu. *end* parametresi ise bu parametrelerin sonuna neyin geleceğini belirler.

print() fonksiyonu öntanımlı olarak, parametrelerin sonuna 'satır başı karakteri' ekler. Peki bu satır başı karakteri denen şey de ne oluyor?

Dilerseniz bunu bir örnek üzerinde görelim.

Şöyle bir kodumuz olsun:

```
>>> print("Pardus ve Ubuntu birer GNU/Linux dağıtımıdır.")
```

Bu kodu yazıp *Enter* tuşuna bastığımız anda print() fonksiyonu iki farklı işlem gerçekleştirir:

1. Öncelikle karakter dizisini ekrana yazdırır.
2. Ardından bir alt satıra geçip bize >>> işaretini gösterir.

İşte bu ikinci işlem, karakter dizisinin sonunda bir adet satır başı karakteri olmasından, daha doğrusu print() fonksiyonunun, satır başı karakterini karakter dizisinin sonuna eklemesinden kaynaklanır. Bu açıklama biraz kafa karıştırıcı gelmiş olabilir. O halde biraz daha açıklayalım. Şu örneğe bakın:

```
>>> print("Pardus\nUbuntu")

Pardus
Ubuntu
```

Burada *"Pardus"* ve *"Ubuntu"* karakter dizilerinin tam ortasında çok özel bir karakter dizisi daha görüyorsunuz. Bu karakter dizisi şudur: *\n*. İşte bu özel karakter dizisine satır başı karakteri (*newline*) adı verilir. Bu karakterin görevi, karakter dizisini, bulunduğu noktadan bölüp, karakter dizisinin geri kalanını bir alt satıra geçirmektir. Zaten çıktıda da bu işlevi yerine getirdiğini görüyorsunuz. Karakter dizisi *"Pardus"* kısmından sonra ikiye bölünüyor ve bu karakter dizisinin geri kalan kısmı olan *"Ubuntu"* karakter dizisi bir alt satıra yazdırılıyor. Bunu daha iyi anlamak için bir örnek daha verelim:

```
>>> print("birinci satır\nikinci satır\nüçüncü satır")

birinci satır
ikinci satır
üçüncü satır
```

Peki size bir soru sorayım: Acaba yukarıdaki kodları daha verimli bir şekilde nasıl yazabiliriz?

Evet, doğru tahmin ettiniz... Tabii ki *sep* parametresini kullanarak:

```
>>> print("birinci satır", "ikinci satır", "üçüncü satır", sep="\n")
```

```
birinci satır
ikinci satır
üçüncü satır
```

Burada yaptığımız şey çok basit. *sep* parametresinin değerini \n, yani yeni satır karakteri olarak değiştirdik. Böylece karakter dizileri arasına birer \n karakteri yerleştirerek her bir karakter dizisinin farklı satıra yazdırılmasını sağladık.

İşte *end* parametresinin öntanımlı değeri de bu \n karakteridir ve bu parametre print() fonksiyonunda görünmese bile her zaman oradadır.

Yani diyelim ki şöyle bir kod yazdık:

```
>>> print("Bugün günlerden Salı")
```

Burada herhangi bir *end* parametresi görmüyoruz. Ancak Python yukarıdaki kodu aslında şöyle algılar:

```
>>> print("Bugün günlerden Salı", end="\n")
```

Biraz önce de dediğimiz gibi, bu kodu yazıp *Enter* tuşuna bastığımız anda print() fonksiyonu iki farklı işlem gerçekleştirir:

1. Öncelikle karakter dizisini ekrana yazdırır.
2. Ardından bir alt satıra geçip bize >>> işaretini gösterir.

Bunun ne demek olduğunu anlamak için *end* parametresinin değerini değiştirmemiz yeterli olacaktır:

```
>>> print("Bugün günlerden Salı", end=".")

Bugün günlerden Salı.>>>
```

Gördüğünüz gibi, *end* parametresinin öntanımlı değeri olan \n karakterini silip yerine . (nokta) işareti koyduğumuz için, komutu yazıp *Enter* tuşuna bastığımızda print() fonksiyonu satır başına geçmedi. Yeni satıra geçebilmek için *Enter* tuşuna kendimiz basmalıyız. Elbette, eğer yukarıdaki kodları şöyle yazarsanız, print() fonksiyonu hem karakter dizisinin sonuna nokta ekleyecek, hem de satır başına geçecektir:

```
>>> print("Bugün günlerden Salı", end=".\n")

Bugün günlerden Salı.
```

Şimdi bu öğrendiklerimizi türkümüze uygulayalım:

```
>>> print("bir", "iki", "üç", "dört", "on dört",
```

```
... sep=" mumdur ", end=" mumdur\n")
```

Not

Burada kodlarımızın sağa doğru çirkin bir şekilde uzamasını engellemek için *"on dört"* karakter dizisini yazıp virgülü koyduktan sonra *Enter* tuşuna basarak bir alt satıra geçtik. Bir alt satıra geçtiğimizde >>> işaretinin ... işaretine dönüştüğüne dikkat edin. Python'da doğru kod yazmak kadar, yazdığımız kodların düzgün görünmesi de önemlidir. O yüzden yazdığımız her bir kod satırının mümkün olduğunca 79 karakteri geçmemesini sağlamalıyız. Eğer yazdığınız bir satır 79 karakteri aşıyorsa, aşan kısmı yukarıda gösterdiğimiz şekilde alt satıra alabilirsiniz.

end parametresi de, tıpkı *sep* parametresi gibi, her zaman ismiyle birlikte kullanılması gereken bir parametredir. Yani eğer *end* parametresinin ismini belirtmeden sadece değerini kullanmaya çalışırsak Python ne yapmaya çalıştığımızı anlayamaz.

file

Not

Burada henüz öğrenmediğimiz bazı şeyler göreceksiniz. Hiç endişe etmeyin. Bunları ilerde bütün ayrıntılarıyla öğreneceğiz. Şimdilik konu hakkında biraz olsun fikir sahibi olmanızı sağlayabilirsek kendimizi başarılı sayacağız.

print() fonksiyonunun *sep* ve *end* dışında üçüncü bir özel parametresi daha bulunur. Bu parametrenin adı *file*'dır. Görevi ise, print() fonksiyonuna verilen karakter dizisi ve/veya sayıların, yani parametrelerin nereye yazılacağını belirtmektir.

Bu parametrenin öntanımlı değeri *sys.stdout*'tur. Peki bu ne anlama geliyor? *sys.stdout*, 'standart çıktı konumu' anlamına gelir. Peki 'standart çıktı konumu' ne demek?

Standart çıktı konumu; bir programın, ürettiği çıktıları verdiği yerdir. Aslında bu kavramın ne demek olduğu adından da anlaşılıyor:

standart çıktı konumu = çıktıların standart olarak verildiği konum.

Mesela Python öntanımlı olarak, ürettiği çıktıları ekrana verir. Eğer o anda etkileşimli kabukta çalışıyorsanız, Python ürettiği çıktıları etkileşimli kabuk üzerinde gösterir. Eğer yazdığınız bir programı komut satırında çalıştırıyorsanız, üretilen çıktılar komut satırında görünür. Dolayısıyla Python'ın standart çıktı konumu etkileşimli kabuk veya komut satırıdır. Yani print() fonksiyonu yardımıyla bastığınız çıktılar etkileşimli kabukta ya da komut satırında görünecektir.

Şimdi bu konuyu daha iyi anlayabilmek için birkaç örnek yapalım.

Normal şartlar altında print() fonksiyonunun çıktısını etkileşimli kabukta görürüz:

```
>>> print("Ben Python, Monty Python!")

Ben Python, Monty Python!
```

Ama eğer istersek print() fonksiyonunun, çıktılarını ekrana değil, bir dosyaya yazdırmasını da sağlayabiliriz. Mesela biz şimdi print() fonksiyonunun *deneme.txt* adlı bir dosyaya çıktı vermesini sağlayalım.

Bunun için sırasıyla şu kodları yazalım:

```
>>> dosya = open("deneme.txt", "w")
>>> print("Ben Python, Monty Python!", file=dosya)
>>> dosya.close()
```

Herhangi bir çıktı almadınız, değil mi? Evet. Çünkü yazdığımız bu kodlar sayesinde print() fonksiyonu, çıktılarını *deneme.txt* adlı bir dosyaya yazdırdı.

Gelin isterseniz yukarıdaki kodları satır satır inceleyelim:

1. Öncelikle *deneme.txt* adlı bir dosya oluşturduk ve bu dosyayı *dosya* adlı bir değişkene atadık. Burada kullandığımız open() fonksiyonuna çok takılmayın. Bunu birkaç bölüm sonra inceleyeceğiz. Biz şimdilik bu şekilde dosya oluşturulduğunu bilelim yeter. Bu arada open fonksiyonunun da biçim olarak type(), len(), pow() ve print() fonksiyonlarına ne kadar benzediğine dikkat edin. Gördüğünüz gibi open() fonksiyonu da tıpkı type(), len(), pow() ve print() fonksiyonları gibi birtakım parametreler alıyor. Bu fonksiyonun ilk parametresi *"deneme.txt"* adlı bir karakter dizisi. İşte bu karakter dizisi bizim oluşturmak istediğimiz dosyanın adını gösteriyor. İkinci parametre ise *"w"* adlı başka bir karakter dizisi. Bu da *deneme.txt* dosyasının yazma kipinde (modunda) açılacağını gösteriyor. Ama dediğim gibi, siz şimdilik bu ayrıntılara fazla takılmayın. İlerleyen derslerde, bu konuları adınızı bilir gibi bileceğinizden emin olabilirsiniz.

2. Oluşturduğumuz bu *deneme.txt* adlı dosya, o anda bulunduğunuz dizin içinde oluşacaktır. Bu dizinin hangisi olduğunu öğrenmek için şu komutları verebilirsiniz:

```
>>> import os
>>> os.getcwd()
```

Bu komutun çıktısında hangi dizinin adı görünüyorsa, *deneme.txt* dosyası da o dizinin içindedir. Mesela bendeki çıktı */home/istihza/Desktop*. Demek ki oluşturduğum *deneme.txt* adlı dosya masaüstündeymiş. Ben bu komutları Ubuntu üzerinde verdim. Eğer Windows üzerinde verseydim şuna benzer bir çıktı alacaktım: *C:\Users\falanca\Desktop*

3. Ardından da normal bir şekilde print() fonksiyonumuzu çalıştırdık. Ama gördüğünüz gibi print() fonksiyonu bize herhangi bir çıktı vermedi. Çünkü, daha önce de söylediğimiz gibi, print()

fonksiyonunu biz ekrana değil, dosyaya çıktı verecek şekilde ayarladık. Burada *file* adlı bir parametreye, biraz önce tanımladığımız *dosya* değişkenini yazdığımıza özellikle dikkat ediyoruz.

4. Son komut yardımıyla da, yaptığımız değişikliklerin dosyada görünebilmesi için ilk başta açtığımız dosyayı kapatıyoruz.

Şimdi *deneme.txt* adlı dosyayı açın. Biraz önce print() fonksiyonuyla yazdırdığımız *"Ben Python, Monty Python!"* karakter dizisinin dosyaya işlenmiş olduğunu göreceksiniz.

Böylece print() fonksiyonunun standart çıktı konumunu değiştirmiş olduk. Yani print() fonksiyonunun *file* adlı parametresine farklı bir değer vererek, print() fonksiyonunun etkileşimli kabuğa değil dosyaya yazmasını sağladık.

Tıpkı *sep* ve *end* parametreleri gibi, *file* parametresi de, siz görmeseniz bile her zaman print() fonksiyonunun içinde vardır. Yani diyelim ki şöyle bir komut verdik:

```
>>> print("Tahir olmak da ayıp değil", "Zühre olmak da")
```

Python bu komutu şöyle algılar:

```
>>> print("Tahir olmak da ayıp değil", "Zühre olmak da",
... sep=" ", end="\n", file=sys.stdout)
```

Yani kendisine parametre olarak verilen değerleri ekrana yazdırırken sırasıyla şu işlemleri gerçekleştirir:

1. Parametrelerin arasına birer boşluk koyar (sep=" "),
2. Ekrana yazdırma işlemi bittikten sonra parametrelerin sonuna satır başı karakteri ekler (end="\n")
3. Bu çıktıyı standart çıktı konumuna gönderir (file=sys.stdout).

İşte biz burada *file* parametresinin değeri olan standart çıktı konumuna başka bir değer vererek bu konumu değiştiriyoruz.

Gelin isterseniz bununla ilgili bir örnek daha yapalım. Mesela kişisel bilgilerimizi bir dosyaya kaydedelim. Öncelikle bilgileri kaydedeceğimiz dosyayı oluşturalım:

```
>>> f = open("kişisel_bilgiler.txt", "w")
```

Bu kodlarla, *kişisel_bilgiler.txt* adını taşıyan bir dosyayı yazma kipinde (*w*) açmış ve bu dosyayı *f* adlı bir değişkene atamış olduk. Şimdi bilgileri yazmaya başlayabiliriz:

```
>>> print("Fırat Özgül", file=f)
>>> print("Adana", file=f)
>>> print("Ubuntu", file=f)
```

İşimiz bittiğinde dosyayı kapatmayı unutmuyoruz. Böylece bütün bilgiler dosyaya yazılmış oluyor:

```
>>> f.close()
```

Oluşturduğumuz *kişisel_bilgiler.txt* adlı dosyayı açtığımızda, print() fonksiyonuna verdiğimiz parametrelerin dosyaya yazdırıldığını görüyoruz.

En başta da söylediğim gibi, bu bölümde henüz öğrenmediğimiz bazı şeylerle karşılaştık. Eğer yukarıda verilen örnekleri anlamakta zorlandıysanız hiç endişe etmenize gerek yok. Birkaç bölüm sonra burada anlattığımız şeyler size çocuk oyuncağı gibi gelecek...

flush

Şimdiye kadar print() fonksiyonunun *sep*, *end* ve *file* adlı özel birtakım parametreleri olduğunu öğrendik. print() fonksiyonunun bunların dışında başka bir özel parametresi daha bulunur. Bu parametrenin adı *flush*. İşte şimdi biz print() fonksiyonunun bu *flush* adlı parametresinden söz edeceğiz.

Bildiğiniz gibi, print() gibi bir komut verdiğimizde Python, yazdırmak istediğimiz bilgiyi standart çıktı konumuna gönderir. Ancak Python'da bazı işlemler standart çıktı konumuna gönderilmeden önce bir süre tamponda bekletilir ve daha sonra bekleyen bu işlemler topluca standart çıktı konumuna gönderilir. Peki ilk başta çok karmaşıkmış gibi görünen bu ifade ne anlama geliyor?

Aslında siz bu olguya hiç yabancı değilsiniz. *file* parametresini anlatırken verdiğimiz şu örneği tekrar ele alalım:

```
>>> f = open("kişisel_bilgiler.txt", "w")
```

Bu komutla *kişisel_bilgiler.txt* adlı bir dosyayı yazma kipinde açtık. Şimdi bu dosyaya bazı bilgiler ekleyelim:

```
>>> print("Fırat Özgül", file=f)
```

Bu komutla *kişisel_bilgiler.txt* adlı dosyaya 'Fırat Özgül' diye bir satır eklemiş olduk.

Şimdi bilgisayarınızda oluşan bu *kişisel_bilgiler.txt* dosyasını açın. Gördüğünüz gibi dosyada hiçbir bilgi yok. Dosya şu anda boş görünüyor. Halbuki biz biraz önce bu dosyaya 'Fırat Özgül' diye bir satır eklemiştik, değil mi?

Python bizim bu dosyaya eklemek istediğimiz satırı tampona kaydetti. Dosyaya yazma işlemleri sona erdiğinde ise Python, tamponda bekleyen bütün bilgileri standart çıktı konumuna (yani bizim durumumuzda *f* adlı değişkenin tuttuğu *kişisel_bilgiler.txt* adlı dosyaya) boşaltacak.

Dosyaya başka bilgiler de yazalım:

```
>>> print("Adana", file=f)
```

```
>>> print("Ubuntu", file=f)
```

Dosyaya yazacağımız şeyler bu kadar. Artık yazma işleminin sona erdiğini Python'a bildirmek için şu komutu veriyoruz:

```
>>> f.close()
```

Böylece dosyamızı kapatmış olduk. Şimdi *kişisel_bilgiler.txt* adlı dosyaya çift tıklayarak dosyayı tekrar açın. Orada 'Fırat Özgül', 'Adana' ve 'Ubuntu' satırlarını göreceksiniz.

Gördüğünüz gibi, gerçekten de Python dosyaya yazdırmak istediğimiz bütün verileri önce tamponda bekletti, daha sonra dosya kapatılınca tamponda bekleyen bütün verileri dosyaya boşalttı. İşte *flush* parametresi ile, bahsettiğimiz bu boşaltma işlemini kontrol edebilirsiniz. Şimdi dikkatlice inceleyin:

```
>>> f = open("kişisel_bilgiler.txt", "w")
```

Dosyamızı oluşturduk. Şimdi bu dosyaya bazı bilgiler ekleyelim:

```
>>> print("Merhaba Dünya!", file=f, flush=True)
```

Gördüğünüz gibi, burada *flush* adlı yeni bir parametre kullandık. Bu parametreye verdiğimiz değer *True*. Şimdi dosyaya çift tıklayarak dosyayı açın. Gördüğünüz gibi, henüz dosyayı kapatmadığımız halde bilgiler dosyaya yazıldı. Bu durum, tahmin edebileceğiniz gibi, *flush* parametresine *True* değeri vermemiz sayesindedir. Bu parametre iki değer alabilir: *True* ve *False*. Bu parametrenin öntanımlı değeri *False*'tur. Yani eğer biz bu parametreye herhangi bir değer belirtmezsek Python bu parametrenin değerini *False* olarak kabul edecek ve bilgilerin dosyaya yazılması için dosyanın kapatılmasını bekleyecektir. Ancak bu parametreye *True* değerini verdiğimizde ise veriler tamponda bekletilmeksizin standart çıktı konumuna gönderilecektir.

Yazdığınız bir programda, yapmak istediğiniz işin niteliğine göre, bir dosyaya yazmak istediğiniz bilgilerin bir süre tamponda bekletilmesini veya hiç bekletilmeden doğrudan dosyaya yazılmasını isteyebilirsiniz. İhtiyacınıza bağlı olarak da *flush* parametresinin değerini *True* veya *False* olarak belirleyebilirsiniz.

Birkaç Pratik Bilgi

Buraya gelene kadar print() fonksiyonu ve bu fonksiyonun parametreleri hakkında epey söz söyledik. Dilerseniz şimdi de, programcılık maceranızda işinize yarayacak, işlerinizi kolaylaştıracak bazı ipuçları verelim.

Yıldızlı Parametreler

Şimdi size şöyle bir soru sormama izin verin: Acaba aşağıdaki gibi bir çıktıyı nasıl elde ederiz?

```
L.i.n.u.x
```

Aklınıza hemen şöyle bir cevap gelmiş olabilir:

```
>>> print("L", "i", "n", "u", "x", sep=".")

L.i.n.u.x
```

Yukarıdaki, gerçekten de doğru bir çözümdür. Ancak bu soruyu çözmenin çok daha basit bir yolu var. Şimdi dikkatle bakın:

```
>>> print(*"Linux", sep=".")

L.i.n.u.x
```

Konuyu açıklamaya geçmeden önce bir örnek daha verelim:

```
>>> print(*"Galatasaray")

G a l a t a s a r a y
```

Burada neler döndüğünü az çok tahmin ettiğinizi zannediyorum. Son örnekte de gördüğünüz gibi, *"Galatasaray"* karakter dizisinin başına eklediğimiz yıldız işareti; *"Galatasaray"* karakter dizisinin herbir öğesini parçalarına ayırarak, bunları tek tek print() fonksiyonuna yolluyor. Yani sanki print() fonksiyonunu şöyle yazmışız gibi oluyor:

```
>>> print("G", "a", "l", "a", "t", "a", "s", "a", "r", "a", "y")

G a l a t a s a r a y
```

Dediğimiz gibi, bir fonksiyona parametre olarak verdiğimiz bir karakter dizisinin başına eklediğimiz yıldız işareti, bu karakter dizisini tek tek öğelerine ayırıp, bu öğeleri yine tek tek ve sanki herbir öğe ayrı bir parametreymiş gibi o fonksiyona gönderdiği için doğal olarak yıldız işaretini ancak, birden fazla parametre alabilen fonksiyonlara uygulayabiliriz.

Örneğin len() fonksiyonu sadece tek bir parametre alabilir:

```
>>> len("Galatasaray")

11
```

Bu fonksiyonu birden fazla parametre ile kullanamayız:

```
>>> len("Galatasaray", "Fenerbahçe", "Beşiktaş")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: len() takes exactly one argument (3 given)
```

Hata mesajında da söylendiği gibi, len() fonksiyonu yalnızca tek bir parametre alabilirken, biz *3* parametre vermeye çalışmışız...

Dolayısıyla yıldızlı parametreleri len() fonksiyonuna uygulayamayız:

```
>>> len(*"Galatasaray")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: len() takes exactly one argument (11 given)
```

Bir parametrenin başına yıldız eklediğimizde, o parametreyi oluşturan bütün öğeler tek tek fonksiyona gönderildiği için, sanki len() fonksiyonuna *1* değil de, *11* ayrı parametre vermişiz gibi bir sonuç ortaya çıkıyor.

Yıldızlı parametreleri bir fonksiyona uygulayabilmemiz için o fonksiyonun birden fazla parametre alabilmesinin yanısıra, yapısının da yıldızlı parametre almaya uygun olması gerekir. Mesela open(), type() ve biraz önce bahsettiğimiz len() fonksiyonlarının yapısı yıldızlı parametre almaya uygun değildir. Dolayısıyla yıldızlı parametreleri her fonksiyonla birlikte kullanamayız, ama print() fonksiyonu yıldızlı parametreler için son derece uygun bir fonksiyondur:

```
>>> print(*"Galatasaray")

G a l a t a s a r a y

>>> print(*"TBMM", sep=".")

T.B.M.M

>>> print(*"abcçdefgğh", sep="/")

a/b/c/ç/d/e/f/g/ğ/h
```

Bu örneklerden de gördüğünüz gibi, print() fonksiyonuna verdiğimiz bir parametrenin başına yıldız eklediğimizde, o parametre tek tek parçalarına ayrılıp print() fonksiyonuna gönderildiği için, sonuç olarak *sep* parametresinin karakter dizisi öğelerine tek tek uygulanmasını sağlamış oluyoruz.

Hatırlarsanız *sep* parametresinin öntanımlı değerinin bir adet boşluk karakteri olduğunu söylemiştik. Yani aslında Python yukarıdaki ilk komutu şöyle görüyor:

```
>>> print(*"Galatasaray", sep=" ")
```

Dolayısıyla, yıldız işareti sayesinde *"Galatasaray"* adlı karakter dizisinin herbir öğesinin arasına bir adet boşluk karakteri yerleştiriliyor. Bir sonraki *"TBMM"* karakter dizisinde ise, *sep* parametresinin değerini nokta işareti olarak değiştirdiğimiz için *"TBMM"* karakter dizisinin her bir öğesinin arasına bir adet nokta işareti yerleştiriliyor. Aynı şekilde *"abcçdefgğh"* karakter dizisinin herbir öğesini tek tek print() fonksiyonuna yollayarak, *sep* parametresine verdiğimiz / işareti yardımıyla her öğenin arasına bu / işaretini yerleştirebiliyoruz.

Bu arada yıldızlı parametrelerin sayılarla birlikte kullanılamayacağına dikkat ediyoruz:

```
>>> print(*2345)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: print() argument after * must be a sequence, not int
```

Çünkü yıldızlı parametreler ancak ve ancak dizi özelliği taşıyan veri tipleriyle birlikte kullanılabilir. Mesela karakter dizileri bu türden bir veri tipidir. İlerde dizi özelliği taşıyan ve bu sayede yıldızlı parametrelerle birlikte kullanılabilecek başka veri tiplerini de öğreneceğiz.

Yıldızlı parametreler ile ilgili bir başka kısıtlama da şudur: Yıldızlı parametrelerden sonra sadece isimli parametreler gelebilir. Peki bu ne demek?

Dilerseniz bu kısıtlamayı bir örnek üzerinden anlatalım:

```
>>> print(*"falanca", sep=".")
```

Bu kod düzgün çalışır. Çünkü yıldızlı parametre olan *"falanca"* öğesinden sonra isimli bir parametre olan *sep* parametresi geliyor. Ama şu kod çalışmaz:

```
>>> print(*"falanca", "filanca")
```

Bu kod şuna benzer bir hata verecektir:

```
Traceback (most recent call last):
  File "<stdin>", line 1
SyntaxError: only named arguments may follow *expression
```

Bu kodun çalışmamasının sebebi, yıldızlı parametreden sonra gelen parametrenin isimli bir parametre olmaması. Yukarıdaki kodun çalışması için parametrelerin yerini değiştirmeyi deneyebilirsiniz:

```
>>> print("filanca", *"falanca")
```

Ayrıca bir fonksiyon içinde birden fazla yıldızlı parametre de kullanılamaz. Yani şu kod yanlıştır:

```
>>> print(*"filanca", *"falanca")
```

Bütün bu örnekler bize yıldızlı parametrelerin son derece kullanışlı araçlar olduğunu gösteriyor. İleride de bu parametrelerden bol bol yararlanacağız. Biz şimdi bu konuyu burada kapatıp başka bir şeyden söz edelim.

sys.stdout'u Kalıcı Olarak Değiştirmek

Önceki başlıklar altında verdiğimiz örneklerden de gördüğünüz gibi, print() fonksiyonunun *file* parametresi yardımıyla Python'ın standart çıktı konumunu geçici olarak değiştirebiliyoruz. Ama bazı durumlarda, yazdığınız programlarda, o programın işleyişi boyunca standart dışı bir çıktı konumu belirlemek isteyebilirsiniz. Yani standart çıktı konumunu geçici olarak değil, kalıcı olarak değiştirmeniz gerekebilir. Mesela yazdığınız programda bütün çıktıları bir dosyaya yazdırmayı tercih edebilirsiniz. Elbette bu işlemi her defasında *file* parametresini, çıktıları yazdırmak istediğiniz dosyanın adı olarak belirleyerek yapabilirsiniz. Tıpkı şu örnekte olduğu gibi:

```
>>> f = open("dosya.txt", "w")
>>> print("Fırat Özgül", file=f)
>>> print("Adana", file=f)
>>> print("Ubuntu", file=f)
>>> f.close()
```

Gördüğünüz gibi, her defasında *file* parametresine *f* değerini vererek işimizi hallettik. Ama bunu yapmanın daha pratik bir yöntemi var. Dilerseniz yazdığınız programın tüm işleyişi boyunca çıktıları başka bir konuma yönlendirebilirsiniz. Bunun için hem şimdiye kadar öğrendiğimiz, hem de henüz öğrenmediğimiz bazı bilgileri kullanacağız.

İlk önce şöyle bir kod yazalım:

```
>>> import sys
```

Bu kod yardımıyla *sys* adlı özel bir 'modülü' programımıza dahil etmiş, yani içe aktarmış olduk. Peki 'modül' nedir, 'içe aktarmak' ne demek?

Aslında biz bu 'modül' ve 'içe aktarma' kavramlarına hiç de yabancı değiliz. Önceki derslerde, pek üzerinde durmamış da olsak, biz Python'daki birkaç modülle zaten tanışmıştık. Mesela *os* adlı bir modül içindeki getcwd() adlı bir fonksiyonu kullanarak, o anda hangi dizinde bulunduğumuzu öğrenebilmiştik:

```
>>> import os
>>> os.getcwd()
```

Aynı şekilde *keyword* adlı başka bir modül içindeki *kwlist* adlı değişkeni kullanarak, hangi kelimelerin Python'da değişken adı olarak kullanılamayacağını da listeleyebilmiştik:

```
>>> import keyword
>>> keyword.kwlist
```

İşte şimdi de, *os* ve *keyword* modüllerine ek olarak *sys* adlı bir modülden söz ediyoruz. Gelin isterseniz öteki modülleri şimdilik bir kenara bırakıp, bu *sys* denen modüle dikkatimizi verelim.

Dediğimiz gibi, *sys* modülü içinde pek çok önemli değişken ve fonksiyon bulunur. Ancak bir modül içindeki değişken ve fonksiyonları kullanabilmek için o modülü öncelikle programımıza dahil etmemiz, yani içe aktarmamız gerekiyor. Bunu import komutuyla yapıyoruz:

```
>>> import sys
```

Artık *sys* modülü içindeki bütün fonksiyon ve değişkenlere ulaşabileceğiz.

sys modülü içinde bulunan pek çok değişken ve fonksiyondan biri de *stdout* adlı değişkendir. Bu değişkenin değerine şöyle ulaşabilirsiniz:

```
>>> sys.stdout
```

Bu komut şuna benzer bir çıktı verir:

```
<_io.TextIOWrapper name='<stdout>' mode='w' encoding='cp1254'>
```

Bu çıktıdaki *name='<stdout>'* kısmına dikkat edin. Bu ifadeye birazdan geri döneceğiz. Biz şimdi başka bir şeyden söz edelim.

Hatırlarsanız etkileşimli kabuğu nasıl kapatabileceğimizi anlatırken, etkileşimli kabuktan çıkmanın bir yolunun da şu komutları vermek olduğunu söylemiştik:

```
>>> import sys; sys.exit()
```

Bu komutu tek satırda yazmıştık, ama istersek şöyle de yazabiliriz elbette:

```
>>> import sys
>>> sys.exit()
```

Dedik ya, *sys* modülü içinde pek çok değişken ve fonksiyon bulunur. Nasıl *stdout sys* modülü içindeki değişkenlerden biri ise, exit() de *sys* modülü içinde bulunan fonksiyonlardan biridir.

Biz 'modüller' konusunu ilerleyen derslerde ayrıntılı bir şekilde inceleyeceğiz. Şimdilik modüllere ilişkin olarak yalnızca şunları bilelim yeter:

1. Python'da modüller import komutu ile içe aktarılır. Örneğin *sys* adlı modülü içe aktarmak için import sys komutunu veriyoruz.

2. Modüller içinde pek çok faydalı değişken ve fonksiyon bulunur. İşte bir modülü içe aktardığımızda, o modül içindeki bu değişken ve fonksiyonları kullanma imkanı elde ederiz.

3. *sys* modülü içindeki değişkenlere bir örnek *stdout*; fonksiyonlara örnek ise exit() fonksiyonudur.

Bir modül içindeki bu değişken ve fonksiyonlara 'modül_adı.değişken_ya_da_fonksiyon' formülünü kullanarak erişebiliriz. Örneğin:

```
>>> sys.stdout
>>> sys.exit()
```

4. Hatırlarsanız bundan önce de, open() fonksiyonu ile dosya oluşturmayı anlatırken, oluşturulan dosyanın hangi dizinde olduğunu bulabilmek amacıyla, o anda içinde bulunduğumuz dizini tespit edebilmek için şu kodları kullanmıştık:

```
>>> import os
>>> os.getcwd()
```

Burada da *os* adlı başka bir modül görüyoruz. İşte *os* da tıpkı *sys* gibi bir modüldür ve tıpkı *sys* modülünde olduğu gibi, *os* modülünün de içinde pek çok yararlı değişken ve fonksiyon bulunur. getcwd() adlı fonksiyon da *os* modülü içinde yer alan ve o anda hangi dizin altında bulunduğumuzu gösteren bir fonksiyondur. Elbette, yine tıpkı *sys* modülünde olduğu gibi, *os* modülü içindeki bu yararlı değişken ve fonksiyonları kullanabilmek için de öncelikle bu *os* modülünü içe aktarmamız, yani programımıza dahil etmemiz gerekiyor. *os* modülünü import komutu aracılığıyla uygun bir şekilde içe aktardıktan sonra, modül içinde yer alan getcwd() adlı fonksiyona yine 'modül_adı.fonksiyon' formülünü kullanarak erişebiliyoruz.

Modüllere ilişkin şimdilik bu kadar bilgi yeter. Modülleri bir kenara bırakıp yolumuza devam edelim...

Eğer sys.exit() komutunu verip etkileşimli kabuktan çıktıysanız, etkileşimli kabuğa tekrar girin ve *sys* modülünü yeniden içe aktarın:

```
>>> import sys
```

Not

Bir modülü aynı etkileşimli kabuk oturumu içinde bir kez içe aktarmak yeterlidir. Bir modülü bir kez içe aktardıktan sonra, o oturum süresince bu modül içindeki değişken ve fonksiyonları kullanmaya devam edebilirsiniz. Ama tabii ki etkileşimli kabuğu kapatıp tekrar açtıktan sonra, bir modülü kullanabilmek için o modülü tekrar içe aktarmanız gerekir.

Şimdi şu kodu yazın:

```
>>> f = open("dosya.txt", "w")
```

Bu kodun anlamını biliyorsunuz. Burada *dosya.txt* adlı bir dosyayı yazma kipinde açmış olduk. Tahmin edebileceğiniz gibi, çıktılarımızı ekran yerine bu dosyaya yönlendireceğiz.

Şimdi de şöyle bir kod yazalım:

```
>>> sys.stdout = f
```

Bildiğiniz gibi, *sys.stdout* değeri Python'ın çıktıları hangi konuma vereceğini belirliyor. İşte biz burada *sys.stdout*'un değerini biraz önce oluşturduğumuz *f* adlı dosya ile değiştiriyoruz. Böylece Python bütün çıktıları *f* değişkeni içinde belirttiğimiz *dosya.txt* adlı dosyaya gönderiyor.

Bu andan sonra yazacağınız her şey *dosya.txt* adlı dosyaya gidecektir:

```
>>> print("deneme metni", flush=True)
```

Gördüğünüz gibi, burada *file* parametresini kullanmadığımız halde çıktılarımız ekrana değil, *dosya.txt* adlı bir dosyaya yazdırıldı. Peki ama bu nasıl oldu? Aslında bunun cevabı çok basit: Biraz önce sys.stdout = f komutuyla *sys.stdout*'un değerini *f* değişkeninin tuttuğu dosya ile değiştirdik. Bu işlemi yapmadan önce sys.stdout'un değeri şuydu hatırlarsanız:

```
<_io.TextIOWrapper name='<stdout>' mode='w' encoding='cp1254'>
```

Ama sys.stdout = f komutundan sonra her şey değişti. Kontrol edelim:

```
>>> print(sys.stdout, flush=True)
```

Elbette bu komuttan herhangi bir çıktı almadınız. Çıktının ne olduğunu görmek için *dosya.txt* adlı dosyayı açın. Orada şu satırı göreceksiniz:

```
<_io.TextIOWrapper name='dosya.txt' mode='w' encoding='cp1254'>
```

Gördüğünüz gibi, özgün *stdout* çıktısındaki *name='<stdout>'* değeri *name='dosya.txt'* olmuş. Dolayısıyla artık bütün çıktılar *dosya.txt* adlı dosyaya gidiyor...

Bu arada, yukarıdaki çıktıda görünen *name*, *mode* ve *encoding* değerlerine şu şekilde ulaşabilirsiniz:

```
>>> sys.stdout.name
>>> sys.stdout.mode
>>> sys.stdout.encoding
```

Burada sys.stdout.name komutu standart çıktı konumunun o anki adını verecektir. sys.stdout.mode komutu ise standart çıktı konumunun hangi kipe sahip olduğunu gösterir. Standart çıktı konumu genellikle yazma kipinde (*w*) bulunur. sys.stdout.encoding kodu ise standart çıktı konumunun sahip olduğu kodlama biçimini gösterir. Kodlama biçimi, standart çıktı konumuna yazdıracağınız karakterlerin hangi kodlama biçimi ile kodlanacağını belirler. Kodlama biçimi Windows'ta genellikle 'cp1254', GNU/Linux'ta ise 'utf-8'dir. Eğer bu kodlama biçimi yanlış olursa, mesela dosyaya yazdıracağınız karakterler içindeki Türkçe harfler düzgün görüntülenemez. Eğer burada söylediklerimiz size şu anda anlaşılmaz geliyorsa, söylediklerimizi dikkate almadan yolunuza devam edebilirsiniz. Birkaç bölüm sonra bu söylediklerimiz size daha fazla şey ifade etmeye

başlayacak nasıl olsa.

Peki standart çıktı konumunu eski haline döndürmek isterseniz ne yapacaksınız? Bunun için etkileşimli kabuktan çıkıp tekrar girebilirsiniz. Etkileşimli kabuğu tekrar açtığınızda her şeyin eski haline döndüğünü göreceksiniz. Aynı şekilde, eğer bu kodları bir program dosyasına yazmış olsaydınız, programınız kapandığında her şey eski haline dönecekti.

Peki standart çıktı konumunu, etkileşimli kabuktan çıkmadan veya programı kapatmadan eski haline döndürmenin bir yolu var mı? Elbette var. Dikkatlice bakın:

```
>>> import sys
>>> f = open("dosya.txt", "w")
>>> sys.stdout, f = f, sys.stdout
>>> print("deneme", flush=True)
>>> f, sys.stdout = sys.stdout, f
>>> print("deneme")

deneme
```

Aslında burada anlayamayacağınız hiçbir şey yok. Burada yaptığımız şeyi geçen bölümlerde değişkenlerin değerini nasıl takas edeceğimizi anlatırken de yapmıştık. Hatırlayalım:

```
>>> osman = "Araştırma Geliştirme Müdürü"
>>> mehmet = "Proje Sorumlusu"
>>> osman, mehmet = mehmet, osman
```

Bu kodlarla Osman ve Mehmet'in unvanlarını birbiriyle takas etmiştik. İşte yukarıda yaptığımız şey de bununla aynıdır. `sys.stdout, f = f, sys.stdout` dediğimizde *f* değerini *sys.stdout*'a, *sys.stdout*'un değerini ise *f*'e vermiş oluyoruz. `f, sys.stdout = sys.stdout, f` dediğimizde ise, bu işlemin tam tersini yaparak her şeyi eski haline getirmiş oluyoruz.

Python'ın bize sunduğu bu kolaylıktan faydalanarak değişkenlerin değerini birbiriyle kolayca takas edebiliyoruz. Eğer böyle bir kolaylık olmasaydı yukarıdaki kodları şöyle yazabilirdik:

```
>>> import sys
>>> f = open("dosya.txt", "w")
>>> özgün_stdout = sys.stdout
>>> sys.stdout = f
>>> print("deneme", flush=True)
>>> sys.stdout = özgün_stdout
>>> print("deneme")

deneme
```

Gördüğünüz gibi, *sys.stdout*'un değerini kaybetmemek için, *sys.stdout* değerini *f* adlı dosyaya

göndermeden önce şu kod yardımıyla yedekliyoruz:

```
>>> özgün_stdout = sys.stdout
```

sys.stdout'un özgün değerini *özgün_stdout* değişkenine atadığımız için, bu değere sonradan tekrar ulaşabileceğiz. Zaten yukarıdaki kodlardan da gördüğünüz gibi, *sys.stdout*'un özgün değerine dönmek istediğimizde şu kodu yazarak isteğimizi gerçekleştirebiliyoruz:

```
>>> sys.stdout = özgün_stdout
```

Böylece *stdout* değeri eski haline dönmüş oluyor ve bundan sonra yazdırdığımız her şey yeniden ekrana basılmaya başlıyor.

...ve böylece uzun bir bölümü daha geride bıraktık. Bu bölümde hem print() fonksiyonunu bütün ayrıntılılarıyla incelemiş olduk, hem de Python programlama diline dair başka çok önemli kavramlardan söz ettik. Bu bakımdan bu bölüm bize epey şey öğretti. Artık öğrendiğimiz bu bilgileri de küfemize koyarak alnımız dik bir şekilde yola devam edebiliriz.

başa dön

Kaçış Dizileri

Python'da karakter dizilerini tanımlayabilmek için tek, çift veya üç tırnak işaretlerinden faydalandığımızı geçen bölümde öğrenmiştik. Python bir verinin karakter dizisi olup olmadığına bu tırnak işaretlerine bakarak karar verdiği için, tek, çift ve üç tırnak işaretleri Python açısından özel bir önem taşıyor. Zira Python'ın gözünde bir başlangıç tırnağı ile bitiş tırnağı arasında yer alan her şey bir karakter dizisidir.

Örneğin ilk olarak bir *"* işareti koyup ardından *"elma* şeklinde devam ettiğinizde, Python ilk tırnağı gördükten sonra karakter dizisini tanımlayabilmek için ikinci bir tırnak işareti aramaya başlar. Siz *"elma"* şeklinde kodunuzu tamamladığınızda ise Python bellekte *"elma"* adlı bir karakter dizisi oluşturur.

Bu noktada size şöyle bir soru sormama izin verin: Acaba tırnak işaretleri herhangi bir metin içinde kaç farklı amaçla kullanılabilir? İsterseniz bu sorunun cevabını örnekler üzerinde vermeye çalışalım:

Ahmet, "Bugün sinemaya gidiyorum," dedi.

Burada tırnak işaretlerini, bir başkasının sözlerini aktarmak için kullandık.

'book' kelimesi Türkçede 'kitap' anlamına gelir.

Burada ise tırnak işaretlerini bazı kelimeleri vurgulamak için kullandık.

Bir de şuna bakalım:

Yarın Adana'ya gidiyorum.

Burada da tırnak işaretini, çekim eki olan '-(y)a' ile özel isim olan 'Adana' kelimesini birbirinden ayırmak için kesme işareti görevinde kullandık.

Şimdi yukarıda verdiğimiz ilk cümleyi bir karakter dizisi olarak tanımlamaya çalışalım:

```
>>> 'Ahmet, "Bugün sinemaya gidiyorum," dedi.'
```

Burada karakter dizisini tanımlamaya tek tırnak işareti ile başladık. Böylece Python bu karakter dizisini tanımlama işlemini bitirebilmek için ikinci bir tek tırnak işareti daha aramaya koyuldu ve aradığı tek tırnak işaretini cümlenin sonunda bularak, karakter dizisini düzgün bir şekilde oluşturabildi.

Dediğimiz gibi, Python'ın gözünde tırnak işaretleri bir karakter dizisini başka veri tiplerinden ayırt etmeye yarayan bir ölçüttür. Ama biz insanlar, yukarıda verdiğimiz örnek cümlelerden de göreceğiniz gibi, programlama dillerinden farklı olarak, tırnak işaretlerini bir metin içinde daha farklı amaçlar için de kullanabiliyoruz.

Şimdi yukarıdaki karakter dizisini şöyle tanımlamaya çalıştığımızı düşünün:

```
>>> "Ahmet, "Bugün sinemaya gidiyorum," dedi."
```

İşte burada Python'ın çıkarları ile bizim çıkarlarımız birbiriyle çatıştı. Python karakter dizisini başlatan ilk çift tırnak işaretini gördükten sonra, karakter dizisini tanımlama işlemini bitirebilmek için ikinci bir tırnak işareti daha aramaya koyuldu. Bu arayış sırasında da 'Bugün' kelimesinin başındaki çift tırnak işaretini gördü ve karakter dizisinin şu olduğunu zannetti:

```
>>> "Ahmet, "
```

Buraya kadar bir sorun yok. Bu karakter dizisi Python'ın sözdizimi kurallarına uygun.

Karakter dizisi bu şekilde tanımlandıktan sonra Python cümlenin geri kalanını okumaya devam ediyor ve herhangi bir tırnak işareti ile başlamayan ve kendisinden önce gelen öğeden herhangi bir virgül işareti ile ayrılmamış 'Bugün' kelimesini görüyor. Eğer bir kelime tırnak işareti ile başlamıyorsa bu kelime ya bir değişkendir ya da sayıdır. Ama 'Bugün' kelimesi ne bir değişken, ne de bir sayı olduğu, üstelik önceki öğeden de virgülle ayrılmadığı için Python'ın hata vermekten başka çaresi kalmıyor. Çünkü biz burada 'Bugün' kelimesinin baş tarafındaki çift tırnak işaretini karakter dizisi tanımlamak için değil, başkasının sözlerini aktarmak amacıyla kullandık. Ancak elbette bir programlama dili bizim amacımızın ne olduğunu kestiremez ve hata mesajını suratımıza yapıştırır:

```
  File "<stdin>", line 1
    "Ahmet, "Bugün sinemaya gidiyorum," dedi."
              ^
SyntaxError: invalid syntax
```

Peki biz böyle bir durumda ne yapmalıyız?

Bu hatayı engellemek için karakter dizisini tanımlamaya çift tırnak yerine tek tırnakla ya da üç tırnakla başlayabiliriz:

```
>>> 'Ahmet, "Bugün sinemaya gidiyorum," dedi.'
```

... veya:

```
>>> """Ahmet, "Bugün sinemaya gidiyorum," dedi."""
```

Böylece karakter dizisini başlatan işaret 'Bugün sinemaya gidiyorum,' cümlesinin başındaki ve sonundaki işaretlerden farklı olduğu için, Python okuma esnasında bu cümleye takılmaz ve doğru bir şekilde, karakter dizisini kapatan tırnak işaretini bulabilir.

Bu yöntem tamamen geçerli ve mantıklıdır. Ama eğer istersek, aynı karakter dizisini çift tırnakla tanımlayıp, yine de hata almayı engelleyebiliriz. Peki ama nasıl?

İşte burada 'kaçış dizileri' adı verilen birtakım araçlardan faydalanacağız.

Peki nedir bu 'kaçış dizisi' denen şey?

Kaçış dizileri, Python'da özel anlam taşıyan işaret veya karakterleri, sahip oldukları bu özel anlam dışında bir amaçla kullanmamızı sağlayan birtakım araçlardır. Mesela yukarıda da örneklerini verdiğimiz gibi, tırnak işaretleri Python açısından özel anlam taşıyan işaretlerdir. Normalde Python bu işaretleri karakter dizilerini tanımlamak için kullanır. Ama eğer siz mesela bir metin içinde bu tırnak işaretlerini farklı bir amaçla kullanacaksanız Python'ı bu durumdan haberdar etmeniz gerekiyor. İşte kaçış dizileri, Python'ı böyle bir durumdan haberdar etmemize yarayan araçlardır.

Python'da pek çok kaçış dizisi bulunur. Biz burada bu kaçış dizilerini tek tek inceleyeceğiz. O halde hemen işe koyulalım.

Ters Taksim (\)

Yukarıda verdiğimiz örneklerde, çift tırnakla gösterdiğimiz karakter dizilerinin içinde de çift tırnak işareti kullanabilmek için birkaç farklı yöntemden yararlanabildiğimizi öğrenmiştik. Buna göre, eğer bir karakter dizisi içinde çift tırnak işareti geçiyorsa, o karakter dizisini tek tırnakla; eğer tek tırnak geçiyorsa da o karakter dizisini çift tırnakla tanımlayarak bu sorunun üstesinden gelebiliyorduk. Ama daha önce de söylediğimiz gibi, 'kaçış dizileri' adı verilen birtakım araçları kullanarak, mesela içinde çift tırnak geçen karakter dizilerini yine çift tırnakla tanımlayabiliriz.

Dilerseniz, kaçış dizisi kavramını açıklamaya geçmeden önce bununla ilgili birkaç örnek verelim. Bu sayede ne ile karşı karşıya olduğumuz, zihnimizde biraz daha belirginleşebilir:

```
>>> print('Yarın Adana\'ya gidiyorum.')

Yarın Adana'ya gidiyorum.
```

Bir örnek daha verelim:

```
>>> print("\"book\" kelimesi Türkçede \"kitap\" anlamına gelir.")

"book" kelimesi Türkçede "kitap" anlamına gelir.
```

Burada da cümle içinde çift tırnak işaretlerini kullandığımız halde, \ işaretleri sayesinde karakter dizilerini yine çift tırnakla tanımlayabildik.

Bir de şu örneğe bakalım:

```
>>> print("Python programlama dilinin adı \"piton\" yılanından gelmez")
```

Bütün bu örneklerde, karakter dizisini hem çift tırnakla tanımlayıp hem de karakter dizisi içinde çift tırnak işaretlerini kullandığımız halde, herhangi bir hata almadığımızı görüyorsunuz. Yukarıdaki kodlarda hata almamızı önleyen şeyin \ işareti olduğu belli. Ama dilerseniz bu işaretin, hata almamızı nasıl önlediğini anlatmadan önce son bir örnek daha verelim.

Hatırlarsanız önceki sayfalarda şöyle bir karakter dizisi ile karşılaşmıştık:

```
>>> print('İstanbul'un 5 günlük hava durumu tahmini')

  File "<stdin>", line 1
    print('İstanbul'un 5 günlük hava durumu tahmini')
                    ^
SyntaxError: invalid syntax
```

Burada da ‘İstanbul’un’ kelimesi içinde geçen tırnak işareti nedeniyle karakter dizisini tek tırnak kullanarak tanımlayamıyorduk. Bu karakter dizisini hatasız bir şekilde tanımlayabilmek için ya çift tırnak ya da üç tırnak kullanmamız gerekiyordu:

```
>>> print("İstanbul'un 5 günlük hava durumu tahmini")

İstanbul'un 5 günlük hava durumu tahmini
```

... veya:

```
>>> print("""İstanbul'un 5 günlük hava durumu tahmini""")

İstanbul'un 5 günlük hava durumu tahmini
```

Tıpkı önceki örneklerde olduğu gibi, yukarıdaki karakter dizisini de aslında tek tırnakla tanımlayıp hata oluşmasını önleyebiliriz. Hemen görelim:

```
>>> print('İstanbul\'un 5 günlük hava durumu tahmini')

İstanbul'un 5 günlük hava durumu tahmini
```

Bütün örneklerde \ işaretini kullandığımızı görüyorsunuz. İşte bu tür işaretlere Python’da kaçış dizisi (*escape sequence*) adı verilir. Bu işaretler karakter dizilerini tanımlarken oluşabilecek hatalardan kaçmamızı sağlar. Peki bu \ işareti nasıl oluyor da karakter dizisini tanımlarken hata almamızı önlüyor? Gelin bu süreci adım adım tarif edelim:

Python bir karakter dizisi tanımladığımızda, karakter dizisini soldan sağa doğru okumaya başlar. Mesela yukarıdaki örnekte ilk olarak karakter dizisini tanımlamaya tek tırnakla başladığımızı görür.

Python karakter dizisini başlatan bu tek tırnak işaretini gördüğü zaman, soldan sağa doğru ilerleyerek karakter dizisini bitirecek olan tek tırnak işaretini aramaya başlar.

Soldan sağa doğru ilerlerken ‘İstanbul’un’ kelimesi içinde geçen kesme işaretini görür ve karakter dizisinin burada sona erdiğini düşünür. Ancak karakter dizisini sona erdiren işaret bu olmadığı için Python’ın hata vermekten başka çaresi kalmaz.

İşte biz ‘İstanbul’un’ kelimesi içinde geçen bu kesme işaretinin sol tarafına bir adet \ işareti yerleştirerek Python’a, ‘Aradığın işaret bu değil. Sen karakter dizisini okumaya devam et. Biraz sonra aradığın tırnağı bulacaksın!’ mesajı vermiş, yani orada tırnak işaretini farklı bir amaçla kullandığımız

konusunda Python'ı bilgilendirmiş oluruz.

Şurada da aynı durum sözkonusu:

```
>>> print("Python programlama dilinin adı \"piton\" yılanından gelmez")
```

Tıpkı bir önceki örnekte olduğu gibi, burada da Python karakter dizisini soldan sağa doğru okumaya başlıyor, karakter dizisini başlatan çift tırnak işaretini görüyor ve bunun üzerine Python karakter dizisini bitirecek olan çift tırnak işaretini aramaya koyuluyor.

Karakter dizisini soldan sağa doğru okuduğu sırada, karakter dizisi içinde geçen 'piton' kelimesini görüyor. Eğer burada bir önlem almazsak Python bu kelimenin başındaki çift tırnak işaretini, karakter dizisini sona erdiren tırnak olarak algılar ve durum aslında böyle olmadığı için de hata verir.

Bu hatayı önlemek için 'piton' kelimesinin başındaki çift tırnağın soluna bir adet \ işareti yerleştirerek Python'a, 'Aradığın tırnak bu değil!' mesajı veriyoruz. Yani bir bakıma, \ adlı kaçış dizisi kendisini tırnak işaretine siper edip Python'ın bu tırnağı görmesine mani oluyor...

Bunun üzerine Python bu çift tırnak işaretini görmezden gelerek, soldan sağa doğru okumaya devam eder ve yol üzerinde 'piton' kelimesinin sonundaki çift tırnak işaretini görür. Eğer burada da bir önlem almazsak Python yine bir hata verecektir.

Tıpkı biraz önce yaptığımız gibi, bu tırnak işaretinin de soluna bir adet \ işareti yerleştirerek Python'a, 'Aradığın tırnak bu da değil. Sen yine okumaya devam et!' mesajı veriyoruz.

Bu mesajı alan Python karakter dizisini soldan sağa doğru okumaya devam ediyor ve sonunda karakter dizisini bitiren çift tırnak işaretini bularak bize hatasız bir çıktı veriyor.

Böylece \ işareti üzerinden hem kaçış dizilerinin ne olduğunu öğrenmiş, hem de bu kaçış dizisinin nasıl kullanılacağına dair örnekler vermiş olduk. Ancak \ kaçış dizisinin yetenekleri yukarıdakilerle sınırlı değildir. Bu kaçış dizisini, uzun karakter dizilerini bölmek için de kullanabiliriz. Şimdi şu örneği dikkatlice inceleyin:

```
>>> print("Python 1990 yılında Guido Van Rossum \
... tarafından geliştirilmeye başlanmış, oldukça \
... güçlü ve yetenekli bir programlama dilidir.")

Python 1990 yılında Guido Van Rossum tarafından geliştirilmeye
başlanmış, oldukça güçlü ve yetenekli bir programlama dilidir.
```

Normal şartlar altında, bir karakter dizisini tanımlamaya tek veya çift tırnakla başlamışsak, karakter dizisinin kapanış tırnağını koymadan *Enter* tuşuna bastığımızda Python bize bir hata mesajı gösterir:

```
>>> print("Python 1990 yılında Guido Van Rossum

  File "<stdin>", line 1
```

```
    print("Python 1990 yılında Guido Van Rossum
                                               ^
SyntaxError: EOL while scanning string literal
```

İşte \ kaçış dizisi bizim burada olası bir hatadan kaçmamızı sağlar. Eğer *Enter* tuşuna basmadan önce bu işareti kullanırsak Python tıpkı üç tırnak işaretlerinde şahit olduğumuz gibi, hata vermeden bir alt satıra geçecektir. Bu sırada, yani \ kaçış dizisini koyup *Enter* tuşuna bastığımızda >>> işaretinin ... işaretine dönüştüğünü görüyorsunuz. Bu işaretin, Python'ın bize verdiği bir 'Yazmaya devam et!' mesajı olduğunu biliyorsunuz.

Satır Başı (\n)

Python'daki en temel kaçış dizisi biraz önce örneklerini verdiğimiz \ işaretidir. Bu kaçış dizisi başka karakterlerle birleşerek, farklı işlevlere sahip yeni kaçış dizileri de oluşturabilir. Aslında bu olguya yabancı değiliz. Önceki sayfalarda bu duruma bir örnek vermiştik. Hatırlarsanız print() fonksiyonunu anlatırken *end* parametresinin ön tanımlı değerinin *\n*, yani satır başı karakteri olduğunu söylemiştik.

Not

Satır başı karakterine 'yeni satır karakteri' dendiği de olur.

Satır başı karakterini ilk öğrendiğimizde bu karakteri anlatırken bazı örnekler de vermiştik:

```
>>> print("birinci satır\nikinci satır\nüçüncü satır")

birinci satır
ikinci satır
üçüncü satır
```

Gördüğünüz gibi, *\n* adlı kaçış dizisi, bir alt satıra geçilmesini sağlıyor. İşte aslında *\n* kaçış dizisi de, \ ile 'n' harfinin birleşmesinden oluşmuş bir kaçış dizisidir. Burada \ işaretinin görevi, 'n' harfinin özel bir anlam kazanmasını sağlamaktır. \ işareti ile 'n' harfi birleştiğinde 'satır başı karakteri' denen özel bir karakter dizisi ortaya çıkarıyor.

Gelin bu kaçış dizisi ile ilgili bir örnek verelim. Şimdi şu kodları dikkatlice inceleyin:

```
>>> başlık = "Türkiye'de Özgür Yazılımın Geçmişi"
>>> print(başlık, "\n", "-"*len(başlık), sep="")

Türkiye'de Özgür Yazılımın Geçmişi
----------------------------------
```

Burada, *başlık* adlı değişkenin tuttuğu *"Türkiye'de Özgür Yazılımın Geçmişi"* adlı karakter dizisinin

altını çizdik. Dikkat ederseniz, başlığın altına koyduğumuz çizgiler başlığın uzunluğunu aşmıyor. Yazdığımız program, başlığın uzunluğu kadar çizgiyi başlığın altına ekliyor. Bu programda başlık ne olursa olsun, programımız çizgi uzunluğunu kendisi ayarlayacaktır. Örneğin:

```
>>> başlık = "Python Programlama Dili"
>>> print(başlık, "\n", "-"*len(başlık), sep="")

Python Programlama Dili
----------------------

>>> başlık = "Alışveriş Listesi"
>>> print(başlık, "\n", "-"*len(başlık), sep="")

Alışveriş Listesi
-----------------
```

Gelin isterseniz bu kodlardaki print() satırını şöyle bir inceleyelim. Kodumuz şu:

```
>>> print(başlık, "\n", "-"*len(başlık), sep="")
```

Burada öncelikle *başlık* adlı değişkeni print() fonksiyonunun parantezleri içine yazdık. Böylece *başlık* değişkeninin değeri ekrana yazdırılacak.

print() fonksiyonunun ikinci parametresinin *\n* adlı kaçış dizisi olduğunu görüyoruz. Bu kaçış dizisini eklememiz sayesinde Python ilk parametreyi çıktı olarak verdikten sonra bir alt satıra geçiyor. Bu parametrenin tam olarak ne işe yaradığını anlamak için, yukarıdaki satırı bir de o parametre olmadan çalıştırmayı deneyebilirsiniz:

```
>>> print(başlık, "-"*len(başlık), sep="")

Alışveriş Listesi----------------
```

print() fonksiyonunun üçüncü parametresinin ise şu olduğunu görüyoruz: "-"*len(başlık).

İşte *başlık* değişkeninin altına gerekli sayıda çizgiyi çizen kodlar bunlardır. Burada len() fonksiyonunu nasıl kullandığımıza çok dikkat edin. Bu kod sayesinde *başlık* değişkeninin uzunluğu (len(başlık)) sayısınca - işaretini ekrana çıktı olarak verebiliyoruz.

Yukarıdaki kodlarda print() fonksiyonunun son parametresi ise *sep=''*. Peki bu ne işe yarıyor? Her zaman olduğu gibi, bu kod parçasının ne işe yaradığını anlamak için programı bir de o kodlar olmadan çalıştırmayı deneyebilirsiniz:

```
>>> print(başlık, "\n", "-"*len(başlık))

Alışveriş Listesi
```

```
---------------
```

Gördüğünüz gibi, *başlık* değişkeninin tam altına gelmesi gereken çizgi işaretleri sağa kaymış. Bunun nedeni *sep* parametresinin öntanımlı değerinin bir adet boşluk karakteri olmasıdır. *sep* parametresinin öntanımlı değeri nedeniyle çizgilerin baş tarafına bir adet boşluk karakteri ekleniyor çıktıda. O yüzden bu çizgiler sağa kaymış görünüyor. İşte biz yukarıdaki kodlarda *sep* parametresinin öntanımlı değerini değiştirip, boşluk karakteri yerine boş bir karakter dizisi yerleştiriyoruz. Böylece çizgiler çıktıda sağa kaymıyor.

Satır başı karakteri, programlama maceramız sırasında en çok kullanacağımız kaçış dizilerinden biri ve hatta belki de birincisidir. O yüzden bu kaçış dizisini çok iyi öğrenmenizi tavsiye ederim.

Ayrıca bu kaçış dizisini (ve tabii öteki kaçış dizilerini) tanıyıp öğrenmeniz, yazacağınız programların selameti açısından da büyük önem taşır. Eğer bir karakter dizisi içinde geçen kaçış dizilerini ayırt edemezseniz Python size hiç beklemediğiniz çıktılar verebilir. Hatta yazdığınız programlar kaçış dizilerini tanımıyor olmanızdan ötürü bir anda hata verip çökebilir. Peki ama nasıl?

Şimdi şu örneğe dikkatlice bakın:

Diyelim ki bilgisayarınızın 'C:\' dizinindeki 'nisan' adlı bir klasörün içinde yer alan *masraflar.txt* adlı bir dosyayı yazdığınız bir program içinde kullanmanız gerekiyor. Mesela bu dosyayı, tam adresiyle birlikte kullanıcılarınıza göstermek istiyorsunuz.

İlk denememizi yapalım:

```
>>> print("C:\nisan\masraflar.txt")
```

Buradan şöyle bir çıktı aldık:

```
C:
isan\masraflar.txt
```

Gördüğünüz gibi, bu çıktıyı normal yollardan vermeye çalıştığımızda Python bize bir hata mesajı gösteriyor. Peki ama neden?

Python'da karakter dizileri ile çalışırken asla aklımızdan çıkarmamamız gereken bir şey var: Eğer yazdığımız herhangi bir karakter dizisinin herhangi bir yerinde \ işaretini kullanmışsak, bu işaretten hemen sonra gelen karakterin ne olduğuna çok dikkat etmemiz gerekir. Çünkü eğer dikkat etmezsek, farkında olmadan Python için özel anlam taşıyan bir karakter dizisi oluşturmuş olabiliriz. Bu da kodlarımızın beklediğimiz gibi çalışmasını engeller.

Yukarıdaki sorunun kaynağını anlamak için "C:\\nisan\\masraflar.txt" adlı karakter dizisine çok dikkatlice bakın. Python bu karakter dizisinde bizim '\nisan' olarak belirttiğimiz kısmın başındaki *\n* karakterlerini bir kaçış dizisi olarak algıladı. Çünkü *\n* adlı karakter dizisi, 'satır başı kaçış dizisi' adını verdiğimiz, Python açısından özel anlam taşıyan bir karakter dizisine işaret ediyor. Zaten yukarıdaki tuhaf görünen çıktıya baktığınızda da, bu kaçış dizisinin olduğu noktadan itibaren karakter

dizisinin bölünüp yeni bir satıra geçildiğini göreceksiniz. İşte biz yukarıdaki örnekte alelade bir dizin adı belirttiğimizi zannederken aslında hiç farkında olmadan bir kaçış dizisi üretmiş oluyoruz. Bu nedenle, daha önce de söylediğimiz gibi, karakter dizileri içinde farkında olarak veya olmayarak kullandığımız kaçış dizilerine karşı her zaman uyanık olmalıyız. Aksi takdirde, yukarıda olduğu gibi hiç beklemediğimiz çıktılarla karşılaşabiliriz.

Esasen yukarıdaki problem bir dereceye kadar (ve yerine göre) 'masum bir kusur' olarak görülebilir. Çünkü bu hata programımızın çökmesine yol açmıyor. Ama bir karakter dizisi içindeki gizli kaçış dizilerini gözden kaçırmak, bazı durumlarda çok daha yıkıcı sonuçlara yol açabilir. Mesela yukarıdaki sorunlu dizin adını ekrana yazdırmak yerine open() fonksiyonunu kullanarak, bu karakter dizisi içinde belirttiğimiz *masraflar.txt* adlı dosyayı açmaya çalıştığımızı düşünün:

```
>>> open("C:\nisan\masraflar.txt")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 22] Invalid argument: 'C:\nisan\\masraflar.txt'
```

Eğer sorunun gözden kaçan bir kaçış dizisinden kaynaklandığını farkedemezsiniz, bu sorunu çözebilmek için saatlerinizi ve hatta günlerinizi harcamak zorunda kalabilirsiniz. Çünkü yukarıdaki hata mesajı sorunun nedenine dair hiçbir şey söylemiyor. Ancak ve ancak yukarıdaki karakter dizisi içinde sinsice gizlenen bir *\n* kaçış dizisi olduğu gözünüze çarparsa bu sorunu çözme yolunda bir adım atabilirsiniz.

Diyelim ki sorunun '\nisan' ifadesinin başındaki *\n* karakterlerinin Python tarafından bir kaçış dizisi olarak algılanmasından kaynaklandığını farkettiniz. Peki bu sorunu nasıl çözeceksiniz?

Bu sorunun birkaç farklı çözüm yolu var. Biz şimdilik sadece ikisini göreceğiz. Bu bölümün sonuna vardığınızda öteki çözüm yolunu da öğrenmiş olacaksınız.

Yukarıdaki problemi, ilgili kaçış dizisi içindeki ters taksim işaretini çiftleyerek çözebilirsiniz:

```
>>> open("C:\\nisan\masraflar")
```

Tabii tutarlılık açısından karakter dizisi içindeki bütün ters taksim işaretlerini çiftlemek mantıklı olacaktır:

```
>>> open("C:\\nisan\\masraflar")
```

Bunun dışında, bu örnek için, dizin adlarını ters taksim yerine düz taksim işaretiyle ayırmayı tercih edebilirsiniz:

```
>>> open("C:/nisan/masraflar")
```

Dediğimiz gibi, üçüncü (ve aslında daha kullanışlı olan) yöntemi biraz sonra inceleyeceğiz. Biz şimdilik kaçış dizilerini anlatmaya devam edelim.

Sekme (\t)

Python'da \ işareti sadece 'n' harfiyle değil, başka harflerle de birleşebilir. Örneğin \ işaretini 't' harfiyle birleştirerek yine özel bir anlam ifade eden bir kaçış dizisi elde edebiliriz:

```
>>> print("abc\tdef")

abc def
```

Burada *\t* adlı kaçış dizisi, *"abc"* ifadesinden sonra sanki *Tab* (sekme) tuşuna basılmış gibi bir etki oluşturarak *"def"* ifadesini sağa doğru itiyor. Bir de şu örneğe bakalım:

```
>>> print("bir", "iki", "üç", sep="\t")

bir iki üç
```

Bir örnek daha:

```
>>> print(*"123456789", sep="\t")

1 2 3 4 5 6 7 8 9
```

Gördüğünüz gibi, parametreler arasında belli aralıkta bir boşluk bırakmak istediğimizde *\t* adlı kaçış dizisinden yararlanabiliyoruz.

Tıpkı *\n* kaçış dizisinde olduğu gibi, karakter dizilerinde *\t* kaçış dizisinin varlığına karşı da uyanık olmalıyız:

```
>>> open("C:\nisan\masraflar\toplam_masraf.txt")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 22] Invalid argument: 'C:\nisan\\masraflar\toplam_masraf.txt'
```

Burada da *\n* ile yaşadığımız soruna benzer bir durum var. Biz *toplam_masraf.txt* adlı bir dosyaya atıfta bulunmaya çalışıyoruz, ama Python bu ifadenin başındaki 't' harfinin, kendisinden önce gelen \ işareti ile birleşmesinden ötürü, bunu *\t* kaçış dizisi olarak algılıyor ve ona göre davranıyor.

Belki yukarıdaki kodları şöyle yazarsak durumu anlamak daha kolay olabilir:

```
>>> print("C:\nisan\masraflar\toplam_masraf.txt")

C:
isan\masraflar      oplam_masraf.txt
```

Gördüğünüz gibi, Python \n kaçış dizisini gördüğü noktada alt satırın başına geçiyor ve \t kaçış dizisini gördüğü noktada da önceki ve sonraki öğeler arasında bir sekme boşluğu bırakıyor. Bu durumu engellemek için ne yapmanız gerektiğini biliyorsunuz: Ya ters taksim işaretlerini çiftleyeceksiniz:

```
>>> print("C:\\nisan\\masraflar\\toplam_masraf.txt")
```

Ya da dizin adı ayracı olarak düz taksim işaretini kullanacaksınız:

```
>>> print("C:/nisan/masraflar/toplam_masraf.txt")
```

Daha önce de söylediğimiz gibi, üçüncü ve daha pratik olan yolu biraz sonra göreceğiz. Şimdilik sadece biraz sabır...

Zil Sesi (\a)

\ işaretinin birleştiğinde farklı bir anlam türettiği bir başka harf de 'a' harfidir. \ işareti harfiyle birleşerek !bip! benzeri bir zil sesi üretilmesini sağlayabilir:

```
>>> print("\a")

!bip!
```

İsterseniz yukarıdaki komutu şu şekilde yazarak, kafa şişirme katsayısını artırabilirsiniz:

```
>>> print("\a" * 10)
```

Bu şekilde !bip! sesi *10* kez tekrar edilecektir. Ancak bu kaçış dizisi çoğunlukla sadece Windows üzerinde çalışacaktır. Bu kaçış dizisinin GNU/Linux üzerinde çalışma garantisi yoktur. Hatta bu kaçış dizisi bütün Windows sistemlerinde dahi çalışmayabilir. Dolayısıyla bu kaçış dizisinin işlevine bel bağlamak pek mantıklı bir iş değildir.

Tıpkı \n ve \t kaçış dizilerinde olduğu gibi bu kaçış dizisinin varlığına karşı da uyanık olmalıyız. Burada da mesela 'C:\aylar' gibi bir dizin adı tanımlamaya çalışırken aslında \a kaçış dizisini oluşturuyor olabilirsiniz farkında olmadan.

Aynı Satır Başı (\r)

Bu kaçış dizisi, bir karakter dizisinde aynı satırın en başına dönülmesini sağlar. Bu kaçış dizisinin işlevini tanımına bakarak anlamak biraz zor olabilir. O yüzden dilerseniz bu kaçış dizisinin ne işe yaradığını bir örnek üzerinde göstermeye çalışalım:

```
>>> print("Merhaba\rZalim Dünya!")

Zalim Dünya!
```

Burada olan şey şu: Normal şartlar altında, print() fonksiyonu içine yazdığımız bir karakter dizisindeki bütün karakterler soldan sağa doğru tek tek ekrana yazdırılır:

```
>>> print("Merhaba Zalim Dünya!")

Merhaba Zalim Dünya!
```

Ancak eğer karakter dizisinin herhangi bir yerine *\r* adlı kaçış dizisini yerleştirirsek, bu kaçış dizisinin bulunduğu konumdan itibaren satırın başına dönülecek ve *\r* kaçış dizisinden sonra gelen bütün karakterler satır başındaki karakterlerin üzerine yazacaktır. Şu örnek daha açıklayıcı olabilir:

```
>>> print("Merhaba\rDünya")

Dünyaba
```

Burada, *"Merhaba"* karakter dizisi ekrana yazdırıldıktan sonra *\r* kaçış dizisinin etkisiyle satır başına dönülüyor ve bu kaçış dizisinden sonra gelen *"Dünya"* karakter dizisi *"Merhaba"* karakter dizisinin üzerine yazıyor. Tabii *"Dünya"* karakter dizisi içinde *5* karakter, *"Merhaba"* karakter dizisi içinde ise *7* karakter olduğu için, *"Merhaba"* karakter dizisinin son iki karakteri (*"ba"*) dışarda kalıyor. Böylece ortaya *"Dünyaba"* gibi bir şey çıkıyor.

Önceki kaçış dizilerinde olduğu gibi, bu kaçış dizisini de farkında olmadan karakter dizisi içinde kullanırsanız beklemediğiniz çıktılar alırsınız:

```
>>> print("C:\ülke\türkiye\iller\rize\nüfus.txt")

izeülke     ürkiye\iller
üfus.txt
```

Burada farkında olmadan sadece bir değil, üç kaçış dizisi birden oluşturduk!

Düşey Sekme (\v)

Eğer \ işaretini 'v' harfiyle birlikte kullanırsak düşey sekme denen şeyi elde ederiz. Hemen bir örnek verelim:

```
>>> print("düşey\vsekme")

düşey
    sekme
```

Yalnız bu *\v* adlı kaçış dizisi her işletim sisteminde çalışmayabilir. Dolayısıyla, birden fazla platform üzerinde çalışmak üzere tasarladığınız programlarınızda bu kaçış dizisini kullanmanızı önermem.

Geri Silme (\b)

\ kaçış dizisinin, biraraya geldiğinde özel bir anlam kazandığı bir başka harf de b'dir. *\b* kaçış dizisini şöyle bir örnek içinde kullanabiliriz:

```
>>> print("istihza", "\b.", "\bcom")
```

Burada *\b* adlı kaçış dizisi, kendisinden önceki bir karakteri silme görevini üstleniyor. Bu örnekte eğer *\b* kaçış dizisini kullanmazsanız ne olacağını kendiniz deneyerek gözlerinizle görebilirsiniz.

Örnekten de gördüğünüz gibi, *\b* kaçış dizisi gereksiz boşluk karakterlerini siliyor. *"b"* kaçış dizisinin görevi klavyedeki *Backspace* tuşuna benzer.

\b kaçış dizisini bir örnekte daha görelim:

```
>>> print("merhaba\b")
```

Ne oldu? Bu karakter dizisi hiç bir şeyi değiştirmedi, değil mi? Çok normal. Çünkü bu kaçış dizisi biraz kaprislidir. *\b* adlı kaçış dizisi, çalışabilmek için kendisinden sonra da en az bir karakter olmasını ister. Dolayısıyla yukarıdaki örnekte *\b* kaçış dizisi görevini yerine getirmiyor, ama şu örnekte bu kaçış dizisi görevini eksiksiz olarak yerine getirecektir:

```
>>> print("merhaba\b dünya")

merhab dünya
```

Burada *\b* kaçış dizisinden sonra boşluk karakteri var. Bu yüzden bu kaçış dizisi böyle bir ortamda görevini yerine getirebiliyor. Eğer sadece print("merhaba") yazıp "merhab" çıktısını elde etmek isterseniz şöyle bir hileye başvurabilirsiniz:

```
>>> print("merhaba\b ")

merhab
```

Burada gördüğünüz gibi, *\b* kaçış dizisinden sonra bir boşluk bırakarak kandırmaca yoluna gittik...

Bu kaçış dizisini art arda yazarak tuhaf şeyler de yapabilirsiniz...

```
>>> print("merhaba\b\b ")

merha a
```

Burada *\b\b* kaçış dizilerinin yaptığı şey, sola doğru iki karakter atlayıp, ulaştığı noktaya bir adet boşluk karakteri yerleştirmektir. Şu örnekte durum biraz daha açık görünecektir:

```
>>> print("merhaba\b\bf")
```

```
merhafa
```

Yani bu karakter dizisini kullanarak şöyle saçma bir şey de yapabilirsiniz:

```
>>> print("merhaba\b\b\b\b\b\bgülegüle")

gülegüle
```

Gördüğünüz gibi, bu karakter dizisi rahatlıkla suyu çıkarılabilecek bir araçtır. Bu kaçış dizisinin Python'da çok nadir kullanıldığı bilgisini vererek, yolumuza devam edelim...

Küçük Unicode (\u)

Tıpkı bundan önceki kaçış dizileri gibi, karakter dizileri içindeki varlığı konusunda dikkatli olmamız gereken bir başka kaçış dizisi de *\u* adlı kaçış dizisidir. Eğer bu kaçış dizisini tanımaz ve dikkatli kullanmazsak, yazdığımız programlar tespit etmesi çok güç hatalar üretebilir.

Örneğin şöyle bir çıktı vermek istediğinizi düşünün:

Dosya konumu: C:\users\zeynep\gizli\dosya.txt

Bu çıktıyı normal yollardan vermeye çalışırsak Python bize bir hata mesajı gösterecektir:

```
>>> print("Dosya konumu: C:\users\zeynep\gizli\dosya.txt")

  File "<stdin>", line 1
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in
position 16-18: truncated \uXXXX escape
```

Belki sağda solda 'UNICODE' diye bir şey duymuşsunuzdur. Eğer şimdiye kadar böyle bir şey duymadıysanız veya duyduysanız bile ne olduğunu bilmiyorsanız hiç ziyanı yok. Birkaç bölüm sonra bunun ne anlama geldiğini bütün ayrıntılarıyla anlatacağız. Biz şimdilik sadece şunu bilelim: UNICODE, karakterlerin, harflerin, sayıların ve bilgisayar ekranında gördüğümüz öteki bütün işaretlerin her biri için tek ve benzersiz bir numaranın tanımlandığı bir sistemdir. Bu sistemde, 'kod konumu' (*code point*) adı verilen bu numaralar özel bir şekilde gösterilir. Örneğin 'ı' harfi UNICODE sisteminde şu şekilde temsil edilir:

```
u+0131
```

Aynı şekilde 'a' harfi bu sistemde şu kod konumu ile gösterilir:

```
u+0061
```

Python programlama dilinde ise, yukarıdaki kod konumu düzeni şöyle gösterilir:

```
\\u0131
```

Gördüğünüz gibi, Python UNICODE sistemindeki her bir kod konumunu gösterebilmek için, önce *\u* şeklinde bir kaçış dizisi tanımlıyor, ardından UNICODE sisteminde + işaretinden sonra gelen sayıyı bu kaçış dizisinin hemen sağına ekliyor. Gelin kendi kendimize birkaç deneme çalışması yapalım:

```
>>> '\u0130'

'İ'

>>> '\u0070'

'p'

>>> "\ufdsf"

  File "<stdin>", line 1
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in
position 0-4: truncated \uXXXX escape
```

Gördüğünüz gibi, eğer *\u* kaçış dizisinden sonra doğru bir kod konumu belirtmezsek Python bize bir hata mesajı gösteriyor...

Bu hata mesajının, biraz önce print("Dosya konumu: C:\users\zeynep\gizli\dosya.txt") kodunu yazdıktan sonra aldığımız hata ile aynı olduğuna dikkat edin. Tıpkı *\ufdsf* örneğinde olduğu gibi, *\users* ifadesi de varolan bir UNICODE kod konumuna karşılık gelmediği için, Python'ın hata vermekten başka çaresi kalmıyor.

Biz bu örnekte 'users' kelimesini kullanmaya çalışıyoruz, ama 'u' harfinden hemen önce gelen \ kaçış dizisi nedeniyle, hiç farkında olmadan Python açısından önemli bir karakter dizisi (*\u*) meydana getirmiş oluyoruz. O yüzden, böyle can sıkıcı hatalarla karşılaşmamak için olası kaçış dizilerine karşı her zaman uyanık olmamız gerekiyor.

Pekı biz bu kaçış dizisi yüzünden, yazdığımız programlarda *Dosya konumu: C:\users\zeynep\gizli\dosya.txt")* gibi bir çıktı veremeyecek miyiz?

Verebileceğimizi, ama bunun bir yolu yordamı olduğunu biliyorsunuz. Biz yine de tekrar edelim:

```
>>> print("Dosya konumu: C:\\users\\zeynep\\gizli\\dosya.txt")

Dosya konumu: C:\users\zeynep\gizli\dosya.txt
```

Gördüğünüz gibi, karakter dizisi içinde geçen bütün \ işaretlerini çiftleyerek sorunumuzu çözdük. Buradaki gibi bir sorunla karşılaşmamak için, dizin adlarını ayırırken ters taksim işareti yerine düz taksim işaretini kullanmayı da tercih edebilirsiniz:

```
>>> print("Dosya konumu: C:/users/zeynep/gizli/dosya.txt")
```

Biraz sonra bu sorunu halletmenin üçüncü ve daha kolay bir yönteminden daha söz edeceğiz. Ama biz şimdilik bu kaçış dizisini bir kenara bırakıp başka bir kaçış dizisini incelemeye geçelim.

Büyük Unicode (\U)

Bu kaçış dizisi biraz önce gördüğümüz *\u* adlı kaçış dizisiyle hemen hemen aynı anlama gelir. Bu kaçış dizisi de, tıpkı *\u* gibi, UNICODE kod konumlarını temsil etmek için kullanılır. Ancak *U* ile gösterilen kod konumları *u* ile gösterilenlere göre biraz daha uzundur. Örneğin, hatırlarsanız *u* kaçış dizisini kullanarak 'ı' harfinin UNICODE kod konumunu şöyle temsil ediyorduk:

```
>>> '\u0131'

'ı'
```

Eğer aynı kod konumunu *U* adlı kaçış dizisi ile göstermek istersek şöyle bir şey yazmamız gerekir:

```
>>> '\U00000131'
```

Gördüğünüz gibi, burada *\U* kaçış dizisinden sonra gelen kısım toplam 8 haneli bir sayıdan oluşuyor. *u* kaçış dizisinde ise bu kısmı toplam 4 haneli bir sayı olarak yazıyorduk. İşte *\u* kaçış dizisi ile *U* kaçış dizisi arasındaki fark budur. *u* kaçış dizisi hakkında söylediğimiz öteki her şey *U* kaçış dizisi için de geçerlidir.

Uzun Ad (\N)

UNICODE sistemi ile ilgili bir başka kaçış dizisi de *\N* adlı kaçış dizisidir.

Dediğimiz gibi, UNICODE sistemine ilişkin ayrıntılardan ilerleyen derslerde söz edeceğiz, ama bu sistemle ilgili ufak bir bilgi daha verelim.

UNICODE sisteminde her karakterin tek ve benzersiz bir kod konumu olduğu gibi, tek ve benzersiz bir de uzun adı vardır. Örneğin 'a' harfinin UNICODE sistemindeki uzun adı şudur:

```
LATIN SMALL LETTER A
```

Bir karakterin UNICODE sistemindeki uzun adını öğrenmek için *unicodedata* adlı bir modülden yararlanabilirsiniz:

```
>>> import unicodedata
>>> unicodedata.name('a')

LATIN SMALL LETTER A

>>> unicodedata.name('Ş')
```

Bu arada, daha önce de söylediğimiz gibi, bu 'modül' kavramına şimdilik takılmayın. İlerde modülleri ayrıntılı olarak inceleyeceğiz. Şimdilik *unicodedata* denen şeyin, (tıpkı daha önce örneklerini gördüğümüz *os*, *sys* ve *keyword* gibi) bir modül olduğunu ve bu modül içindeki *name* adlı bir fonksiyonu kullanarak, parantez içinde belirttiğimiz herhangi bir karakterin UNICODE sistemindeki uzun adını elde edebileceğimizi bilelim yeter.

İşte *\N* kaçış dizisi bu uzun isimleri, Python programlarımızda kullanma imkanı verir bize. Bu kaçış dizisini, karakterlerin UNICODE sistemindeki uzun adları ile birlikte kullanarak asıl karakterleri elde edebiliriz. Dikkatlice bakın:

```
>>> print("\N{LATIN SMALL LETTER A}")

a

>>> print("\N{LATIN CAPITAL LETTER S WITH CEDILLA}")

Ş

>>> print("\Nisan")

  File "<stdin>", line 1
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in
position 0-1: malformed \N character escape
```

Gördüğünüz gibi, herhangi bir karşılığı olmayan bir uzun ad belirttiğimizde Python bize bir hata mesajı gösteriyor. Çünkü Python *\N* kaçış dizisinin hemen ardından *{* işaretinin getirilmesini ve sonra da UNICODE sistemi dahilinde geçerli bir uzun ad belirtilmesini bekliyor. Yukarıdaki örnekte *\N* kaçış dizisinden sonra *{* işareti yok. Zaten *\N* kaçış dizisinin hemen ardından gelen 'isan' ifadesi de doğru bir uzun ada işaret etmiyor. Dolayısıyla da Python'ın bize hata mesajı göstermekten başka çaresi kalmıyor...

\u, *\U* ve *\N* kaçış dizileri, UNICODE sistemi ile ilgili çalışmalar yapmak isteyen programcılar için Python programlama dilinin sunduğu faydalı araçlardan yalnızca birkaçıdır. Ancak bu araçların sizin işinize yaramayacağını asla düşünmeyin. Zira *\u*, *\U* ve *\N* kaçış dizileri ile ilgili yukarıdaki durum hiç beklemediğiniz bir anda sizi de vurabilir. Çünkü bu kaçış dizilerinin oluşturduğu risk hiç de öyle nadir karşılaşılacak bir sorun değildir.

Bildiğiniz gibi Windows 7'de kullanıcının dosyalarını içeren dizin adı *C:\Users\kullanıcı_adı* şeklinde gösteriliyor. Dolayısıyla Windows kullananlar UNICODE kaçış dizilerinden kaynaklanan bu tuzağa her an düşebilir. Ya da eğer adınız 'u' veya 'n' harfi ile başlıyorsa yine bu tuzağa düşme ihtimaliniz epey yüksek olacak, *C:\Users\umut* veya *C:\Users\Nihat* gibi bir dizin adı belirtirken çok dikkatli olmanız gerekecektir. Zira özellikle dosyalar üzerinde işlem yaparken, bu tür dizin adlarını sık sık kullanmak durumunda kalacaksınız. Bu yüzden, alelade bir kelime yazdığınızı zannederken hiç

farkında olmadan bir kaçış dizisi tanımlıyor olma ihtimalini her zaman göz önünde bulundurmalı ve buna uygun önlemleri almış olmalısınız.

Onaltılı Karakter (\x)

'x' harfi de \ işareti ile birleştiğinde özel anlam kazanarak bir kaçış dizisi meydana getirir.

\x kaçış dizisini kullanarak, onaltılı (*hexadecimal*) sayma sistemindeki bir sayının karakter karşılığını gösterebilirsiniz. Dikkatlice bakın:

```
>>> "\x41"

'A'
```

Onaltılı sayma sistemindeki *41* sayısı 'A' harfine karşılık gelir. Eğer hangi karakterlerin hangi sayılara karşılık geldiğini merak ediyorsanız http://www.ascii.cl/ adresindeki tabloyu inceleyebilirsiniz. Bu tabloda 'hex' sütunu altında gösterilen sayılar onaltılı sayılar olup, 'symbol' sütununda gösterilen karakterlere karşılık gelirler. Örneğin 'hex' sütunundaki *4E* sayısı 'symbol' sütunundaki 'N' harfine karşılık gelir. Bu durumu Python'la da teyit edebilirsiniz:

```
>>>"\x4E"

N
```

Eğer sayılarla karakterler arasındaki bağlantının tam olarak ne olduğunu bilmiyorsanız hiç endişe etmeyin. Birkaç bölüm sonra sayılarla karakterler arasında nasıl bir bağ olduğunu gayet ayrıntılı bir şekilde anlatacağız. Biz şimdilik yalnızca \x karakter dizisinin özel bir kaçış dizisine karşılık geldiğini ve bu kaçış dizisini karakter dizileri içinde kullanırken dikkatli olmamız gerektiğini bilelim yeter:

```
>>> print("C:\Users\Ayşe\xp_dosyaları")

  File "<stdin>", line 1
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in
position 2-4: truncated \UXXXXXXXX escape
```

Gördüğünüz gibi, Python \x ifadesinden sonra onaltılı bir sayı belirtmenizi bekliyor. Halbuki biz burada \x ifadesini 'xp_dosyaları' adlı dizini gösterebilmek için kullanmıştık. Ama görünüşe göre yanlışlıkla Python için özel bir anlam ifade eden bir karakter dizisi oluşturmuşuz...

Etkisizleştirme (r)

Dediğimiz gibi, Python'daki en temel kaçış dizisi \ işaretidir. Bu işaret bazı başka harflerle birleşerek yeni kaçış dizileri de oluşturabilir.

Python'da \ işaretinin dışında temel bir kaçış dizisi daha bulunur. Bu kaçış dizisi 'r' harfidir. Şimdi bu kaçış dizisinin nasıl kullanılacağını ve ne işe yaradığını inceleyelim:

Şöyle bir çıktı vermek istediğimizi düşünün:

```
Kurulum dizini: C:\\aylar\\nisan\\toplam masraf
```

Bildiğimiz yoldan bu çıktıyı vermeye çalışırsak neler olacağını adınız gibi biliyorsunuz:

```
>>> print("Kurulum dizini: C:\aylar\nisan\toplam masraf")

Kurulum dizini: C:ylar
isan	oplam masraf
```

Not

Eğer Windows üzerinde çalışıyorsanız bu komutu verdikten sonra bir !bip! sesi de duymuş olabilirsiniz...

Python tabii ki, karakter dizisi içinde geçen '**a**ylar', '**n**isan', ve '**t**oplam masraf' ifadelerinin ilk karakterlerini yanlış anladı! *\a*, *\n* ve *\t* gibi ifadeler Python'ın gözünde birer kaçış dizisi. Dolayısıyla Python *\a* karakterlerini görünce bir !bip! sesi çıkarıyor, *\n* karakterlerini görünce satır başına geçiyor ve *\t* karakterlerini görünce de *Tab* tuşuna basılmış gibi bir tepki veriyor. Sonuç olarak da yukarıdaki gibi bir çıktı üretiyor.

Daha önce bu durumu şöyle bir kod yazarak engellemiştik:

```
>>> print("Kurulum dizini: C:\\aylar\\nisan\\toplam masraf")

Kurulum dizini: C:\aylar\nisan\toplam masraf
```

Burada, \ işaretlerinin her birini çiftleyerek sorunun üstesinden geldik. Yukarıdaki yöntem doğru ve kabul görmüş bir çözümdür. Ama bu sorunun üstesinden gelmenin çok daha basit ve pratik bir yolu var. Bakalım:

```
>>> print(r"Kurulum dizini: C:\aylar\nisan\toplam masraf")

Kurulum dizini: C:\aylar\nisan\toplam masraf
```

Gördüğünüz gibi, karakter dizisinin baş kısmının dış tarafına bir adet *r* harfi yerleştirerek sorunun üstesinden geliyoruz. Bu kaçış dizisinin, kullanım açısından öteki kaçış dizilerinden farklı olduğuna dikkat edin. Öteki kaçış dizileri karakter dizisinin içinde yer alırken, bu kaçış dizisi karakter dizisinin dışına yerleştiriliyor.

Bu kaçış dizisinin tam olarak nasıl işlediğini görmek için dilerseniz bir örnek daha verelim:

```
>>> print("Kaçış dizileri: \, \n, \t, \a, \\, r")

Kaçış dizileri: \,
, , , \, r
```

Burada da Python bizim yapmak istediğimiz şeyi anlayamadı ve karakter dizisi içinde geçen kaçış dizilerini doğrudan ekrana yazdırmak yerine bu kaçış dizilerinin işlevlerini yerine getirmesine izin verdi. Tıpkı biraz önceki örnekte olduğu gibi, istersek kaçış dizilerini çiftleyerek bu sorunu aşabiliriz:

```
>>> print("Kaçış dizileri: \\, \\n, \\t, \\a, \\\, r")

Kaçış dizileri: \, \n, \t, \a, \\, r
```

Ama tabii ki bunun çok daha kolay bir yöntemi olduğunu biliyorsunuz:

```
>>> print(r"Kaçış dizileri: \, \n, \t, \a, \\, r")

Kaçış dizileri: \, \n, \t, \a, \\, r
```

Gördüğünüz gibi, karakter dizisinin başına getirdiğimiz *r* kaçış dizisi, karakter dizisi içinde geçen kaçış dizilerinin işlevlerini yerine getirmesine engel olarak, istediğimiz çıktıyı elde etmemizi sağlıyor.

Bu arada bu kaçış dizisini, daha önce öğrendiğimiz *\r* adlı kaçış dizisi ile karıştırmamaya dikkat ediyoruz.

Python'daki bütün kaçış dizilerinden söz ettiğimize göre, konuyu kapatmadan önce önemli bir ayrıntıdan söz edelim.

Python'da karakter dizilerinin sonunda sadece çift sayıda \ işareti bulunabilir. Tek sayıda \ işareti kullanıldığında karakter dizisini bitiren tırnak işareti etkisizleşeceği için çakışma sorunu ortaya çıkar. Bu etkisizleşmeyi, karakter dizisinin başına koyduğunuz 'r' kaçış dizisi de engelleyemez. Yani:

```
>>> print("Kaçış dizisi: \")
```

Bu şekilde bir tanımlama yaptığımızda Python bize bir hata mesajı gösterir. Çünkü kapanış tırnağının hemen öncesine yerleştirdiğimiz \ kaçış dizisi, Python'ın karakter dizisini kapatan tırnak işaretini görmezden gelmesine yol açarak bu tırnağı etkisizleştiriyor. Böylece sanki karakter dizisini tanımlarken kapanış tırnağını hiç yazmamışız gibi bir sonuç ortaya çıkıyor:

```
>>> print("Kaçış dizisi: \")
  File "<stdin>", line 1
    print("Kaçış dizisi: \")
                          ^
SyntaxError: EOL while scanning string literal
```

Üstelik bu durumu, *r* adlı kaçış dizisi de engelleyemiyor:

```
>>> print(r"Kaçış dizisi: \")
  File "<stdin>", line 1
    print(r"Kaçış dizisi: \")
                          ^
SyntaxError: EOL while scanning string literal
```

Çözüm olarak birkaç farklı yöntemden yararlanabilirsiniz. Mesela karakter dizisini kapatmadan önce karakter dizisinin sonundaki \ işaretinin sağına bir adet boşluk karakteri yerleştirmeyi deneyebilirsiniz:

```
>>> print("Kaçış dizisi: \ ")
```

Veya kaçış dizisini çiftleyebilirsiniz:

```
>>> print("Kaçış dizisi: \\")
```

Ya da karakter dizisi birleştirme yöntemlerinden herhangi birini kullanabilirsiniz:

```
>>> print("Kaçış dizisi: " + "\\")
>>> print("Kaçış dizisi:", "\\")
>>> print("Kaçış dizisi: " "\\")
```

Böyle bir durumla ilk kez karşılaştığınızda bunun Python programlama dilinden kaynaklanan bir hata olduğunu düşünebilirsiniz, ancak bu durum Python'ın resmi internet sitesinde 'Sıkça Sorulan Sorular' bölümüne alınacak kadar önemli bir tasarım tercihidir: http://goo.gl/i3tkk

Sayfa Başı (\f)

\f artık günümüzde pek kullanılmayan bir kaçış dizisidir. Bu kaçış dizisinin görevi, özellikle eski yazıcılarda, bir sayfanın sona erip yeni bir sayfanın başladığını göstermektir. Dolayısıyla eski model yazıcılar, bu karakteri gördükleri noktada mevcut sayfayı sona erdirip yeni bir sayfaya geçer.

Bu kaçış dizisinin tam olarak ne işe yaradığını test etmek için şu kodları çalıştırın:

```
>>> f = open("deneme.txt", "w")
>>> print("deneme\fdeneme", file=f)
>>> f.close()
```

Şimdi bu kodlarla oluşturduğunuz *deneme.txt* adlı dosyayı LibreOffice veya Microsoft Word gibi bir programla açın. 'deneme' satırlarının iki farklı sayfaya yazdırıldığını göreceksiniz. Bu arada, eğer Microsoft Word dosyayı açarken bir hata mesajı gösterirse, o hata mesajına birkaç kez 'tamam' diyerek hata penceresini kapatın. Dosya normal bir şekilde açılacaktır.

Dediğimiz gibi, bu kaçış dizisi artık pek kullanılmıyor. Ama yine de bu kaçış dizisine karşı da uyanık

olmalısınız. Çünkü bu kaçış dizisi de beklemediğiniz çıktılar almanıza yol açabilir. Mesela şu örneğe bir bakalım:

```
>>> "\fırat"

'\x0cırat'
```

Gördüğünüz gibi, siz aslında '\fırat' yazmak isterken, Python bu kelimenin baş tarafındaki *\f* karakter dizisini bir kaçış dizisi olarak değerlendirip ona göre bir çıktı verdi.

Bütün bu anlattıklarımızın ardından, kaçış dizilerinin, birleştirildikleri karakterlerin farklı bir anlam yüklenmesini sağlayan birtakım işaretler olduğunu anlıyoruz. Örneğin \ işareti ' (tek tırnak) işareti ile bir araya gelerek, tek tırnak işaretinin karakter dizisi tanımlama dışında başka bir anlam yüklenmesini sağlıyor. Aynı şekilde yine \ işareti " (çift tırnak) işareti ile birleşerek çift tırnak işaretinin de karakter dizisi tanımlama dışında bir anlama kavuşmasını sağlıyor. Böylece tırnak işaretlerini karakter dizileri içinde rahatlıkla kullanabiliyoruz.

Ya da yine \ işareti 'n' harfi ile bir araya gelip, bu harfin satır başına geçilmesini sağlayan bir kaçış dizisi oluşturmasını mümkün kılıyor. Veya aynı işaret 't' harfiyle birleşip, öğeler arasında sekme oluşturulmasını sağlayabiliyor. Bu araçlar sayesinde ekrana yazdırdığımız bir metnin akışını kontrol etme imkanına kavuşuyoruz.

Kaçış Dizilerine Toplu Bakış

Biraz sonra bu önemli konuyu kapatacağız. Ama dilerseniz kapatmadan önce, bu bölümde öğrendiğimiz kaçış dizilerini şöyle bir topluca görelim:

Kaçış Dizisi	**Anlamı**
\'	Karakter dizisi içinde tek tırnak işaretini kullanabilmemizi sağlar.
\"	Karakter dizisi içinde çift tırnak işaretini kullanabilmemizi sağlar.
**	Karakter dizisi içinde \ işaretini kullanabilmemizi sağlar.
\n	Yeni bir satıra geçmemizi sağlar.
\t	Karakterler arasında sekme boşluğu bırakmamızı sağlar.
\u	UNICODE kod konumlarını gösterebilmemizi sağlar.
\U	UNICODE kod konumlarını gösterebilmemizi sağlar.
\N	Karakterleri UNICODE adlarına göre kullanabilmemizi sağlar.
\x	Onaltılı sistemdeki bir sayının karakter karşılığını gösterebilmemizi sağlar.
\a	Destekleyen sistemlerde, kasa hoparlöründen bir 'bip' sesi verilmesini sağlar.
\r	Aynı satırın başına dönülmesini sağlar.
\v	Destekleyen sistemlerde düşey sekme oluşturulmasını sağlar.
\b	Kendisinden önce gelen karakterin silinmesini sağlar.
\f	Yeni bir sayfaya geçilmesini sağlar.

r	Karakter dizisi içinde kaçış dizilerini kullanabilmemizi sağlar.

Kaçış dizileriyle ilgili son olarak şunu söyleyebiliriz: Kaçış dizileri, görmezden gelebileceğiniz, ‘öğrenmesem de olur,’ diyebileceğiniz önemsiz birtakım işaretler değildir. Bu konu boyunca verdiğimiz örneklerden de gördüğünüz gibi, kaçış dizileri, kullanıcıya göstereceğiniz metinlerin biçimini doğrudan etkiliyor. Bütün bu örnekler, bu kaçış dizilerinin yersiz veya yanlış kullanılmasının ya da bunların bir metin içinde gözden kaçmasının, yazdığınız programların hata verip çökmesine, yani programınızın durmasına sebep olabileceğini de gösteriyor bize.

Böylece bir bölümü daha bitirmiş olduk. Artık Python’la ‘gerçek’ programlar yazmamızın önünde hiçbir engel kalmadı.

başa dön

Temel Program Kaydetme ve Çalıştırma Mantığı

Bu noktaya kadar bütün işlerimizi Python'ın etkileşimli kabuğu üzerinden hallettik. Her ne kadar etkileşimli kabuk son derece kullanışlı bir ortam da olsa, bizim asıl çalışma alanımız değildir. Daha önce de dediğimiz gibi, etkileşimli kabuğu genellikle ufak tefek Python kodlarını test etmek için kullanacağız. Ama asıl programlarımızı tabii ki etkileşimli kabuğa değil, program dosyasına yazacağız.

Ne dedik? Özellikle küçük kod parçaları yazıp bunları denemek için etkileşimli kabuk mükemmel bir ortamdır. Ancak kodlar çoğalıp büyümeye başlayınca bu ortam yetersiz gelmeye başlayacaktır. Üstelik tabii ki yazdığınız kodları bir yere kaydedip saklamak isteyeceksiniz. İşte burada metin düzenleyiciler devreye girecek.

Python kodlarını yazmak için istediğiniz herhangi bir metin düzenleyiciyi kullanabilirsiniz. Hatta Notepad bile olur. Ancak Python kodlarını ayırt edip renklendirebilen bir metin düzenleyici ile yola çıkmak her bakımdan hayatınızı kolaylaştıracaktır.

Not

Python kodlarınızı yazmak için Microsoft Word veya OpenOffice.Org OOWriter gibi, belgeleri ikili (*binary*) düzende kaydeden programlar uygun değildir. Kullanacağınız metin düzenleyici, belgelerinizi düz metin (*plain text*) biçiminde kaydedebilmeli.

Biz bu bölümde farklı işletim sistemlerinde, metin düzenleyici kullanılarak Python programlarının nasıl yazılacağını ve bunların nasıl çalıştırılacağını tek tek inceleyeceğiz.

Daha önce de söylediğimiz gibi, hangi işletim sistemini kullanıyor olursanız olun, hem Windows hem de GNU/Linux başlığı altında yazılanları okumalısınız.

Dilerseniz önce GNU/Linux ile başlayalım:

GNU/Linux

Eğer kullandığınız sistem GNU/Linux'ta Unity veya GNOME masaüstü ortamı ise başlangıç düzeyi için Gedit adlı metin düzenleyici yeterli olacaktır.

Eğer kullandığınız sistem GNU/Linux'ta KDE masaüstü ortamı ise Kwrite veya Kate adlı metin düzenleyicilerden herhangi birini kullanabilirsiniz. Şu aşamada kullanım kolaylığı ve sadeliği nedeniyle Kwrite önerilebilir.

İşe yeni bir Gedit belgesi açarak başlayalım. Yeni bir Gedit belgesi açmanın en kolay yolu *Alt+F2*

tuşlarına bastıktan sonra çıkan ekranda:

```
gedit
```

yazıp *Enter* düğmesine basmaktır.

Eğer Gedit yerine mesela Kwrite kullanıyorsanız, yeni bir Kwrite belgesi oluşturmak için *Alt+F2* tuşlarına bastıktan sonra:

```
kwrite
```

komutunu vermelisiniz. Elbette kullanacağınız metin düzenleyiciye, komut vermek yerine, dağıtımınızın menüleri aracılığıyla da ulaşabilirsiniz.

Python kodlarımızı, karşımıza çıkan bu boş metin dosyasına yazıp kaydedeceğiz.

Aslında kodları metin dosyasına yazmakla etkileşimli kabuğa yazmak arasında çok fazla fark yoktur. Dilerseniz hemen bir örnek vererek ne demek istediğimizi anlatmaya çalışalım:

1. Boş bir Gedit ya da Kwrite belgesi açıyoruz ve bu belgeye şu kodları eksiksiz bir şekilde yazıyoruz:

```
tarih = "02.01.2012"
gün = "Pazartesi"
vakit = "öğleden sonra"

print(tarih, gün, vakit, "buluşalım", end=".\n")
```

2. Bu kodları yazıp bitirdikten sonra dosyayı masaüstüne *randevu.py* adıyla kaydedelim.

3. Sonra işletim sistemimize uygun bir şekilde komut satırına ulaşalım.

4. Ardından komut satırı üzerinden masaüstüne gelelim. (Bunun nasıl yapılacağını hatırlıyorsunuz, değil mi?)

5. Son olarak şu komutla programımızı çalıştıralım:

```
python3 randevu.py
```

Şöyle bir çıktı almış olmalıyız:

```
02.01.2012 Pazartesi öğleden sonra buluşalım.
```

Eğer bu çıktı yerine bir hata mesajı alıyorsanız bunun birkaç farklı sebebi olabilir:

1. Kodlarda yazım hatası yapmış olabilirsiniz. Bu ihtimali bertaraf etmek için yukarıdaki kodlarla kendi yazdığınız kodları dikkatlice karşılaştırın.

2. Kodlarınızı kaydettiğiniz *randevu.py* adlı dosyanın adını yanlış yazmış olabilirsiniz. Dolayısıyla python3 randevu.py komutu, var olmayan bir dosyaya atıfta bulunuyor olabilir.
3. python3 randevu.py komutunu verdiğiniz dizin konumu ile *randevu.py* dosyasının bulunduğu dizin konumu birbirinden farklı olabilir. Yani siz *randevu.py* dosyasını masaüstüne kaydetmişsinizdir, ama python3 randevu.py komutunu yanlışlıkla başka bir dizin altında veriyor olabilirsiniz. Bu ihtimali ortadan kaldırmak için, önceki derslerde öğrendiğimiz yöntemleri kullanarak hangi dizin altında bulunduğunuzu kontrol edin. O anda içinde bulunduğunuz dizinin içeriğini listeleyerek, *randevu.py* dosyasının orada görünüp görünmediğini kontrol edebilirsiniz. Eğer program dosyanız bu listede görünmüyorsa, elbette python3 randevu.py komutu çalışmayacaktır.
4. Geçen derslerde anlattığımız şekilde Python3'ü kaynaktan *root* haklarıyla derlemenize rağmen, derleme sonrasında */usr/bin/* dizini altına *python3* adlı bir sembolik bağ oluşturmadığınız için python3 komutu çalışmıyor olabilir.
5. Eğer Python3'ü yetkisiz kullanıcı olarak derlediyseniz, *$HOME/python/bin/* dizini altında hem python3 adlı bir sembolik bağ oluşturmuş, hem de *$HOME/python/bin/* dizinini YOL'a (*PATH*) eklemiş olmanız gerekirken bunları yapmamış olabilirsiniz.
6. Asla unutmayın, Python'ın etkileşimli kabuğunu başlatmak için hangi komutu kullanıyorsanız, *randevu.py* dosyasını çalıştırmak için de aynı komutu kullanacaksınız. Yani eğer Python'ın etkileşimli kabuğunu python3.3 gibi bir komutla çalıştırıyorsanız, programınızı da python3.3randevu.py şeklinde çalıştırmanız gerekir. Aynı şekilde, eğer etkileşimli kabuğu mesela python (veya py3) gibi bir komutla çalıştırıyorsanız, programınızı da python randevu.py (veya py3 randevu.py) şeklinde çalıştırmalısınız. Neticede etkileşimli kabuğu çalıştırırken de, bir program dosyası çalıştırırken de aslında temel olarak Python programlama dilini çalıştırmış oluyorsunuz. Python programını çalıştırırken bir dosya adı belirtmezseniz, yani Python'ı başlatan komutu tek başına kullanırsanız etkileşimli kabuk çalışmaya başlar. Ama eğer Python'ı başlatan komutla birlikte bir program dosyası ismi de belirtirseniz, o belirttiğiniz program dosyası çalışmaya başlar.

Kodlarınızı düzgün bir şekilde çalıştırabildiğinizi varsayarak yolumuza devam edelim...

Gördüğünüz gibi, kod dosyamızı çalıştırmak için python3 komutundan yararlanıyoruz. Bu arada tekrar etmekte fayda var: Python'ın etkileşimli kabuğunu çalıştırmak için hangi komutu kullanıyorsanız, dosyaya kaydettiğiniz programlarınızı çalıştırmak için de aynı komutu kullanacaksınız. Ben bu belgelerde sizin Python3'ün etkileşimli kabuğunu python3 komutuyla çalıştırdığınızı varsayacağım.

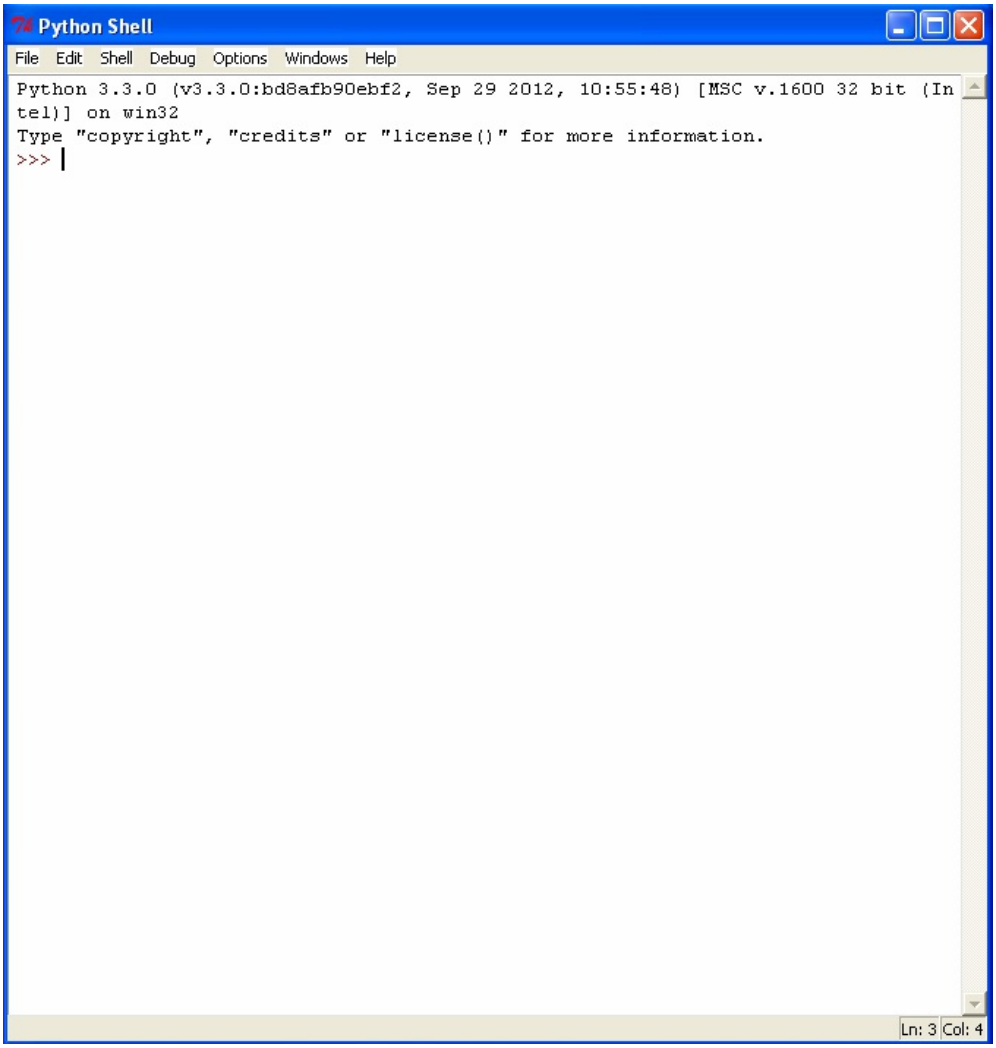
Gelelim Windows kullanıcılarına...

Windows

Daha önce de söylediğimiz gibi, Python kodlarımızı yazmak için istediğimiz bir metin düzenleyiciyi kullanabiliriz. Hatta Notepad'i bile kullansak olur. Ancak Notepad'den biraz daha gelişmiş bir metin düzenleyici ile başlamak işinizi kolaylaştıracaktır.

Python programlama dilini öğrenmeye yeni başlayan Windows kullanıcıları için en uygun metin düzenleyici IDLE'dır. *Başlat > Tüm Programlar > Python*3.3 *> IDLE (Python GUI)* yolunu takip ederek IDLE'a ulaşabilirsiniz.

IDLE'ı açtığınızda şöyle bir ekranla karşılaşacaksınız:

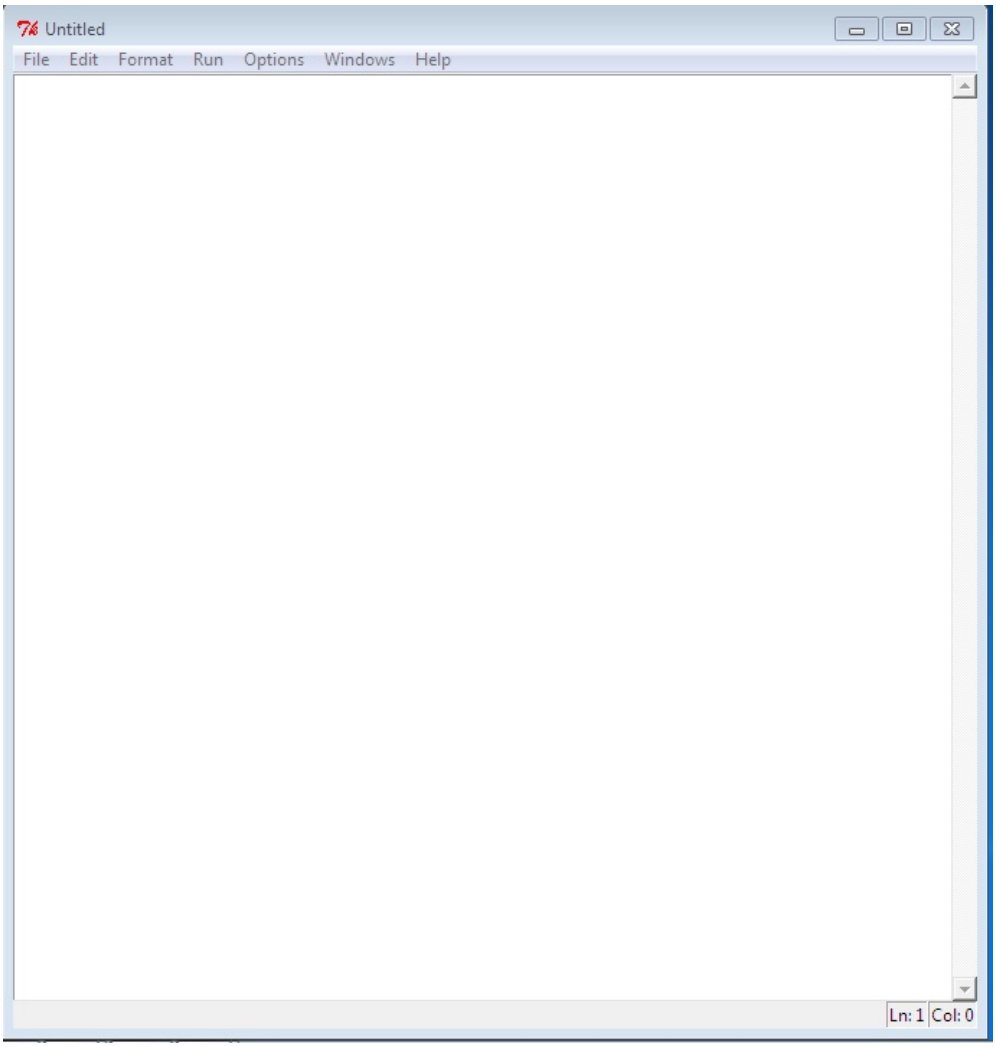


Aslında bu ekran size bir yerlerden tanıdık geliyor olmalı. Dikkat ederseniz beyaz ekranın en sonunda bordo renkli bir >>> işareti var. Evet, tahmin ettiğiniz gibi, burası aslında Python'ın etkileşimli kabuğudur. Yani o siyah etkileşimli kabuk ekranında ne yapabilirseniz burada da aynı şeyi yapabilirsiniz. Dilerseniz kendi kendinize bazı denemeler yapın. Ama şu anda biz IDLE'ın bu özelliğini değil, metin düzenleyici olma özelliğini kullanacağız. O yüzden yolumuza devam ediyoruz.

Not

Dediğimiz gibi, yukarıda görünen ekran aslında Python'ın etkileşimli kabuğudur. Dolayısıyla biraz sonra göstereceğimiz kodları buraya yazmayacağız. Python programlama diline yeni başlayanların en sık yaptığı hatalardan biri de, kaydetmek istedikleri kodları yukarıda görünen ekrana yazmaya çalışmalarıdır. Unutmayın, Python'ın etkileşimli kabuğunda ne yapabiliyorsanız, IDLE'ı açtığınızda ilk karşınıza çıkan ekranda da onu yapabilirsiniz. Python'ın etkileşimli kabuğunda yazdığınız kodlar etkileşimli kabuğu kapattığınızda nasıl kayboluyorsa, yukarıdaki ekrana yazdığınız kodlar da IDLE'ı kapattığınızda kaybolur...

Bir önceki ekranda sol üst köşede *File* [Dosya] menüsü görüyorsunuz. Oraya tıklayın ve menü içindeki *New Window* [Yeni Pencere] düğmesine basın. Şöyle bir ekranla karşılaşacaksınız:



İşte Python kodlarımızı bu beyaz ekrana yazacağız. Şimdi bu ekrana şu satırları yazalım:

```
tarih = "02.01.2012"
gün = "Pazartesi"
vakit = "öğleden sonra"

print(tarih, gün, vakit, "buluşalım", end=".\n")
```

Bu noktadan sonra yapmamız gereken şey dosyamızı kaydetmek olacak. Bunun için *File* > *Save as* yolunu takip ederek programımızı masaüstüne *randevu.py* adıyla kaydediyoruz.

Şu anda programımızı yazdık ve kaydettik. Artık programımızı çalıştırabiliriz. Bunun için IDLE'da *Run* > *Run Module* yolunu takip etmeniz veya kısaca *F5* tuşuna basmanız yeterli olacaktır. Bu iki yöntemden birini kullanarak programınızı çalıştırdığınızda şöyle bir çıktı elde edeceksiniz:

```
02.01.2012 Pazartesi öğleden sonra buluşalım.
```

Tebrikler! İlk Python programınızı yazıp çalıştırdınız... Eğer çalıştıramadıysanız veya yukarıdaki çıktı

yerine bir hata mesajı aldıysanız muhtemelen kodları yazarken yazım hatası yapmışsınızdır. Kendi yazdığınız kodları buradaki kodlarla dikkatlice karşılaştırıp tekrar deneyin.

Şimdi gelin isterseniz yukarıda yazdığımız kodları şöyle bir kısaca inceleyelim.

Programımızda üç farklı değişken tanımladığımıza dikkat edin. Bu değişkenler *tarih*, *gün* ve *vakit* adlı değişkenlerdir. Daha sonra bu değişkenleri birbiriyle birleştiriyoruz. Bunun için print() fonksiyonundan nasıl yararlandığımızı görüyorsunuz. Ayrıca print() fonksiyonunu kullanış biçimimize de dikkat edin. Buradaki *end* parametresinin anlamını ve bunun ne işe yaradığını artık gayet iyi biliyorsunuz. *end* parametresi yardımıyla cümlenin en sonuna bir adet nokta yerleştirip, *\n* adlı kaçış dizisi yardımıyla da bir alt satıra geçiyoruz.

Böylece basit bir Python programının temel olarak nasıl yazılıp bir dosyaya kaydedileceğini ve bu programın nasıl çalıştırılacağını öğrenmiş olduk.

başa dön

Program Çalıştırmada Alternatif Yöntemler

Python programlarını nasıl çalıştıracağımızı bir önceki bölümde anlatmıştık. Gerçekten de Python'da bir programı çalıştırmanın en temel yöntemi bir önceki bölümde anlattığımız gibidir. Ama bunlar Python programlarını çalıştırmanın tek yolu değildir. Python programlarını çalıştırmanın alternatif yöntemleri de bulunur. İşte biz bu bölümde farklı işletim sistemlerine göre bu alternatif yöntemlerden söz edeceğiz.

Dilerseniz önce GNU/Linux ile başlayalım.

GNU/Linux

Elimizde şöyle bir program olduğunu düşünelim:

```
kartvizit = """
İstihza Anonim Şirketi
Fırat Özgül
Tel: 0212 123 23 23
Faks: 0212 123 23 24
e.posta: kistihza@yahoo.com"""

print(kartvizit)
```

Bu programı çalıştırabilmek için, önce programımızı masaüstüne *kartvizit.py* adı ile kaydediyoruz. Sonra da, daha önce gösterdiğimiz şekilde python3 kartvizit.py komutunu vererek programımızı çalıştırıyoruz.

Daha önce de söylediğimiz gibi, Python programlarını çalıştırmanın temel yöntemidir bu. Bu temel yöntemde, yazdığımız Python programını çalıştırabilmek için *Python'ı başlatan komut + program dosyasının adı* (python3 kartvizit.py) yapısını kullanıyoruz.

Peki başa python3 gibi bir komut getirmeden, sadece program adını vererek (yani sadece kartvizit komutuyla) programımızı çalıştıramaz mıyız? Elbette çalıştırabiliriz. Ancak bunun için programımız üzerinde bazı değişiklikler yapmamız gerekiyor. Gelin bu değişikliklerin neler olduğuna bakalım.

Programları Sadece İsmi ile Çalıştırmak

Elimizdeki programa tekrar bir bakalım:

```
kartvizit = """
İstihza Anonim Şirketi
```

```
Fırat Özgül
Tel: 0212 123 23 23
Faks: 0212 123 23 24
e.posta: kistihza@yahoo.com"""

print(kartvizit)
```

Programımızın adının *kartvizit.py* olduğunu varsayarsak, bu programı python3 kartvizit.py komutuyla çalıştırabileceğimizi biliyoruz. Ama şimdi biz bu programı komut satırında sadece kartvizit komutunu vererek çalıştıracağız.

Dediğimiz gibi, programımızı sadece kartvizit gibi bir komutla çalıştırabilmek için programımız üzerinde bazı değişiklikler yapmamız gerekiyor.

Peki nedir bu değişiklikler?

Dedik ki programımızın adı *kartvizit.py*. Şimdi ilkin programımızın adını *kartvizit* olarak değiştirelim. Yani uzantısını silelim.

Daha önce de söylediğimiz gibi, GNU/Linux'ta dosyalar üzerinde öntanımlı olarak çalıştırma yetkisine sahip değiliz. Yani o dosya bir program da olsa, bu programı çalıştırabilmemiz için, program dosyasını çalıştırma yetkisine sahip olmamız gerekiyor. chmod adlı bir sistem komutu yardımıyla gereken bu çalıştırma yetkisini nasıl alabileceğimizi öğrenmiştik:

```
sudo chmod +x kartvizit
```

Not

chmod komutunun bir Python komutu olmadığını biliyorsunuz. Bu komut bir sistem komutudur. Dolayısıyla Python'ın etkileşimli kabuğunda değil, işletim sisteminin komut satırında çalıştırılır.

Program dosyasını çalıştırabilmemiz için gereken yetkileri elimize aldık. Ama yapmamız gereken birkaç işlem daha var.

Öncelikle program dosyamızı açarak ilk satıra şu ibareyi ekleyelim:

```
#!/usr/bin/python3
```

Yani programımız tam olarak şöyle görünsün:

```
#!/usr/bin/python3

kartvizit = """
İstihza Anonim Şirketi
```

```
Fırat Özgül
Tel: 0212 123 23 23
Faks: 0212 123 23 24
e.posta: kistihza@yahoo.com"""

print(kartvizit)
```

Son olarak da programımızı `echo $PATH` çıktısında görünen dizinlerden herhangi birinin içine kopyalayalım. Mesela */usr/bin* dizini bu iş için en uygun yerdir:

```
sudo cp kartvizit /usr/bin/kartvizit
```

Artık komut satırında sadece `kartvizit` komutu vererek programımızı çalıştırabiliriz.

Eğer kullandığımız sistemde *root* haklarına sahip değilsek, */usr/bin* dizinine herhangi bir şey kopyalayamayız. Böyle bir durumda *$HOME* dizini altında *bin* adlı bir dizin oluşturup, daha sonra bu dizini YOL'a ekleyebilir, programınızı da bu *$HOME/bin* dizini içine atabiliriz.

Peki, kodların en başına yerleştirdiğimiz `#!/usr/bin/python3` satırı ne anlama geliyor?

Şöyle düşünün: Bir Python programı yazdınız. Bu programı `python3 kartvizit.py` gibi bir komut ile çalıştırdığınızda bunun bir Python programı olduğu ve bu programın da Python tarafından çalıştırılacağı anlaşılıyor. Ancak baştaki python3 ifadesini kaldırdığımızda işletim sistemi bu programla ne yapması gerektiğine karar veremez. İşte en başa yerleştirdiğimiz `#!/usr/bin/python3` satırı yazdığımız kodların birer Python kodu olduğunu ve bu kodların da */usr/bin/* dizini altındaki *python3* adlı program ile çalıştırılacağını gösteriyor.

GNU/Linux sistemlerinde Python'ın çalıştırılabilir dosyası genellikle */usr/bin* dizini altındadır. Dolayısıyla yukarıdaki satırı şöyle yazabiliyoruz:

```
#!/usr/bin/python3
```

Böylece işletim sistemimiz Python'ı */usr/bin* dizini altında arayacak ve yazdığımız programı Python'la çalıştırması gerektiğini anlayacaktır.

Ancak bazı GNU/Linux sistemlerinde, Python'ın nasıl kurulduğuna bağlı olarak Python'ın çalıştırılabilir dosyası başka bir dizinin içinde de olabilir. Mesela eğer Python programlama dilini yetkisiz kullanıcı olarak ev dizininin altındaki *python* adlı bir dizin içine kurduysak, çalıştırılabilir dosya *$HOME/python/bin/* dizini altında olacaktır. Bu durumda, çalıştırılabilir dosya */usr/bin* altında bulunamayacağı için, sistemimiz yazdığımız programı çalıştıramaz.

Python'ın çalıştırılabilir dosyasının her sistemde aynı dizin altında bulunmama ihtimalinden ötürü yukarıdaki gibi sabit bir dizin adı vermek iyi bir fikir olmayabilir. Bu tür sistem farklılıklarına karşı önlem olarak GNU/Linux sistemlerindeki env adlı bir betikten yararlanabiliriz. */usr/bin* dizini altında bulunduğunu varsayabileceğimiz bu env betiği, YOL dizinlerini tek tek tarayarak, Python'ın hangi

YOL dizini içinde olduğunu bulabilir. Böylece yazdığımız bir programın, Python'ın */usr/bin* dizini haricinde bir konuma kurulduğu sistemlerde çalıştırılması konusunda endişe etmemize gerek kalmaz. Bunun için programımızın en başına eklediğimiz satırı şöyle yazıyoruz:

```
#/usr/bin/env python3
```

Yani programımız tam olarak şöyle görünüyor:

```
#!/usr/bin/env python3

kartvizit = """
İstihza Anonim Şirketi
Fırat Özgül
Tel: 0212 123 23 23
Faks: 0212 123 23 24
e.posta: kistihza@yahoo.com"""

print(kartvizit)
```

İlk satırı bu şekilde yazmamız sayesinde Python YOL içinde nereye kurulmuş olursa olsun kolaylıkla tespit edilebilecektir.

Uzun lafın kısası, #!/usr/bin/python3 yazdığımızda sisteme şu emri vermiş oluyoruz: 'Python'ı */usr/bin* dizini içinde ara!' #!/usr/bin/env python3 yazdığımızda ise şu emri: 'Python'ı YOL içinde nereye saklandıysa bul!'

Eğer bu satırı #!/usr/bin/python3 şeklinde yazarsanız ve eğer kullandığımız sistemde Python'ın çalıştırılabilir dosyası */usr/bin/* dizini altında değilse, programınızı çalıştırmak istediğimizde şuna benzer bir hata çıktısı alırız:

```
bash: deneme: /usr/bin/python3: bad interpreter: No such file or directory
```

Böyle bir hata almamak için, o satırı #!/usr/bin/env python3 şeklinde yazmaya özen gösteriyoruz. Böylece Python YOL içindeki dizinler arasında nereye kurulmuş olursa olsun işletim sistemimiz Python'ı tespit edebiliyor.

Bu noktada size şöyle bir soru sormama izin verin: Acaba Python'ın *python3* adıyla değil de, mesela *python33* (veya py3) adıyla kurulu olduğu sistemlerde #!/usr/bin/env python3 satırı görevini yerine getirebilir mi?

Elbette getiremez...

Çünkü bu satır yardımıyla biz sistemde *python3* adlı bir dosya aratıyoruz aslında. Ama eğer sistemde Python3'ü çalıştıran dosyanın adı 'python3' değil de, mesela 'py3' ise tabii ki o satır bir işe yaramayacaktır. Programımızın böyle bir sistemde çalışabilmesi için bu satırın

#!/usr/bin/envpython3.3 şeklinde yazılması gerekir.

Bu manzara karşısında aklınıza şöyle bir soru gelmiş olabilir:

> GNU/Linux sistemlerinde Python3'ü çalıştırmanın standart ve tek bir yolu yok. Çünkü Python3'ü çalıştıran dosyanın adı farklı dağıtımlarda farklı ada sahip olabiliyor. Böyle bir durumda ben yazdığım bir Python programının bütün GNU/Linux dağıtımlarında çalışabileceğinden nasıl emin olacağım?

Bu sorunun cevabı, 'Emin olamazsınız,' olacaktır...

GNU/Linux'ta sadece Python için değil başka programlama dilleriyle yazılmış programlar için de aynı sorun geçerlidir. Çünkü GNU/Linux dağıtımları arasında bir uyumluluktan bahsetmek mümkün değil. Sistem farklılıklarından ötürü, bir GNU/Linux dağıtımında çalışan bir program başka bir GNU/Linux dağıtımında çalışmayabilir.

GNU/Linux işletim sistemine yönelik programlar yazan bir Python programcısı olarak sizin sorumluluğunuz programınızı yazıp bitirmekten ibarettir. Bu programın herhangi bir GNU/Linux dağıtımında çalışabilmesi o dağıtımın geliştiricilerinin sorumluluğu altındadır. Yani mesela siz kendiniz Ubuntu dağıtımını kullanıyorsanız programlarınızı Ubuntu'yu temel alarak yazarsınız. Burada sizin sorumluluğunuz, yazdığınız programın Ubuntu altında çalışmasını sağlamaktan ibarettir. Aynı programın mesela Pardus altında çalışabilmesi Pardus'a uygun paket yazan geliştiricilerin görevidir. Tabii ki siz isterseniz programınız için hem Ubuntu paketini (*DEB*), hem de Pardus paketini (*PISI*) ayrı ayrı hazırlayabilirsiniz. Ama elbette bütün dağıtımlara uygun paketleri tek tek hazırlamanız mümkün değil.

Ayrıca programlarınızın kaynak kodlarını yayımlarken, programın nasıl çalıştırılacağına ilişkin ayrıntılı açıklamaları *README* yada *BENİOKU* adlı bir dosya içine yazabilirsiniz. Böylece farklı dağıtımların kullanıcıları, yazdığınız programı kendi dağıtımlarında nasıl çalıştırmaları gerektiği konusunda bir fikir sahibi olabilir. Kullanıcılar, yazdığınız açıklamalardan yola çıkarak programınızı çalıştırabilirse ne âlâ! Ama eğer çalıştıramazsa, bu doğrudan sizin sorununuz değil. Böyle bir durumda zaten GNU/Linux kullanıcıları da sizi suçlamayacak, kendi kullandıkları dağıtımın geliştiricilerinden sizin yazdığınız program için paket isteğinde bulunacaktır.

Programları Çift Tıklayarak Çalıştırmak

GNU/Linux işletim sistemlerinde, özellikle grafik bir arayüze sahip olmayan, yani yalnızca komut satırı üzerinde çalışabilen bir programı başlatmak için program simgesi üzerine çift tıklamak pek uygulanan bir yol değildir. Eğer bir programın grafik arayüzü varsa *.desktop* dosyaları veya menü girdileri aracılığıyla programa ulaşabilirsiniz. Ama komut satırı üzerinden çalışan uygulamalar genellikle komut satırı üzerinden çalıştırılır. Dolayısıyla elimizde şöyle bir program olduğunu varsayarsak:

```
kartvizit = """
İstihza Anonim Şirketi
```

```
Fırat Özgül
Tel: 0212 123 23 23
Faks: 0212 123 23 24
e.posta: kistihza@yahoo.com"""

print(kartvizit)
```

Bu program GNU/Linux üzerinde komut satırı aracılığıyla şu komut verilerek çalıştırılır:

```
python3 kartvizit.py
```

Veya eğer isterseniz, daha önce gösterdiğimiz yöntemleri uygulayarak, programı yalnızca ismiyle de çalıştırabilirsiniz.

GNU/Linux dağıtımlarında, yalnızca komut arayüzüne sahip bir programı çift tıklayarak çalıştırmanın önündeki bir başka engel, bütün GNU/Linux dağıtımlarının, simge üzerine çift tıklamaya aynı tepkiyi vermemesidir. Ama teorik olarak, GNU/Linux'ta komut satırı üzerinde çalışabilen bir programın dosyası üzerine çift tıklayarak çalıştırabilmek için şu yolu takip edebilirsiniz:

Öncelikle programımızın ilk satırına, programımızı hangi Python sürümü ile çalıştırmak istediğimizi belirten şu kodu ekliyoruz:

```
#!/usr/bin/env python3
```

Daha sonra, program kodlarının en son satırına da şu kodu ekliyoruz:

```
input()
```

Eğer bu satırı eklemezseniz, program dosyasına çift tıkladığınızda programınız çok hızlı bir şekilde açılıp kapanır. Bunu engellemek için yukarıda gördüğümüz bu input() adlı fonksiyonu kullanıyoruz. Bu fonksiyonu eklememiz sayesinde programımız açıldıktan sonra kapanmayacak, kapanmak için bizim *Enter* düğmesine basmamızı bekleyecektir.

Bu arada bu input() fonksiyonuna şimdilik takılmayın. Birkaç bölüm sonra bu fonksiyonu da bütün ayrıntılarıyla inceleyeceğiz.

Programımızın son hali şöyle:

```
#!/usr/bin/env python3

kartvizit = """
İstihza Anonim Şirketi
Fırat Özgül
Tel: 0212 123 23 23
Faks: 0212 123 23 24
```

```
e.posta: kistihza@yahoo.com"""

print(kartvizit)

input()
```

Bütün bu işlemlerin ardından programımızı çalıştırılabilir olarak işaretliyoruz:

```
chmod +x kartvizit.py
```

Artık program dosyası üzerine çift tıklayarak programımızı çalıştırmayı deneyebiliriz. Ama dediğimiz gibi, farklı GNU/Linux dağıtımlarında çift tıklama işlemi farklı yanıtlar verebilir. O yüzden komut satırı uygulamalarını çift tıklayarak başlatmak pek tercih edilen bir yöntem değildir.

GNU/Linux işletim sistemi altında bir Python programını çalıştırmanın alternatif yöntemlerini öğrendiğimize göre, aynı işlemi Windows altında nasıl yapabileceğimizi tartışmaya geçebiliriz. Bu arada siz GNU/Linux kullanıcıları da yukarıda verdiğimiz bilgileri iyice sindirebilmek için kendi kendinize örnek programlar yazıp bunları farklı şekillerde çalıştırmayı deneyebilirsiniz.

Bu arada, her zaman söylediğimiz gibi, yukarıda anlatılanları ezberlemeye çalışmak yerine, neyi neden yaptığımızı anlamaya çalışmak çok daha faydalı bir iş olacaktır. Örneğin bir programı adıyla çağırabilmek için bütün o işlemleri yaparken nereye varmaya çalışıyoruz? Amacımız ne? Her bir adım hangi maksada yönelik? Mesela programımıza neden çalıştırma izni veriyoruz? env betiği ne işe yarar? Programımızın konumunu neden değiştiriyoruz?

Yukarıda anlatılanları okurken kendinize bu tür sorular sorup cevaplamanız, süreci ezberlemek yerine sürecin mantığını kavramanızı sağlayacaktır. Unutmayın, buradaki bilgiler sadece Python programları için geçerli değildir. Burada sözü edilen program çalıştırma mantığı, etrafta gördüğünüz bütün programlar için geçerlidir. Dolayısıyla, eğer buradaki bilgileri hakkıyla öğrenirseniz, bir taşla iki kuş vurmuş olursunuz.

Windows

Bir önceki bölümde, yazdığımız bir Python programını IDLE üzerinden nasıl çalıştırabileceğimizi gördük. Eğer IDLE adlı metin düzenleyiciyi kullanıyorsanız, Python programlarınızı çalıştırmak için muhtemelen çoğunlukla bu yöntemi kullanacaksınız. Ama elbette bir Python programını çalıştırmanın tek yöntemi bu değildir. İşte biz bu bölümde, Windows'ta Python programlarını çalıştırmanın alternatif yöntemlerinden söz edeceğiz.

Programları Komut Satırından Çalıştırmak

Python programlarınızı yazmak için kullandığınız metin düzenleyiciler, tıpkı IDLE'da olduğu gibi, *F5* benzeri kısayollar aracılığıyla programlarınızı çalıştırmanızı sağlayabilir. Ama bazı metin düzenleyicilerde bu tür kolaylıklar bulunmaz. Yazdığınız Python programlarını her koşulda çalıştırabilmek için, bu programları komut satırı üzerinden nasıl çalıştıracağınızı da bilmeniz gerekir.

Bunun için şu adımları takip ediyoruz.

Önclelikle daha önce öğrendiğimiz yöntemlerden birini kullanarak MS-DOS komut satırına ulaşın.

MS-DOS komut satırına ulaştıktan sonra, daha önce öğrendiğimiz şekilde, komut satırı üzerinden masaüstüne, yani dosyayı kaydettiğiniz yere gelin. (Bu işlemin nasıl yapılacağını hatırladığınızı varsayıyorum.)

Masaüstüne geldikten sonra şu komutu vererek programınızı çalıştırabilirsiniz:

```
python program_adı.py
```

Elbette burada Python'ın etkileşimli kabuğuna ulaşmak için hangi komutu kullanıyorsanız onu kullanacaksınız. Ben sizin Python'ın etkileşimli kabuğuna ulaşmak için python komutunu kullandığınızı varsaydım.

Yukarıdaki gibi python komutunu vermek yerine, aynı işlev için önceki derslerde öğrendiğimiz 'py' adlı betiği de kullanabilirsiniz:

```
py program_adı.py
```

Unutmayın ki ben burada py komutunun Python3'ü başlattığını varsayıyorum. Eğer sisteminizde hem Python2 hem de Python3 kurulu ise py komutunu tek başına kullandığınızda Python2 de başlıyor olabilir.

Dolayısıyla farklı Python sürümlerinin bir arada bulunduğu sistemlerde programınızı Python3 ile çalıştırabilmek için yukarıdaki komutu şu şekilde vermeniz de gerekebilir:

```
py -3 program_adı.py
```

py adlı betiği çalıştırırken, istediğiniz Python sürümünün çalışmasını temin etmek için, yukarıda gösterdiğimiz şekilde sürüm numarasını da belirtmenin yanısıra, yazdığınız bir programın istediğiniz Python sürümü ile çalışmasını sağlamanın bir yolu daha var:

Bunun için öncelikle boş bir metin düzenleyici açıp şu kodları yazın:

```
#! python3

print("Merhaba Zalim Dünya!")
```

Dosyayı *zalim.py* adıyla kaydedip şu komutu verin:

```
py zalim.py
```

Programınız normal bir şekilde çıktı verir. Şimdi aynı kodları bir de şöyle yazın:

```
#! python2

print("Merhaba Zalim Dünya!")
```

Bu defa programınız çalıştırma esnasında hata verecektir. Peki neden?

Yazdığımız programların ilk satırlarına dikkat edin. İlk programda şöyle bir satır yazdık:

```
#! python3
```

İşte bu satır, Python programımızı Python'ın 3.x sürümlerinden biri ile çalıştırmamızı sağladı. Python'ın 3.x sürümlerinde öntanımlı karakter kodlama biçimi *UTF-8* olduğu için (yani Python3'te Türkçe desteği daha kuvvetli olduğu için) *Merhaba Zalim Dünya* karakter dizisindeki Türkçe karakterler düzgün bir şekilde gösterilebildi ve programımız düzgün bir şekilde çalıştı.

İkinci programda ise şu şatırı kullandık:

```
#! python2
```

Bu satır ise programımızı Python'ın 2.x sürümlerinden biri ile çalıştırmamızı sağlıyor. Python'ın 2.x sürümlerinde öntanımlı karakter kodlama biçimi *ASCII* olduğundan, Türkçe karakterler içeren bir programı çalıştırabilmek için ilave işlemler yapmamız gerekir. Yukarıdaki program dosyası bu ilave işlemleri içermediği için programımız Python'ın 2.x sürümlerinde hata verecektir.

Bu satırların oluşturduğu etkiyi daha net bir şekilde görebilmek için şu kodları da kullanabilirsiniz:

```
#! python3

import sys
print(sys.version)
```

Hatırlarsanız, print() fonksiyonunu anlatırken Python'da 'modül' diye bir şey olduğundan söz etmiş, örnek olarak da *sys* adlı modülden bahsetmiştik. Orada bu *sys* modülü içindeki *stdout* adlı bir değişkeni incelemiştik. Burada da yine *sys* modülünü görüyoruz. Bu defa bu modülün içindeki *version* adlı bir değişkeni kullandık. İsterseniz etkileşimli kabuğu açıp şu komutları vererek kendi kendinize denemeler yapabilirsiniz:

```
>>> import sys
>>> sys.version
```

Gördüğünüz gibi bu değişken, kullandığımız Python'ın sürümünü gösteriyor.

Burada ilk satır yardımıyla *sys* modülünü içe aktardığımızı, ikinci satır yardımıyla da bu modül içindeki *version* adlı değişkeni kullandığımızı biliyorsunuz. İşte yukarıda gösterdiğimiz program örneğinde de bu *sys* modülünü ve bunun içindeki *version* değişkenini kullanıyoruz:

```
#! python3

import sys
print(sys.version)
```

Şimdi bu programı kaydedip çalıştırın. Şuna benzer bir çıktı alacaksınız:

'3.3.2 (default, May 20 2013, 16:10:12) n[GCC 4.4.7 20120313 (Red Hat 4.4.7-3)]'

Gördüğünüz gibi, elde ettiğimiz şey bir karakter dizisi. Bu karakter dizisinin ilk satırlarına bakarak programımızın Python'ın 3.3.2 sürümü ile çalışmış olduğunu anlıyoruz.

Aynı kodları bir de şöyle yazalım:

```
#! python2

import sys
print(sys.version)
```

Bu da bize şöyle bir çıktı verecek:

'2.7.5 (default, May 20 2013, 15:59:26) n[GCC 4.4.7 20120313 (Red Hat 4.4.7-3)]'

Demek ki bu defa programımız Python'ın 2.7.5 sürümü ile çalışmış...

Elbette ben yukarıdaki kodları anlatırken, sisteminizde Python3 ile birlikte Python2'nin de kurulu olduğunu varsaydım. Eğer sisteminizde Python2 kurulu değilse, aldığınız hata mesajının sebebi, doğal olarak, kodlarda Türkçe karakterlerin bulunması değil, Python2'nin kurulu olmaması olacaktır...

Gördüğünüz gibi, programlarımızın ilk satırına eklediğimiz bu kodlar, programlarımızı çalıştırmak istediğimiz sürümleri kontrol etmemizi sağlıyor. Bu satırları kullanarak, programlarımızın istediğimiz Python sürümüyle çalışmasını temin edebiliriz.

Hatırlarsanız buna benzer bir satırın GNU/Linux işletim sistemleri için de geçerli olduğunu söylemiştik. GNU/Linux'ta bu satırı şöyle yazıyorduk:

```
#!/usr/bin/env python3
```

Eğer Windows kullanıcıları olarak, Windows'ta yazdığınız programların GNU/Linux dağıtımları altında da çalışmasını istiyorsanız Windows'ta `#! python3` şeklinde yazdığınız satırı tıpkı GNU/Linux'ta olduğu gibi `#!/usr/bin/env python3` şeklinde yazmayı da tercih edebilirsiniz. Çünkü Python Windows'ta yazdığınız satırın sadece *python3* kısmıyla ilgilenecek, dolayısıyla `#!/usr/bin/env python3` şeklinde yazdığınız kod hem Windows'ta hem de GNU/Linux'ta çalışacaktır.

Programları Sadece İsmi ile Çalıştırmak

Peki yazdığımız programı, başa python gibi bir komut getirmeden çalıştırma imkanımız var mı?

Python Windows'a kurulurken kendini kütüğe (*Registry*) kaydeder. Dolayısıyla Windows Python programlarını nasıl çalıştırması gerektiğini bilir. Bu sayede Windows üzerinde yazdığımız Python programlarını, programın bulunduğu dizin içinde sadece ismini kullanarak çalıştırmamız da mümkündür. Yani yazdığımız programı masaüstüne kaydettiğimizi varsayarsak, masaüstüne geldikten sonra şu komutu vererek programımızı çalıştırma imkanına sahibiz:

```
program_adı.py
```

Tıpkı bir önceki konuda da anlattığımız gibi, programımızı hangi Python sürümünün çalıştıracağını yine program dosyasının ilk satırına yazacağımız #! python3 gibi bir kod yardımıyla kontrol edebiliriz.

Burada hemen şöyle bir soru akla geliyor: Windows'ta mesela Notepad ve Calc gibi programları, hiç *.EXE* uzantısını belirtmeden doğrudan isimleriyle çağırabiliyoruz. Yani örneğin MS-DOS komut satırında notepad komutu verirsek Notepad programı, calc komutu verirsek de Hesap Makinesi programı çalışmaya başlayacaktır. Ama mesela *deneme.py* adlı programımızı çalıştırabilmek için *.PY* uzantısını da belirtmemiz gerekti. Peki bu durumun nedeni nedir?

Windows'ta *PATHEXT* adlı bir çevre değişkeni vardır. Bu değişken, sistemin çalıştırılabilir kabul ettiği uzantıları tutar. MS-DOS ekranında şu komutu verelim:

```
echo %PATHEXT%
```

Buradan şu çıktıyı alıyoruz:

```
.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH
```

Windows'ta eğer bir program yukarıda görülen uzantılardan birine sahipse çalıştırılabilir olarak kabul edilecektir. Gördüğünüz gibi, *EXE* de bu uzantılardan biri. Dolayısıyla bu uzantıya sahip bir dosyayı, uzantısını belirtmeden de çağırabiliyoruz. Eğer isterseniz bu listeye *.PY* uzantısını da ekleyebilirsiniz. Bunun için şu işlemleri yapabilirsiniz:

1. *Başlat > Denetim Masası > Sistem ve Güvenlik > Sistem > Gelişmiş Sistem Ayarları* yolunu takip edin.
2. Açılan pencerede 'Gelişmiş' sekmesine tıklayın ve 'Ortam Değişkenleri' düğmesine basın.
3. 'Sistem Değişkenleri' bölümünde *PATHEXT* öğesini bulup buna çift tıklayın.
4. En son girdi olan *.WSH*'den sonra *;.PY* girdisini ekleyin.
5. *TAMAM*'a basıp çıkın.

Böylece artık *.PY* uzantılı dosyaları da, uzantı belirtmeden çalıştırabilirsiniz. Ancak bu komutun işe yarayabilmesi için, MS-DOS'ta o anda programın bulunduğu dizin içinde olmamız gerek. Yani eğer programınız *Belgeler* dizini içinde ise ve siz de MS-DOSta *C:/Users/Kullanici_adi/Desktop* dizini altındaysanız bu komut bir işe yaramayacaktır. Eğer yazdığınız programı konum farketmeksizin her yerden çağırabilmek istiyorsanız, programınızın bulunduğu dizini daha önce anlattığımız şekilde

YOL'a eklemelisiniz.

Programları Çift Tıklayarak Çalıştırmak

PY uzantılı Python programları üzerine çift tıklanarak da bu programlar çalıştırılabilir. Ama öncelikle şunu söylemeliyiz: Python programları geliştirirken takip etmemiz gereken yöntem, program simgeleri üzerine çift tıklayarak bunları çalıştırmak değil, bu programları komut satırından çalıştırmak olmalıdır. Çünkü programları çift tıklayarak çalıştırdığınızda programların hangi hataları verdiğini göremezsiniz. Program geliştirirken, programınızın verdiği olası hata ve uyarıları görebilmek ve bunları düzeltebilmek için programlarınızı komut satırından çalıştırmalısınız. Dolayısıyla program geliştirirken simge üzerine çift tıklayarak bunları çalıştırmak tamamen anlamsızdır.

Ama yine de en azından bir Python programının üzerine çift tıklanarak nasıl çalıştırılabileceğini öğrenmeniz için biz burada bu konuyu da ele alacağız.

Şimdi elimizde şöyle bir program olsun:

```
kartvizit = """
İstihza Anonim Şirketi
Fırat Özgül
Tel: 0212 123 23 23
Faks: 0212 123 23 24
e.posta: kistihza@yahoo.com"""

print(kartvizit)
```

Kullandığımız işletim sisteminde birden fazla Python sürümü kurulu olabilir. O yüzden, programımızın hangi Python sürümüyle çalışmasını istiyorsak ona uygun şekilde bir ilk satır yazmamız gerekiyor öncelikle. Biz programımızı Python3 ile çalıştırmak istediğimizi varsayalım ve ilk satıra şunu ekleyelim:

```
#! python3
```

Böylece Windows, programımızı hangi Python sürümü ile çalıştırması gerektiğini bilecek.

Şimdi program simgesi üzerine iki kez tıklayalım.

Biz bu programın simgesi üzerine çift tıkladığımızda siyah bir komut ekranının çok hızlı bir şekilde açılıp kapandığını görürüz. Aslında programımız çalışıyor, ama programımız yapması gereken işi yaptıktan hemen sonra kapandığı için biz program penceresini görmüyoruz. Programımızın çalıştıktan sonra hemen kapanmamasını sağlayacak bir yol bulmamız gerekiyor. Bunun için dosyamızın en sonuna bir adet `input()` satırı ekleyeceğiz. Yani programımız şöyle görünecek:

```
#! python3

```

```
kartvizit = """
İstihza Anonim Şirketi
Fırat Özgül
Tel: 0212 123 23 23
Faks: 0212 123 23 24
e.posta: kistihza@yahoo.com"""

print(kartvizit)

input()
```

Şimdi programı bu şekilde kaydedip simgesi üzerine çift tıkladığımızda program penceresi açılacak ve biz *Enter* düğmesine basana kadar da açık kalacaktır. Burada şimdilik input() fonksiyonunun ne olduğuna takılmayın. Birkaç bölüm sonra bu fonksiyonu bütün ayrıntılarıyla inceleyeceğiz.

Bu arada, elbette en sona eklediğimiz input() satırının görevini yerine getirebilmesi için, ondan önce gelen bütün kodların doğru ve hatasız olması gerekiyor. Eğer bu satırdan önceki kısımda herhangi bir hata yapmışsanız, programın akışı hiçbir zaman input() satırına ulaşmayacağı için yine siyah bir komut ekranının hızla yanıp söndüğünü görürsünüz. O yüzden programlarınızı çift tıklayarak çalıştırmadan önce, komut satırında test ederek hatasız olduğundan emin olmalısınız.

Programları .EXE Haline Dönüştürmek

Şimdiye kadar anlattıklarımızdan anlayacağınız gibi, bir Python programını çalıştırabilmek için Python programlama dilinin bilgisayarımızda kurulu olması gerekiyor. Bir program yazdıktan sonra o programı dağıtırken kullanıcılarınızdan Python programlama dilini de kurmalarını isteyebilirsiniz. Ama eğer bilgisayarlarında Python programlama dili kurulu olmayanların da Python ile yazdığınız programları çalıştırabilmelerini istiyorsanız, bu programı çeşitli araçlar yardımıyla bir *.EXE* dosyası haline getirmeniz de mümkündür. Böylelikle bilgisayarında Python kurulu olmayan kişiler de yazdığınız programı çalıştırma imkanına sahip olabilir.

cx_Freeze

Python3 programlarını *.EXE* haline dönüştürmek için cx_Freeze adlı bir programdan yararlanacağız. Bu programın, kullandığınız Python sürümüne uygun sürümünü cx-freeze.sourceforge.net/ adresinden indirebilirsiniz.

Programı kurduktan sonra, cx_Freeze programının çalışıp çalışmadığını kontrol etmek için MS-DOS komut satırında şu komutu veriyoruz:

```
cxfreeze
```

Bu komutu verdikten sonra şuna benzer bir çıktı almış olmalıyız:

```
Usage: cxfreeze [options] [SCRIPT]

Freeze a Python script and all of its referenced modules to a base
executable which can then be distributed without requiring a Python
installation.

cxfreeze: error: script or a list of modules must be specified
```

Bu çıktı yerine şöyle bir hata çıktısı almış da olabilirsiniz:

```
'cxfreeze' iç ya da dış komut, çalıştırılabilir program ya da toplu iş
dosyası olarak tanınmıyor.
```

Eğer aldığınız çıktı buysa okumaya devam edin.

cx_Freeze Çalışmıyor!

cx_Freeze programını kurduktan sonra eğer cxfreeze komutuyla programı çalıştıramadıysanız cx_Freeze sisteminize yanlış kurulmuş demektir. Bu sorunu düzeltmek için aşağıdaki adımları takip ediyoruz.

1. *C:\Python33\Scripts* adlı dizinin içine girip *cxfreeze.bat* adlı dosyayı buluyoruz.

2. Bu dosyayı Notepad ile açıyoruz. Şuna benzer bir içerikle karşılaşmış olmalıyız:

```
@echo off

C:\Python\32-bit\3.3.2\python.exe C:\Python\32-bit\3.3.2\Scripts\cxfreeze %*
```

Gördüğünüz gibi dosyadaki Python adresi tamamen yanlış. Bunu düzeltmemiz lazım. Dolayısıyla dosyayı şu şekilde değiştiriyoruz:

```
@echo off

C:\Python33\python.exe C:\Python33\Scripts\cxfreeze %*
```

3. Dosyadaki değişiklikleri tamamladıktan sonra dosyayı kaydedip kapatıyoruz. Şimdi komut satırında tekrar şu komutu verelim:

```
cxfreeze
```

Muhtemelen yine aynı hatayı aldınız ve yine cx_Freeze'i çalıştıramadınız. Peki ama neden?

4. Üzerinde değişiklik yaptığımız *cxfreeze.bat* adlı dosyanın içeriğini incelediğimizde cx_Freeze adlı programın çalıştırılabilir dosyasının (cxfreeze) *C:\Python33\Scripts* adlı bir dizin içinde olduğunu

görüyoruz. Komut satırında `echo %PATH%` komutunu verirseniz, bu dizinin YOL üzerinde olmadığını görürsünüz. Bu dizin YOL üzerinde olmadığı için, bu dizinde bulunan program dosyalarını isimleri ile çağıramayız. Yani `cxfreeze` komutunu verdiğimizde cx_Freeze programının çalışmamasının nedeni, bu programın çalıştırılabilir dosyasını barındıran *C:\Python33\Scripts* adlı dizinin YOL üzerinde olmamasıdır. O halde yapmamız gereken şey bu dizini YOL'a eklemek olmalı. Bunun nasıl yapılacağını hatırlıyorsunuz, değil mi?

5. İlgili dizini YOL'a ekledikten sonra komut satırında tekrar `cxfreeze` komutunu verin. Eğer yukarıda anlatılan işlemleri doğru bir şekilde yaptıysanız bu komut artık çalışıyor olmalı.

cx_Freeze Nasıl Kullanılır?

cx_Freeze adlı programı indirip kurduktan ve eğer hatalı bir şekilde kurulduysa bu hatayı giderdikten sonra, kendi yazdığınız Python programının bulunduğu dizin altında şu komutu verin:

```
cxfreeze kartvizit.py
```

Bu komutu verdiğinizde *dist* adlı bir dizinin oluştuğunu göreceksiniz. Şimdi bu dizinin içine girin. İşte orada *deneme.exe* adlı dosya, sizin yazdığınız Python programının *.EXE*'ye dönüştürülmüş hali. Programınızı dağıtırken sadece *.exe* uzantılı dosyayı değil, *dist* adlı dizini bütün içeriğiyle birlikte dağıtacaksınız.

Dizin adını kendiniz sonradan elle değiştirebileceğiniz gibi, daha *.EXE* dosyasını oluştururken de değiştirebilirsiniz. Bunun için yukarıdaki komutu şu şekilde verebilirsiniz:

```
cxfreeze kartvizit.py --install-dir=program_adı
```

Burada *program_adı* değerinin yerine, program dizininin adının ne olmasını istiyorsanız onu yazacaksınız.

Cxfreeze oldukça kapsamlı ve ayrıntılı bir programdır. Elbette yazdığınız programlar karmaşıklaştıkça, programınızı *.EXE* haline getirmek için yukarıdaki gösterdiğimiz `cxfreeze` komutunu daha farklı parametrelerle çalıştırmanız gerektiğini göreceksiniz. Biz şimdilik sizi sadece cxfreeze adlı bu programın varlığından haberdar etmekle yetiniyoruz. İlerde bu programı daha ayrıntılı bir şekilde inceleyeceğiz.

başa dön

Çalışma Ortamı Tavsiyesi

Bu bölümde, Python programları geliştirirken rahat bir çalışma ortamı elde edebilmek için yapmanız gerekenleri sıralayacağız. Öncelikle Windows kullanıcılarından başlayalım.

Windows Kullanıcıları

Windows'ta bir Python programı yazıp kaydettikten sonra bu programı komut satırından çalıştırabilmek için, MS-DOS'u açıp, öncelikle `cd` komutuyla programın bulunduğu dizine ulaşmamız gerekir. İlgili dizine ulaştıktan sonra programımızı `python program_adı` komutuyla çalıştırabiliriz. Ancak bir süre sonra, programı çalıştırmak için her defasında programın bulunduğu dizine ulaşmaya çalışmak sıkıcı bir hal alacaktır. Ama bu konuda çaresiz değiliz.

Windows 7, istediğimiz dizin altında bir MS-DOS ekranı açabilmemiz için bize çok güzel bir kolaylık sunuyor. Normal şartlar altında mesela masaüstünde bir MS-DOS ekranı açabilmek için şu yolu izlemeniz gerekiyor:

1. Windows logolu tuşa ve *R* tuşuna birlikte bas,
2. Açılan pencereye `cmd` yazıp *Enter* düğmesine bas,
3. Bu şekilde ulaştığın MS-DOS ekranında `cd Desktop` komutunu ver.

Bu üç adımla, MS-DOS ekranı üzerinden masaüstüne ulaşmış oluyoruz. Ama aslında bunun çok daha kolay bir yolu var: Masaüstüne sağ tıklarken *Shift* tuşunu da basılı tutarsanız, sağ-tık menüsünde 'Komut penceresini burada aç' adlı bir satır görürsünüz. İşte bu satıra tıklayarak, MS-DOS komut satırını tek harekette masaüstü konumunda çalıştırabilirsiniz. Elbette bu özellik sadece masaüstü için değil, bütün konumlar için geçerlidir. Yani bilgisayarınızda herhangi bir yere sağ tıklarken *Shift* tuşunu da basılı tutarak o konumda bir MS-DOS penceresi açabilirsiniz.

İkinci olarak, çalışma kolaylığı açısından Windows'ta dosya uzantılarının her zaman görünmesini sağlamanızı da tavsiye ederim. Windows ilk kurulduğunda hiçbir dosyanın uzantısı görünmez. Yani mesela *deneme.txt* adlı bir dosya Windows ilk kurulduğunda *deneme* şeklinde görünecektir. Bu durumda, bir dosyanın uzantısını değiştirmek istediğinizde bazı sıkıntılar yaşarsınız. Örneğin, masaüstünde bir metin dosyası oluşturduğunuzu varsayalım. Diyelim ki amacınız bu dosyanın içine bir şeyler yazıp daha sonra mesela bu dosyanın uzantısını *.BAT* veya *.PY* yapmak olsun. Böyle bir durumda, dosya uzantılarını göremediğiniz için, metin dosyasının uzantısını değiştirmeye çalıştığınızda *deneme.bat.txt* gibi bir dosya adı elde edebilirsiniz. Tabii ki bu dosya bir *.BAT* dosyası değil, bir *.TXT*, yani metin dosyasıdır. Dolayısıyla aslında dosya uzantısını değiştirememiş oluyorsunuz.

Yukarıdaki nedenlerden ötürü, ben size şu yolu takip ederek dosya uzantılarını her zaman görünür hale getirmenizi öneririm:

1. *Başlat > Denetim Masası* yolunu takip ederek denetim masasına ulaşın,
2. Denetim masasında 'Görünüm ve Kişiselleştirme' seçeneğine tıklayın,

3. Açılan menünün sağ tarafında ‘Klasör Seçenekleri’ satırına tıklayın,
4. Açılan pencerede ‘Görünüm’ sekmesine tıklayın,
5. ‘Gelişmiş Ayarlar’ listesinde ‘Bilinen dosya türleri için uzantıları gizle’ seçeneğinin yanındaki onay işaretini kaldırın,
6. *Uygula* ve *Tamam* düğmelerine basarak bütün pencereleri kapatın,
7. Artık bütün dosyalarınızın uzantısı da görüneceği için, uzantı değiştirme işlemlerini çok daha kolay bir şekilde halledebilirsiniz.

GNU/Linux Kullanıcıları

Eğer KDE temelli bir GNU/Linux dağıtımı kullanıyorsanız, yazıp kaydettiğiniz Python programını barındıran dizin açıkken *F4* tuşuna bastığınızda, komut satırı o dizin altında açılacaktır.

Unity ve GNOME kullanıcılarının ise benzer bir kolaylığa ulaşmak için nautilus-open-terminal adlı betiği sistemlerine kurmaları gerekiyor. Eğer Ubuntu kullanıyorsanız bu betiği şu komutla kurabilirsiniz:

```
sudo apt-get install nautilus-open-terminal
```

Bu betiği kurduktan sonra bilgisayarınızı yeniden başlatın veya şu komutu verin:

```
killall nautilus
```

Artık komut satırını hangi dizin altında başlatmak istiyorsanız o dizine sağ tıklayın. Menüler arasında *Open in Terminal* [Uçbirimde aç] adlı bir seçenek göreceksiniz. Buna tıkladığınızda o dizin altında bir komut satırı penceresi açılacaktır.

Metin Düzenleyici Ayarları

Daha önce de söylediğimiz gibi, Python ile program yazmak için istediğiniz metin düzenleyiciyi kullanabilirsiniz. Ama kodlarınızın kusursuz görünmesi ve hatasız çalışması için kullandığınız metin düzenleyicide birtakım ayarlamalar yapmanız gerekir. İşte bu bölümde bu ayarların neler olduğunu göstereceğiz.

Eğer programlarınızı IDLE ile yazıyorsanız aslında bir şey yapmanıza gerek yok. IDLE Python ile program yazmak üzere tasarlanmış bir düzenleyici olduğu için bu programın bütün ayarları Python ile uyumludur. Ama eğer IDLE dışında bir metin düzenleyici kullanıyorsanız bu düzenleyicide temel olarak şu ayarları yapmanız gerekir:

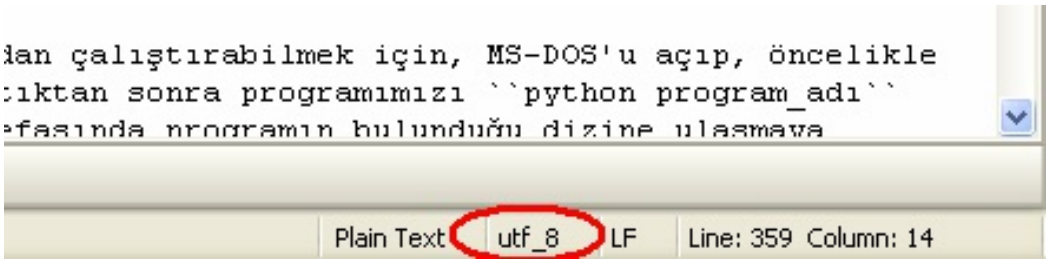
1. Sekme genişliğini [*TAB width*] 4 olarak ayarlayın.
2. Girinti genişliğini [*Indent width*] 4 olarak ayarlayın.
3. Girintilemede sekme yerine boşluk kullanmayı tercih edin [*Use spaces instead of tabs*]
4. Tercih edilen kodlama biçimini [*Preferred encoding*] utf-8 olarak ayarlayın.

Özellikle son söylediğimiz ‘kodlama biçimi’ ayarı çok önemlidir. Bu ayarın yanlış olması halinde,

yazdığınız programı çalıştırmak istediğinizde şöyle bir hata alabilirsiniz:

```
SyntaxError: Non-UTF-8 code starting with '\xfe' in file deneme.py on line 1,
but no encoding declared; see http://python.org/dev/peps/pep-0263/ for details
```

Eğer yazdığınız bir program böyle bir hata mesajı üretiyorsa, ilk olarak metin düzenleyicinizin kodlama biçimi (*encoding*) ayarlarını kontrol edin. Metin düzenleyiciler genellikle tercih edilen kodlama biçimini aşağıdaki örnek resimde görüldüğü gibi, durum çubuğunda sürekli olarak gösterir.



Ancak kodlama biçimi doğru bir şekilde utf-8 olarak ayarlanmış metin düzenleyicilerde, özellikle internet üzerinden kod kopyalanıp yapıştırılması sırasında bu ayar siz farkında olmadan değişebilir. Böyle bir durumda da program çalışırken yukarıda bahsedilen hatayı alabilirsiniz. Dolayısıyla, programınızı yazdığınız metin düzenleyicinin kodlama ayarlarının siz farkında olmadan değişme ihtimaline karşı uyanık olmanız gerekir.

Elbette piyasada yüzlerce metin düzenleyici olduğu için yukarıda bahsedilen ayarların her metin düzenleyicide nasıl yapılacağını tek tek göstermemiz mümkün değil. Ancak iyi bir metin düzenleyicide yukarıdaki ayarların hepsi bulunur. Tek yapmanız gereken, bu ayarların, kullandığınız metin düzenleyicide nereden yapıldığını bulmak. Eğer kullandığınız metin düzenleyiciyi ayarlamakta zorlanıyorsanız, her zamanki gibi istihza.com/forum adresinde sıkıntınızı dile getirebilirsiniz.

'Kodlama biçimi' kavramından söz etmişken, Python'la ilgili önemli bir konuya daha değinelim. En başta da söylediğimiz gibi, şu anda piyasada Python iki farklı seri halinde geliştiriliyor. Bunlardan birinin 2.x serisi, öbürünün de 3.x serisi olduğunu biliyoruz. Python'ın 2.x serisinde Türkçe karakterlerin gösterimi ile ilgili çok ciddi problemler vardı. Örneğin Python'ın 2.x serisinde şöyle bir kod yazamıyorduk:

```
print("Günaydın Şirin Baba!")
```

Bu kodu bir dosyaya kaydedip, Python'ın 2.x serisine ait bir sürümle çalıştırmak istediğimizde Python bize şöyle bir hata mesajı veriyordu:

```
SyntaxError: Non-ASCII character '\xc3' in file test.py on line 1, but no
encoding declared; see http://www.python.org/peps/pep-0263.html for details
```

Bunun sebebi, Python'ın 2.x sürümlerinde *ASCII* adlı kodlama biçiminin kullanılıyor olmasıdır. Zaten hata mesajına baktığımızda da, Python'ın ASCII olmayan karakterlerin varlığından şikayet ettiğini görüyoruz.

Yukarıdaki kodların çalışabilmesi için programımıza şöyle bir ekleme yapmamız gerekiyordu:

```
# -*- coding: utf-8 -*-
print("Günaydın Şirin Baba!")
```

Buradaki ilk satıra dikkat edin. Bu kodlarla yaptığımız şey, Python'ın *ASCII* adlı kodlama biçimi yerine *UTF-8* adlı kodlama biçimini kullanmasını sağlamaktır. *ASCII* adlı kodlama biçimi Türkçe karakterleri gösteremez, ama *UTF-8* adlı kodlama biçimi Türkçe karakterleri çok rahat bir şekilde gösterebilir.

Not

Kodlama biçimlerinden, ileride ayrıntılı bir şekilde söz edeceğiz. O yüzden bu anlattıklarımızda eğer anlamadığınız yerler olursa bunlara takılmanıza gerek yok.

Python'ın 3.x serisinin gelişi ile birlikte Python'da öntanımlı olarak *ASCII* yerine *UTF-8* kodlama biçimi kullanılmaya başlandı. Dolayısıyla yazdığımız programlara `# -*- coding: utf-8 -*-` satırını eklememize gerek kalmadı. Çünkü zaten Python *UTF-8* kodlama biçimini öntanımlı olarak kendisi kullanıyor. Ama eğer *UTF-8* dışında başka bir kodlama biçimine ihtiyaç duyarsanız yine bu satırdan yararlanabilirsiniz.

Örneğin GNU/Linux dağıtımlarının geleneksel olarak UTF-8 kodlama biçimi ile arası iyidir. Dolayısıyla eğer GNU/Linux üzerinde Python programları geliştiriyorsanız bu satırı hiç yazmadan bir ömür geçirebilirsiniz. Ama Windows işletim sistemleri *UTF-8*'i desteklemekle birlikte, bu destek GNU/Linux'taki kadar iyi değildir. Dolayısıyla zaman zaman Windows'ta *UTF-8* dışında başka bir kodlama biçimini kullanmanız gerekebilir. Örneğin yazdığınız bir programda Türkçe karakterleri göremiyorsanız, programınızın ilk satırını şöyle düzenleyebilirsiniz:

```
# -*- coding: cp1254 -*-
```

Burada *UTF-8* yerine *cp1254* adlı kodlama biçimini kullanmış oluyoruz. Windows işletim sisteminde *cp1254* adlı kodlama biçimi *UTF-8*'e kıyasla daha fazla desteklenir.

MS-DOS Komut Satırı Ayarları

Eğer yukarıda anlattığımız bütün ayarları doğru bir şekilde yapmanıza rağmen, özellikle MS-DOS komut satırında hala Türkçe karakterleri düzgün görüntüleyemiyorsanız, kullandığınız Windows sürümünün komut satırı, öntanımlı olarak Türkçe karakterleri gösteremeyen bir yazı tipine ayarlanmış olabilir. Dolayısıyla Türkçe karakterleri gösterebilmek için öncelikle uygun bir yazı tipi seçmeniz gerekir. Bunun için şu basamakları takip ediyoruz:

1. Komut satırını açıyoruz,
2. Açılan pencerenin başlık çubuğuna sağ tıklayarak, 'özellikler' menüsüne giriyoruz,
3. 'Yazı tipi' sekmesinde yazı tipi olarak 'Lucida Console'u (veya varsa 'Consolas'ı) seçiyoruz,
4. 'Tamam' düğmesine basıyoruz.
5. Eğer önünüze bir onay penceresi açılırsa, 'Özellikleri aynı başlıkla ileride oluşturulacak

pencereler için kaydet' seçeneğini işaretleyip 'Tamam'a bastıktan sonra çıkıyoruz.
6. Bu işlemin nasıl yapılacağını gösteren bir videoyu http://media.istihza.com/videos/ms-dos.swf adresinden izleyebilirsiniz.

Böylece MS-DOS komut satırı için Türkçe karakterleri gösterebilen bir yazı tipi belirlemiş olduk. Ancak bu, Türkçe karakterleri düzgün görüntülemeye yetmeyebilir. Eğer Türkçe karakterler hala düzgün görünmüyorsa, kullandığınız sistemde MS-DOS'un dil kodlaması Türkçe karakterleri görüntülemeye uygun olmayabilir. Türkçe karakterleri gösterebilen bir dil kodlaması belirlemek için komut satırında şu komutu verin:

```
chcp 1254
```

CP1254, Türkçe karakterleri de içeren bir dil kodlamasıdır. Bu komutu verdikten sonra artık Türkçe karakterleri düzgün görüntüleyebiliyor olmanız lazım.

Program Örnekleri

Yukarıda Python ve programlamaya ilişkin pek çok teknik bilgi verdik. Bunları öğrenmemiz, işlerimizi kuru kuruya ezberleyerek değil, anlayarak yapmamızı sağlaması açısından büyük önem taşıyordu. Ancak yukarıda pratiğe yönelik pek bir şey sunamadık. İşte bu bölümde pratik eksikliğimizi biraz olsun kapamaya dönük örnekler yapacağız.

Hatırlarsanız Python'la tanışmamızı sağlayan ilk örneğimiz ekrana basit bir *"Merhaba Zalim Dünya!"* cümlesi yazdırmaktı. Bu ilk örneği etkileşimli kabukta verdiğimizi hatırlıyorsunuz:

```
>>> "Merhaba Zalim Dünya!"
```

Ama artık programlarımızı dosyaya kaydetmeyi öğrendiğimize göre bu kodları etkileşimli kabuğa yazmak yerine bir dosyaya yazmayı tercih edebiliriz. Bu sayede yazdığımız kodlar kalıcılık kazanacaktır.

Hemen bir deneme yapalım. Boş bir metin belgesi açıp oraya şu satırı yazalım:

```
"Merhaba Zalim Dünya!"
```

Şimdi de bu dosyayı daha önce anlattığımız şekilde masaüstüne *deneme.py* adıyla kaydedip programımızı çalıştıralım.

Ne oldu? Programınız hiçbir çıktı vermeden kapandı, değil mi?

Hemen hatırlayacağınız gibi, `print()` fonksiyonu içine alınmayan ifadelerin ekrana çıktı olarak verilebilmesi sadece etkileşimli kabuğa özgü bir durumdur. Programlarımızı dosyadan çalıştırırken, `print()` fonksiyonu içine alınmayan ifadeler ekranda görünmeyecektir. Yukarıdaki örnek bu durumun bir göstergesidir. Dolayısıyla yukarıdaki ifadenin ekrana çıktı olarak verilebilmesi için o kodu şöyle yazmamız gerekiyor:

```
print("Merhaba Zalim Dünya!")
```

Programınızı bu şekilde tekrar çalıştırdığınızda şöyle bir çıktı alıyoruz:

```
Merhaba Zalim Dünya!
```

Bu oldukça basit bir örnekti. Şimdi biraz daha karmaşık bir örnek verelim.

Yine hatırlayacağınız gibi, önceki bölümlerden birinde aylık yol masrafımızı hesaplayan bir program yazmıştık.

Orada elimizdeki verilerin şunlar olduğunu varsaymıştık:

1. Cumartesi-Pazar günleri çalışmıyoruz.
2. Dolayısıyla ayda *22* gün çalışıyoruz.
3. Evden işe gitmek için kullandığımız vasıtanın ücreti *1.5* TL
4. İşten eve dönmek için kullandığımız vasıtanın ücreti *1.4* TL

Elimizdeki bu bilgilere göre aylık yol masrafımızı hesaplamak için de şöyle bir formül üretmiştik:

```
aylık yol masrafı = çalışılan gün sayısı x (işe gidiş ücreti + işten dönüş
ücreti)
```

Gelin şimdi yukarıdaki bilgileri kullanarak programımızı dosyaya yazalım:

```
çalışılan_gün_sayısı = 22
işe_gidiş_ücreti = 1.5
işten_dönüş_ücreti = 1.4

aylık_yol_masrafı = çalışılan_gün_sayısı * (işe_gidiş_ücreti + işten_dönüş_ücreti)

print(aylık_yol_masrafı)
```

Tıpkı öncekiler gibi, bu programı da masaüstüne *deneme.py* adıyla kaydedelim ve komut satırında masaüstünün bulunduğu konuma giderek `python3 deneme.py` komutuyla programımızı çalıştıralım. Programı çalıştırdığımızda şöyle bir çıktı alıyoruz:

```
63.8
```

Programımız gayet düzgün çalışıyor. Ancak gördüğünüz gibi, elde ettiğimiz çıktı çok yavan. Ama eğer isterseniz yukarıdaki programa daha profesyonel bir görünüm de kazandırabilirsiniz. Dikkatlice inceleyin:

```
çalışılan_gün_sayısı = 22
işe_gidiş_ücreti = 1.5
işten_dönüş_ücreti = 1.4
```

```
aylık_yol_masrafı = çalışılan_gün_sayısı * (işe_gidiş_ücreti + işten_dönüş_ücreti)

print("-"*30)
print("çalışılan gün sayısı\t:", çalışılan_gün_sayısı)
print("işe gidiş ücreti\t:", işe_gidiş_ücreti)
print("işten dönüş ücreti\t:", işten_dönüş_ücreti)
print("-"*30)

print("AYLIK YOL MASRAFI\t:", aylık_yol_masrafı)
```

Bu defa programımız şöyle bir çıktı verdi:

```
------------------------------
çalışılan gün sayısı    : 22
işe gidiş ücreti        : 1.5
işten dönüş ücreti      : 1.4
------------------------------
AYLIK YOL MASRAFI       : 63.8
```

Gördüğünüz gibi, bu kodlar sayesinde kullanıcıya daha ayrıntılı bilgi vermiş olduk. Üstelik elde ettiğimiz çıktı daha şık görünüyor.

Yukarıdaki kodlarda şimdiye kadar öğrenmediğimiz hiçbir şey yok. Yukarıdaki kodların tamamını anlayabilecek kadar Python bilgimiz var. Bu kodlarda çok basit parçaları bir araya getirerek istediğimiz çıktıyı nasıl elde ettiğimizi dikkatlice inceleyin. Mesela elde etmek istediğimiz çıktının görünüşünü güzelleştirmek için iki yerde şu satırı kullandık:

```
print("-"*30)
```

Böylece *30* adet - işaretini yan yana basmış olduk. Bu sayede elde ettiğimiz çıktı daha derli toplu bir görünüme kavuştu. Ayrıca kodlarımız içinde *\t* adlı kaçış dizisinden de yararlandık. Böylelikle ekrana basılan çıktılar alt alta düzgün bir şekilde hizalanmış oldu.

Bu arada, yukarıdaki kodlar sayesinde değişken kullanımının işlerimizi ne kadar kolaylaştırdığına da birebir tanık olduk. Eğer değişkenler olmasaydı yukarıdaki kodları şöyle yazacaktık:

```
print("-"*30)
print("çalışılan gün sayısı\t:", 22)
print("işe gidiş ücreti\t:", 1.5)
print("işten dönüş ücreti\t:", 1.4)
print("-"*30)

print("AYLIK YOL MASRAFI\t:", 22 * (1.5 + 1.4))
```

Eğer günün birinde mesela çalışılan gün sayısı değişirse yukarıdaki kodların iki farklı yerinde değişiklik yapmamız gerekecekti. Bu kodların çok büyük bir programın parçası olduğunu düşünün. Kodların içinde değer arayıp bunları tek tek değiştirmeye kalkışmanın ne kadar hataya açık bir yöntem olduğunu tahmin edebilirsiniz. Ama değişkenler sayesinde, sadece tek bir yerde değişiklik yaparak kodlarımızı güncel tutabiliriz. Mesela çalışılan gün sayısı *20*'ye düşmüş olsun:

```
çalışılan_gün_sayısı = 20
işe_gidiş_ücreti = 1.5
işten_dönüş_ücreti = 1.4

aylık_yol_masrafı = çalışılan_gün_sayısı * (işe_gidiş_ücreti + işten_dönüş_ücreti)

print("-"*30)
print("çalışılan gün sayısı\t:", çalışılan_gün_sayısı)
print("işe gidiş ücreti\t:", işe_gidiş_ücreti)
print("işten dönüş ücreti\t:", işten_dönüş_ücreti)
print("-"*30)

print("AYLIK YOL MASRAFI\t:", aylık_yol_masrafı)
```

Gördüğünüz gibi, sadece en baştaki *çalışılan_gün_sayısı* adlı değişkenin değerini değiştirerek istediğimiz sonucu elde ettik.

Kendiniz isterseniz yukarıdaki örnekleri çeşitlendirebilirsiniz.

Gördüğünüz gibi, Python'da az da olsa işe yarar bir şeyler yazabilmek için çok şey bilmemize gerek yok. Sırf şu ana kadar öğrendiklerimizi kullanarak bile ufak tefek programlar yazabiliyoruz.

başa dön

Yorum ve Açıklama Cümleleri

Python'la ilgili şimdiye kadar öğrendiğimiz bilgileri kullanarak yazabileceğimiz en karmaşık programlardan biri herhalde şöyle olacaktır:

```
isim    = "Fırat"
soyisim = "Özgül"
işsis   = "Ubuntu"
şehir   = "İstanbul"

print("isim          : ", isim,    "\n",
      "soyisim       : ", soyisim, "\n",
      "işletim sistemi: ", işsis,  "\n",
      "şehir         : ", şehir,   "\n",
      sep="")
```

Yukarıdaki kodları rahatlıkla anlayabildiğinizi zannediyorum. Ama isterseniz yine de bu kodları satır satır inceleyelim:

İlk olarak *isim*, *soyisim*, *işsis* ve *şehir* adında dört farklı değişken tanımladık. Bu değişkenlerin değeri sırasıyla *Fırat*, *Özgül*, *Ubuntu* ve *İstanbul*.

Daha sonra da tanımladığımız bu değişkenleri belli bir düzen içinde kullanıcılarımıza gösterdik, yani ekrana yazdırdık. Elbette bu iş için print() fonksiyonunu kullandık. Bildiğiniz gibi, print() birden fazla parametre alabilen bir fonksiyondur. Yani print() fonksiyonunun parantezleri içine istediğimiz sayıda öğe yazabiliriz.

Eğer print() fonksiyonunun yukarıdaki kullanımı ilk bakışta gözünüze anlaşılmaz göründüyse, fonksiyonda geçen ve ne işe yaradığını anlayamadığınız öğeleri, bir de çıkartarak yazmayı deneyebilirsiniz bu fonksiyonu.

Python'la yazılmış herhangi bir programın tam olarak nasıl işlediğini anlamanın en iyi yolu program içindeki kodlarda bazı değişiklikler yaparak ortaya çıkan sonucu incelemektir. Örneğin print() fonksiyonunda *sep* parametresinin değerini boş bir karakter dizisi yapmamızın nedenini anlamak için, fonksiyondaki bu *sep* parametresini kaldırıp, programı bir de bu şekilde çalıştırmayı deneyebilirsiniz.

Yukarıdaki örnekte bütün öğeleri tek bir print() fonksiyonu içine yazdık. Ama tabii eğer isterseniz birden fazla print() fonksiyonu da kullanabilirsiniz. Şöyle:

```
isim    = "Fırat"
soyisim = "Özgül"
işsis   = "Ubuntu"
şehir   = "İstanbul"
```

```
print("isim          : ", isim)
print("soyisim       : ", soyisim)
print("işletim sistemi: ", işsis)
print("şehir         : ", şehir)
```

Yukarıdaki kodlarla ilgili birkaç noktaya daha dikkatinizi çekmek istiyorum:

Birincisi, gördüğünüz gibi kodları yazarken biraz şekil vererek yazdık. Bunun sebebi kodların görünüş olarak anlaşılır olmasını sağlamak. Daha önce de dediğimiz gibi, Python'da doğru kod yazmak kadar, yazdığınız kodların anlaşılır olması da önemlidir. Bu sebepten, Python'la kod yazarken, mesela kodlarımızdaki her bir satırın uzunluğunun 79 karakteri geçmemesine özen gösteriyoruz. Bunu sağlamak için, kodlarımızı yukarıda görüldüğü şekilde belli noktalardan bölmemiz gerekebilir.

Esasında yukarıdaki kodları şöyle de yazabilirdik:

```
isim = "Fırat"
soyisim = "Özgül"
işsis = "Ubuntu"
şehir = "İstanbul"

print("isim: ", isim, "\n", "soyisim: ", soyisim, "\n", "işletim sistemi: ",
işsis, "\n", "şehir: ", şehir, "\n", sep="")
```

Ancak bu şekilde kod yapısı biraz karmaşık görünüyor. Ayrıca parantez içindeki öğeleri yan yana yazdığımız için, *isim:*, *soyisim:*, *işletim sistemi:* ve *şehir:* ifadelerini alt alta düzgün bir şekilde hizalamak da kolay olmayacaktır.

Belki bu basit kodlarda çok fazla dikkati çekmiyordur, ama özellikle büyük boyutlu programlarda kodlarımızı hem yapı hem de görüntü olarak olabildiğince anlaşılır bir hale getirmek hem kodu okuyan başkaları için, hem de kendimiz için büyük önem taşır. Unutmayın, bir programı yazdıktan 5-6 ay sonra geri dönüp baktığınızda kendi yazdığınız kodlardan siz dahi hiçbir şey anlamadığınızı farkedebilirsiniz!

Bir program yazarken kodların olabildiğince okunaklı olmasını sağlamanın bir kaç yolu vardır. Biz bunlardan bazılarını yukarıda gördük. Ancak bir programı okunaklı hale getirmenin en iyi yolu kodlar içine bazı yorum cümleleri ekleyerek kodları açıklamaktır.

İşte bu bölümde, Python programlama dili ile yazdığımız kodlara nasıl yorum ve açıklama cümleleri ekleyeceğimizi inceleyeceğiz.

Yorum İşareti

Programcılıkta en zor şey başkasının yazdığı kodları okuyup anlamaktır. Hatta yazılmış bir programı

düzeltmeye çalışmak, bazen o programı sıfırdan yazmaktan daha zor olabilir. Bunun nedeni, program içindeki kodların ne işe yaradığını anlamanın zorluğudur. Programı yazan kişi kendi düşünüşüne göre bir yol izlemiş ve programı geliştirirken karşılaştığı sorunları çözmek için kimi yerlerde enteresan çözümler üretmiş olabilir. Ancak kodlara dışarıdan bakan birisi için o programın mantık düzenini ve içindeki kodların tam olarak ne yaptığını anlamak bir hayli zor olacaktır. Böyle durumlarda, kodları okuyan programcının en büyük yardımcısı, programı geliştiren kişinin kodlar arasına eklediği notlar olacaktır. Tabii programı geliştiren kişi kodlara yorum ekleme zahmetinde bulunmuşsa...

Python'da yazdığımız kodları başkalarının da anlayabilmesini sağlamak için, programımızın yorumlarla desteklenmesi tavsiye edilir. Elbette programınızı yorumlarla desteklemeseniz de programınız sorunsuz bir şekilde çalışacaktır. Ama programı yorumlarla desteklemek en azından nezaket gereğidir.

Ayrıca işin başka bir boyutu daha var. Sizin yazdığınız kodları nasıl başkaları okurken zorlanıyorsa, kendi yazdığınız kodları okurken siz bile zorlanabilirsiniz. Özellikle uzun süredir ilgilenmediğiniz eski programlarınızı gözden geçirirken böyle bir sorunla karşılaşabilirsiniz. Programın içindeki bir kod parçası, programın ilk yazılışının üzerinden 5-6 ay geçtikten sonra size artık hiçbir şey ifade etmiyor olabilir. Kodlara bakıp, 'Acaba burada ne yapmaya çalışmışım?' diye düşündüğünüz zamanlar da olacaktır. İşte bu tür sıkıntıları ortadan kaldırmak veya en aza indirmek için kodlarımızın arasına açıklayıcı notlar ekleyeceğiz.

Python'da yorumlar # işareti ile gösterilir. Mesela bu bölümün ilk başında verdiğimiz kodları yorumlarla destekleyelim:

```
isim    = "Fırat"
soyisim = "Özgül"
işsis   = "Ubuntu" #işletim sistemi
şehir   = "İstanbul"

#isim, soyisim, işsis ve şehir adlı değişkenleri
#alt alta, düzgün bir şekilde ekrana basıyoruz.
#Uygun yerlerde alt satıra geçebilmek için "\n"
#adlı kaçış dizisini kullanıyoruz.
print("isim          : ", isim,    "\n",
    "soyisim       : ", soyisim, "\n",
    "işletim sistemi: ", işsis,   "\n",
    "şehir         : ", şehir,   "\n",
    sep="") #parametreler arasında boşluk bırakmıyoruz.
```

Burada dikkat edeceğimiz nokta her yorum satırının başına # işaretini koymayı unutmamaktır.

Yazdığımız yorumlar Python'a hiç bir şey ifade etmez. Python bu yorumları tamamen görmezden gelecektir. Bu yorumlar bilgisayardan ziyade kodları okuyan kişi için bir anlam taşır.

Elbette yazdığınız yorumların ne kadar faydalı olacağı, yazdığınız yorumların kalitesine bağlıdır.

Dediğimiz gibi, yerli yerinde kullanılmış yorumlar bir programın okunaklılığını artırır, ama her tarafı yorumlarla kaplı bir programı okumak da bazen hiç yorum girilmemiş bir programı okumaktan daha zor olabilir! Dolayısıyla Python'da kodlarımıza yorum eklerken önemli olan şey, kaş yapmaya çalışırken göz çıkarmamaktır. Yani yorumlarımızı, bir kodun okunaklılığını artırmaya çalışırken daha da bozmayacak şekilde yerleştirmeye dikkat etmeliyiz.

Yorum İşaretinin Farklı Kullanımları

Yukarıda yorum (#) işaretini kullanarak, yazdığımız Python kodlarını nasıl açıklayacağımızı öğrendik. Python'da yorum işaretleri çoğunlukla bu amaç için kullanılır. Yani kodları açıklamak, bu kodları hem kendimiz hem de kodları okuyan başkaları için daha anlaşılır hale getirmek için... Ama Python'da # işareti asıl amacının dışında bazı başka amaçlara da hizmet edebilir.

Etkisizleştirme Amaçlı

Dediğimiz gibi, yorum işaretinin birincil görevi, tabii ki, kodlara açıklayıcı notlar eklememizi sağlamaktır. Ama bu işaret başka amaçlar için de kullanılabilir. Örneğin, diyelim ki yazdığımız programa bir özellik eklemeyi düşünüyoruz, ama henüz bu özelliği yeni sürüme eklemek istemiyoruz. O zaman şöyle bir şey yapabiliriz:

```
isim    = "Fırat"
soyisim = "Özgül"
işsis   = "Ubuntu"
şehir   = "İstanbul"
#uyruğu = "T.C"

print("isim          : ", isim,    "\n",
    "soyisim       : ", soyisim, "\n",
    "işletim sistemi: ", işsis,   "\n",
    "şehir         : ", şehir,   "\n",
    #"uyruğu       : ", uyruğu,  "\n",
    sep="")
```

Burada, programa henüz eklemek istemediğimiz bir özelliği, yorum içine alarak şimdilik iptal ediyoruz yani etkisizleştiriyoruz (İngilizcede bu yorum içine alma işlemine *comment out* deniyor). Python yorum içinde bir kod bile yer alsa o kodları çalıştırmayacaktır. Çünkü Python # işareti ile başlayan satırların içeriğini görmez (`#!/usr/bin/env python3` ve `# -*- coding: utf-8 -*-` satırları hariç).

Peki eklemek istemediğimiz özelliği yorum içine almaktansa doğrudan silsek olmaz mı? Elbette olur. Ama programın daha sonraki bir sürümüne ilave edeceğimiz bir özelliği yorum içine almak yerine silecek olursak, vakti geldiğinde o özelliği nasıl yaptığımızı hatırlamakta zorlanabiliriz! Hatta bir süre sonra programımıza hangi özelliği ekleyeceğimizi dahi unutmuş olabiliriz. 'Hayır, ben hafızama güveniyorum!' diyorsanız karar sizin.

Yorum içine alarak iptal ettiğiniz bu kodları programa ekleme vakti geldiğinde yapacağınız tek şey, kodların başındaki # işaretlerini kaldırmak olacaktır. Hatta bazı metin düzenleyiciler bu işlemi tek bir tuşa basarak da gerçekleştirme yeteneğine sahiptir. Örneğin IDLE ile çalışıyorsanız, yorum içine almak istediğiniz kodları fare ile seçtikten sonra *Alt+3* tuşlarına basarak ilgili kodları yorum içine alabilirsiniz. Bu kodları yorumdan kurtarmak için ise ilgili alanı seçtikten sonra *Alt+4* tuşlarına basmanız yeterli olacaktır (yorumdan kurtarma işlemine İngilizcede *uncomment* diyorlar).

Süsleme Amaçlı

Bütün bunların dışında, isterseniz yorum işaretini kodlarınızı süslemek için dahi kullanabilirsiniz:

```
##########################################################
##
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~#
#                FALANCA v.1                 #
#             Yazan: Keramet Su                #
#              Lisans: GPL v2                 #
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~#
##########################################################
##

isim    = "Fırat"
soyisim = "Özgül"
işsis   = "Ubuntu"
şehir   = "İstanbul"

print("isim          : ", isim,    "\n",
      "soyisim       : ", soyisim, "\n",
      "işletim sistemi: ", işsis,   "\n",
      "şehir         : ", şehir,   "\n",
      sep="")
```

Yani kısaca, Python'un görmesini, çalıştırmasını istemediğimiz her şeyi yorum içine alabiliriz. Unutmamamız gereken tek şey, yorumların yazdığımız programların önemli bir parçası olduğu ve bunları mantıklı, makul bir şekilde kullanmamız gerektiğidir.

başa dön

Kullanıcıyla Veri Alışverişi

Şimdiye kadar Python programlama dili ile ilgili epey bilgi edindik. Ama muhtemelen buraya kadar öğrendiklerimiz sizi heyecanlandırmaktan bir hayli uzaktı. Zira şu ana kadar hep tek yönlü bir programlama faaliyeti yürüttük.

Mesela şimdiye kadar öğrendiklerimizi kullanarak ancak şöyle bir program yazabildik:

```
isim = "Mübeccel"

print("Merhaba", isim, end="!\n")
```

Bu programı çalıştırdığımızda şöyle bir çıktı alacağımızı biliyorsunuz:

```
Merhaba Mübeccel!
```

Bu programın ne kadar sıkıcı olduğunu herhalde söylemeye gerek yok. Bu programda *isim* değişkenini doğrudan kendimiz yazdığımız için programımız hiçbir koşulda *Merhaba Mübeccel* dışında bir çıktı veremez. Çünkü bu program, tek yönlü bir programlama faaliyetinin ürünüdür.

Halbuki bu değişkenin değerini kendimiz yazmasak, bu değeri kullanıcıdan alsak ne hoş olurdu, değil mi?

Python'da kullanıcıdan herhangi bir veri alıp, yazdığımız programları tek taraflı olmaktan kurtarmak için input() adlı bir fonksiyondan faydalanıyoruz.

İşte biz bu bölümde, programcılık maceramızı bir üst seviyeye taşıyacak çok önemli bir araç olan bu input() fonksiyonunu derinlemesine inceleyeceğiz. Ama bu bölümde sadece bu fonksiyonu ele almayacağız elbette. Burada kullanıcıdan veri almanın yanısıra, aldığımız bu veriyi nasıl dönüştüreceğimizi ve bu veriyi, yazdığımız programlarda nasıl kullanacağımızı da derin derin inceleyeceğiz.

İlkin input() fonksiyonunu anlatarak yola koyulalım.

input() Fonksiyonu

input() da daha önce öğrendiğimiz type(), len() ve print() gibi bir fonksiyondur. Esasında biz bu fonksiyonu ilk kez burada görmüyoruz. Windows ve GNU/Linux kullanıcıları, yazdıkları bir programı çift tıklayarak çalıştırabilmek için bu fonksiyonu kullandıklarını hatırlıyor olmalılar. Mesela şu programı ele alalım:

```
#!/usr/bin/env python3

kartvizit = """
```

```
İstihza Anonim Şirketi
Fırat Özgül
Tel: 0212 123 23 23
Faks: 0212 123 23 24
e.posta: kistihza@yahoo.com
"""

print(kartvizit)
```

Bu programı yazıp kaydettikten sonra bu programın simgesi üzerine çift tıkladığımızda siyah bir komut ekranının çok hızlı bir şekilde açılıp kapandığını görürüz. Aslında programımız çalışıyor, ama programımız yapması gereken işi yaptıktan hemen sonra kapandığı için biz program penceresini görmüyoruz.

Programımızın çalıştıktan sonra hemen kapanmamasını sağlamak için son satıra bir input() fonksiyonu yerleştirmemiz gerektiğini biliyoruz:

```
#!/usr/bin/env python3

kartvizit = """
İstihza Anonim Şirketi
Fırat Özgül
Tel: 0212 123 23 23
Faks: 0212 123 23 24
e.posta: kistihza@yahoo.com
"""

print(kartvizit)

input()
```

Bu sayede programımız kullanıcıdan bir giriş bekleyecek ve o girişi alana kadar da kapanmayacaktır. Programı kapatmak için *Enter* düğmesine basabiliriz.

input() bir fonksiyondur dedik. Henüz fonksiyon kavramının ayrıntılarını öğrenmemiş olsak da, şimdiye kadar pek çok fonksiyon gördüğümüz için artık bir fonksiyonla karşılaştığımızda bunun nasıl kullanılacağını az çok tahmin edebiliyoruz. Tıpkı düşündüğünüz ve yukarıdaki örnekten de gördüğünüz gibi, birer fonksiyon olan type(), print(), len() ve open() fonksiyonlarını nasıl kullanıyorsak input() fonksiyonunu da öyle kullanacağız.

Dilerseniz lafı daha fazla uzatmadan örnek bir program yazalım:

```
isim = input("İsminiz nedir? ")
```

```
print("Merhaba", isim, end="!\n")
```

Bu programı kaydedip çalıştırdığınızda, sorulan soruya verdiğiniz cevaba göre çıktı farklı olacaktır. Örneğin eğer bu soruya 'Niyazi' cevabını vermişseniz çıktınız *Merhaba Niyazi!* şeklinde olacaktır.

Görüyorsunuz ya, tıpkı daha önce gördüğümüz fonksiyonlarda olduğu gibi, input() fonksiyonunda da parantez içine bir parametre yazıyoruz. Bu fonksiyona verilen parametre, kullanıcıdan veri alınırken kullanıcıya sorulacak soruyu gösteriyor. Gelin isterseniz bir örnek daha yapalım elimizin alışması için:

```
yaş = input("Yaşınız: ")

print("Demek", yaş, "yaşındasın.")
print("Genç mi yoksa yaşlı mı olduğuna karar veremedim.")
```

input() fonksiyonunun ne kadar kullanışlı bir araç olduğu ortada. Bu fonksiyon sayesinde, şimdiye kadar tek sesli bir şekilde yürüttüğümüz programcılık faaliyetlerimizi çok sesli bir hale getirebileceğiz. Mesela önceki bölümlerden birinde yazdığımız, daire alanı hesaplayan programı hatırlarsınız. O zaman henüz dosyalarımızı kaydetmeyi ve input() fonksiyonunu öğrenmediğimiz için o programı etkileşimli kabukta şu şekilde yazmıştık:

```
>>> çap = 16
>>> yarıçap = çap / 2
>>> pi = 3.14159
>>> alan = pi * (yarıçap * yarıçap)
>>> alan

201.06176
```

Ama artık hem dosyalarımızı kaydetmeyi biliyoruz, hem de input() fonksiyonunu öğrendik. Dolayısıyla yukarıdaki programı şu şekilde yazabiliriz:

```
#Kullanıcıdan dairenin çapını girmesini istiyoruz.
çap = input("Dairenin çapı: ")

#Kullanıcının verdiği çap bilgisini kullanarak
#yarıçapı hesaplayalım. Buradaki int() fonksiyonunu
#ilk kez görüyoruz. Biraz sonra bunu açıklayacağız
yarıçap = int(çap) / 2

#pi sayımız sabit
pi = 3.14159

#Yukarıdaki bilgileri kullanarak artık
```

```
#dairenin alanını hesaplayabiliriz
alan = pi * (yarıçap * yarıçap)

#Son olarak, hesapladığımız alanı yazdırıyoruz
print("Çapı", çap, "cm olan dairenin alanı: ", alan, "cm2'dir")
```

Gördüğünüz gibi, input() fonksiyonunu öğrenmemiz sayesinde artık yavaş yavaş işe yarar programlar yazabiliyoruz.

Ancak burada, daha önce öğrenmediğimiz bir fonksiyon dikkatinizi çekmiş olmalı. Bu fonksiyonun adı int(). Bu yeni fonksiyon dışında, yukarıdaki bütün kodları anlayabilecek kadar Python bilgisine sahibiz.

int() fonksiyonunun ne işe yaradığını anlamak için isterseniz ilgili satırı yarıçap = çap / 2 şeklinde yazarak çalıştırmayı deneyin bu programı.

Dediğim gibi, eğer o satırdaki int() fonksiyonunu kaldırarak programı çalıştırdıysanız şuna benzer bir hata mesajı almış olmalısınız:

```
Traceback (most recent call last):
  File "deneme.py", line 8, in <module>
    yarıçap = çap / 2
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Gördüğünüz gibi programımız bölme işlemini yapamadı. Buradan anlıyoruz ki, bu int() fonksiyonu programımızdaki aritmetik işlemin düzgün bir şekilde yapılabilmesini sağlıyor. Gelelim bu fonksiyonun bu işlevi nasıl yerine getirdiğini incelemeye.

Tip Dönüşümleri

Bir önceki bölümün sonunda verdiğimiz örnek programda int() adlı bir fonksiyon görmüş, bu fonksiyonu anlatmayı o zaman ertelemiştik. Çok gecikmeden, bu önemli fonksiyonun ne işe yaradığını öğrenmemiz gerekiyor. İsterseniz bir örnek üzerinden gidelim.

Diyelim ki kullanıcıdan aldığı sayının karesini hesaplayan bir program yazmak istiyoruz. Öncelikle şöyle bir şey deneyelim:

```
sayı = input("Lütfen bir sayı girin: ")

#Girilen sayının karesini bulmak için sayı değişkeninin 2.
#kuvvetini alıyoruz. Aynı şeyi pow() fonksiyonu ile de
#yapabileceğimizi biliyorsunuz. Örn.: pow(sayı, 2)
print("Girdiğiniz sayının karesi: ", sayı ** 2)
```

Bu kodları çalıştırdığımız zaman, programımız kullanıcıdan bir sayı girmesini isteyecek, ancak kullanıcı bir sayı girip *Enter* tuşuna bastığında şöyle bir hata mesajıyla karşılaşacaktır:

```
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    print("Girdiğiniz sayının karesi: ", sayı ** 2)
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

Hata mesajına baktığınızda, 'TypeError' ifadesinden, bunun veri tipine ilişkin bir hata olduğunu tahmin edebilirsiniz. Eğer İngilizce biliyorsanız yukarıdaki hata mesajının anlamını rahatlıkla çıkarabilirsiniz. İngilizce bilmeseniz de en sondaki 'str' ve 'int' kelimeleri size karakter dizisi ve sayı adlı veri tiplerini hatırlatacaktır. Demek ki ortada veri tiplerini ilgilendiren bir sorun var...

Peki burada tam olarak neler dönüyor?

Hatırlayacaksınız, geçen derslerden birinde len() fonksiyonunu anlatırken şöyle bir şey söylemiştik:

> Biz henüz kullanıcıdan nasıl veri alacağımızı bilmiyoruz. Ama şimdilik şunu söyleyebiliriz: Python'da kullanıcıdan herhangi bir veri aldığımızda, bu veri bize bir karakter dizisi olarak gelecektir.

Gelin isterseniz yukarıda anlattığımız durumu teyit eden bir program yazalım:

```
#Kullanıcıdan herhangi bir veri girmesini istiyoruz
sayı = input("Herhangi bir veri girin: ")

#Kullanıcının girdiği verinin tipini bir
#değişkene atıyoruz
tip = type(sayı)

#Son olarak kullanıcının girdiği verinin tipini
#ekrana basıyoruz.
print("Girdiğiniz verinin tipi: ", tip)
```

Bu programı çalıştırdığımızda ne tür bir veri girersek girelim, girdiğimiz verinin tipi *str*, yani karakter dizisi olacaktır. Demek ki gerçekten de, kullanıcıdan veri almak için kullandığımız input() fonksiyonu bize her koşulda bir karakter dizisi veriyormuş.

Geçen derslerde şöyle bir şey daha söylemiştik:

> Python'da, o anda elinizde bulunan bir verinin hangi tipte olduğunu bilmek son derece önemlidir. Çünkü bir verinin ait olduğu tip, o veriyle neler yapıp neler yapamayacağınızı belirler.

Şu anda karşı karşıya olduğumuz durum da buna çok güzel bir örnektir. Eğer o anda elimizde bulunan verinin tipini bilmezsek tıpkı yukarıda olduğu gibi, o veriyi programımızda kullanmaya çalışırken programımız hata verir ve çöker.

Her zaman üstüne basa basa söylediğimiz gibi, aritmetik işlemler yalnızca sayılarla yapılır. Karakter dizileri ile herhangi bir aritmetik işlem yapılamaz. Dolayısıyla, input() fonksiyonundan gelen veri bir karakter dizisi olduğu için ve biz de programımızda girilen sayının karesini hesaplamak amacıyla bu fonksiyondan gelen verinin *2.* kuvvetini, yani karesini hesaplamaya çalıştığımız için programımız hata verecektir.

Yukarıdaki programda neler olup bittiğini daha iyi anlayabilmek için Python'ın etkileşimli kabuğunda şu işlemleri yapabiliriz:

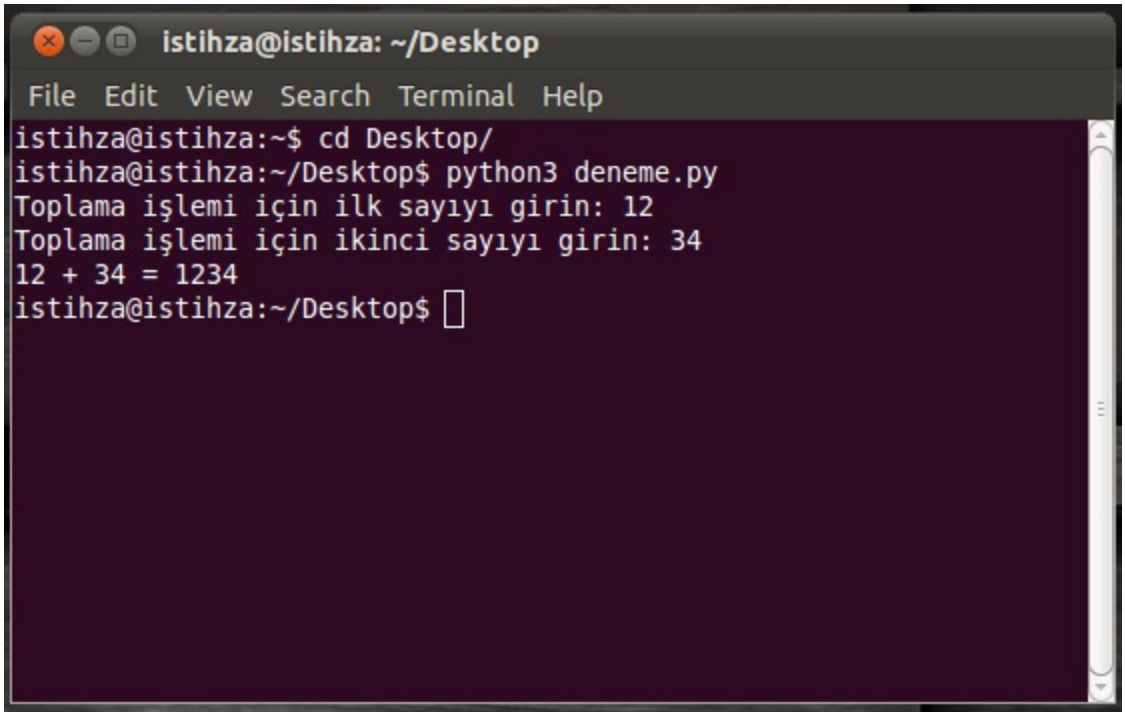
```
>>> "23" ** 2

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

Gördüğünüz gibi, programımızdan aldığımız hata ile yukarıdaki hata tamamen aynı (hata mesajlarında bizi ilgilendiren kısım en son satırdır). Tıpkı burada olduğu gibi, hata veren programda da 'Lütfen bir sayı girin: ' sorusuna örneğin *23* cevabını verdiğimizde programımız aslında "23" ** 2 gibi bir işlem yapmaya çalışıyor. Bir karakter dizisinin kuvvetini hesaplamak mümkün olmadığı, kuvvet alma işlemi yalnızca sayılarla yapılabileceği için de hata vermekten başka çaresi kalmıyor.

Ancak bazen öyle durumlarla karşılaşırsınız ki, programınız hiçbir hata vermez, ama elde edilen sonuç aslında tamamen beklentinizin dışındadır. Mesela şu basit örneği inceleyelim:

```
sayı1 = input("Toplama işlemi için ilk sayıyı girin: ")
sayı2 = input("Toplama işlemi için ikinci sayıyı girin: ")

print(sayı1, "+", sayı2, "=", sayı1 + sayı2)
```

Bu kodları çalıştırdığımızda şöyle bir manzarayla karşılaşırız:



input() fonksiyonunun alttan alta neler çevirdiğini bu örnek yardımıyla çok daha iyi anladığınızı zannediyorum. Gördüğünüz gibi yukarıdaki program herhangi bir hata vermedi. Ama beklediğimiz çıktıyı da vermedi. Zira biz programımızın iki sayıyı toplamasını istiyorduk. O ise kullanıcının girdiği sayıları yan yana yazmakla yetindi. Yani bir aritmetik işlem yapmak yerine, verileri birbiriyle bitiştirdi. Çünkü, dediğim gibi, input() fonksiyonunun kullanıcıdan aldığı şey bir karakter dizisidir. Dolayısıyla bu fonksiyon yukarıdaki gibi bir durumla karşılaştığı zaman karakter dizileri arasında bir birleştirme işlemi gerçekleştirir. Tıpkı ilk derslerimizde etkileşimli kabukta verdiğimiz şu örnekte olduğu gibi:

```
>>> "23" + "23"

2323
```

Bu son örnekten ayrıca şunu çıkarıyoruz: Yazdığınız bir programın herhangi bir hata vermemesi o programın doğru çalıştığı anlamına gelmeyebilir. Dolayısıyla bu tür durumlara karşı her zaman uyanık olmanızda fayda var.

Peki yukarıdaki gibi durumlarla karşılaşmamak için ne yapacağız?

İşte bu noktada devreye tip dönüştürücü adını verdiğimiz birtakım fonksiyonlar girecek.

int()

Dediğimiz gibi, input() fonksiyonundan gelen veri her zaman bir karakter dizisidir. Dolayısıyla bu fonksiyondan gelen veriyle herhangi bir aritmetik işlem yapabilmek için öncelikle bu veriyi bir sayıya dönüştürmemiz gerekir. Bu dönüştürme işlemi için int() adlı özel bir dönüştürücü fonksiyondan yararlanacağız. Gelin isterseniz Python'ın etkileşimli kabuğunda bu fonksiyonla bir kaç deneme yaparak bu fonksiyonun ne işe yaradığını ve nasıl kullanıldığını anlamaya çalışalım. Zira etkileşimli

kabuk bu tür deneme işlemleri için biçilmiş kaftandır:

```
>>> karakter_dizisi = "23"
>>> sayı = int(karakter_dizisi)
>>> print(sayı)

23
```

Burada öncelikle *"23"* adlı bir karakter dizisi tanımladık. Ardından da int() fonksiyonunu kullanarak bu karakter dizisini bir tamsayıya (*integer*) dönüştürdük. İsminden de anlayacağınız gibi int() fonksiyonu İngilizce *integer* (tamsayı) kelimesinin kısaltmasıdır ve bu fonksiyonun görevi bir veriyi tamsayıya dönüştürmektir.

Ancak burada dikkat etmemiz gereken bir şey var. Herhangi bir verinin sayıya dönüştürülebilmesi için o verinin sayı değerli bir veri olması gerekir. Örneğin *"23"*, sayı değerli bir karakter dizisidir. Ama mesela *"elma"* sayı değerli bir karakter dizisi değildir. Bu yüzden *"elma"* karakter dizisi sayıya dönüştürülemez:

```
>>> karakter_dizisi = "elma"
>>> sayı = int(karakter_dizisi)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'elma'
```

Gördüğünüz gibi, sayı değerli olmayan bir veriyi sayıya dönüştürmeye çalıştırdığımızda Python bize bir hata mesajı gösteriyor. Yazdığımız programlarda bu duruma özellikle dikkat etmemiz gerekiyor.

Şimdi bu bölümün başında yazdığımız ve hata veren programımıza dönelim yine:

```
sayı = input("Lütfen bir sayı girin: ")

print("Girdiğiniz sayının karesi: ", sayı ** 2)
```

Bu kodların hata vereceğini biliyoruz. Ama artık, öğrendiğimiz int() dönüştürücüsünü kullanarak programımızı hata vermeyecek şekilde yeniden yazabiliriz:

```
veri = input("Lütfen bir sayı girin: ")

#input() fonksiyonundan gelen karakter dizisini
#sayıya dönüştürüyoruz.
sayı = int(veri)

print("Girdiğiniz sayının karesi: ", sayı ** 2)
```

Artık programımız hatasız bir şekilde çalışıyor.

Bir de öteki örneğimizi ele alalım:

```
sayı1 = input("Toplama işlemi için ilk sayıyı girin: ")
sayı2 = input("Toplama işlemi için ikinci sayıyı girin: ")

print(sayı1, "+", sayı2, "=", sayı1 + sayı2)
```

Bu kodların beklediğimiz çıktıyı vermeyeceğini biliyoruz. Ama eğer bu kodları şöyle yazarsak işler değişir:

```
v1 = input("Toplama işlemi için ilk sayıyı girin: ")
v2 = input("Toplama işlemi için ikinci sayıyı girin: ")

sayı1 = int(v1) #v1 adlı karakter dizisini sayıya dönüştürüyoruz.
sayı2 = int(v2) #v2 adlı karakter dizisini sayıya dönüştürüyoruz.

print(sayı1, "+", sayı2, "=", sayı1 + sayı2)
```

Gördüğünüz gibi, input() fonksiyonundan gelen karakter dizilerini sayıya dönüştürerek istediğimiz çıktıyı alabiliyoruz.

str()

Python'daki tip dönüştürücüleri elbette sadece int() fonksiyonuyla sınırlı değildir. Gördüğünüz gibi, int() fonksiyonu sayı değerli verileri (mesela karakter dizilerini) tam sayıya dönüştürüyor. Bunun bir de tersi mümkündür. Yani karakter dizisi olmayan verileri karakter dizisine dönüştürmemiz de mümkündür. Bu işlem için str() adlı başka bir tip dönüştürücüden yararlanıyoruz:

```
>>> sayı = 23
>>> kardiz = str(sayı)
>>> print(kardiz)

23

>>> print(type(kardiz))

<class 'str'>
```

Gördüğünüz gibi, bir tam sayı olan *23*'ü str() adlı bir fonksiyondan yararlanarak karakter dizisi olan *"23"* ifadesine dönüştürdük. Son satırda da, elde ettiğimiz şeyin bir karakter dizisi olduğundan emin olmak için type() fonksiyonunu kullanarak verinin tipini denetledik.

Yukarıdaki örneklerden gördüğümüz gibi, aritmetik işlemler yapmak istediğimizde karakter dizilerini sayıya çevirmemiz gerekiyor. Peki acaba hangi durumlarda bunun tersini yapmamız, yani sayıları karakter dizilerine çevirmemiz gerekir? Python bilginiz ve tecrübeniz arttıkça bunların hangi durumlar olduğunu kendiniz de göreceksiniz. Mesela biz daha şimdiden, sayıları karakter dizisine çevirmemiz gereken bir durumla karşılaştık. Hatırlarsanız, len() fonksiyonunu anlatırken, bu fonksiyonun sayılarla birlikte kullanılamayacağını söylemiştik:

```
>>> len(12343423432)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
```

Peki ya yazdığınız programda bir sayının kaç haneden oluştuğunu hesaplamanız gerekirse ne yapacaksınız? Yani mesela yukarıdaki sayının 11 haneli olduğunu bilmeniz gerekiyorsa ne olacak?

İşte böyle bir durumda str() fonksiyonundan yararlanabilirsiniz:

```
>>> sayı = 12343423432
>>> kardiz = str(sayı)
>>> len(kardiz)

11
```

Bildiğiniz gibi, len() fonksiyonu, şu ana kadar öğrendiğimiz veri tipleri içinde sadece karakter dizileri üzerinde işlem yapabiliyor. Biz de bu yüzden, sayımızın kaç haneli olduğunu öğrenebilmek için, öncelikle bu sayıyı bir karakter dizisine çeviriyoruz. Daha sonra da elde ettiğimiz bu karakter dizisini len() fonksiyonuna parametre olarak veriyoruz. Böylece sayının kaç haneli olduğu bilgisini elde etmiş oluyoruz.

Bu arada elbette yukarıdaki işlemi tek satırda da halledebilirsiniz:

```
>>> len(str(12343423432))

11
```

Bu şekilde iç içe geçmiş fonksiyonlar yazdığımızda, Python fonksiyonları içten dışa doğru tek tek değerlendirecektir. Mesela yukarıdaki örnekte Python önce str(12343423432) ifadesini değerlendirecek ve çıkan sonucu len() fonksiyonuna gönderecektir. İç içe geçmiş fonksiyonları yazarken dikkat etmemiz gereken önemli bir nokta da, açtığımız her bir parantezi tek tek kapatmayı unutmamaktır.

float()

Hatırlarsanız ilk bölümlerde sayılardan söz ederken tamsayıların (*integer*) dışında kayan noktalı

sayıların (*float*) da olduğundan söz etmiştik. İşte eğer bir tamsayıyı veya sayı değerli bir karakter dizisini kayan noktalı sayıya dönüştürmek istersek float() adlı başka bir dönüştürücüden yararlanacağız:

```
>>> a = 23
>>> type(a)

<class 'int'>

>>> float(a)

23.0
```

Gördüğünüz gibi, *23* tamsayısı, float() fonksiyonu sayesinde *23.0*'a yani bir kayan noktalı sayıya dönüştü.

Aynı şeyi, sayı değerli karakter dizileri üzerine uygulamak da mümkündür:

```
>>> b = "23"
>>> type(b)

<class 'str'>

>>> float(b)

23.0
```

complex()

Sayılardan söz ederken, eğer matematikle çok fazla içli dışlı değilseniz pek karşılaşmayacağınız, 'karmaşık sayı' adlı bir sayı türünden de bahsetmiştik. Karmaşık sayılar Python'da 'complex' ifadesiyle gösteriliyor. Mesela şunun bir karmaşık sayı olduğunu biliyoruz:

```
>>> 12+0j
```

Kontrol edelim:

```
>>> type(12+0j)

<class 'complex'>
```

İşte eğer herhangi bir sayıyı karmaşık sayıya dönüştürmeniz gerekirse complex() adlı bir fonksiyondan yararlanabilirsiniz. Örneğin:

```
>>> complex(15)
```

```
(15+0j)
```

Böylece Python'daki bütün sayı dönüştürücüleri öğrenmiş olduk.

Gelin isterseniz, bu bölümde anlattığımız konuları şöyle bir tekrar ederek bilgilerimizi sağlamlaştırmaya çalışalım.

```
>>> a = 56
```

Bu sayı bir tamsayıdır. İngilizce olarak ifade etmek gerekirse, *integer*. Bunun bir tamsayı olduğunu şu şekilde teyit edebileceğimizi gayet iyi biliyorsunuz:

```
>>> type(a)

<class 'int'>
```

Burada aldığımız *<class int>* çıktısı, bize *a* değişkeninin tuttuğu sayının bir tamsayı olduğunu söylüyor. 'int' ifadesi, *integer* (tamsayı) kelimesinin kısaltmasıdır.

Bir de şu sayıya bakalım:

```
>>> b = 34.5
>>> type(b)

<class 'float'>
```

Bu çıktı ise bize *34.5* sayısının bir kayan noktalı sayı olduğunu söylüyor. *float* kelimesi *Floats* veya *Floating Point Number* ifadesinin kısaltmasıdır. Yani 'kayan noktalı sayı' demektir.

Bu arada, bu type() adlı fonksiyonu sadece sayılara değil, başka şeylere de uygulayabileceğimizi biliyorsunuz. Mesela bir örnek vermek gerekirse:

```
>>> meyve = "karpuz"
>>> type(meyve)

<class 'str'>
```

Gördüğünüz gibi, type() fonksiyonu bize *meyve* adlı değişkenin değerinin bir 'str' yani *string* yani karakter dizisi olduğunu bildirdi.

Bu veri tipleri arasında, bazı özel fonksiyonları kullanarak dönüştürme işlemi yapabileceğimizi öğrendik. Mesela:

```
>>> sayı = 45
```

sayı adlı değişkenin tuttuğu verinin değeri bir tamsayıdır. Biz bu tamsayıyı kayan noktalı sayıya dönüştürmek istiyoruz. Yapacağımız işlem çok basit:

```
>>> float(sayı)

45.0
```

Gördüğünüz gibi, *45* adlı tamsayıyı, *45.0* adlı bir kayan noktalı sayıya dönüştürdük. Şimdi type(45.0) komutu bize *<class 'float'>* çıktısını verecektir.

Eğer kayan noktalı bir sayıyı tamsayıya çevirmek istersek şu komutu veriyoruz. Mesela kayan noktalı sayımız, *56.5* olsun:

```
>>> int(56.5)

56
```

Yukarıdaki örneği tabii ki şöyle de yazabiliriz:

```
>>> a = 56.5
>>> int(a)

56
```

Dönüştürme işlemini sayılar arasında yapabileceğimiz gibi, sayılar ve karakter dizileri arasında da yapabiliriz. Örneğin şu bir karakter dizisidir:

```
>>> nesne = "45"
```

Yukarıdaki değeri tırnak içinde belirttiğimiz için bu değer bir karakter dizisidir. Şimdi bunu bir tamsayıya çevireceğiz:

```
>>> int(nesne)

45
```

Dilersek, aynı karakter dizisini kayan noktalı sayıya da çevirebiliriz:

```
>>> float(nesne)

45.0
```

Hatta bir sayıyı karakter dizisine de çevirebiliriz. Bunun için *string* (karakter dizisi) kelimesinin kısaltması olan *str* ifadesini kullanacağız:

```
>>> s = 6547
```

```
>>> str(s)

'6547'
```

Bir örnek de kayan noktalı sayılarla yapalım:

```
>>> s = 65.7
>>> str(s)

'65.7'
```

Yalnız şunu unutmayın: Bir karakter dizisinin sayıya dönüştürülebilmesi için o karakter dizisinin sayı değerli olması lazım. Yani *"45"* değerini sayıya dönüştürebiliriz. Çünkü *"45"* değeri, tırnaklardan ötürü bir karakter dizisi de olsa, neticede sayı değerli bir karakter dizisidir. Ama mesela *"elma"* karakter dizisi böyle değildir. Dolayısıyla, şöyle bir maceraya girişmek bizi hüsrana uğratacaktır:

```
>>> nesne = "elma"
>>> int(nesne)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'elma'
```

Gördüğünüz gibi, Python böyle bir işlem denemesi karşısında hata veriyor...

Bu bölümde pek çok yeni şey öğrendik. Bu bölümün en önemli getirisi input() fonksiyonunu öğrenmemiz oldu. Bu fonksiyon sayesinde kullanıcıyla iletişim kurmayı başardık. Artık kullanıcıdan veri alıp, bu verileri programlarımız içinde işleyebiliyoruz.

Yine bu bölümde dikkatinizi çektiğimiz başka bir konu da sayılar ve karakter dizileri arasındaki ilişkiydi. input() fonksiyonuyla elde edilen çıktının bir karakter dizisi olduğunu öğrendik. Bildiğimiz gibi, aritmetik işlemler ancak sayılar arasında yapılabilir. Dolayısıyla input() fonksiyonuyla gelen karakter dizisini bir sayıyla çarpmaya kalkarsak hata alıyoruz. Burada yapmamız gereken şey, elimizdeki verileri dönüştürmek. Yani input() fonksiyonundan gelen karakter dizisini bir sayıyla çarpmak istiyorsak, öncelikle aldığımız karakter dizisini sayıya dönüştürmemiz gerekiyor. Dönüştürme işlemleri için kullandığımız fonksiyonlar şunlardı:

int()
: Sayı değerli bir karakter dizisini veya kayan noktalı sayıyı tamsayıya (*integer*) çevirir.

float()
: Sayı değerli bir karakter dizisini veya tamsayıyı kayan noktalı sayıya (*float*) çevirir.

str()
: Bir tamsayı veya kayan noktalı sayıyı karakter dizisine (*string*) çevirir.

complex()
Herhangi bir sayıyı veya sayı değerli karakter dizisini karmaşık sayıya (*complex*) çevirir.

Ayrıca bu bölümde öğrendiklerimiz, şöyle önemli bir tespitte bulunmamıza da olanak tanıdı:

Her tamsayı ve/veya kayan noktalı sayı bir karakter dizisine dönüştürülebilir. Ama her karakter dizisi tamsayıya ve/veya kayan noktalı sayıya dönüştürülemez.

Örneğin, *5654* gibi bir tamsayıyı veya *543.34* gibi bir kayan noktalı sayıyı str() fonksiyonu yardımıyla karakter dizisine dönüştürebiliriz:

```
>>> str(5654)
>>> str(543.34)
```

"5654" veya *"543.34"* gibi bir karakter dizisini int() veya float() fonksiyonu yardımıyla tamsayıya ya da kayan noktalı sayıya da dönüştürebiliriz:

```
>>> int("5654")
>>> int("543.34")

>>> float("5654")
>>> float("543.34")
```

Ama *"elma"* gibi bir karakter dizisini ne int() ne de float() fonksiyonuyla tamsayıya veya kayan noktalı sayıya dönüştürebiliriz! Çünkü *"elma"* verisi sayı değerli değildir.

Bu bölümü kapatmadan önce, dilerseniz şimdiye kadar öğrendiklerimizi de içeren örnek bir program yazalım. Bu program, Python maceramız açısından bize yeni kapılar da açacak.

Önceki derslerimizin birinde verdiğimiz doğalgaz faturası hesaplayan programı hatırlarsınız. İşte artık input() fonksiyonu sayesinde bu doğalgaz faturası hesaplama programını da daha ilginç bir hale getirebiliriz:

```
#Her bir ayın kaç gün çektiğini tanımlıyoruz
ocak = mart = mayıs = temmuz = ağustos = ekim = aralık = 31
nisan = haziran = eylül = kasım = 30
şubat = 28

#Doğalgazın vergiler dahil metreküp fiyatı
birimFiyat = 0.79

#Kullanıcı ayda ne kadar doğalgaz tüketmiş?
aylıkSarfiyat = input("Aylık doğalgaz sarfiyatınızı metreküp olarak giriniz: ")

#Kullanıcı hangi aya ait faturasını öğrenmek istiyor?
```

```
dönem = input("""Hangi aya ait faturayı hesaplamak istersiniz?
(Lütfen ay adını tamamı küçük harf olacak şekilde giriniz)\n""")

#Yukarıdaki input() fonksiyonundan gelen veriyi
#Python'ın anlayabileceği bir biçime dönüştürüyoruz
ay = eval(dönem)

#Kullanıcının günlük doğalgaz sarfiyatı
günlükSarfiyat = int(aylıkSarfiyat) / ay

#Fatura tutarı
fatura = birimFiyat * günlükSarfiyat * ay

print("günlük sarfiyatınız: \t", günlükSarfiyat, " metreküp\n",
"tahmini fatura tutarı: \t", fatura, " TL", sep="")
```

Burada yine bilmediğimiz bir fonksiyonla daha karşılaştık. Bu fonksiyonun adı eval(). Biraz sonra eval() fonksiyonunu derinlemesine inceleyeceğiz. Ama bu fonksiyonu anlatmaya geçmeden önce dilerseniz yukarıdaki kodları biraz didikleyelim.

İlk satırların ne işe yaradığını zaten biliyorsunuz. Bir yıl içindeki bütün ayların kaç gün çektiğini gösteren değişkenlerimizi tanımladık. Burada her bir değişkeni tek tek tanımlamak yerine değişkenleri topluca tanımladığımıza dikkat edin. İsteseydik tabii ki yukarıdaki kodları şöyle de yazabilirdik:

```
#Her bir ayın kaç gün çektiğini tanımlıyoruz
ocak    = 31
şubat   = 28
mart    = 31
nisan   = 30
mayıs   = 31
haziran = 30
temmuz  = 31
ağustos = 31
eylül   = 30
ekim    = 31
kasım   = 30
aralık  = 31

#Doğalgazın vergiler dahil m3 fiyatı
birimFiyat = 0.79

#Kullanıcı ayda ne kadar doğalgaz tüketmiş?
aylıkSarfiyat = input("Aylık doğalgaz sarfiyatınızı m3 olarak giriniz: ")
```

```
#Kullanıcı hangi aya ait faturasını öğrenmek istiyor?
dönem = input("""Hangi aya ait faturayı hesaplamak istersiniz?
(Lütfen ay adını tamamı küçük harf olacak şekilde giriniz)\n""")

#Yukarıdaki input() fonksiyonundan gelen veriyi
#Python'ın anlayabileceği bir biçime dönüştürüyoruz
ay = eval(dönem)

#Kullanıcının günlük doğalgaz sarfiyatı
günlükSarfiyat = int(aylıkSarfiyat) / ay

#Fatura tutarı
fatura = birimFiyat * günlükSarfiyat * ay

print("günlük sarfiyatınız: \t", günlükSarfiyat, " metreküp\n",
"tahmini fatura tutarı: \t", fatura, " TL", sep="")
```

Ama tabii ki, değişkenleri tek tek tanımlamak yerine topluca tanımlamak, daha az kod yazmanızı sağlamasının yanısıra, programınızın çalışma performansı açısından da daha iyidir. Yani değişkenleri bu şekilde tanımladığınızda programınız daha hızlı çalışır.

Programımızı incelemeye devam edelim...

Değişkenleri tanımladıktan sonra doğalgazın vergiler dahil yaklaşık birim fiyatını da bir değişken olarak tanımladık. *0.79* değerini zaten birkaç bölüm önce hesaplayıp bulduğumuz için, aynı işlemleri tekrar programımıza eklememize gerek yok. Doğrudan nihai değeri programımıza yazsak yeter...

Birim fiyatı belirledikten sonra kullanıcıya aylık doğalgaz sarfiyatını soruyoruz. Kullanıcının bu değeri m^3 olarak girmesini bekliyoruz. Elbette bu veriyi kullanıcıdan alabilmek için input() fonksiyonunu kullanıyoruz.

Daha sonra kullanıcıya hangi aya ait doğalgaz faturasını ödemek istediğini soruyoruz. Bu bilgi, bir sonraki satırda günlük doğalgaz sarfiyatını hesaplarken işimize yarayacak. Çünkü kullanıcının girdiği ayın çektiği gün sayısına bağlı olarak günlük sarfiyat değişecektir. Günlük sarfiyatı hesaplamak için aylık sarfiyatı, ilgili ayın çektiği gün sayısına bölüyoruz. Bu arada bir önceki satırda *dönem* değişkenini eval() adlı bir fonksiyonla birlikte kullandığımızı görüyorsunuz. Bunu biraz sonra inceleyeceğiz. O yüzden bu satırları atlayıp son satıra gelelim.

Son satırda print() fonksiyonunu kullanarak, kullanıcıdan aldığımız verileri düzgün bir şekilde kendisine gösteriyoruz. Programımız kullanıcıya günlük doğalgaz sarfiyatını ve ay sonunda karşılaşacağı tahmini fatura tutarını bildiriyor. print() fonksiyonu içinde kullandığımız kaçış dizilerine özellikle dikkatinizi çekmek istiyorum. Burada düzgün bir çıktı elde etmek için *\t* ve *\n* adlı kaçış dizilerinden nasıl yararlandığımızı görüyorsunuz. Bu kaçış dizilerinin buradaki işlevini tam olarak anlayabilmek için, bu kodları bir de bu kaçış dizileri olmadan yazmayı deneyebilirsiniz.

Bu bilgileri, önemlerinden ötürü aklımızda tutmaya çalışalım. Buraya kadar anlatılan konular hakkında zihnimizde belirsizlikler varsa veya bazı noktaları tam olarak kavrayamadıysak, şimdiye kadar öğrendiğimiz konuları tekrar gözden geçirmemiz bizim için epey faydalı olacaktır. Zira bundan sonraki bölümlerde, yeni bilgilerin yanısıra, buraya kadar öğrendiğimiz şeyleri de yoğun bir şekilde pratiğe dökeceğiz. Bundan sonraki konuları takip edebilmemiz açısından, buraya kadar verdiğimiz temel bilgileri iyice sindirmiş olmak işimizi bir hayli kolaylaştıracaktır.

eval() ve exec() Fonksiyonları

Bir önceki bölümün son örnek programında eval() adlı bir fonksiyonla karşılaşmıştık. İşte şimdi bu önemli fonksiyonun ne işe yaradığını anlamaya çalışacağız. Ancak eval() fonksiyonunu anlatmaya başlamadan önce şu uyarıyı yapalım:

eval() ŞEYTANİ GÜÇLERİ OLAN BİR FONKSİYONDUR!

Bunun neden böyle olduğunu hem biz anlatacağız, hem de zaten bu fonksiyonu tanıdıkça neden eval()'e karşı dikkatli olmanız gerektiğini kendiniz de anlayacaksınız.

Dilerseniz işe basit bir eval() örneği vererek başlayalım:

```
print("""
Basit bir hesap makinesi uygulaması.

İşleçler:

   +   toplama
   -   çıkarma
   *   çarpma
   /   bölme

Yapmak istediğiniz işlemi yazıp ENTER
tuşuna basın. (Örneğin 23 ve 46 sayılarını
çarpmak için 23 * 46 yazdıktan sonra
ENTER tuşuna basın.)
""")

veri = input("İşleminiz: ")
hesap = eval(veri)

print(hesap)
```

İngilizcede *evaluate* diye bir kelime bulunur. Bu kelime, 'değerlendirmeye tabi tutmak, işleme sokmak, işlemek' gibi anlamlar taşır. İşte eval() fonksiyonundaki *eval* kelimesi bu *evaluate* kelimesinin kısaltmasıdır. Yani bu fonksiyonun görevi, kendisine verilen karakter dizilerini

değerlendirmeye tabi tutmak ya da işlemektir. Peki bu tam olarak ne anlama geliyor?

Aslında yukarıdaki örnek programı çalıştırdığımızda bu sorunun yanıtını kendi kendimize verebiliyoruz. Bu programı çalıştırarak, *"İşleminiz: "* ifadesinden sonra, örneğin, 45 * 76 yazıp *Enter* tuşuna basarsak programımız bize *3420* çıktısı verecektir. Yani programımız hesap makinesi işlevini yerine getirip *45* sayısı ile *76* sayısını çarpacaktır. Dolayısıyla, yukarıdaki programı kullanarak her türlü aritmetik işlemi yapabilirsiniz. Hatta bu program, son derece karmaşık aritmetik işlemlerin yapılmasına dahi müsaade eder.

Peki programımız bu işlevi nasıl yerine getiriyor? İsterseniz kodların üzerinden tek tek geçelim.

Öncelikle programımızın en başına kullanım kılavuzuna benzer bir metin yerleştirdik ve bu metni print() fonksiyonu yardımıyla ekrana bastık.

Daha sonra kullanıcıdan alacağımız komutları *veri* adlı bir değişkene atadık. Tabii ki kullanıcıyla iletişimi her zaman olduğu gibi input() fonksiyonu yardımıyla sağlıyoruz.

Ardından, kullanıcıdan gelen veriyi eval() fonksiyonu yardımıyla değerlendirmeye tabi tutuyoruz. Yani kullanıcının girdiği komutları işleme sokuyoruz. Örneğin, kullanıcı 46 / 2 gibi bir veri girdiyse, biz eval() fonksiyonu yardımıyla bu 46 / 2 komutunu işletiyoruz. Bu işlemin sonucunu da *hesap* adlı başka bir değişken içinde depoluyoruz.

Eğer burada eval() fonksiyonunu kullanmazsak, programımız, kullanıcının girdiği 45 * 76 komutunu hiçbir işleme sokmadan dümdüz ekrana basacaktır. Yani:

```
print("""
Basit bir hesap makinesi uygulaması.

İşleçler:

   +   toplama
   -   çıkarma
   *   çarpma
   /   bölme

Yapmak istediğiniz işlemi yazıp ENTER
tuşuna basın. (Örneğin 23 ve 46 sayılarını
çarpmak için 23 * 46 yazdıktan sonra
ENTER tuşuna basın.)
""")

veri = input("İşleminiz: ")

print(veri)
```

Eğer programımızı yukarıdaki gibi, eval() fonksiyonu olmadan yazarsak, kullanıcımız 45 * 76 gibi bir komut girdiğinde alacağı cevap dümdüz bir *45 * 76* çıktısı olacaktır. İşte eval() fonksiyonu, kullanıcının girdiği her veriyi bir Python komutu olarak algılar ve bu veriyi işleme sokar. Yani 45 * 76 gibi bir şey gördüğünde, bu şeyi doğrudan ekrana yazdırmak yerine, işlemin sonucu olan *3420* sayısını verir.

eval() fonksiyonunun, yukarıda anlattığımız özelliklerini okuduktan sonra, 'Ne güzel bir fonksiyon! Her işimi görür bu!' dediğinizi duyar gibiyim. Ama aslında durum hiç de öyle değil. Neden mi?

Şimdi yukarıdaki programı tekrar çalıştırın ve *"İşleminiz: "* ifadesinden sonra şu cevabı verin:

```
print("Merhaba Python!")
```

Bu komut şöyle bir çıktı vermiş olmalı:

```
Merhaba Python!
None
```

Not

Buradaki *None* değerini görmezden gelin. Bunu fonksiyonlar konusunu anlatırken inceleyeceğiz.

Gördüğünüz gibi, yazdığımız program, kullanıcının girdiği Python komutunun işletilmesine sebep oldu. Bu noktada, 'Eee, ne olmuş!' demiş olabilirsiniz. Gelin bir de şuna bakalım. Şimdi programı tekrar çalıştırıp şu cevabı verin:

```
open("deneme.txt", "w")
```

Bu cevap, bilgisayarınızda *deneme.txt* adlı bir dosya oluşturulmasına sebep oldu. Belki farkındasınız, belki farkında değilsiniz, ama aslında şu anda kendi yazdığınız program sizin kontrolünüzden tamamen çıktı. Siz aslında bir hesap makinesi programı yazmıştınız. Ama eval() fonksiyonu nedeniyle kullanıcıya rastgele Python komutlarını çalıştırma imkanı verdiğiniz için programınız sadece aritmetik işlemleri hesaplamak için kullanılmayabilir. Böyle bir durumda kötü niyetli (ve bilgili) bir kullanıcı size çok büyük zarar verebilir. Mesela kullanıcının, yukarıdaki programa şöyle bir cevap verdiğini düşünün:

```
__import__("os").system("dir")
```

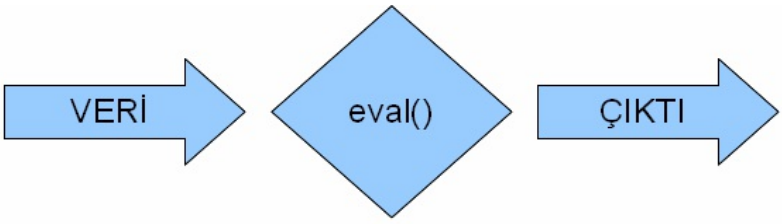
Burada anlamadığınız şeyleri şimdilik bir kenara bırakıp, bu komutun sonuçlarına odaklanın. Gördüğünüz gibi, yukarıdaki programa bu cevabı vererek mevcut dizin altındaki bütün dosyaları listeleyebildik. Yani programımız bir anda amacını aştı. Artık bu aşamadan sonra bu programı şeytani bir amaca yönelik olarak kullanmak tamamen programı kullanan kişiye kalmış... Bu programın, bir web sunucusu üzerinde çalışan bir uygulama olduğunu ve bu programı kullananların yukarıdaki gibi masumane bir şekilde dizin içindeki dosyaları listeleyen bir komut yerine, dizin içindeki dosyaları ve

hatta sabit disk üzerindeki her şeyi silen bir komut yazdığını düşünün... Yanlış yazılmış bir program yüzünden bütün verilerinizi kaybetmeniz işten bile değildir. (Bahsettiğim o, 'bütün sabit diski silen komutu' kendi sisteminizde vermemeniz gerektiğini söylememe gerek yok, değil mi?)

Eğer *SQL Injection* kavramını biliyorsanız, yukarıdaki kodların yol açtığı güvenlik açığını gayet iyi anlamış olmalısınız. Zaten internet üzerinde yaygın bir şekilde kullanılan ve web sitelerini hedef alan *SQL Injection* tarzı saldırılar da aynı mantık üzerinden gerçekleştiriliyor. *SQL Injection* metoduyla bir web sitesine saldıran *cracker*'lar, o web sitesini programlayan kişinin (çoğunlukla farkında olmadan) kullanıcıya verdiği rastgele SQL komutu işletme yetkisini kötüye kullanarak gizli ve özel bilgileri ele geçirebiliyorlar. Örneğin *SQL Injection* metodu kullanılarak, bir web sitesine ait veritabanının içeriği tamamen silinebilir. Aynı şekilde, yukarıdaki eval() fonksiyonu da kullanıcılarınıza rastgele Python komutlarını çalıştırma yetkisi verdiği için kötü niyetli bir kullanıcının programınıza sızmasına yol açabilecek potansiyele sahiptir.

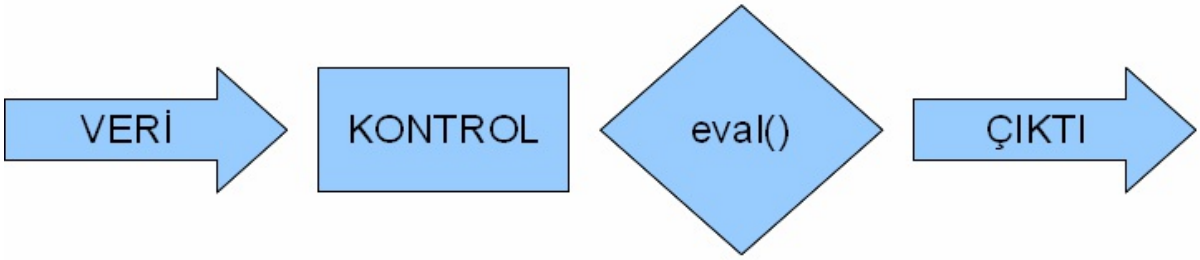
Peki eval() fonksiyonunu asla kullanmayacak mıyız? Elbette kullanacağız. Bu fonksiyonun kullanımını gerektiren durumlarla da karşılaşabilirsiniz. Ama şunu asla aklınızdan çıkarmayın: eval() fonksiyonu her ne kadar son derece yetenekli ve güçlü bir araç da olsa yanlış ellerde yıkıcı sonuçlar doğurabilir. Program yazarken, eğer eval() kullanmanızı gerektiren bir durumla karşı karşıya olduğunuzu düşünüyorsanız, bir kez daha düşünün. eval() ile elde edeceğiniz etkiyi muhtemelen başka ve çok daha iyi yöntemlerle de elde edebilirsiniz. Üstelik performans açısından eval() pek iyi bir tercih değildir, çünkü bu fonksiyon (çoğu durumda farketmeseniz de) aslında yavaş çalışır. O yüzden, eval() fonksiyonunu kullanacağınız zaman, bunun artı ve eksilerini çok iyi tartın: Bu fonksiyonu kullanmak size ne kazandırıyor, ne kaybettiriyor?

Ayrıca eval() fonksiyonu kullanılacağı zaman, kullanıcıdan gelen veri bu fonksiyona parametre olarak verilmeden önce sıkı bir kontrolden geçirilir. Yani kullanıcının girdiği veri eval() aracılığıyla doğrudan değerlendirmeye tabi tutulmaz. Araya bir kontrol mekanizması yerleştirilir. Örneğin, yukarıdaki hesap makinesi programında kullanıcının gireceği verileri sadece sayılar ve işleçlerle sınırlandırabilirsiniz. Yani kullanıcınızın, izin verilen değerler harici bir değer girmesini engelleyebilirsiniz. Bu durumu somutlaştırmak için şöyle bir diyagram çizebiliriz:



Yukarıdaki diyagram eval() fonksiyonunun yanlış uygulanış biçimini gösteriyor. Gördüğünüz gibi, veri doğrudan eval() fonksiyonuna gidiyor ve çıktı olarak veriliyor. Böyle bir durumda, eval() fonksiyonu kullanıcıdan gelen verinin ne olduğuna bakmadan, veriyi doğrudan komut olarak değerlendirip işleteceği için programınızı kullanıcının insafına terketmiş oluyorsunuz.

Aşağıdaki diyagram ise eval() fonksiyonunun doğru uygulanış biçimini gösteriyor:



Burada ise, veri eval() fonksiyonuna ulaşmadan önce kontrolden geçiriliyor. Eğer veri ancak kontrol aşamasından geçerse eval() fonksiyona ulaşabilecek ve oradan da çıktı olarak verilebilecektir. Böylece kullanıcıdan gelen komutları süzme imkanına sahip oluyoruz.

Gördüğünüz gibi, Python eval() gibi bir fonksiyon yardımıyla karakter dizileri içinde geçen Python kodlarını ayıklayıp bunları çalıştırabiliyor. Bu sayede, mesela bize input() fonksiyonu aracılığıyla gelen bir karakter dizisi içindeki Python kodlarını işletme imkanına sahip olabiliyoruz. Bu özellik, dikkatli kullanıldığında, işlerinizi epey kolaylaştırabilir.

Python'da eval() fonksiyonuna çok benzeyen exec() adlı başka bir fonksiyon daha bulunur. eval() ile yapamadığımız bazı şeyleri exec() ile yapabiliriz. Bu fonksiyon yardımıyla, karakter dizileri içindeki çok kapsamlı Python kodlarını işletebilirsiniz.

Örneğin eval() fonksiyonu bir karakter dizisi içindeki değişken tanımlama işlemini yerine getiremez. Yani eval() ile şöyle bir şey yapamazsınız:

```
>>> eval("a = 45")
```

Ama exec() ile böyle bir işlem yapabilirsiniz:

```
>>> exec("a = 45")
```

Böylece *a* adlı bir değişken tanımlamış olduk. Kontrol edelim:

```
>>> print(a)

45
```

eval() ve exec() fonksiyonları özellikle kullanıcıdan alınan verilerle doğrudan işlem yapmak gereken durumlarda işinize yarar. Örneğin bir hesap makinesi yaparken eval() fonksiyonundan yararlanabilirsiniz.

Aynı şekilde mesela insanlara Python programlama dilini öğreten bir program yazıyorsanız exec() fonksiyonunu şöyle kullanabilirsiniz:

```
d1 = """
```

```
Python'da ekrana çıktı verebilmek için print() adlı bir
fonksiyondan yararlanıyoruz. Bu fonksiyonu şöyle kullanabilirsiniz:

>>> print("Merhaba Dünya")

Şimdi de aynı kodu siz yazın!

>>> """

girdi = input(d1)

exec(girdi)

d2 = """

Gördüğünüz gibi print() fonksiyonu, kendisine
parametre olarak verilen değerleri ekrana basıyor.

Böylece ilk dersimizi tamamlamış olduk. Şimdi bir
sonraki dersimize geçebiliriz."""

print(d2)
```

Burada exec() ile yaptığımız işi eval() ile de yapabiliriz. Ama mesela eğer bir sonraki derste 'Python'da değişkenler' konusunu öğreteceksenız, eval() yerine exec() fonksiyonunu kullanmak durumunda kalabilirsiniz.

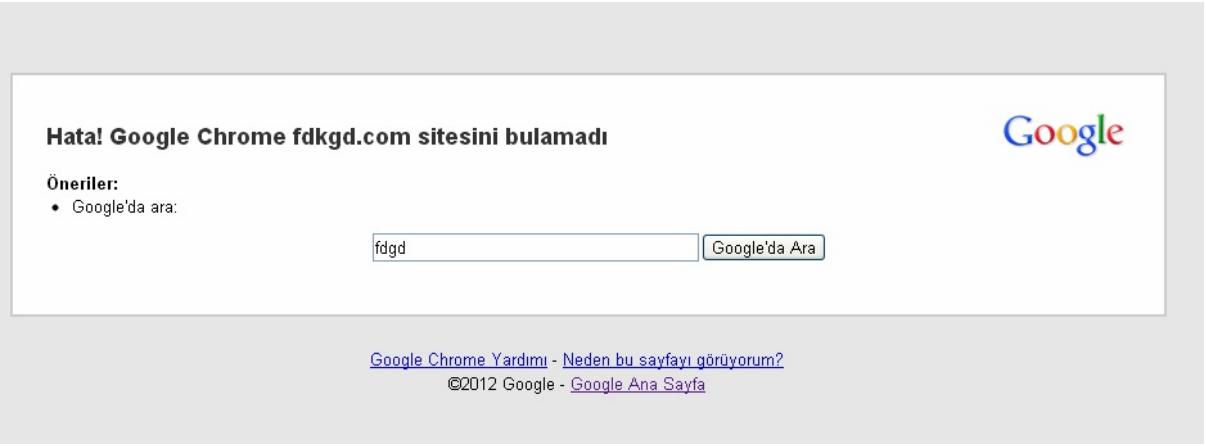
eval() fonksiyonunu anlatırken güvenlik ile ilgili olarak söylediğimiz her şey exec() fonksiyonu için de geçerlidir. Dolayısıyla bu iki fonksiyonu çok dikkatli bir şekilde kullanmanız ve bu fonksiyonların doğurduğu güvenlik açığının bilincinde olmanız gerekiyor.

Henüz Python bilgilerimiz çok kısıtlı olduğu için eval() ve exec() fonksiyonlarını bütün ayrıntılarıyla inceleyemiyoruz. Ama bilgimiz arttıkça bu fonksiyonların ne kadar güçlü (ve tehlikeli) araçlar olduğunu siz de göreceksiniz.

format() Metodu

Python programlama dili içindeki çok temel bazı araçları incelediğimize göre, bu noktada Python'daki küçük ama önemli bir konuya değinelim bu bölümü kapatmadan önce.

İnternette dolaşırken mutlaka şuna benzer bir sayfayla karşılaşmış olmalısınız:



Burada belli ki adres çubuğuna fdkgd.com diye bir URL yazmışız, ama böyle bir internet adresi olmadığı için, kullandığımız internet tarayıcısı bize şöyle bir mesaj vermiş:

Hata! Google Chrome fdkgd.com sitesini bulamadı

Şimdi de dadasdaf.com adresini arayalım...

Yine böyle bir adres olmadığı için, bu defa tarayıcımız bize şöyle bir uyarı gösterecek:

Hata! Google Chrome dadasdaf.com sitesini bulamadı

Gördüğünüz gibi, hata mesajlarında değişen tek yer, aradığımız sitenin adresi. Yani internet tarayıcımız bu hata için şöyle bir taslağa sahip:

Hata! Google Chrome ... sitesini bulamadı

Burada ... ile gösterdiğimiz yere, bulunamayan URL yerleştiriliyor. Peki böyle bir şeyi Python programlama dili ile nasıl yapabiliriz?

Çok basit:

```
#Öncelikle kullanıcıdan bir internet adresi girmesini istiyoruz
url = input("Lütfen ulaşmak istediğiniz sitenin adresini yazın: ")

#Şimdi de bu adresin bulunamadığı konusunda kullanıcıyı bilgilendiriyoruz
print("Hata! Google Chrome", url, "sitesini bulamadı")
```

Gördüğünüz gibi, şimdiye kadar öğrendiğimiz bilgileri kullanarak böyle bir programı rahatlıkla yazabiliyoruz.

Peki ya biz kullanıcının girdiği internet adresini mesela tırnak içinde göstermek istersek ne olacak? Yani örneğin şöyle bir çıktı vermek istersek:

Hata! Google Chrome 'fdsfd.com' sitesini bulamadı

Bunun için yine karakter dizisi birleştirme yönteminden yararlanabilirsiniz:

```
#Öncelikle kullanıcıdan bir internet adresi girmesini istiyoruz
url = input("Lütfen ulaşmak istediğiniz sitenin adresini yazın: ")

#Şimdi de bu adresin bulunamadığı konusunda kullanıcıyı bilgilendiriyoruz
print("Hata! Google Chrome", """ + url + """, "sitesini bulamadı")
```

Burada, + işaretlerini kullanarak, kullanıcının girdiği adresin sağına ve soluna birer tırnak işaretini nasıl yerleştirdiğimize dikkat edin.

Gördüğünüz gibi bu yöntem işe yarıyor, ama ortaya çıkan karakter dizisi de oldukça karmaşık görünüyor. İşte bu tür 'karakter dizisi biçimlendirme' işlemleri için Python bize çok faydalı bir araç sunuyor. Bu aracın adı format().

Bu aracı şöyle kullanıyoruz:

```
#Öncelikle kullanıcıdan bir internet adresi girmesini istiyoruz
url = input("Lütfen ulaşmak istediğiniz sitenin adresini yazın: ")

#Şimdi de bu adresin bulunamadığı konusunda kullanıcıyı bilgilendiriyoruz
print("Hata! Google Chrome {} sitesini bulamadı".format(url))
```

Bir de bulunamayan internet adresini tırnak içine alalım:

```
print("Hata! Google Chrome '{}' sitesini bulamadı".format(url))
```

Görüyorsunuz ya, biraz önce karakter dizisi birleştirme yöntemini kullanarak gerçekleştirdiğimiz işlemi, çok daha basit bir yolla gerçekleştirme imkanı sunuyor bize bu format() denen araç...

Peki format() nasıl çalışıyor?

Bunu anlamak için şu basit örneklere bir bakalım:

```
>>> print("{} ve {} iyi bir ikilidir".format("Python", "Django"))

'Python ve Django iyi bir ikilidir

>>> print("{} {}'yi seviyor!".format("Ali", "Ayşe"))

'Ali Ayşe'yi seviyor!'

>>> print("{} {} yaşında bir {}dur".format("Ahmet", "18", "futbolcu"))

'Ahmet 18 yaşında bir futbolcudur'
```

Elbette bu örnekleri şöyle de yazabilirdik:

```
>>> metin = "{} ve {} iyi bir ikilidir"
>>> metin.format("Python", "Django")

'Python ve Django iyi bir ikilidir

>>> metin = "{} {}'yi seviyor!"
>>> metin.format("Ali", "Ayşe")

'Ali Ayşe'yi seviyor!'

>>> metin = "{} {} yaşında bir {}dur"
>>> metin.format("Ahmet", "18", "futbolcu")

'Ahmet 18 yaşında bir futbolcudur'
```

Burada taslak metni doğrudan format() metoduna parametre olarak vermeden önce bir değişkene atadık. Böylece bu metni daha kolay bir şekilde kullanabildik.

Bu örneklerin, format() denen aracı anlamak konusunda size epey fikir verdiğini zannediyorum. Ama isterseniz bu aracın ne olduğunu ve nasıl çalıştığını daha ayrıntılı olarak incelemeye geçmeden önce başka bir örnek daha verelim.

Varsayalım ki kullanıcıdan aldığı bilgiler doğrultusunda, özel bir konu üzerine dilekçe oluşturan bir program yazmak istiyorsunuz.

Dilekçe taslağımız şu şekilde olsun:

```
                                                tarih:

T.C.
... ÜNİVERSİTESİ
... Fakültesi Dekanlığına



Fakülteniz .....Bölümü ..... numaralı öğrencisiyim. Ekte sunduğum
belgede belirtilen mazeretim gereğince ..... Eğitim-Öğretim Yılı  .....
yarıyılında öğrenime ara izni (kayıt dondurma) istiyorum.

   Bilgilerinizi ve gereğini arz ederim.

   İmza

Ad-Soyadı      :
T.C. Kimlik No. :
```

```
Adres          :
Tel.           :
Ekler          :
```

Amacınız bu dilekçedeki boşluklara gelmesi gereken bilgileri kullanıcıdan alıp, eksiksiz bir dilekçe ortaya çıkarmak.

Kullanıcıdan bilgi alma kısmı kolay. input() fonksiyonunu kullanarak gerekli bilgileri kullanıcıdan alabileceğimizi biliyorsunuz:

```
tarih          = input("tarih: ")
üniversite     = input("üniversite adı: ")
fakülte        = input("fakülte adı: ")
bölüm          = input("bölüm adı: ")
öğrenci_no     = input("öğrenci no. :")
öğretim_yılı   = input("öğretim yılı: ")
yarıyıl        = input("yarıyıl: ")
ad             = input("öğrencinin adı: ")
soyad          = input("öğrencinin soyadı: ")
tc_kimlik_no   = input("TC Kimlik no. :")
adres          = input("adres: ")
tel            = input("telefon: ")
ekler          = input("ekler: ")
```

Bilgileri kullanıcıdan aldık. Peki ama bu bilgileri dilekçe taslağı içindeki boşluklara nasıl yerleştireceğiz?

Şu ana kadar öğrendiğimiz print() fonksiyonunu ve *\t* ve *\n* gibi kaçış dizilerini kullanarak istediğiniz çıktıyı elde etmeyi deneyebilirsiniz. Ama denediğinizde siz de göreceksiniz ki, bu tür yöntemleri kullanarak yukarıdaki dilekçe taslağını doldurmak inanılmaz zor ve vakit alıcı olacaktır. Halbuki bunların hiçbirine gerek yok. Çünkü Python bize bu tür durumlarda kullanılmak üzere çok pratik bir araç sunuyor. Şimdi çok dikkatlice inceleyin şu kodları:

```
dilekçe = """
                                        tarih: {}


T.C.
{} ÜNİVERSİTESİ
{} Fakültesi Dekanlığına


Fakülteniz {} Bölümü {} numaralı öğrencisiyim. Ekte sunduğum belgede
belirtilen mazeretim gereğince {} Eğitim-Öğretim Yılı  {}.
```

```
yarıyılında öğrenime ara izni (kayıt dondurma) istiyorum.

   Bilgilerinizi ve gereğini arz ederim.

       İmza

Ad             : {}
Soyad          : {}
T.C. Kimlik No. : {}
Adres          : {}
Tel.           : {}
Ekler          : {}
"""


tarih          = input("tarih: ")
üniversite     = input("üniversite adı: ")
fakülte        = input("fakülte adı: ")
bölüm          = input("bölüm adı: ")
öğrenci_no     = input("öğrenci no. :")
öğretim_yılı   = input("öğretim yılı: ")
yarıyıl        = input("yarıyıl: ")
ad             = input("öğrencinin adı: ")
soyad          = input("öğrencinin soyadı: ")
tc_kimlik_no   = input("TC Kimlik no. :")
adres          = input("adres: ")
tel            = input("telefon: ")
ekler          = input("ekler: ")

print(dilekçe.format(tarih, üniversite, fakülte, bölüm,
                     öğrenci_no, öğretim_yılı, yarıyıl,
                     ad, soyad, tc_kimlik_no,
                     adres, tel, ekler))
```

Bu kodlara (ve bundan önceki örneklere) bakarak birkaç tespitte bulunalım:

1. Taslak metinde kullanıcıdan alınacak bilgilerin olduğu yerlere birer *{}* işareti yerleştirdik.
2. Taslaktaki eksiklikleri tamamlayacak verileri input() fonksiyonu yardımıyla kullanıcıdan tek tek aldık.
3. Son olarak, print() fonksiyonu yardımıyla metni tam bir şekilde ekrana çıktı olarak verdik.

Şimdi son tespitimizi biraz açıklayalım. Gördüğünüz gibi, print() fonksiyonu içinde dilekçe.format() gibi bir yapı var. Burada *dilekçe* değişkenine nokta işareti ile bağlanmış format()

adlı, fonksiyon benzeri bir araç görüyoruz. Bu araca teknik dilde 'metot' adı verilir. format() metodunun parantezleri içinde ise, kullanıcıdan alıp birer değişkene atadığımız veriler yer alıyor.

Dilerseniz yukarıda olan biteni daha net anlayabilmek için bu konunun başına verdiğimiz örneklere geri dönelim.

İlk olarak şöyle bir örnek vermiştik:

```
#Öncelikle kullanıcıdan bir internet adresi girmesini istiyoruz
url = input("Lütfen ulaşmak istediğiniz sitenin adresini yazın: ")

#Şimdi de bu adresin bulunamadığı konusunda kullanıcıyı bilgilendiriyoruz
print("Hata! Google Chrome {} sitesini bulamadı".format(url))
```

Burada kullanıcının gireceği internet adresinin yerini tutması için *{}* işaretlerinden yararlanarak şöyle bir karakter dizisi oluşturduk:

```
"Hata! Google Chrome {} sitesini bulamadı"
```

Gördüğünüz gibi, *{}* işareti karakter dizisi içinde URL'nin geleceği yeri tutuyor. Bu *{}* işaretinin yerine neyin geleceğini format() metodunun parantezleri içinde belirtiyoruz. Dikkatlice bakın:

```
print("Hata! Google Chrome {} sitesini bulamadı".format(url))
```

Elbette eğer istersek yukarıdaki örneği şöyle de yazabilirdik:

```
url = input("Lütfen ulaşmak istediğiniz sitenin adresini yazın: ")

#Kullanıcıya gösterilecek hata için bir taslak metin oluşturuyoruz
hata_taslağı = "Hata! Google Chrome {} sitesini bulamadı"

print(hata_taslağı.format(url))
```

Burada hata metnini içeren karakter dizisini doğrudan format() metoduna bağlamak yerine, bunu bir değişkene atayıp, format() metodunu bu değişkene bağladık.

Bunun dışında şu örnekleri de vermiştik:

```
>>> metin = "{} ve {} iyi bir ikilidir"
>>> metin.format("Python", "Django")

'Python ve Django iyi bir ikilidir

>>> metin = "{} {}'yi seviyor!"
>>> metin.format("Ali", "Ayşe")
```

```
'Ali Ayşe'yi seviyor!'

>>> metin = "{} {} yaşında bir {}dur"
>>> metin.format("Ahmet", "18", "futbolcu")

'Ahmet 18 yaşında bir futbolcudur'
```

Burada da, gördüğüz gibi, öncelikle bir karakter dizisi tanımlıyoruz. Bu karakter dizisi içindeki değişken değerleri ise *{}* işaretleri ile gösteriyoruz. Daha sonra format() metodunu alıp bu karakter dizisine bağlıyoruz. Karakter dizisi içindeki *{}* işaretleri ile gösterdiğimiz yerlere gelecek değerleri de format() metodunun parantezleri içinde gösteriyoruz. Yalnız burada şuna dikkat etmemiz lazım: Karakter dizisi içinde kaç tane *{}* işareti varsa, format() metodunun parantezleri içinde de o sayıda değer olması gerekiyor.

Bu yapının, yazdığımız programlarda işimizi ne kadar kolaylaştıracağını tahmin edebilirsiniz. Kısa karakter dizilerinde pek belli olmayabilir, ama özellikle çok uzun ve boşluklu karakter dizilerini biçimlendirirken format() metodunun hayat kurtardığına kendiniz de şahit olacaksınız.

İlerleyen derslerimizde format() metodunu ve karakter dizisi biçimlendirme konusunu çok daha ayrıntılı bir şekilde inceleyeceğiz. Ancak yukarıda verdiğimiz bilgiler format() metodunu verimli bir şekilde kullanabilmenizi sağlamaya yetecek düzeydedir.

başa dön

Koşullu Durumlar

Artık Python programlama dilinde belli bir noktaya geldik sayılır. Ama eğer farkettiyseniz, yine de elimizi kolumuzu bağlayan, istediğimiz şeyleri yapmamıza engel olan bir şeyler var. İşte bu bölümde, Python programlama dilinde hareket alanımızı bir hayli genişletecek araçları tanıyacağız.

Aslında sadece bu bölümde değil, bu bölümü takip eden her bölümde, hareket alanımızı kısıtlayan duvarları tek tek yıktığımıza şahit olacaksınız. Özellikle bu bölümde inceleyeceğimiz 'koşullu durumlar' konusu, tabir yerindeyse, Python'da boyut atlamamızı sağlayacak.

O halde hiç vakit kaybetmeden yola koyulalım...

Şimdiye kadar öğrendiğimiz Python bilgilerini kullanarak şöyle bir program yazabileceğimizi biliyorsunuz:

```
yaş = 15

print("""Programa hoşgeldiniz!

Programımızı kullanabilmek için en az
13 yaşında olmalısınız.""")

print("Yaşınız: ", yaş)
```

Burada yaptığımız şey çok basit. Öncelikle, değeri *15* olan, *yaş* adlı bir değişken tanımladık. Daha sonra, programımızı çalıştıran kullanıcılar için bir hoşgeldin mesajı hazırladık. Son olarak da *yaş* değişkeninin değerini ekrana yazdırdık.

Bu programın özelliği tek sesli bir uygulama olmasıdır. Yani bu programda kullanıcıyla herhangi bir etkileşim yok. Burada bütün değerleri/değişkenleri programcı olarak kendimiz belirliyoruz. Bu programın ne kadar yavan olduğunu herhalde söylemeye gerek yok.

Ancak yine önceki derslerde öğrendiğimiz input() fonksiyonu yardımıyla yukarıdaki programın üzerindeki yavanlığı bir nebze de olsa atabilir, bu programı rahatlıkla çok sesli bir hale getirebilir, yani kullanıcıyla etkileşim içine girebiliriz.

Yukarıdaki tek sesli uygulamayı, input() fonksiyonunu kullanarak çok sesli bir hale nasıl getireceğimizi gayet iyi bildiğinize eminim:

```
print("""Programa hoşgeldiniz!

Programımızı kullanabilmek için en az
13 yaşında olmalısınız.""")
```

```
print("Lütfen yaşınızı girin.\n")

yaş = input("Yaşınız: \t")

print("Yaşınız: ", yaş)
```

Tıpkı bir önceki uygulamada olduğu gibi, burada da yaptığımız şey çok basit. İlk örnekte *yaş* değişkeninin değerini kendimiz elle yazmıştık. İkinci örnekte ise bu *yaş* değişkenini kullanıcıdan alıyoruz ve tıpkı ilk örnekte olduğu gibi, bu değişkenin değerini ekrana yazdırıyoruz.

Bu arada, yukarıdaki uygulamada yer verdiğimiz *\n* ve *\t* adlı kaçış dizileri de artık sizin için oldukça tanıdık. *\n* kaçış dizisi yardımıyla bir alt satıra geçtiğimizi, *\t* adlı kaçış dizisi yardımıyla da bir sekmelik boşluk bıraktığımızı biliyorsunuz.

Gördüğünüz gibi, şu ana kadar öğrendiklerimizle ancak kullanıcıdan gelen yaş bilgisini ekrana yazdırabiliyoruz. Öğrendiğimiz input() fonksiyonu bize kullanıcıdan bilgi alma imkanı sağlıyor. Ama kullanıcıdan gelen bu bilgiyi şimdilik ancak olduğu gibi kullanabiliyoruz. Yani mesela yukarıdaki örneği dikkate alarak konuşacak olursak, kullanıcının yaşı eğer 13'ün üzerindeyse onu programa kabul edecek, yok eğer 13 yaşın altındaysa da programdan atacak bir mekanizma üretemiyoruz. Yapabildiğimiz tek şey, kullanıcının girdiği veriyi ekrana yazdırmak.

Yukarıda verdiğimiz örneklerle nereye varmaya çalıştığımızı az çok tahmin etmişsinizdir. Dikkat ederseniz yukarıda sözünü ettiğimiz şey koşullu bir durum. Yani aslında yapmak istediğimiz şey, kullanıcının yaşını denetleyip, onun programa kabul edilmesini 13 yaşından büyük olma koşuluna bağlamak.

İsterseniz tam olarak neden bahsettiğimizi anlayabilmek için, birkaç vaka örneği verelim.

Diyelim ki Google'ın Gmail hizmeti aracılığıyla bir e.posta hesabı aldınız. Bu hesaba gireceğiniz zaman Gmail size bir kullanıcı adı ve parola sorar. Siz de kendinize ait kullanıcı adını ve parolayı sayfadaki kutucuklara yazarsınız. Eğer yazdığınız kullanıcı adı ve parola doğruysa hesabınıza erişebilirsiniz. Ama eğer kullanıcı adınız ve parolanız doğru değilse hesabınıza erişemezsiniz. Yani e.posta hesabınıza erişmeniz, kullanıcı adı ve parolayı doğru girme koşuluna bağlıdır.

Ya da şu vaka örneğini düşünelim: Diyelim ki Pardus'ta komut satırı aracılığıyla güncelleme işlemi yapacaksınız. sudo pisi up komutunu verdiğiniz zaman güncellemelerin listesi size bildirilecek, bu güncellemeleri yapmak isteyip istemediğiniz sorulacaktır. Eğer evet cevabı verirseniz güncelleme işlemi başlar. Ama eğer hayır cevabı verirseniz güncelleme işlemi başlamaz. Yani güncelleme işleminin başlaması kullanıcının evet cevabı vermesi koşuluna bağlıdır.

İşte bu bölümde biz bu tür koşullu durumlardan söz edeceğiz.

Koşul Deyimleri

Hiç kuşkusuz, koşula bağlı durumlar Python'daki en önemli konulardan biridir. Giriş bölümünde

bahsettiğimiz koşullu işlemleri yapabilmek için ‘koşul deyimleri’ adı verilen birtakım araçlardan yararlanacağız. Gelin şimdi bu araçların neler olduğunu görelim.

if

Python programlama dilinde koşullu durumları belirtmek için üç adet deyimden yararlanıyoruz:

- if
- elif
- else

İsterseniz önce if deyimi ile başlayalım...

Eğer daha önceden herhangi bir programlama dilini az da olsa kurcalama fırsatınız olduysa, bir programlama dilinde if deyimlerinin ne işe yaradığını az çok biliyorsunuzdur. Daha önceden hiç programcılık deneyiminiz olmamışsa da ziyanı yok. Zira bu bölümde if deyimlerinin ne işe yaradığını ve nerelerde kullanıldığını enine boyuna tartışacağız.

İngilizce bir kelime olan ‘*if*‘, Türkçede ‘eğer’ anlamına gelir. Anlamından da çıkarabileceğimiz gibi, bu kelime bir koşul bildiriyor. Yani ‘*eğer bir şey falanca ise...*‘ ya da ‘*eğer bir şey filanca ise...*‘ gibi... İşte biz Python’da bir koşula bağlamak istediğimiz durumları if deyimi aracılığıyla göstereceğiz.

Gelin isterseniz bu deyimi nasıl kullanacağımıza dair ufacık bir örnek vererek işe başlayalım:

Öncelikle elimizde şöyle bir değişken olsun:

```
n = 255
```

Yukarıda verdiğimiz değişkenin değerinin bir karakter dizisi değil, aksine bir sayı olduğunu görüyoruz. Şimdi bu değişkenin değerini sorgulayalım:

```
if n > 10:
```

Burada sayının 10’dan büyük olup olmadığına bakıyoruz.

Burada gördüğümüz > işaretinin ne demek olduğunu açıklamaya gerek yok sanırım. Hepimizin bildiği ‘büyüktür’ işareti Python’da da aynen bildiğimiz şekilde kullanılıyor. Mesela ‘küçüktür’ demek isteseydik, < işaretini kullanacaktık. İsterseniz hemen şurada araya girip bu işaretleri yeniden hatırlayalım:

İşleç	Anlamı
>	büyüktür
<	küçüktür
>=	büyük eşittir

<=	küçük eşittir
==	eşittir
!=	eşit değildir

Gördüğünüz gibi hiçbiri bize yabancı gelecek gibi değil. Yalnızca en sondaki ‘eşittir’ (==) ve ‘eşit değildir’ (*!*=) işaretleri biraz değişik gelmiş olabilir. Burada ‘eşittir’ işaretinin = olmadığına dikkat edin. Python’da = işaretini değer atama işlemleri için kullanıyoruz. == işaretini ise iki adet değerin birbirine eşit olup olmadığını denetlemek için... Mesela:

```
>>> a = 26
```

Burada değeri *26* olan *a* adlı bir değişken belirledik. Yani *a* değişkenine değer olarak *26* sayısını atadık. Ayrıca burada, değer atama işleminin ardından *Enter* tuşuna bastıktan sonra Python hiçbir şey yapmadan bir alt satıra geçti. Bir de şuna bakalım:

```
>>> a == 26

True
```

Burada ise yaptığımız şey *a* değişkeninin değerinin *26* olup olmadığını sorgulamak `a == 26` komutunu verdikten sonra Python bize *True* diye bir çıktı verdi. Bu çıktının anlamını biraz sonra öğreneceğiz. Ama şimdi isterseniz konuyu daha fazla dağıtmayalım. Biz şimdilik sadece = ve == işaretlerinin birbirinden tamamen farklı anlamlara geldiğini bilelim yeter.

Ne diyorduk?

```
if n > 10:
```

Bu ifadeyle Python’a şöyle bir şey demiş oluyoruz:

> Eğer *n* sayısının değeri 10’dan büyükse...

Burada kullandığımız işaretlere dikkat edin. En sonda bir adet *:* işaretinin olduğunu gözden kaçırmıyoruz. Bu tür işaretler Python için çok önemlidir. Bunları yazmayı unutursak Python gözümüzün yaşına bakmayacaktır.

Dedik ki, `if n > 10:` ifadesi, ‘eğer n değişkeninin değeri 10’dan büyükse...’ anlamına gelir. Bu ifadenin eksik olduğu apaçık ortada. Yani belli ki bu cümlenin bir de devamı olması gerekiyor. O halde biz de devamını getirelim:

```
if n > 10:
    print("sayı 10'dan büyüktür!")
```

Burada çok önemli bir durumla karşı karşıyayız. Dikkat ederseniz, ikinci satırı ilk satıra göre girintili yazdık. Elbette bunu şirinlik olsun diye yapmadık. Python programlama dilinde girintiler çok büyük önem taşır. Hatta ne kadarlık bir girinti verdiğiniz bile önemlidir. Eğer Python kodlarına duyarlı bir

metin düzenleyici kullanıyorsanız, kullandığınız metin düzenleyici çoğu durumda sizin yerinize uygun bir şekilde girintilemeyi yapacaktır. Mesela IDLE adlı geliştirme ortamını kullananlar, ilk satırdaki *:* işaretini koyup *Enter* tuşuna bastıklarında otomatik olarak girinti verildiğini farkedeceklerdir. Eğer kullandığınız metin düzenleyici, satırları otomatik olarak girintilemiyorsa sizin bu girintileme işlemini elle yapmanız gerekecektir. Yalnız elle girintilerken, ne kadar girinti vereceğimize dikkat etmeliyiz. Genel kural olarak *4* boşlukluk bir girintileme uygun olacaktır. Girintileme işlemini klavyedeki sekme (*Tab*) tuşuna basarak da yapabilirsiniz. Ama aynı program içinde sekmelerle boşlukları karıştırmayın. Yani eğer girintileme işlemini klavyedeki boşluk (*Space*) tuşuna basarak yapıyorsanız, program boyunca aynı şekilde yapın. (Ben size girinti verirken *Tab* tuşu yerine *Space* tuşunu kullanmanızı tavsiye ederim). Kısaca söylemek gerekirse; Python'da girintileme ve girintilemede tutarlılık çok önemlidir. Özellikle büyük programlarda, girintilemeler açısından tutarsızlık gösterilmesi programın çalışmamasına sebep olabilir.

Not

Python'da girintileme konusuyla ilgili daha ayrıntılı bilgi için: http://www.istihza.com/blog/python-ve-metin-duzenleyiciler.html/

Eğer yukarıdaki if bloğunu bir metin düzenleyici içine değil de doğrudan etkileşimli kabuğa yazmışsanız bazı şeyler dikkatinizi çekmiş olmalı. Etkileşimli kabukta `if sayı > 10:` satırını yazıp *Enter* tuşuna bastığınızda şöyle bir görüntüyle karşılaşmış olmalısınız:

```
>>> if n > 10:
...
```

Dikkat ederseniz, >>> işareti, ... işaretine dönüştü. Eğer bu noktada herhangi bir şey yazmadan *Enter* tuşuna basacak olursanız Python size şöyle bir hata mesajı verecektir:

```
File "<stdin>", line 2
    ^
IndentationError: expected an indented block
```

Hata mesajında da söylendiği gibi, Python bizden girintilenmiş bir blok beklerken, biz onun bu beklentisini karşılamamışız. Dolayısıyla bize yukarıdaki hata mesajını göstermiş. ... işaretini gördükten sonra yapmamız gereken şey, dört kez boşluk (*Space*) tuşuna basarak girinti oluşturmak ve if bloğunun devamını yazmak olmalıydı. Yani şöyle:

```
>>> if n > 10:
...     print("sayı 10'dan büyüktür!")
...
```

Gördüğünüz gibi, print() fonksiyonunu yazıp *Enter* tuşuna bastıktan sonra yine ... işaretini gördük. Python burada bizden yeni bir satır daha bekliyor. Ama bizim yazacak başka bir kodumuz olmadığı için tekrar *Enter* tuşuna basıyoruz ve nihai olarak şöyle bir görüntü elde ediyoruz:

```
>>> if n > 10:
...     print("sayı 10'dan büyüktür!")
...
sayı 10'dan büyüktür!
>>>
```

Demek ki *250* sayısı *10*'dan büyükmüş! Ne büyük bir buluş! Merak etmeyin, daha çok şey öğrendikçe daha mantıklı programlar yazacağız. Burada amacımız işin temelini kavramak. Bunu da en iyi, (çok mantıklı olmasa bile) basit programlar yazarak yapabiliriz.

Şimdi metin düzenleyicimizi açarak daha mantıklı şeyler yazmaya çalışalım. Zira yukarıdaki örnekte değişkeni kendimiz belirlediğimiz için, bu değişkenin değerini if deyimleri yardımıyla denetlemek pek akla yatkın görünmüyor. Ne de olsa değişkenin değerinin ne olduğunu biliyoruz. Dolayısıyla bu değişkenin *10* sayısından büyük olduğunu da biliyoruz! Bunu if deyimiyle kontrol etmek çok gerekli değil. Ama şimdi daha makul bir iş yapacağız. Değişkeni biz belirlemek yerine kullanıcıya belirleteceğiz:

```
sayı = int(input("Bir sayı giriniz: "))

if sayı > 10:
    print("Girdiğiniz sayı 10'dan büyüktür!")

if sayı < 10:
    print("Girdiğiniz sayı 10'dan küçüktür!")

if sayı == 10:
    print("Girdiğiniz sayı 10'dur!")
```

Gördüğünüz gibi, art arda üç adet if bloğu kullandık. Bu kodlara göre, eğer kullanıcının girdiği sayı *10*'dan büyükse, ilk if bloğu işletilecek; eğer sayı *10*'dan küçükse ikinci if bloğu işletilecek; eğer sayı 10'a eşit ise üçüncü if bloğu işletilecektir. Peki ya kullanıcı muziplik yapıp sayı yerine harf yazarsa ne olacak? Böyle bir ihtimal için programımıza herhangi bir denetleyici yerleştirmedik. Dolayısıyla eğer kullanıcı sayı yerine harf girerse programımız hata verecek, yani çökecektir. Bu tür durumlara karşı nasıl önlem alacağımızı ilerleyen derslerimizde göreceğiz. Biz şimdilik bildiğimiz yolda yürüyelim.

Yukarıdaki örnekte input() ile gelen karakter dizisini, int() fonksiyonu yardımıyla bir sayıya dönüştürdüğümüze dikkat edin. Kullanıcıdan gelen veriyi büyüklük-küçüklük ölçütüne göre inceleyeceğimiz için, gelen veriyi bir sayıya dönüştürmemiz gerekiyor. Bunu da int() fonksiyonu ile yapabileceğimizi biliyorsunuz.

Elbette yukarıdaki dönüştürme işlemini şöyle de yapabilirdik:

```
sayı = input("Bir sayı giriniz: ")
```

```
sayı = int(sayı)
```

Burada önce input() fonksiyonuyla veriyi aldık, daha sonra bu veriyi ayrı bir yerde sayıya dönüştürüp tekrar *sayı* adlı değişkene atadık.

if deyimlerini kullanıcı adı veya parola denetlerken de kullanabiliriz. Mesela şöyle bir program taslağı yazabiliriz:

```
print("""
Dünyanın en gelişmiş e.posta hizmetine
hoşgeldiniz. Yalnız hizmetimizden
yararlanmak için önce sisteme giriş
yapmalısınız.
""")

parola = input("Parola: ")

if parola == "12345678":
    print("Sisteme Hoşgeldiniz!")
```

Gördüğünüz gibi, programın başında üç tırnak işaretlerinden yararlanarak uzun bir metni kullanıcıya gösterdik. Bu bölümü, kendiniz göze hoş gelecek bir şekilde süsleyebilirsiniz de. Eğer kullanıcı, kendisine parola sorulduğunda cevap olarak *"12345678"* yazarsa kullanıcıyı sisteme alıyoruz.

Yukarıdaki örnekte, kullanıcının girdiği parola *"12345678"* ise kendisine *"Sisteme Hoşgeldiniz!"* mesajını gösteriyoruz. Mantık olarak bunun tersini yapmak da mümkündür. Yani:

```
if parola != "12345678":
    print("Ne yazık ki yanlış parola girdiniz!")
```

Burada ise bir önceki örneğin mantığını ters çevirdik. Önceki örnekte *parola* değişkeni *"12345678"* adlı karakter dizisine eşitse (if parola == "12345678") bir işlem yapıyorduk. Yukarıdaki örnekte ise *parola* değişkeni *"12345678"* adlı karakter dizisine eşit değilse (if parola != "12345678") bir işlem yapıyoruz.

Bu iki örneğin de aslında aynı kapıya çıktığını görüyorsunuz. Tek değişiklik, kullanıcıya gösterilen mesajlardadır.

Böylece Python'daki koşullu durumlar üzerindeki incelememizin ilk ve en önemli aşamasını geride bırakmış olduk. Dikkat ettiyseniz if deyimi sayesinde programlarımıza karar vermeyi öğrettik. Bu deyim yardımıyla, kullanıcıdan aldığımız herhangi bir verinin niteliği üzerinde kapsamlı bir karar verme işlemi yürütebiliyoruz. Yani artık programlarımız kullanıcıdan alınan veriyi olduğu gibi kabul etmekle yetinmiyor. Kullanıcının girdiği verinin ne olduğuna bağlı olarak programlarımızın farklı işlemler yapmasını da sağlayabiliyoruz.

Daha önce de söylediğimiz gibi, if deyimi dışında Python'da koşullu durumları ifade etmek için kullandığımız, elif ve else adlı iki deyim daha vardır. Bunlar if ile birlikte kullanılırlar. Gelin isterseniz bu iki deyimden, adı elif olana bakalım.

elif

Python'da, if deyimleriyle birlikte kullanılan ve yine koşul belirten bir başka deyim de elif deyimidir. Buna şöyle bir örnek verebiliriz:

```
yaş = int(input("Yaşınız: "))

if yaş == 18:
    print("18 iyidir!")

elif yaş < 0:
    print("Yok canım, daha neler!...")

elif yaş < 18:
    print("Genç bir kardeşimizsin!")

elif yaş > 18:
    print("Eh, artık yaş yavaş yavaş kemale eriyor!")
```

Yukarıdaki örneği şöyle yazmayı da deneyebilirsiniz:

```
yaş = int(input("Yaşınız: "))

if yaş == 18:
    print("18 iyidir!")

if yaş < 0:
    print("Yok canım, daha neler!...")

if yaş < 18:
    print("Genç bir kardeşimizsin!")

if yaş > 18:
    print("Eh, artık yaş yavaş yavaş kemale eriyor!")
```

Bu iki programın da aynı işlevi gördüğünü düşünebilirsiniz. Ancak ilk bakışta pek belli olmasa da, aslında yukarıdaki iki program birbirinden farklı davranacaktır. Örneğin ikinci programda eğer kullanıcı eksi değerli bir sayı girerse hem if yaş < 0 bloğu, hem de if yaş < 18 bloğu çalışacaktır. İsterseniz yukarıdaki programı çalıştırıp, cevap olarak eksi değerli bir sayı verin. Ne demek istediğimiz gayet net anlaşılacaktır.

Bu durum if ile elif arasındaki çok önemli bir farktan kaynaklanır. Buna göre if bize olası bütün sonuçları listeler, elif ise sadece doğru olan ilk sonucu verir. Bu soyut tanımlamayı biraz daha somutlaştıralım:

```
a = int(input("Bir sayı giriniz: "))

if a < 100:
    print("verdiğiniz sayı 100'den küçüktür.")

if a < 50:
    print("verdiğiniz sayı 50'den küçüktür.")

if a == 100:
    print("verdiğiniz sayı 100'dür.")

if a > 100:
    print("verdiğiniz sayı 100'den büyüktür.")

if a > 150:
    print("verdiğiniz sayı 150'den büyüktür.")
```

Yukarıdaki kodları çalıştırdığımızda, doğru olan bütün sonuçlar listelenecektir. Yani mesela kullanıcı *40* sayısını girmişse, ekrana verilecek çıktı şöyle olacaktır:

```
verdiğiniz sayı 100'den küçüktür.
verdiğiniz sayı 50'den küçüktür.
```

Burada *40* sayısı hem *100*'den, hem de *50*'den küçük olduğu için iki sonuç da çıktı olarak verilecektir. Ama eğer yukarıdaki kodları şöyle yazarsak:

```
a = int(input("Bir sayı giriniz: "))

if a < 100:
    print("verdiğiniz sayı 100'den küçüktür.")

elif a < 50:
    print("verdiğiniz sayı 50'den küçüktür.")

elif a == 100:
    print("verdiğiniz sayı 100'dür.")

elif a > 150:
    print("verdiğiniz sayı 150'den büyüktür.")
```

```
elif a > 100:
    print("verdiğiniz sayı 100'den büyüktür.")
```

Kullanıcının *40* sayısını girdiğini varsaydığımızda, bu defa programımımız yalnızca şu çıktıyı verecektir:

```
verdiğiniz sayı 100'den küçüktür.
```

Gördüğünüz gibi, elif deyimlerini kullandığımız zaman, ekrana yalnızca doğru olan ilk sonuç veriliyor. Yukarıda *40* sayısı hem *100*'den hem de *50*'den küçük olduğu halde, Python bu sayının *100*'den küçük olduğunu görür görmez sonucu ekrana basıp, öteki koşul bloklarını incelemeyi bırakıyor. if deyimlerini arka arkaya sıraladığımızda ise, Python bütün olasılıkları tek tek değerlendirip, geçerli olan bütün sonuçları ekrana döküyor.

Bir sonraki bölümde else deyimini öğrendiğimiz zaman, elif'in tam olarak ne işe yaradığını çok daha iyi anlamanızı sağlayacak bir örnek vereceğiz.

Not

Şimdiye kadar verdiğimiz örneklerden de rahatlıkla anlayabileceğiniz gibi, ilk koşul bloğunda asla elif deyimi kullanılamaz. Bu deyimin kullanılabilmesi için kendisinden önce en az bir adet if bloğu olmalıdır. Yani Python'da koşullu durumları ifade ederken ilk koşul bloğumuz her zaman if deyimi ile başlamalıdır.

elif'i de incelediğimize göre, koşul bildiren deyimlerin sonuncusuna göz atabiliriz: else

else

Şimdiye kadar Python'da koşul bildiren iki deyimi öğrendik. Bunlar if ve elif idi. Bu bölümde ise koşul deyimlerinin sonuncusu olan else deyimini göreceğiz. Öğrendiğimiz şeyleri şöyle bir gözden geçirecek olursak, temel olarak şöyle bir durumla karşı karşıya olduğumuzu görürüz:

```
if falanca:
    bu işlemi yap

if filanca:
    şu işlemi yap
```

Veya şöyle bir durum:

```
if falanca:
    bu işlemi yap
```

```
elif filanca:
    şu işlemi yap
```

if ile elif arasındaki farkı biliyoruz. Eğer if deyimlerini art arda sıralayacak olursak, Python doğru olan bütün sonuçları listeleyecektir. Ama eğer if deyiminden sonra elif deyimini kullanırsak, Python doğru olan ilk sonucu listelemekle yetinecektir.

Bu bölümde göreceğimiz else deyimi, yukarıdaki tabloya bambaşka bir boyut kazandırıyor. Dikkat ederseniz şimdiye kadar öğrendiğimiz deyimleri kullanabilmek için ilgili bütün durumları tanımlamamız gerekiyordu. Yani:

```
eğer böyle bir durum varsa:
    bunu yap

eğer şöyle bir durum varsa:
    şunu yap

eğer filancaysa:
    şöyle git

eğer falancaysa:
    böyle gel
```

gibi...

Ancak her durum için bir if bloğu yazmak bir süre sonra yorucu ve sıkıcı olacaktır. İşte bu noktada devreye else deyimi girecek. else'in anlamı kabaca şudur:

> Eğer yukarıdaki koşulların hiçbiri gerçekleşmezse...

Gelin isterseniz bununla ilgili şöyle bir örnek verelim:

```
soru = input("Bir meyve adı söyleyin bana:")

if soru == "elma":
    print("evet, elma bir meyvedir...")

elif soru == "karpuz":
    print("evet, karpuz bir meyvedir...")

elif soru == "armut":
    print("evet, armut bir meyvedir...")

else:
```

```
    print(soru, "gerçekten bir meyve midir?")
```

Eğer kullanıcı soruya ‘elma’, ‘karpuz’ veya ‘armut’ cevabı verirse, *evet, ... bir meyvedir* çıktısı verilecektir. Ama eğer kullanıcı bu üçü dışında bir cevap verirse, ... *gerçekten bir meyve midir?* çıktısını görürüz. Burada else deyimi, programımıza şu anlamı katıyor:

> Eğer kullanıcı yukarıda belirlenen meyve adlarından hiç birini girmez, bunların yerine bambaşka bir şey yazarsa, o zaman else bloğu içinde belirtilen işlemi gerçekleştir.

Dikkat ederseniz yukarıdaki kodlarda if deyimlerini art arda sıralamak yerine ilk if‘ten sonra elif ile devam ettik. Peki şöyle bir şey yazarsak ne olur?

```
soru = input("Bir meyve adı söyleyin bana:")

if soru == "elma":
    print("evet, elma bir meyvedir...")

if soru == "karpuz":
    print("evet, karpuz bir meyvedir...")

if soru == "armut":
    print("evet, armut bir meyvedir...")

else:
    print(soru, "gerçekten bir meyve midir?")
```

Bu kodlar beklediğiniz sonucu vermeyecektir. İsterseniz yukarıdaki kodları çalıştırıp ne demek istediğimizi daha iyi anlayabilirsiniz. Eğer yukarıda olduğu gibi if deyimlerini art arda sıralar ve son olarak da bir else bloğu tanımlarsak, ekrana ilk bakışta anlamsız gibi görünen bir çıktı verilecektir:

```
evet, elma bir meyvedir...
elma gerçekten bir meyve midir?
```

Burada olan şey şu:

Soruya ‘elma’ cevabını verdiğimizi düşünelim. Bu durumda, Python ilk olarak ilk if bloğunu değerlendirecek ve soruya verdiğimiz cevap ‘elma’ olduğu için *evet, elma bir meyvedir...* çıktısını verecektir.

if ile elif arasındaki farkı anlatırken, hatırlarsanız art arda gelen if bloklarında Python’ın olası bütün sonuçları değerlendireceğini söylemiştik. İşte burada da böyle bir durum söz konusu. Gördüğünüz gibi, ilk if bloğundan sonra yine bir if bloğu geliyor. Bu nedenle Python olası bütün sonuçları değerlendirebilmek için blokları okumaya devam edecek ve sorunun cevabı ‘karpuz’ olmadığı için ikinci if bloğunu atlayacaktır.

Sonraki blok yine bir if bloğu olduğu için Python kodları okumaya devam ediyor. Ancak sorunun cevabı 'armut' da olmadığı için, Python sonraki if bloğunu da geçiyor ve böylece else bloğuna ulaşıyor.

Yukarıda verdiğimiz örnekteki gibi art arda if deyimlerinin sıralanıp en sona else deyiminin yerleştirildiği durumlarda else deyimi sadece bir önceki if deyimini dikkate alarak işlem yapar. Yani yukarıdaki örnekte kullanıcının verdiği cevap 'armut' olmadığı için else deyiminin olduğu blok çalışmaya başlar. Yukarıdaki örneğe 'armut' cevabını verirseniz ne demek istediğimi biraz daha iyi anlayabilirsiniz. 'armut' cevabı verilmesi durumunda sadece if soru == "armut" ifadesinin olduğu blok çalışır, else bloğu ise çalışmaz. Çünkü dediğim gibi, eğer else bloğundan önce art arda gelen if blokları varsa, else deyimi yalnızca kendisinden önceki son if bloğunu dikkate alır ve sanki yukarıdaki örnek şöyleymiş gibi davranır:

```
if soru == "armut":
    print("evet, armut bir meyvedir...")

else:
    print(soru, "gerçekten bir meyve midir?")
```

Bu tür durumlarda else deyimi bir önceki if bloğundan önce gelen bütün if bloklarını görmezden gelir ve böylece şu anlamsız görünen çıktı elde edilir:

```
evet, elma bir meyvedir...
elma gerçekten bir meyve midir?
```

Sözün özü, kullanıcının cevabı 'elma' olduğu için, yukarıdaki çıktıda yer alan ilk cümle ilk if bloğunun çalışması sonucu ekrana basılıyor. İkinci cümle ise else bloğundan bir önceki if bloğu kullanıcının cevabıyla uyuşmadığı için ekrana basılıyor.

Yalnız bu dediğimizden, else ifadesi if ile birlikte kullanılmaz, anlamı çıkarılmamalı. Mesela şöyle bir örnek yapılabilir:

```
soru = input("Programdan çıkmak istediğinize emin misiniz? \
Eminseniz 'e' harfine basın : ")

if soru == "e":
    print("Güle güle!")

else:
    print("Peki, biraz daha sohbet edelim!")
```

Burada eğer kullanıcının cevabı 'e' ise if bloğu işletilecek, eğer cevap 'e' dışında herhangi bir şey ise else bloğu çalışacaktır. Gayet mantıklı bir süreç. Ama eğer yukarıdaki örneğe bir if bloğu daha eklerseniz işler beklediğiniz gibi gitmez:

```
soru = input("Programdan çıkmak istediğinize emin misiniz? \
Eminseniz 'e' harfine basın : ")

if soru == "e":
    print("Güle güle!")

if soru == "b":
    print("Kararsız kaldım şimdi!")

else:
    print("Peki, biraz daha sohbet edelim!")
```

Bu soruya ‘e’ cevabı verdiğimizi düşünelim. Bu cevap ilk if bloğuyla uyuşuyor ve böylece ekrana *Güle güle!* çıktısı veriliyor. İlk if bloğundan sonra tekrar bir if bloğu daha geldiği için Python bütün olasılıkları değerlendirmek amacıyla blokları okumaya devam ediyor ve cevap ‘b’ olmadığı için ikinci if bloğunu atlıyor ve böylece else bloğuna ulaşıyor. Bir önceki örnekte de söylediğimiz gibi, else bloğu art arda gelen if blokları gördüğünde sadece bir önceki if bloğunu dikkate aldığı ve kullanıcının cevabı da ‘b’ olmadığı için ekrana *Peki, biraz daha sohbet edelim!* çıktısını veriyor ve ilk bakışta tuhaf görünen şöyle bir çıktı üretiyor:

```
Güle güle!
Peki, biraz daha sohbet edelim!
```

Dolayısıyla, eğer programınızda bir else bloğuna yer verecekseniz, ondan önce gelen koşullu durumların ilkini if ile sonrakileri ise elif ile bağlayın. Yani:

```
if koşul_1:
    sonuç_1

elif koşul_2:
    sonuç_2

elif koşul_3:
    sonuç_3

else:
    sonuç_4
```

Ama eğer else bloğundan önce sadece tek bir koşul bloğu yer alacaksa bunu if ile bağlayın. Yani:

```
if koşul_1:
    sonuç_1
```

```
else:
    sonuç_2
```

Programlarımızın doğru çalışması ve istediğimiz sonucu verebilmesi için bu tür ayrıntılara olabildiğince dikkat etmemiz gerekiyor. Neticede koşullu durumlar mantıkla ilgilidir. Dolayısıyla koşullu durumlarla muhatap olurken mantığınızı hiçbir zaman devre dışı bırakmamalısınız.

Bir önceki bölümde elif deyiminin tam olarak ne işe yaradığını anlamamızı sağlayacak bir örnek vereceğimizi söylemiştik. Şimdi bu örneğe bakalım:

```
boy = int(input("boyunuz kaç cm?"))

if boy < 170:
    print("boyunuz kısa")

elif boy < 180:
    print("boyunuz normal")

else:
    print("boyunuz uzun")
```

Yukarıda yedi satırla hallettiğimiz işi sadece if deyimleriyle yapmaya çalışırsanız bunun ne kadar zor olduğunu göreceksiniz. Diyelim ki kullanıcı '165' cevabını verdi. Python bu *165* sayısının *170*'ten küçük olduğunu görünce *boyunuz kısa* cevabını verecek, öteki satırları değerlendirmeyecektir. *165* sayısı, elif ile gösterdiğimiz koşullu duruma da uygun olduğu halde (165 < 180), koşul ilk blokta karşılandığı için ikinci blok değerlendirmeye alınmayacaktır.

Kullanıcının '175' cevabını verdiğini varsayalım: Python *175* sayısını görünce önce ilk koşula bakacak, verilen *175* sayısının ilk koşulu karşılamadığını görecektir (175 > 170). Bunun üzerine Python kodları incelemeye devam edecek ve elif bloğunu değerlendirmeye alacaktır. *175* sayısının *180*'den küçük olduğunu görünce de çıktı olarak *boyunuz normal* cevabını verecektir.

Peki ya kullanıcı '190' cevabını verirse ne olacak? Python yine önce ilk if bloğuna bakacak ve *190* cevabının bu bloğa uymadığını görecektir. Dolayısıyla ilk bloğu bırakıp ikinci bloğa bakacaktır. *190* cevabının bu bloğa da uymadığını görünce, bir sonraki bloğu değerlendirmeye alacaktır. Bir sonraki blokta ise else deyimimiz var. Bu bölümde öğrendiğimiz gibi, else deyimi, 'eğer kullanıcının cevabı yukarıdaki koşulların hiçbirine uymazsa bu bloğu çalıştır,' anlamına geliyor. Kullanıcının girdiği *190* cevabı ne birinci ne de ikinci bloktaki koşula uyduğu için, normal bir şekilde else bloğu işletilecek, dolayısıyla da ekrana *boyunuz uzun* çıktısı verilecektir.

Böylece Python'da if, elif ve else deyimlerini incelemiş olduk. Ancak tabii ki bu deyimlerle işimiz henüz bitmedi. Elimizdeki bilgiler şimdilik bu deyimleri ancak bu kadar incelememize yetiyor, ama ilerleyen sayfalarda bazı başka araçları da bilgi dağarcığımıza kattıktan sonra bu deyimlerin daha farklı yönlerini öğrenme imkanına kavuşacağız.

Örnek Uygulama

Önceki derslerimizde len() fonksiyonunu anlatırken şöyle bir program tasarısından bahsetmiştik hatırlarsanız:

> Diyelim ki sisteme kayıt için kullanıcı adı ve parola belirlenmesini isteyen bir program yazıyorsunuz. Yazacağınız bu programda, belirlenebilecek kullanıcı adı ve parolanın toplam uzunluğu *40* karakteri geçmeyecek.

O zaman henüz koşullu durumları öğrenmemiş olduğumuz için, yukarıda bahsettiğimiz programın ancak şu kadarlık kısmını yazabilmiştik:

```
kullanıcı_adı = input("Kullanıcı adınız: ")
parola        = input("Parolanız      : ")

toplam_uzunluk = len(kullanıcı_adı) + len(parola)
```

Burada yapabildiğimiz tek şey, kullanıcıdan kullanıcı adı ve parola bilgilerini alıp, bu bilgilerin karakter uzunluğunu ölçebilmekti. Ama artık koşullu durumları öğrendiğimize göre bu programı eksiksiz olarak yazabiliriz. Şu kodları dikkatlice inceleyin:

```
kullanıcı_adı = input("Kullanıcı adınız: ")
parola        = input("Parolanız      : ")

toplam_uzunluk = len(kullanıcı_adı) + len(parola)

mesaj = "Kullanıcı adı ve parolanız toplam {} karakterden oluşuyor!"

print(mesaj.format(toplam_uzunluk))

if toplam_uzunluk >= 40:
    print("Kullanıcı adınız ile parolanızın toplam uzunluğu 40 karakteri geçmemeli!")
else:
    print("Sisteme hoşgeldiniz!")
```

Burada öncelikle kullanıcıdan kullanıcı adı ve parola bilgilerini alıyoruz. Daha sonra da kullanıcıdan gelen bu bilgilerin toplam karakter uzunluğunu hesaplıyoruz. Bunun için len() fonksiyonundan yararlanmamız gerektiğini hatırlıyor olmalısınız.

Eğer toplam uzunluk 40 karakterden fazla ise, if bloğunda verilen mesajı gösteriyoruz. Bunun dışındaki bütün durumlarda ise else bloğunu devreye sokuyoruz.

başa dön

İşleçler

Bu bölümde, aslında pek de yabancısı olmadığımız ve hatta önceki derslerimizde üstünkörü de olsa değindiğimiz bir konuyu çok daha ayrıntılı bir şekilde ele alacağız. Burada anlatacağımız konu size yer yer sıkıcı gelebilir. Ancak bu konuyu hakkıyla öğrenmenizin, programcılık maceranız açısından hayati önemde olduğunu rahatlıkla söyleyebilirim.

Gelelim konumuza...

Bu bölümün konusu işleçler. Peki nedir bu 'işleç' denen şey?

İngilizce'de *operator* adı verilen işleçler, sağında ve solunda bulunan değerler arasında bir ilişki kuran işaretlerdir. Bir işlecin sağında ve solunda bulunan değerlere ise işlenen (*operand*) adı veriyoruz.

Not

Türkçede işleç yerine operatör, işlenen yerine de operant dendiğine tanık olabilirsiniz.

Biz bu bölümde işleçleri altı başlık altında inceleyeceğiz:

1. Aritmetik İşleçler
2. Karşılaştırma İşleçleri
3. Bool İşleçleri
4. Değer Atama İşleçleri
5. Aitlik İşleçleri
6. Kimlik İşleçleri

Gördüğünüz gibi, işlememiz gereken konu çok, gitmemiz gereken yol uzun. O halde hiç vakit kaybetmeden, aritmetik işleçlerle yolculuğumuza başlayalım.

Aritmetik İşleçler

Dedik ki, sağında ve solunda bulunan değerler arasında bir ilişki kuran işaretlere işleç (*operator*) adı verilir. Önceki derslerimizde temel işleçlerin bazılarını öğrenmiştik. İsterseniz bunları şöyle bir hatırlayalım:

+	toplama
-	çıkarma
*	çarpma
/	bölme
**	kuvvet

Bu işleçlere aritmetik işleçler adı verilir. Aritmetik işleçler; matematikte kullanılan ve sayılarla aritmetik işlemler yapmamızı sağlayan yardımcı araçlardır.

Dilerseniz bu tanımı bir örnekle somutlaştıralım:

```
>>> 45 + 33

78
```

Burada *45* ve *33* değerlerine işlenen (*operand*) adı verilir. Bu iki değer arasında yer alan + işareti ise bir işleçtir (*operator*). Dikkat ederseniz + işleci *45* ve *33* adlı işlenenler arasında bir toplama ilişkisi kuruyor.

Bir örnek daha verelim:

```
>>> 23 * 46

1058
```

Burada da *23* ve *46* değerleri birer işlenendir. Bu iki değer arasında yer alan * işareti ise, işlenenler arasında bir çarpma ilişkisi kuran bir işleçtir.

Ancak bir noktaya özellikle dikkatinizi çekmek istiyorum. Daha önceki derslerimizde de değindiğimiz gibi, + ve * işleçleri Python'da birden fazla anlama gelir. Örneğin yukarıdaki örnekte + işleci, işlenenler arasında bir toplama ilişkisi kuruyor. Ama aşağıdaki durum biraz farklıdır:

```
>>> "istihza" + ".com"

'istihza.com'
```

Burada + işleci işlenenler (*"istihza"* ve *".com"*) arasında bir birleştirme ilişkisi kuruyor.

Tıpkı + işlecinde olduğu gibi, * işleci de Python'da birden fazla anlama gelir. Bu işlecin, çarpma ilişkisi kurma işlevi dışında tekrar etme ilişkisi kurma işlevi de vardır. Yani:

```
>>> "hızlı " * 2

'hızlı hızlı '
```

...veya:

```
>>> "-" * 30

'------------------------------'
```

Burada * işlecinin, sayılar arasında çarpma işlemi yapmak dışında bir görev üstlendiğini görüyoruz.

Python'da bu tür farklar, yazacağınız programın sağlıklı çalışabilmesi açısından büyük önem taşır. O yüzden bu tür farklara karşı her zaman uyanık olmamız gerekiyor.

+ ve * işleçlerinin aksine / ve - işleçleri ise işlenenler arasında sadece bölme ve çıkarma ilişkisi kurar. Bu işleçler tek işlevlidir:

```
>>> 25 / 4

6.25

>>> 10 - 5

5
```

Önceki derslerde gördüğümüz ve yukarıda da tekrar ettiğimiz dört adet temel aritmetik işlece şu iki aritmetik işleci de ekleyelim:

%	modülüs
//	taban bölme

İlk önce modülüsün ne olduğunu ve ne işe yaradığını anlamaya çalışalım.

Şu bölme işlemine bir bakın:

```
 30 | 4
 28 | 7
 --
 02
```

Burada *02* sayısı bölme işleminin kalanıdır. İşte modülüs denen işleç de bölme işleminden kalan bu değeri gösterir. Yani:

```
>>> 30 % 4

2
```

Gördüğünüz gibi modülüs işleci (%) gerçekten de bölme işleminden kalan sayıyı gösteriyor... Peki bu bilgi ne işimize yarar?

Mesela bu bilgiyi kullanarak bir sayının tek mi yoksa çift mi olduğunu tespit edebiliriz:

```
sayı = int(input("Bir sayı girin: "))

if sayı % 2 == 0:
    print("Girdiğiniz sayı bir çift sayıdır.")
else:
```

```
    print("Girdiğiniz sayı bir tek sayıdır.")
```

Eğer bir sayı *2*'ye bölündüğünde kalan değer *0* ise o sayı çifttir. Aksi halde o sayı tektir. Mesela:

```
>>> 14 % 2

0
```

Gördüğünüz gibi, bir çift sayı olan *14*'ü *2*'ye böldüğümüzde kalan sayı *0* oluyor. Çünkü çift sayılar *2*'ye tam bölünürler.

Bir de şuna bakalım:

```
>>> 15 % 2

1
```

Bir tek sayı olan *15* ise *2*'ye bölündüğünde kalan sayı *1* oluyor. Yani *15* sayısı *2*'ye tam bölünmüyor. Bu bilgiden yola çıkarak 15 sayısının bir tek sayı olduğunu söyleyebiliyoruz.

Bir sayının tek mi yoksa çift mi olduğunu tespit etme işlemini küçümsememenizi tavsiye ederim. Bir sayının tek mi yoksa çift mi olduğu bilgisinin, arayüz geliştirirken dahi işinize yarayacağından emin olabilirsiniz.

Elbette modülüs işlecini bir sayının yalnızca *2*'ye tam bölünüp bölünmediğini denetlemek için kullanmıyoruz. Bu işleci kullanarak herhangi bir sayının herhangi bir sayıya tam bölünüp bölünmediğini de denetleyebilirsiniz. Örneğin:

```
>>> 45 % 4

1

>>> 36 % 9

0
```

Bu bilgiyi kullanarak mesela şöyle bir program yazabilirsiniz:

```
bölünen = int(input("Bir sayı girin: "))
bölen = int(input("Bir sayı daha girin: "))

şablon = "{} sayısı {} sayısına tam".format(bölünen, bölen)

if bölünen % bölen == 0:
    print(şablon, "bölünüyor!")
```

```
else:
    print(şablon, "bölünmüyor!")
```

Programımız, kullanıcının girdiği ilk sayının ikinci sayıya tam bölünüp bölünmediğini hesaplıyor ve sonuca göre kullanıcıyı bilgilendiriyor. Bu kodlarda özellikle şu satıra dikkat edin:

```
if bölünen % bölen == 0:
    ...
```

Programımızın temelini bu kod oluşturuyor. Çünkü bir sayının bir sayıya tam bölünüp bölünmediğini bu kodla belirliyoruz. Eğer bir sayı başka bir sayıya bölündüğünde kalan değer, yani modülüs *0* ise, o sayı öbür sayıya tam bölünüyor demektir.

Ayrıca bir sayının son basamağını elde etmek için de modülüsten yararlanabilirsiniz. Herhangi bir tamsayı *10*'a bölündüğünde kalan (yani modülüs), bölünen sayının son basamağı olacaktır:

```
>>> 65 % 10

5

>>> 543 % 10

3
```

Programlama tecrübeniz arttıkça, aslında modülüsün ne kadar faydalı bir araç olduğunu kendi gözlerinizle göreceksiniz.

Modülüs işlecini örnekler eşliğinde ayrıntılı bir şekilde incelediğimize göre sıra geldi taban bölme işlecini açıklamaya...

Öncelikle şu örneği inceleyelim:

```
>>> 5 / 2

2.5
```

Burada, bildiğimiz bölme işlecini (/) kullanarak basit bir bölme işlemi yaptık. Elde ettiğimiz sonuç doğal olarak *2.5*.

Matematikte bölme işleminin sonucunun kesirli olması durumuna 'kesirli bölme' adı verilir. Bunun tersi ise tamsayılı bölme veya taban bölmedir. Eğer herhangi bir sebeple kesirli bölme işlemi değil de taban bölme işlemi yapmanız gerekirse // işlecinden yararlanabilirsiniz:

```
>>> 5 // 2

2
```

Gördüğünüz gibi, // işleci sayesinde bölme işleminin sonucu kesirli değil, tamsayı olarak elde ediliyor.

Yukarıda yaptığımız taban bölme işlemi şununla aynı anlama gelir:

```
>>> int(5 / 2)

2
```

Daha açık ifade etmemiz gerekirse:

```
>>> a = 5 / 2
>>> a

2.5

>>> int(a)

2
```

Burada olan şu: 5 / 2 işleminin sonucu bir kayan noktalı sayıdır (*2.5*). Bunu şu şekilde teyit edebiliriz:

```
>>> a = 5 / 2
>>> type(a)

<class 'float'>
```

Buradaki *float* çıktısının *floating point number*, yani kayan noktalı sayı anlamına geldiğini biliyorsunuz.

Bu kayan noktalı sayının sadece tabanını elde etmek için bu sayıyı tamsayıya (*integer*) çevirmemiz yeterli olacaktır. Yani:

```
>>> int(a)

2
```

Bu arada yeri gelmişken round() adlı bir gömülü fonksiyondan bahsetmeden geçmeyelim. Eğer bir sayının değerini yuvarlamanız gerekirse round() fonksiyonundan yararlanabilirsiniz. Bu fonksiyon şöyle kullanılır:

```
>>> round(2.55)

3
```

Gördüğünüz gibi, round() fonksiyonuna parametre olarak bir sayı veriyoruz. Bu fonksiyon da bize o sayının yuvarlanmış halini döndürüyor. Bu fonksiyonu kullanarak yuvarlanacak sayının noktadan sonraki hassasiyetini de belirleyebilirsiniz. Örneğin:

```
>>> round(2.55, 1)

2.5
```

Burada ikinci parametre olarak *1* sayısını verdiğimiz için, noktadan sonraki bir basamak görüntüleniyor. Bir de şuna bakalım:

```
>>> round(2.68, 1)

2.7
```

Burada da yuvarlama işlemi yapılırken noktadan sonra bir basamak korunuyor. Eğer *1* sayısı yerine *2* sayısını kullanırsanız, yukarıdaki örnek şu çıktıyı verir:

```
>>> round(2.68, 2)

2.68
```

round() fonksiyonunun çalışma prensibini anlamak için kendi kendinize örnekler yapabilirsiniz.

Şimdiye kadar öğrendiğimiz ve yukarıdaki tabloda andığımız bir başka aritmetik işleç de kuvvet işleci (**) idi. Mesela bu işleci kullanarak bir sayının karesini hesaplayabileceğimizi biliyorsunuz:

```
>>> 25 ** 2

625
```

Bir sayının *2.* kuvveti o sayının karesidir. Bir sayının *0.5.* kuvveti ise o sayının kareköküdür:

```
>>> 625 ** 0.5

25.0
```

Bu arada, eğer karekökün kayan noktalı sayı cinsinden olması hoşunuza gitmediyse, bu sayıyı int() fonksiyonu ile tam sayıya çevirebileceğinizi biliyorsunuz:

```
>>> int(625 ** 0.5)

25
```

Kuvvet hesaplamaları için ** işlecinin yanısıra pow() adlı bir fonksiyondan da

yararlanabileceğimizi öğrenmiştik:

```
>>> pow(25, 2)

625
```

Bildiğiniz gibi pow() fonksiyonu aslında toplam üç parametre alabiliyor:

```
>>> pow(25, 2, 5)

0
```

Bu işlemin şununla aynı anlama geliyor:

```
>>> (25 ** 2) % 5

0
```

Yani pow(25, 2, 5) gibi bir komut verdiğimizde, *25* sayısının *2.* kuvvetini alıp, elde ettiğimiz sayının *5*'e bölünmesinden kalan sayıyı hesaplamış oluyoruz.

Böylece aritmetik işleçleri tamamlamış olduk. Artık karşılaştırma işleçlerini inceleyebiliriz.

Karşılaştırma İşleçleri

Adından da anlaşılacağı gibi, karşılaştırma işleçleri, işlenenler (*operands*) arasında bir karşılaştırma ilişkisi kuran işleçlerdir. Bu işleçleri şöyle sıralayabiliriz:

==	eşittir
!=	eşit değildir
>	büyüktür
<	küçüktür
>=	büyük eşittir
<=	küçük eşittir

Bu işleçlerin hiçbiri size yabancı değil, zira bunların hepsini aslında daha önceki derslerde verdiğimiz örneklerde kullanmıştık. Burada da bunlarla ilgili basit bir örnek vererek yolumuza devam edelim:

```
parola = "xyz05"

soru = input("parolanız: ")

if soru == parola:
```

```
    print("doğru parola!")

elif soru != parola:
    print("yanlış parola!")
```

Burada *soru* değişkeniyle kullanıcıdan alınan verinin, programın başında tanımladığımız *parola* değişkeninin değerine eşit olup olmadığını sorguluyoruz. Buna göre, eğer kullanıcıdan gelen veri parolayla eşleşiyorsa (if soru == parola), kullanıcıyı parolanın doğru olduğu konusunda bilgilendiriyoruz (print("doğru parola!")). Ama eğer kullanıcıdan gelen veri parolayla eşleşmiyorsa (elif soru != parola), o zaman da kullanıcıya parolanın yanlış olduğunu bildiriyoruz (print("yanlış parola!")).

Yukarıdaki örnekte == (eşittir) ve *!=* (eşit değildir) işleçlerinin kullanımını örneklendirdik. Öteki karşılaştırma işleçlerinin de nasıl kullanıldığını biliyorsunuz. Basit bir örnek verelim:

```
sayı = input("sayı: ")

if int(sayı) <= 100:
    print("sayı 100 veya 100'den küçük")

elif int(sayı) >= 100:
    print("sayı 100 veya 100'den büyük")
```

Böylece karşılaştırma işleçlerini de incelemiş olduk. O halde gelelim bool işleçlerine...

Bool İşleçleri

Bu bölümde bool işleçlerinden söz edeceğiz, ancak bool işleçlerine geçmeden önce biraz bool kavramından bahsetmemiz yerinde olacaktır.

Nedir bu bool denen şey?

Bilgisayar bilimi iki adet değer üzerine kuruludur: *1* ve *0*. Yani sırasıyla *True* ve *False*. Bilgisayar biliminde herhangi bir şeyin değeri ya *True*, ya da *False*'tur. İşte bu *True* ve *False* olarak ifade edilen değerlere bool değerleri adı verilir (George Boole adlı İngiliz matematikçi ve filozofun adından). Türkçe olarak söylemek gerekirse, *True* değerinin karşılığı *Doğru*, *False* değerinin karşılığı ise *Yanlış*'tır.

Örneğin:

```
>>> a = 1
```

Burada *a* adlı bir değişken tanımladık. Bu değişkenin değeri *1*. Şimdi bu değişkenin değerini sorgulayalım:

```
>>> a == 1 #a değeri 1'e eşit mi?

True
```

Gördüğünüz gibi, *a == 1* sorgusu *True* (Doğru) çıktısı veriyor. Çünkü *a* değişkeninin değeri gerçekten de *1*. Bir de şunu deneyelim:

```
>>> a == 2

False
```

Burada da *a* değişkeninin değerinin *2* sayısına eşdeğer olup olmadığını sorguladık. *a* değişkeninin değeri *2* olmadığı için de Python bize *False* (Yanlış) çıktısı verdi.

Gördüğünüz gibi, bool işleçleri herhangi bir ifadenin doğruluğunu veya yanlışlığını sorgulamak için kullanılabiliyor. Buna göre, eğer bir sorgulamanın sonucu doğru ise *True*, eğer yanlış ise *False* çıktısı alıyoruz.

Bool işleçleri sadece yukarıda verdiğimiz örneklerdeki gibi, salt bir doğruluk-yanlışlık sorgulamaya yarayan araçlar değildir. Bilgisayar biliminde her şeyin bir bool değeri vardır. Bununla ilgili genel kuralımız şu: *0* değeri ve boş veri tipleri *False*'tur. Bunlar dışında kalan her şey ise *True*'dur.

Bu durumu bool() adlı özel bir fonksiyondan yararlanarak teyit edebiliriz:

```
>>> bool(3)

True

>>> bool("elma")

True

>>> bool(" ")

True

>>> bool("     ")

True

>>> bool("fdsdfsdg")

True

>>> bool("0")
```

```
True

>>> bool(0)

False

>>> bool("")

False
```

Gördüğünüz gibi, gerçekten de *0* sayısının ve boş karakter dizilerinin bool değeri *False*'tur. Geri kalan her şey ise *True*'dur.

Not

0'ın bir sayı, *"0"*'ın ise bir karakter dizisi olduğunu unutmayın. Sayı olan *0*'ın bool değeri *False*'tur, ama karakter dizisi olan *"0"*'ın değeri *True*'dur.

Yukarıdaki örneklere göre, içinde herhangi bir değer barındıran karakter dizileri (*0* hariç) *True* çıktısı veriyor. Burada söylediğimiz şey bütün veri tipleri için geçerlidir. Eğer herhangi bir veri tipi herhangi bir değer içermiyorsa o veri tipi *False* çıktısı verir.

Peki bu bilgi bizim ne işimize yarar? Yani mesela boş veri tiplerinin *False*, içinde bir veri barındıran veri tiplerinin ise *True* olması bizim için neden bu kadar önemli? Bunu birazdan açıklayacağız. Ama önce isterseniz, bool değerleri ile ilgili çok önemli bir konuya değinelim.

Belki kendiniz de farketmişsinizdir; bool değerleri Python'da koşul belirten if, elif ve else deyimlerinin de temelini oluşturur. Şu örneği ele alalım mesela:

```
isim = input("İsminiz: ")

if isim == "Ferhat":
    print("Ne güzel bir isim bu!")
else:
    print(isim, "ismini pek sevmem!")
```

Burada if isim == "Ferhat" dediğimizde, aslında Python'a şu emri vermiş oluyoruz:

Eğer isim == "Ferhat" ifadesi *True* ise...

Bunu teyit etmek için şöyle bir kod yazabilirsiniz:

```
isim = input("İsminiz: ")
```

```
print(isim == "Ferhat")
```

Eğer burada kullanıcı 'Ferhat' ismini girecek olursa programımız *True* çıktısı verir. Ama eğer kullanıcı başka bir isim girerse bu kez *False* çıktısını alırız. İşte koşul bildiren deyimler, karar verme görevini, kendilerine verilen ifadelerin bool değerlerine bakarak yerine getirir. Dolayısıyla yukarıdaki örneği şu şekilde Türkçeye çevirebiliriz:

> Eğer isim == "Ferhat" ifadesinin bool değeri *True* ise, *Ne güzel bir isim bu!* çıktısı ver! Ama eğer isim == "Ferhat" ifadesinin bool değeri *True* dışında herhangi bir şey ise (yani *False* ise), ... *ismini pek sevmem!* çıktısı ver!

Koşul bildiren deyimlerle bool değerleri arasındaki ilişkiyi daha iyi anlamak için bir örnek daha verelim:

Hatırlarsanız içi boş veri tiplerinin bool değerinin her zaman *False* olacağını söylemiştik. Yani:

```
>>> a = ""

>>> bool(a)

False
```

Herhangi bir değere sahip veri tiplerinin bool değeri ise her zaman *True* olur (*0* hariç):

```
>>> a = "gdfg"

>>> bool(a)

True
```

İçi boş veri tiplerinin bool değerinin her zaman *False* olacağı bilgisini kullanarak şöyle bir uygulama yazabiliriz:

```
kullanıcı = input("Kullanıcı adınız: ")

if bool(kullanıcı) == True:
    print("Teşekkürler!")
else:
    print("Kullanıcı adı alanı boş bırakılamaz!")
```

Burada şöyle bir emir verdik:

> "Eğer *kullanıcı* değişkeninin bool değeri *True* ise *Teşekkürler!* çıktısı ver! Değilse *Kullanıcı adı alanı boş bırakılamaz!* uyarısını göster!

Eğer kullanıcı, kullanıcı adına herhangi bir şey yazdıktan sonra *Enter* tuşuna basarsa *kullanıcı* değişkeni, kullanıcının girdiği değeri gösterecek ve böylece bool(kullanıcı) komutu *True* çıktısı verecektir. Bu sayede de kodlarımızın içindeki if bloğu çalışmaya başlayacaktır.

Ama eğer kullanıcı, kullanıcı adını yazmadan *Enter* tuşuna basarsa, *kullanıcı* değişkeni boş kalacağı için (yani kullanıcı = "" gibi bir durum ortaya çıkacağı için) bool(kullanıcı) komutu *False* çıktısı verecek ve böylece else bloğu çalışacaktır.

Yalnız bu noktada şöyle bir uyarı yapalım. Yukarıdaki komutlar sözdizimi açısından tamamen doğru olsa da, etrafta yukarıdakine benzer bir kullanımı pek görmezsiniz. Aynı iş için genellikle şöyle bir şeyler yazılır:

```
kullanıcı = input("Kullanıcı adınız: ")

if kullanıcı:
    print("Teşekkürler!")
```

Gördüğünüz gibi, if bool(kullanıcı) == True: kodunu if kullanıcı: şeklinde kısaltabiliyoruz. Bu ikisi tamamen aynı anlama gelir. Yani ikisi de 'kullanıcı değişkeninin bool değeri *True* ise...' demektir.

Bool kavramına aşinalık kazandığımıza göre şimdi bool işleçlerini incelemeye başlayabiliriz.

Bool işleçleri, bool değerlerinden birini elde etmemizi sağlayan işleçlerdir. Bu işleçler şunlardır:

and

or

not

Eğer mantık dersleri aldıysanız bu işleçler size hiç yabancı gelmeyecektir. Eğer lisede mantık dersleri almadıysanız veya aldığınız derslerden hiçbir şey hatırlamıyorsanız, yine de ziyanı yok. Biz burada bu işleçleri bütün ayrıntılarıyla inceleyeceğiz.

Önce *and* ile başlayalım...

Türkçe söylemek gerekirse *and* 've' anlamına gelir. Peki bu *and* ne işimize yarar? Çok basit bir örnek verelim:

Hatırlarsanız geçen bölümde koşullu durumlara örnek verirken şöyle bir durumdan bahsetmiştik:

> Diyelim ki Google'ın Gmail hizmeti aracılığıyla bir e.posta hesabı aldınız. Bu hesaba gireceğiniz zaman Gmail size bir kullanıcı adı ve parola sorar. Siz de kendinize ait kullanıcı adını ve parolayı sayfadaki kutucuklara yazarsınız. Eğer yazdığınız kullanıcı adı ve parola doğruysa hesabınıza erişebilirsiniz. Ama eğer kullanıcı adınız ve parolanız doğru değilse

hesabınıza erişemezsiniz. Yani e.posta hesabınıza erişmeniz, kullanıcı adı ve parolayı doğru girme koşuluna bağlıdır.

Burada çok önemli bir nokta var. Kullanıcının Gmail sistemine girebilmesi için hem kullanıcı adını hem de parolayı doğru yazması gerekiyor. Yani kullanıcı adı veya paroladan herhangi biri yanlış ise sisteme giriş mümkün olmayacaktır.

Yukarıdaki durumu taklit eden bir programı, şu ana kadar olan bilgilerimizi kullanarak şöyle yazabiliyoruz:

```
kullanıcı_adı = input("Kullanıcı adınız: ")
parola = input("Parolanız: ")

if kullanıcı_adı == "aliveli":
    if parola == "12345678":
        print("Programa hoşgeldiniz")

else:
    print("Yanlış kullanıcı adı veya parola!")
```

Burada yeni bir bilgiyle daha karşılaşıyoruz. Gördüğünüz gibi, burada if deyimlerini iç içe kullandık. Python'da istediğiniz kadar iç içe geçmiş if deyimi kullanabilirsiniz. Ancak yazdığınız bir programda eğer üçten fazla iç içe if deyimi kullandıysanız, benimsediğiniz yöntemi yeniden gözden geçirmenizi tavsiye ederim. Çünkü iç içe geçmiş if deyimleri bir süre sonra anlaşılması güç bir kod yapısı ortaya çıkarabilir. Neyse... Biz konumuza dönelim.

Yukarıdaki yazdığımız programda kullanıcının sisteme giriş yapabilmesi için hem kullanıcı adını hem de parolayı doğru girmesi gerekiyor. Kullanıcı adı ve paroladan herhangi biri yanlışsa sisteme girişe izin verilmiyor. Ancak yukarıdaki yöntem dolambaçlıdır. Halbuki aynı işlevi yerine getirmenin, Python'da çok daha kolay bir yolu var. Bakalım:

```
kullanıcı_adı = input("Kullanıcı adınız: ")
parola = input("Parolanız: ")

if kullanıcı_adı == "aliveli" and parola == "12345678":
    print("Programa hoşgeldiniz")

else:
    print("Yanlış kullanıcı adı veya parola!")
```

Burada *and* işlecini nasıl kullandığımızı görüyorsunuz. Bu işleci kullanarak iki farklı ifadeyi birbirine bağladık. Böylece kullanıcının sisteme girişini hem kullanıcı adının hem de parolanın doğru olması koşuluna dayandırdık.

Peki *and* işlecinin çalışma mantığı nedir? Dediğim gibi, *and* Türkçede 've' anlamına geliyor. Bu

işleci daha iyi anlayabilmek için şu cümleler arasındaki farkı düşünün:

1. Toplantıya Ali ve Veli katılacak.
2. Toplantıya Ali veya Veli katılacak.

İlk cümlede 've' bağlacı kullanıldığı için, bu cümlenin gereğinin yerine getirilebilmesi, hem Ali'nin hem de Veli'nin toplantıya katılmasına bağlıdır. Sadece Ali veya sadece Veli'nin toplantıya katılması durumunda bu cümlenin gereği yerine getirilememiş olacaktır.

İkinci cümlede ise toplantıya Ali ve Veli'den herhangi birisinin katılması yeterlidir. Toplantıya sadece Ali'nin katılması, sadece Veli'nin katılması veya her ikisinin birden katılması, bu cümlenin gereğinin yerine getirilebilmesi açısından yeterlidir.

İşte Python'daki *and* işleci de aynı bu şekilde işler. Şu örneklere bir bakalım:

```
>>> a = 23
>>> b = 10
>>> a == 23

True

>>> b == 10

True

>>> a == 23 and b == 10

True
```

Burada değeri *23* olan bir adet *a* değişkeni ve değeri *10* olan bir adet *b* değişkeni tanımladık. Daha sonra bu iki değişkenin değerini tek tek sorguladık ve bunların gerçekten de sırasıyla 23 ve 10 sayısına eşit olduğunu gördük. Son olarak da bunları *and* işleci ile birbirine bağlayarak sorguladık. *a* değişkeninin değeri 23, *b* değişkeninin değeri de 10 olduğu için, yani *and* ile bağlanan her iki önerme de *True* çıktısı verdiği için `a == 23 and b == 10` ifadesi *True* değeri verdi.

Bir de şuna bakalım:

```
>>> a = 23
>>> b = 10
>>> a == 23

True

>>> b == 54
```

```
False

>>> a == 23 and b == 54

False
```

Burada ise *a* değişkenin değeri *23*'tür. Dolayısıyla `a == 23` ifadesi *True* çıktısı verir. Ancak *b* değişkeninin değeri *54* değildir. O yüzden de `b == 54` komutu *False* çıktısı verir. Gördüğünüz gibi, *and* işleci ile bağlanan önermelerden herhangi biri *False* olduğunda çıktımız da *False* oluyor. Unutmayın: *and* işlecinin *True* çıktısı verebilmesi için bu işleç tarafından bağlanan her iki önermenin de *True* olması gerekir. Eğer önermelerden biri bile *True* değilse çıktı da *True* olmayacaktır.

Tahmin edebileceğiniz gibi, *and* işleci en yaygın `if` deyimleriyle birlikte kullanılır. Mesela yukarıda kullanıcıdan kullanıcı adı ve parola alırken de bu *and* işlecinden yararlanmıştık.

Gelelim *or* işlecine...

Tıpkı *and* gibi bir bool işleci olan *or*'un Türkçede karşılığı 'veya'dır. Yukarıda 'Toplantıya Ali veya Veli katılacak.' cümlesini tartışırken aslında bu *or* kelimesinin anlamını açıklamıştık. Hatırlarsanız *and* işlecinin *True* çıktısı verebilmesi için bu işleçle bağlanan bütün önermelerin *True* değerine sahip olması gerekiyordu. *or* işlecinin *True* çıktısı verebilmesi için ise *or* işleciyle bağlanan önermelerden herhangi birinin *True* çıktısı vermesi yeterli olacaktır. Söylediğimiz bu şeyleri birkaç örnek üzerinde somutlaştıralım:

```
>>> a = 23
>>> b = 10
>>> a == 23

True

>>> b == 10

True

>>> a == 11

False

>>> a == 11 or b == 10

True
```

Gördüğünüz gibi, `a == 11` ifadesinin bool değeri *False* olduğu halde, `b == 10` ifadesinin bool değeri *True* olduğu için `a == 11 or b == 10` ifadesi *True* değerini veriyor.

and ve *or* işleçlerini öğrendiğimize göre, bir sınavdan alınan notların harf karşılıklarını gösteren bir uygulama yazabiliriz:

```
x = int(input("Notunuz: "))

if x > 100 or x < 0:
    print("Böyle bir not yok")

elif x >= 90 and x <= 100:
    print("A aldınız.")

elif x >= 80 and x <= 89:
    print("B aldınız.")

elif x >= 70 and x <= 79:
    print("C aldınız.")

elif x >= 60 and x <= 69:
    print("D aldınız.")

elif x >= 0 and x <= 59:
    print("F aldınız.")
```

Bu programda eğer kullanıcı *100*'den büyük ya da *0*'dan küçük bir sayı girerse *Böyle bir not yok* uyarısı alacaktır. 0-100 arası notlarda ise, her bir not aralığına karşılık gelen harf görüntülecektir. Eğer isterseniz yukarıdaki kodları şu şekilde de kısaltabilirsiniz:

```
x = int(input("Notunuz: "))

if x > 100 or x < 0:
    print("Böyle bir not yok")

elif x >= 90 <= 100:
    print("A aldınız.")

elif x >= 80 <= 89:
    print("B aldınız.")

elif x >= 70 <= 79:
    print("C aldınız.")

elif x >= 60 <= 69:
    print("D aldınız.")
```

```
elif x >= 0 <= 59:
    print("F aldınız.")
```

Gördüğünüz gibi, and x kısımlarını çıkardığımızda da bir önceki kodlarla aynı anlamı yakalayabiliyoruz.

Hatta yukarıdaki kodları şöyle de yazabilirsiniz:

```
x = int(input("Notunuz: "))

if x > 100 or x < 0:
    print("Böyle bir not yok")

#90 sayısı x'ten küçük veya x'e eşit,
#x sayısı 100'den küçük veya 100'e eşit ise,
#Yani x, 90 ile 100 arasında bir sayı ise
elif 90 <= x <= 100:
    print("A aldınız.")

#80 sayısı x'ten küçük veya x'e eşit,
#x sayısı 89'dan küçük veya 89'a eşit ise,
#Yani x, 80 ile 89 arasında bir sayı ise
elif 80 <= x <= 89:
    print("B aldınız.")

elif 70 <= x <= 79:
    print("C aldınız.")

elif 60 <= x <= 69:
    print("D aldınız.")

elif 0 <= x <= 59:
    print("F aldınız.")
```

Bu kodlar bir öncekiyle aynı işi yapar. Yorumlardan da göreceğiniz gibi, bu iki kod arasında sadece mantık farkı var.

Son bool işlecimiz *not*. Bu kelimenin İngilizce'deki anlamı 'değil'dir. Bu işleci şöyle kullanıyoruz:

```
>>> a = 23
>>> not a

False
```

```
>>> a = ""
>>> not a

True
```

Bu işleç, özellikle kullanıcı tarafından bir değişkene veri girilip girilmediğini denetlemek için kullanılabilir. Örneğin:

```
parola = input("parola: ")

if not parola:
    print("Parola boş bırakılamaz!")
```

Eğer kullanıcı herhangi bir parola belirlemeden doğrudan *Enter* tuşuna basacak olursa *parola* değişkeninin değeri boş bir karakter dizisi olacaktır. Yani parola = "". Boş veri tiplerinin bool değerinin *False* olacağını biliyoruz. Dolayısıyla, yukarıdaki gibi bir örnekte, kullanıcı parolayı boş geçtiğinde not parola kodu *True* verecek ve böylece ekrana *"Parola boş bırakılamaz!"* karakter dizisi yazdırılacaktır. Eğer yukarıdaki örneğin mantığını kavramakta zorluk çekiyorsanız şu örnekleri incelemenizi öneririm:

```
>>> parola = ""
>>> bool(parola)

False

>>> bool(not parola)

True

>>> parola = "1243"
>>> bool(parola)

True

>>> bool(not parola)

False
```

Aslında yukarıdaki örneklerde şuna benzer sorular sormuş gibi oluyoruz:

```
>>> parola = ""
>>> bool(parola) #parola boş bırakılmamış, değil mi?

>>> False #Hayır, parola boş bırakılmış.
```

```
>>> bool(not parola) #parola boş bırakılmış, değil mi?

>>> True #Evet, parola boş bırakılmış
```

Kendi kendinize pratik yaparak bu işlecin görevini daha iyi anlayabilirsiniz.

Böylece kısmen çetrefilli bir konu olan bool işleçlerini de geride bırakmış olduk. Sırada değer atama işleçleri var.

Değer Atama İşleçleri

Bu noktaya kadar yaptığımız çalışmalarda sadece tek bir değer atama işleci gördük. Bu işleç = işlecidir. Adından da anlaşılacağı gibi, bu işlecin görevi bir değişkene değer atamaktır. Mesela:

```
>>> a = 23
```

Burada = işleci *a* değişkenine *23* değerini atama işlevi görüyor.

Python'daki tek değer atama işleci elbette = değildir. Bunun dışında başka değer atama işleçleri de bulunur. Tek tek inceleyelim:

+= işleci

Bu işlecin ne işe yaradığını anlamak için şöyle bir örnek düşünün:

```
>>> a = 23
```

a değerine mesela *5* ekleyip bu değeri *28*'e eşitlemek için ne yapmamız lazım? Tabii ki şunu:

```
>>> a = a + 5
>>> print(a)

28
```

Burada yaptığımız şey çok basit: *a* değişkeninin taşıdığı değere *5* ilave ediyoruz ve daha sonra bu değeri tekrar *a* değişkenine atıyoruz. Aynı işlemi çok daha kolay bir şekilde de yapabiliriz:

```
>>> a += 5
>>> print(a)

28
```

Bu kod, yukarıdakiyle tamamen aynı anlama gelir. Ama bir önceki koda göre çok daha verimlidir. Çünkü `a += 5` kodunda Python *a* değişkeninin değerini sadece bir kez kontrol ettiği için, işlemi `a =`

a + 5 koduna göre daha hızlı yapacaktır.

-= işleci

Bir önceki += işleci toplama işlemi yapıp, ortaya çıkan değeri tekrar aynı değişkene atıyordu. -= işleci de buna benzer bir işlem gerçekleştirir:

```
>>> a = 23
>>> a -= 5
>>> print(a)

18
```

Yukarıdaki kullanım şununla tamamen aynıdır:

```
>>> a = 23
>>> a = a - 5
>>> print(a)

18
```

Ancak tıpkı += işlecinde olduğu gibi, -= işleci de alternatifine göre daha hızlı çalışan bir araçtır.

/= işleci

Bu işlecin çalışma mantığı da yukarıdaki işleçlerle aynıdır:

```
>>> a = 30
>>> a /= 3
>>> print(a)

10
```

Yukarıdaki işlem de şununla tamamen aynıdır:

```
>>> a = 30
>>> a = a / 3
>>> print(a)

10
```

***= işleci**

Bu da ötekiler gibi, çarpma işlemi yapıp, bu işlemin sonucunu aynı değişkene atar:

```
>>> a = 20
```

```
>>> a *= 2
>>> print(a)

40
```

Bu işlecin eşdeğeri de şudur:

```
>>> a = 20
>>> a = a * 2
>>> print(a)

40
```

%= işleci

Bu işlecimiz ise bölme işleminden kalan sayıyı aynı değişkene atar:

```
>>> a = 40
>>> a %= 3
>>> print(a)

1
```

Bu işleç de şuna eşdeğerdir:

```
>>> a = 40
>>> a = a % 3
>>> print(a)

1
```

****= işleci**

Bu işlecin ne yaptığını tahmin etmek zor değil. Bu işlecimiz, bir sayının kuvvetini hesapladıktan sonra çıkan değeri aynı değişkene atıyor:

```
>>> a = 12
>>> a **= 2
>>> print(a)

144
```

Eşdeğeri:

```
>>> a = 12
```

```
>>> a = a ** 2
>>> print(a)

144
```

//= işleci

Değer atama işleçlerinin sonuncusu olan //= işlecinin görevi ise taban bölme işleminin sonucunu aynı değişkene atamaktır:

```
>>> a = 5
>>> a //= 2
>>> print(a)

2
```

Eşdeğeri:

```
>>> a = 5
>>> a = a // 2
>>> print(a)

2
```

Bu işleçler arasından, özellikle += ve -= işleçleri işinize bir hayli yarayacak.

Bu arada eğer bu işleçleri kullanırken mesela += mi yoksa =+ mı yazacağınızı karıştırıyorsanız, şöyle düşünebilirsiniz:

```
>>> a = 5
>>> a += 5
>>> print(a)

10
```

Burada, değeri *5* olan bir *a* değişkenine *5* daha ekleyip, çıkan sonucu tekrar *a* değişkenine atadık. Böylece değeri *10* olan bir *a* değişkeni elde ettik. += işlecinin doğru kullanımı yukarıdaki gibidir. Bir de yukarıdaki örneği şöyle yazmayı deneyelim:

```
>>> a = 5
>>> a =+ 5
>>> print(a)

5
```

Burada + işleci ile = işlecinin yerini değiştirdik.

`a =+ 5` satırına dikkatlice bakın. Aslında burada yaptığımız şeyin `a = +5` işlemi olduğunu, yani *a* değişkenine *+5* gibi bir değer verdiğimizi göreceksiniz. Durum şu örnekte daha net görünecektir:

```
>>> a = 5
>>> a =- 5
>>> print(a)
>>> -5
```

Gördüğünüz gibi, `a =- 5` yazdığımızda, aslında yaptığımız şey *a* değişkenine *-5* değerini vermekten ibarettir. Yani `a = -5`.

Aitlik İşleçleri

Aitlik işleçleri, bir karakter dizisi ya da sayının, herhangi bir veri tipi içinde bulunup bulunmadığını sorgulamamızı sağlayan işleçlerdir.

Python'da bir tane aitlik işleci bulunur. Bu işleç de *in* işlecidir. Bu işleci şöyle kullanıyoruz:

```
>>> a = "abcd"
>>> "a" in a

True

>>> "f" in a

False
```

Gördüğünüz gibi, *in* adlı bu işleç, bir öğenin, veri tipi içinde bulunup bulunmadığını sorguluyor. Eğer bahsedilen öğe, veri tipi içinde geçiyorsa *True* çıktısı, eğer geçmiyorsa *False* çıktısı alıyoruz.

Henüz bu *in* işlecini verimli bir şekilde kullanmamızı sağlayacak araçlardan yoksunuz. Ancak birkaç sayfa sonra öğreneceğimiz yeni araçlarla birlikte bu işleci çok daha düzgün ve verimli bir şekilde kullanabilecek duruma geleceğiz.

Kimlik İşleçleri

Python'da her şeyin (ya da başka bir deyişle her nesnenin) bir kimlik numarası (*identity*) vardır. Kabaca söylemek gerekirse, bu kimlik numarası denen şey esasında o nesnenin bellekteki adresini gösterir.

Peki bir nesnenin kimlik numarasına nasıl ulaşırız?

Python'da bu işi yapmamızı sağlayacak id() adlı bir fonksiyon bulunur (İngilizcedeki *identity* (kimlik) kelimesinin kısaltması). Şimdi bir örnek üzerinde bu id() fonksiyonunu nasıl kullanacağımıza bakalım:

```
>>> a = 100
>>> id(a)

137990748
```

Çıktıda gördüğümüz *137990748* sayısı *a* değişkeninin tuttuğu *100* sayısının kimlik numarasını gösteriyor.

Bir de şu örneklere bakalım:

```
>>> a = 50
>>> id(a)

505494576

>>> kardiz = "Elveda Zalim Dünya!"
>>> id(kardiz)

14461728
```

Gördüğünüz gibi, Python'daki her nesnenin kimliği eşşiz, tek ve benzersizdir.

Yukarıda verdiğimiz ilk örnekte bir *a* değişkeni tanımlayıp bunun değerini *100* olarak belirlemiş ve id(a) komutuyla da bu nesnenin kimlik numarasına ulaşmıştık. Yani:

```
>>> a = 100
>>> id(a)
137990748
```

Bir de şu örneğe bakalım:

```
>>> b = 100
>>> id(b)

137990748
```

Gördüğünüz gibi, Python *a* ve *b* değişkenlerinin değeri için aynı kimlik numarasını gösterdi. Bu demek oluyor ki, Python iki adet *100* sayısı için bellekte iki farklı nesne oluşturmuyor. İlk kullanımda önbelleğine aldığı sayıyı, ikinci kez ihtiyaç olduğunda bellekten alıp kullanıyor. Bu tür bir önbellekleme mekanizmasının gerekçesi performansı artırmaktır.

Ama bir de şu örneklere bakalım:

```
>>> a = 1000
>>> id(a)

15163440

>>> b = 1000
>>> id(b)

14447040

>>> id(1000)

15163632
```

Bu defa Python *a* değişkeninin tuttuğu *1000* sayısı, *b* değişkeninin tuttuğu 1000 sayısı ve tek başına yazdığımız *1000* sayısı için farklı kimlik numaraları gösterdi. Bu demek oluyor ki, Python *a* değişkeninin tuttuğu *1000* sayısı için, *b* değişkeninin tuttuğu *1000* sayısı için ve doğrudan girdiğimiz *1000* sayısı için bellekte üç farklı nesne oluşturuyor. Yani bu üç adet *1000* sayısı Python açısından birbirinden farklı...

Yukarıdaki durumu görebileceğimiz başka bir yöntem de Python'daki *is* adlı kimlik işlecini kullanmaktır. Deneyelim:

```
>>> a is 1000

False

>>> b is 1000

False
```

Gördüğünüz gibi, Python *False* (Yanlış) çıktısını suratımıza bir tokat gibi çarptı... Peki bu ne anlama geliyor?

Bu şu anlama geliyor: Demek ki görünüşte aynı olan iki nesne aslında birbirinin aynı olmayabiliyor. Bunun neden bu kadar önemli olduğunu ilerleyen derslerde çok daha iyi anlayacağız.

Yukarıdaki durumun bir başka yansıması daha vardır. Özellikle Python'a yeni başlayıp da bu dilde yer alan *is* işlecini öğrenenler, bu işlecin == işleciyle aynı işleve sahip olduğu yanılgısına kapılabiliyor ve *is* işlecini kullanarak iki nesne arasında karşılaştırma işlemi yapmaya kalkışabiliyor.

Ancak Python'da *is* işlecini kullanarak iki nesne arasında karşılaştırma yapmak güvenli değildir. Yani *is* ve == işleçleri birbirleriyle aynı işlevi görmez. Bu iki işleç nesnelerin farklı yönlerini

sorgular: *is* işleci nesnelerin kimliklerine bakıp o nesnelerin aynı nesneler olup olmadığını kontrol ederken, == işleci nesnelerin içeriğine bakarak o nesnelerin aynı değere sahip olup olmadıklarını sorgular. Bu iki tanım arasındaki ince farka dikkat edin.

Yani:

```
>>> a is 1000

False
```

Ama:

```
>>> a == 1000

True
```

Burada *is* işleci *a* değişkeninin tuttuğu veri ile *1000* sayısının aynı kimlik numarasına sahip olup olmadığını sorgularken, == işleci *a* değişkeninin tuttuğu verinin *1000* olup olmadığını denetliyor. Yani *is* işlecinin yaptığı şey kabaca şu oluyor:

```
>>> id(a) == id(1000)

False
```

Şimdiye kadar denediğimiz örnekler hep sayıydı. Şimdi isterseniz bir de karakter dizilerinin durumuna bakalım:

```
>>> a = "python"
>>> a is "python"

True
```

Burada *True* çıktısını aldık. Bir de == işleci ile bir karşılaştırma yapalım:

```
>>> a == "python"

True
```

Bu da normal olarak *True* çıktısı veriyor. Ama şu örneğe bakarsak:

```
>>> a = "python güçlü ve kolay bir programlama dilidir"
>>> a is "python güçlü ve kolay bir programlama dilidir"

False
```

Ama:

```
>>> a == "python güçlü ve kolay bir programlama dilidir"

True
```

is ve == işleçlerinin nasıl da farklı sonuçlar verdiğini görüyorsunuz. Çünkü bunlardan biri nesnelerin kimliğini sorgularken, öbürü nesnelerin içeriğini sorguluyor. Ayrıca burada dikkatimizi çekmesi gereken başka bir nokta da *"python"* karakter dizisinin önbelleğe alınıp gerektiğinde tekrar tekrar kullanılıyorken, *"python güçlü ve kolay bir programlama dilidir"* karakter dizisinin ise önbelleğe alınmıyor olmasıdır. Aynı karakter dizisinin tekrar kullanılması gerektiğinde Python bunun için bellekte yeni bir nesne daha oluşturuyor.

Peki neden Python, örneğin, *100* sayısını ve *"python"* karakter dizisini önbelleklerken *1000* sayısını ve *"python güçlü ve kolay bir programlama dilidir"* karakter dizisini önbelleğe almıyor. Sebebi şu: Python kendi iç mekanizmasının işleyişi gereğince 'ufak' nesneleri önbelleğe alırken 'büyük' nesneler için her defasında yeni bir depolama işlemi yapıyor. Peki ufak ve büyük kavramlarının ölçütü nedir? İsterseniz Python açısından ufak kavramının sınırının ne olabileceğini şöyle bir kod yardımıyla sorgulayabiliriz:

```
>>> for k in range(-1000, 1000):
...     for v in range(-1000, 1000):
...         if k is v:
...             print(k)
```

Not

Burada henüz öğrenmediğimiz şeyler var. Bunları birkaç bölüm sonra ayrıntılı bir şekilde inceleyeceğiz.

Bu kod *-1000* ve *1000* aralığındaki iki sayı grubunu karşılaştırıp, kimlikleri aynı olan sayıları ekrana döküyor. Yani bir bakıma Python'un hangi sayıya kadar önbellekleme yaptığını gösteriyor. Buna göre *-5* ile *257* arasında kalan sayılar Python tarafından ufak olarak değerlendiriliyor ve önbelleğe alınıyor. Bu aralığın dışında kalan sayılar için ise bellekte her defasında ayrı bir nesne oluşturuluyor.

Burada aldığımız sonuca göre şöyle bir denetleme işlemi yapalım:

```
>>> a = 256
>>> a is 256

True

>>> a = 257
>>> a is 257

False
```

```
>>> a = -5
>>> a is -5

True

>>> a = -6
>>> a is -6

False
```

Böylece Python'daki kimlik işleçlerini de incelemiş olduk. Belki programcılık maceranız boyunca id() fonksiyonunu hiç kullanmayacaksınız, ancak bu fonksiyonun arkasındaki mantığı anlamak, Python'ın kimi yerlerde alttan alta neler çevirdiğini çok daha kolay kavramanızı sağlayacaktır.

Not

http://goo.gl/NWDpb adresindeki tartışmaya bakınız.

Böylece Python'daki bütün işleçleri ayrıntılı bir şekilde incelemiş olduk. Dilerseniz şimdi bu konuyla ilgili birkaç uygulama örneği yapalım.

Uygulama Örnekleri

Basit Bir Hesap Makinesi

Şu ana kadar Python'da pek çok şey öğrendik. Bu öğrendiğimiz şeylerle artık kısmen yararlı bazı programlar yazabiliriz. Elbette henüz yazacağımız programlar pek yetenekli olamayacak olsa da, en azından bize öğrendiklerimizle pratik yapma imkanı sağlayacak. Bu bölümde, if, elif, else yapılarını ve öğrendiğimiz temel aritmetik işleçleri kullanarak çok basit bir hesap makinesi yapmayı deneyeceğiz. Bu arada, bu derste yeni şeyler öğrenerek ufkumuzu ve bilgimizi genişletmeyi de ihmal etmeyeceğiz.

İsterseniz önce kullanıcıya bazı seçenekler sunarak işe başlayalım:

```
giriş = """
(1) topla
(2) çıkar
(3) çarp
(4) böl
(5) karesini hesapla
(6) kare kök hesapla
"""
```

```
print(giriş)
```

Burada kullanıcıya bazı seçenekler sunduk. Bu seçenekleri ekrana yazdırmak için üç tırnak işaretlerinden yararlandığımıza dikkat edin. Birden fazla satıra yayılmış bu tür ifadeleri en kolay üç tırnak işaretleri yardımıyla yazdırabileceğimizi biliyorsunuz artık.

Biz burada bütün seçenekleri tek bir değişken içine yerleştirdik. Esasında her bir seçenek için ayrı bir değişken tanımlamak da mümkündür. Yani aslında yukarıdaki kodları şöyle de yazabiliriz:

```
seçenek1 = "(1) topla"
seçenek2 = "(2) çıkar"
seçenek3 = "(3) çarp"
seçenek4 = "(4) böl"
seçenek5 = "(5) karesini hesapla"
seçenek6 = "(6) karekök hesapla"

print(seçenek1, seçenek2, seçenek3, seçenek4, seçenek5)
```

Yalnız burada dikkat ederseniz, seçenekler hep yan yana diziliyor. Eğer programınızda yukarıdaki şekli kullanmak isterseniz, bu seçeneklerin yan yana değil de, alt alta görünmesini sağlamak için, önceki derslerimizde öğrendiğimiz *sep* parametresini kullanabilirsiniz:

```
seçenek1 = "(1) topla"
seçenek2 = "(2) çıkar"
seçenek3 = "(3) çarp"
seçenek4 = "(4) böl"
seçenek5 = "(5) karesini hesapla"
seçenek6 = "(6) karekök hesapla"

print(seçenek1, seçenek2, seçenek3, seçenek4, seçenek5, seçenek6, sep="\n")
```

Burada *sep* parametresinin değeri olarak *\n* kaçış dizisini belirlediğimize dikkat edin. *\n* kaçış dizisinin ne işe yaradığını hatırlıyorsunuz. Bu dizi, satır başına geçmemizi sağlıyordu. Burada, ayraç olarak satır başı kaçış dizisini belirlediğimiz için her bir seçenek yan yana değil, alt alta görünecektir. Elbette *sep* parametresi için istediğiniz değeri belirleyebilirsiniz. Mesela her bir seçeneği satır başı işaretiyle ayırmak yerine, çift tire gibi bir işaretle ayırmayı da tercih edebilirsiniz:

```
print(seçenek1, seçenek2, seçenek3, seçenek4, seçenek5, sep="--")
```

Programınızda nasıl bir giriş paragrafı belirleyeceğiniz konusunda özgürsünüz. Gelin isterseniz biz birinci şekille yolumuza devam edelim:

```
giriş = """
(1) topla
```

```
(2) çıkar
(3) çarp
(4) böl
(5) karesini hesapla
(6) karekök hesapla
"""

print(giriş)
```

Burada *giriş* adlı bir değişken oluşturduk. Bu değişkenin içinde barındırdığı değeri kullanıcıların görebilmesi için print() fonksiyonu yardımıyla bu değişkeni ekrana yazdırıyoruz. Devam edelim:

```
soru = input("Yapmak istediğiniz işlemin numarasını girin: ")
```

Bu kod yardımıyla kullanıcıya bir soru soruyoruz. Kullanıcıdan yapmasını istediğimiz şey, yukarıda belirlediğimiz giriş seçenekleri içinden bir sayı seçmesi. Kullanıcı *1*, *2*, *3*, *4*, *5* veya *6* seçeneklerinden herhangi birini seçebilir. Kullanıcıyı, seçtiği numaranın karşısında yazan işleme yönlendireceğiz. Yani mesela eğer kullanıcı klavyedeki *1* tuşuna basarsa hesap makinemiz toplama işlemi yapacaktır. *2* tuşu ise kullanıcıyı çıkarma işlemine yönlendirir...

input() fonksiyonunu işlediğimiz bölümde, bu fonksiyonun değer olarak her zaman bir karakter dizisi (*string*) verdiğini söylemiştik. Yukarıdaki kodun çıktısı da doğal olarak bir karakter dizisi olacaktır. Bizim şu aşamada kullanıcıdan karakter dizisi almamızın bir sakıncası yok. Çünkü kullanıcının gireceği *1*, *2*, *3*, *4*, *5* veya *6* değerleriyle herhangi bir aritmetik işlem yapmayacağız. Kullanıcının gireceği bu değerler, yalnızca bize onun hangi işlemi yapmak istediğini belirtecek. Dolayısıyla input() fonksiyonunu yukarıdaki şekilde kullanıyoruz.

İsterseniz şimdiye kadar gördüğümüz kısma topluca bakalım:

```
giriş = """
(1) topla
(2) çıkar
(3) çarp
(4) böl
(5) karesini hesapla
(6) karekök hesapla
"""

print(giriş)

soru = input("Yapmak istediğiniz işlemin numarasını girin: ")
```

Bu kodları çalıştırdığımızda, ekranda giriş paragrafımız görünecek ve kullanıcıya, yapmak istediği işlemin ne olduğu sorulacaktır. Henüz kodlarımız eksik olduğu için, kullanıcı hangi sayıyı girerse girsin, programımız hiç bir iş yapmadan kapanacaktır. O halde yolumuza devam edelim:

```
if soru == "1":
```

Böylece ilk if deyimimizi tanımlamış olduk. Buradaki yazım şekline çok dikkat edin. Bu kodlarla Python'a şu emri vermiş oluyoruz:

Eğer *soru* adlı değişkenin değeri *1* ise, yani eğer kullanıcı klavyede *1* tuşuna basarsa...

if deyimlerinin en sonuna *:* işaretini koymayı unutmuyoruz. Python'a yeni başlayanların en çok yaptığı hatalardan birisi, sondaki bu *:* işaretini koymayı unutmalarıdır. Bu işaret bize çok ufak bir ayrıntıymış gibi görünse de Python için manevi değeri çok büyüktür! Python'un bize öfkeli mesajlar göstermesini istemiyorsak bu işareti koymayı unutmayacağız. Bu arada, burada == işaretini kullandığımıza da dikkat edin. Bunun ne anlama geldiğini önceki derslerimizde öğrenmiştik. Bu işaret, iki şeyin aynı değere sahip olup olmadığını sorgulamamızı sağlıyor. Biz burada *soru* adlı değişkenin değerinin *1* olup olmadığını sorguladık. *soru* değişkeninin değeri kullanıcı tarafından belirleneceği için henüz bu değişkenin değerinin ne olduğunu bilmiyoruz. Bizim programımızda kullanıcı klavyeden *1*, *2*, *3*, *4*, *5* veya *6* değerlerinden herhangi birini seçebilir. Biz yukarıdaki kod yardımıyla, eğer kullanıcı klavyede *1* tuşuna basarsa ne yapılacağını belirleyeceğiz. O halde devam edelim:

```
if soru == "1":
    sayı1 = int(input("Toplama işlemi için ilk sayıyı girin: "))
    sayı2 = int(input("Toplama işlemi için ikinci sayıyı girin: "))
    print(sayı1, "+", sayı2, "=", sayı1 + sayı2)
```

Böylece ilk if bloğumuzu tanımlamış olduk.

if deyimimizi yazdıktan sonra ne yaptığımız çok önemli. Buradaki girintileri, programımız güzel görünsün diye yapmıyoruz. Bu girintilerin Python için bir anlamı var. Eğer bu girintileri vermezsek programımız çalışmayacaktır. Eğer Python kodlarına duyarlı bir metin düzenleyici kullanıyorsanız, *:* işaretini koyup *Enter* tuşuna bastıktan sonra otomatik olarak girinti verilecektir. Eğer kullandığınız metin düzenleyici size böyle bir kolaylık sunmuyorsa *Enter* tuşuna bastıktan sonra klavyedeki boşluk (*SPACE*) tuşunu kullanarak dört vuruşluk bir girinti oluşturabilirsiniz. Bu girintiler, ilk satırda belirlediğimiz if deyimiyle gösterilecek işlemlere işaret ediyor. Dolayısıyla burada yazılan kodları Pythoncadan Türkçeye çevirecek olursak şöyle bir şey elde ederiz:

```
eğer sorunun değeri '1' ise:
    Toplama işlemi için ilk sayı girilsin. Bu değere 'sayı1' diyelim.
    Sonra ikinci sayı girilsin. Bu değere de 'sayı2' diyelim.
    En son, 'sayı1', '+' işleci, '=' işleci, 'sayı2' ve 'sayı1 + sayı2'
    ekrana yazdırılsın...
```

Gelin isterseniz buraya kadar olan bölümü yine topluca görelim:

```
giriş = """
(1) topla
(2) çıkar
```

```
(3) çarp
(4) böl
(5) karesini hesapla
(6) karekök hesapla
"""

print(giriş)

soru = input("Yapmak istediğiniz işlemin numarasını girin: ")

if soru == "1":
    sayı1 = int(input("Toplama işlemi için ilk sayıyı girin: "))
    sayı2 = int(input("Toplama işlemi için ikinci sayıyı girin: "))
    print(sayı1, "+", sayı2, "=", sayı1 + sayı2)
```

Bu kodları çalıştırıp, klavyede *1* tuşuna bastığımızda, bizden bir sayı girmemiz istenecektir. İlk sayımızı girdikten sonra bize tekrar bir sayı girmemiz söylenecek. Bu emre de uyup *Enter* tuşuna basınca, girdiğimiz bu iki sayının toplandığını göreceğiz. Fena sayılmaz, değil mi?

Şimdi programımızın geri kalan kısmını yazıyoruz. İşin temelini kavradığımıza göre birden fazla kod bloğunu aynı anda yazabiliriz:

```
elif soru == "2":
    sayı3 = int(input("Çıkarma işlemi için ilk sayıyı girin: "))
    sayı4 = int(input("Çıkarma işlemi için ikinci sayıyı girin: "))
    print(sayı3, "-", sayı4, "=", sayı3 - sayı4)

elif soru == "3":
    sayı5 = int(input("Çarpma işlemi için ilk sayıyı girin: "))
    sayı6 = int(input("Çarpma işlemi için ikinci sayıyı girin: "))
    print(sayı5, "x", sayı6, "=", sayı5 * sayı6)

elif soru == "4":
    sayı7 = int(input("Bölme işlemi için ilk sayıyı girin: "))
    sayı8 = int(input("Bölme işlemi için ikinci sayıyı girin: "))
    print(sayı7, "/", sayı8, "=", sayı7 / sayı8)

elif soru == "5":
    sayı9 = int(input("Karesini hesaplamak istediğiniz sayıyı girin: "))
    print(sayı9, "sayısının karesi =", sayı9 ** 2)

elif soru == "6":
    sayı10 = int(input("Karekökünü hesaplamak istediğiniz sayıyı girin: "))
    print(sayı10, "sayısının karekökü = ", sayı10 ** 0.5)
```

Bunlarla birlikte kodlarımızın büyük bölümünü tamamlamış oluyoruz. Bu bölümdeki tek fark, ilk if bloğunun aksine, burada elif bloklarını kullanmış olmamız. Eğer burada bütün blokları if kullanarak yazarsanız, biraz sonra kullanacağımız else bloğu her koşulda çalışacağı için beklentinizin dışında sonuçlar elde edersiniz.

Yukarıdaki kodlarda az da olsa farklılık gösteren tek yer son iki elif bloğumuz. Esasında buradaki fark da pek büyük bir fark sayılmaz. Neticede tek bir sayının karesini ve karekökünü hesaplayacağımız için, kullanıcıdan yalnızca tek bir giriş istiyoruz.

Şimdi de son bloğumuzu yazalım. Az evvel çıtlattığımız gibi, bu son blok bir else bloğu olacak:

```
else:
    print("Yanlış giriş.")
    print("Aşağıdaki seçeneklerden birini giriniz:", giriş)
```

Çok basit bir else bloğu ile işimizi bitirdik. Bu bloğun ne işe yaradığını biliyorsunuz:

> Eğer kullanıcının girdiği değer yukarıdaki bloklardan hiç birine uymuyorsa bu *else* bloğunu işlet!

gibi bir emir vermiş oluyoruz bu else bloğu yardımıyla. Mesela kullanıcımız *1*, *2*, *3*, *4*, *5* veya *6* seçeneklerini girmek yerine *7* yazarsa, bu blok işletilecek.

Gelin isterseniz son kez kodlarımızı topluca bir görelim:

```
giriş = """
(1) topla
(2) çıkar
(3) çarp
(4) böl
(5) karesini hesapla
(6) karekök hesapla
"""

print(giriş)

soru = input("Yapmak istediğiniz işlemin numarasını girin: ")

if soru == "1":
    sayı1 = int(input("Toplama işlemi için ilk sayıyı girin: "))
    sayı2 = int(input("Toplama işlemi için ikinci sayıyı girin: "))
    print(sayı1, "+", sayı2, "=", sayı1 + sayı2)

elif soru == "2":
    sayı3 = int(input("Çıkarma işlemi için ilk sayıyı girin: "))
```

```
    sayı4 = int(input("Çıkarma işlemi için ikinci sayıyı girin: "))
    print(sayı3, "-", sayı4, "=", sayı3 - sayı4)

elif soru == "3":
    sayı5 = int(input("Çarpma işlemi için ilk sayıyı girin: "))
    sayı6 = int(input("Çarpma işlemi için ikinci sayıyı girin: "))
    print(sayı5, "x", sayı6, "=", sayı5 * sayı6)

elif soru == "4":
    sayı7 = int(input("Bölme işlemi için ilk sayıyı girin: "))
    sayı8 = int(input("Bölme işlemi için ikinci sayıyı girin: "))
    print(sayı7, "/", sayı8, "=", sayı7 / sayı8)

elif soru == "5":
    sayı9 = int(input("Karesini hesaplamak istediğiniz sayıyı girin: "))
    print(sayı9, "sayısının karesi =", sayı9 ** 2)

elif soru == "6":
    sayı10 = int(input("Karekökünü hesaplamak istediğiniz sayıyı girin: "))
    print(sayı10, "sayısının karekökü = ", sayı10 ** 0.5)

else:
    print("Yanlış giriş.")
    print("Aşağıdaki seçeneklerden birini giriniz:", giriş)
```

Genel olarak baktığımızda, bütün programın aslında basit bir ‘if, elif, else’ yapısından ibaret olduğunu görüyoruz. Ayrıca bu kodlardaki simetriye de dikkatinizi çekmek isterim. Gördüğünüz gibi her ‘paragraf’ bir if, elif veya else bloğundan oluşuyor ve her blok kendi içinde girintili bir yapı sergiliyor. Temel olarak şöyle bir şeyle karşı karşıyayız:

```
Eğer böyle bir durum varsa:
    şöyle bir işlem yap

Yok eğer şöyle bir durum varsa:
    böyle bir işlem yap

Eğer bambaşka bir durum varsa:
    şöyle bir şey yap
```

Böylelikle şirin bir hesap makinesine sahip olmuş olduk! Hesap makinemiz pek yetenekli değil, ama olsun... Henüz bildiklerimiz bunu yapmamıza müsaade ediyor. Yine de başlangıçtan bu noktaya kadar epey yol katettiğimizi görüyorsunuz.

Şimdi bu programı çalıştırın ve neler yapabildiğine göz atın. Bu arada kodları da iyice inceleyin.

Programı yeterince anladıktan sonra, program üzerinde kendinize göre bazı değişiklikler yapın, yeni özellikler ekleyin. Eksikliklerini, zayıf yönlerini bulmaya çalışın. Böylece bu dersten azami faydayı sağlamış olacaksınız.

Sürüme Göre İşlem Yapan Program

Bildiğiniz gibi, şu anda piyasada iki farklı Python serisi bulunuyor: Python2 ve Python3. Daha önce de söylediğimiz gibi, Python'ın 2.x serisi ile çalışan bir program Python'ın 3.x serisi ile muhtemelen çalışmayacaktır. Aynı şekilde bunun tersi de geçerlidir. Yani 3.x ile çalışan bir program 2.x ile büyük ihtimalle çalışmayacaktır.

Bu durum, yazdığınız programların farklı Python sürümleri ile çalıştırılma ihtimaline karşı bazı önlemler almanızı gerektirebilir. Örneğin yazdığınız bir programda kullanıcılarınızdan beklentiniz, programınızı Python'ın 3.x sürümlerinden biri ile çalıştırmaları olabilir. Eğer programınız Python'ın 2.x sürümlerinden biri ile çalıştırılırsa kullanıcıya bir uyarı mesajı göstermek isteyebilirsiniz.

Hatta yazdığınız bir program, aynı serinin farklı sürümlerinde dahi çalışmayı engelleyecek özellikler içeriyor olabilir. Örneğin print() fonksiyonunun *flush* adlı parametresi dile *3.3* sürümü ile birlikte eklendi. Dolayısıyla bu parametreyi kullanan bir program, kullanıcının *3.3* veya daha yüksek bir Python sürümü kullanmasını gerektirir. Böyle bir durumda, programınızı çalıştıran Python sürümünün en düşük *3.3* olmasını temin etmeniz gerekir.

Peki bunu nasıl yapacaksınız?

Burada aklınızda ilk olarak, kodlarınıza *#!/usr/bin/env python3.3* veya *#! python3.3* gibi bir satır eklemek gelmiş olabilir. Ama unutmayın, bu çözüm ancak kısıtlı bir işlevsellik sunabilir. Programımıza böyle bir satır eklediğimizde, programımızın Python'ın *3.3* sürümü ile çalıştırılması gerektiğini belirtiyoruz. Ama *3.3* dışı bir sürümle çalıştırıldığında ne olacağını belirtmiyoruz. Böyle bir durumda, eğer programımız *3.3* dışı bir sürümle çalıştırılırsa çökecektir. Bizim burada daha kapsamlı ve esnek bir çözüm bulmamız gerekiyor.

Hatırlarsanız önceki derslerden birinde *sys* adlı bir modülden söz etmiştik. Bildiğiniz gibi, bu modül içinde pek çok yararlı değişken ve fonksiyon bulunuyor. Önceki derslerimizde, bu modül içinde bulunan exit() fonksiyonu ile *stdout* ve *version* değişkenlerini gördüğümüzü hatırlıyor olmalısınız. *sys* modülü içinde bulunan exit() fonksiyonunun programdan çıkmamızı sağladığını, *stdout* değişkeninin standart çıktı konumu bilgisini tuttuğunu ve *version* değişkeninin de kullandığımız Python sürümü hakkında bilgi verdiğini biliyoruz. İşte yukarıda bahsettiğimiz programda da bu *sys* modülünden yararlanacağız.

Bu iş için, *version* değişkenine çok benzeyen *version_info* adlı bir değişkeni kullanacağız.

Bu değişkenin nasıl kullanıldığına etkileşimli kabukta beraberce bakalım...

sys modülü içindeki araçları kullanabilmek için öncelikle bu modülü içe aktarmamız gerektiğini biliyorsunuz:

```
>>> import sys
```

Şimdi de bu modül içindeki *version_info* adlı değişkene erişelim:

```
>>> sys.version_info
```

Bu komut bize şöyle bir çıktı verir:

```
sys.version_info(major=3, minor=3, micro=2, releaselevel='final', serial=0)
```

Gördüğünüz gibi, bu değişken de bize tıpkı *version* adlı değişken gibi, kullandığımız Python sürümü hakkında bilgi veriyor.

Ben yukarıdaki komutu Python3'te verdiğinizi varsaydım. Eğer yukarıdaki komutu Python3 yerine Python2'de verseydik şöyle bir çıktı alacaktık:

```
sys.version_info(major=2, minor=7, micro=5, releaselevel='final', serial=0)
```

version_info ve *version* değişkenlerinin verdikleri çıktının birbirlerinden farklı yapıda olduğuna dikkat edin. *version* değişkeni, *version_info* değişkeninden farklı olarak şöyle bir çıktı verir:

```
'3.3.2 (default, May 20 2013, 16:10:12) n[GCC 4.4.7 20120313 (Red Hat 4.4.7-3)]'
```

version_info değişkeninin verdiği çıktı bizim şu anda yazmak istediğimiz programa daha uygun. Bunun neden böyle olduğunu biraz sonra siz de anlayacaksınız.

Gördüğünüz gibi, *version_info* değişkeninin çıktısında *major* ve *minor* gibi bazı değerler var. Çıktıdan da rahatlıkla anlayabileceğiniz gibi, *major*, kullanılan Python serisinin ana sürüm numarasını; *minor* ise alt sürüm numarasını verir. Çıktıda bir de *micro* adlı bir değer var. Bu da kullanılan Python serisinin en alt sürüm numarasını verir.

Bu değere şu şekilde erişiyoruz:

```
>>> sys.version_info.major
```

Öteki değerlere de aynı şekilde ulaşıyoruz:

```
>>> sys.version_info.minor
>>> sys.version_info.micro
```

İşte bu çıktılardaki *major* (ve yerine göre bununla birlikte *minor* ve *micro*) değerini kullanarak, programımızın hangi Python sürümü ile çalıştırılması gerektiğini kontrol edebiliriz. Şimdi programımızı yazalım:

```
import sys

_2x_metni = """
```

```
Python'ın 2.x sürümlerinden birini kullanıyorsunuz.
Programı çalıştırabilmek için sisteminizde Python'ın
3.x sürümlerinden biri kurulu olmalı."""

_3x_metni = "Programa hoşgeldiniz."

if sys.version_info.major < 3:
    print(_2x_metni)
else:
    print(_3x_metni)
```

Gelin isterseniz öncelikle bu kodları biraz inceleyelim.

İlk olarak modülümüzü içe aktarıyoruz. Bu modül içindeki araçları kullanabilmemiz için bunu yapmamız şart:

```
import sys
```

Ardından Python'ın 2.x sürümlerinden herhangi birini kullananlar için bir uyarı metni oluşturuyoruz:

```
_2x_metni = """
Python'ın 2.x sürümlerinden birini kullanıyorsunuz.
Programı çalıştırabilmek için sisteminizde Python'ın
3.x sürümlerinden biri kurulu olmalı."""
```

Bildiğiniz gibi Python'da değişken adları bir sayıyla başlamaz. O yüzden değişken isminin başına bir tane alt çizgi işareti koyduğumuza dikkat edin.

Bu da Python3 kullanıcıları için:

```
_3x_metni = "Programa hoşgeldiniz."
```

Artık sürüm kontrolü kısmına geçebiliriz. Eğer major parametresinin değeri 3'ten küçükse _2x_metnini yazdırıyoruz. Bunun dışındaki bütün durumlar için ise _3x_metnini basıyoruz:

```
if sys.version_info.major < 3:
    print(_2x_metni)
else:
    print(_3x_metni)
```

Gördüğünüz gibi, kullanılan Python sürümünü kontrol etmek ve eğer program istenmeyen bir Python sürümüyle çalıştırılıyorsa ne yapılacağını belirlemek son derece kolay.

Yukarıdaki çok basit bir kod parçası olsa da bize Python programlama diline ve bu dilin farklı sürümlerine dair son derece önemli bazı bilgiler veriyor.

Eğer bu programı Python'ın 3.x sürümlerinden biri ile çalıştırdıysanız şu çıktıyı alacaksınız:

```
Programa hoşgeldiniz.
```

Ama eğer bu programı Python'ın 2.x sürümlerinden biri ile çalıştırdıysanız, beklentinizin aksine, şöyle bir hata mesajı alacaksınız:

```
  File "test.py", line 5
SyntaxError: Non-ASCII character '\xc4' in file test.py on line 6, but no
encoding declared; see http://www.python.org/peps/pep-0263.html for details
```

Biz *_2x_metni* adlı değişkenin ekrana basılmasını beklerken Python bize bir hata mesajı gösterdi. Aslında siz bu hata mesajına hiç yabancı değilsiniz. Bunu daha önce de görmüştünüz. Hatırlarsanız önceki derslerimizde karakter kodlamalarından bahsederken, Python'ın 2.x sürümlerinde öntanımlı karakter kodlamasının ASCII olduğundan söz etmiştik. Bu yüzden programlarımızda Türkçe karakterleri kullanırken bazı ilave işlemler yapmamız gerekiyordu.

Burada ilk olarak karakter kodlamasını *UTF-8* olarak değiştirmemiz gerekiyor. Bunun nasıl yapılacağını biliyorsunuz. Programımızın ilk satırına şu kodu ekliyoruz:

```
# -*- coding: utf-8 -*-
```

Bu satır Python3 için gerekli değil. Çünkü Python3'te öntanımlı karakter kodlaması zaten *UTF-8*. Ama Python2'de öntanımlı karakter kodlaması *ASCII*. O yüzden Python2 kullanıcılarını da düşünerek *UTF-8* kodlamasını açıkça belirtiyoruz. Böylece programımızın Python'ın 2.x sürümlerinde Türkçe karakterler yüzünden çökmesini önlüyoruz.

Ama burada bir problem daha var. Programımız Türkçe karakterler yüzünden çökmüyor çökmemesine ama, bu defa da Türkçe karakterleri düzgün göstermiyor:

```
Python'Ä±n 2.x sÃ¼rÃ¼mlerinden birini kullanÄ±yorsunuz.
ProgramÄ± Ã§alÄ±ÅŸtÄ±rabilmek iÃ§in sisteminizde Python'Ä±n
3.x sÃ¼rÃ¼mlerinden biri kurulu olmalÄ±.
```

Programımızı Python'ın 2.x sürümlerinden biri ile çalıştıranların uyarı mesajını düzgün bir şekilde görüntüleyebilmesini istiyorsanız, Türkçe karakterler içeren karakter dizilerinin en başına bir 'u' harfi eklemelisiniz. Yani *_2x_metni* adlı değişkeni şöyle yazmalısınız:

```
_2x_metni = u"""
Python'ın 2.x sürümlerinden birini kullanıyorsunuz.
Programı çalıştırabilmek için sisteminizde Python'ın
3.x sürümlerinden biri kurulu olmalı."""
```

Bu karakter dizisinin en başına bir 'u' harfi ekleyerek bu karakter dizisini 'unicode' olarak tanımlamış olduk. Eğer 'unicode' kavramını bilmiyorsanız endişe etmeyin. İlerde bu kavramdan bolca söz edeceğiz. Biz şimdilik, içinde Türkçe karakterler geçen karakter dizilerinin Python2 kullanıcıları

tarafından düzgün görüntülenebilmesi için başlarına bir ‘u’ harfi eklenmesi gerektiğini bilelim yeter.

Eğer siz bir Windows kullanıcısıysanız ve bütün bu işlemlerden sonra bile Türkçe karakterleri düzgün görüntüleyemiyorsanız, bu durum muhtemelen MS-DOS komut satırının kullandığı yazı tipinin Türkçe karakterleri gösterememesinden kaynaklanıyordur. Bu problemi çözmek için MS-DOS komut satırının başlık çubuğuna sağ tıklayıp ‘özellikler’ seçeneğini seçerek yazı tipini ‘Lucida Console’ olarak değiştirin. Bu işlemin ardından da komut satırında şu komutu verin:

```
chcp 1254
```

Böylece Türkçe karakterleri düzgün görüntüleyebilirsiniz.

Not

MS-DOS’taki Türkçe karakter problemi hakkında daha ayrıntılı bilgi için http://goo.gl/eRY1P adresindeki makalemizi inceleyebilirsiniz.

Şimdiye kadar anlattıklarımızdan öğrendiğiniz gibi, *sys* modülü içinde sürüm denetlemeye yarayan iki farklı değişken var. Bunlardan biri *version*, öbürü ise *version_info*.

Python3’te bu değişkenlerin şu çıktıları verdiğiniz biliyoruz:

version:

```
'3.3.2 (default, May 20 2013, 16:10:12) n[GCC 4.4.7 20120313 (Red Hat 4.4.7-3)]'
```

version_info:

```
sys.version_info(major=3, minor=3, micro=2, releaselevel='final', serial=0)
```

Gördüğünüz gibi, çıktıların hem yapıları birbirinden farklı, hem de verdikleri bilgiler arasında bazı farklar da var. Mesela *version* değişkeni, kullandığımız Python sürümünün hangi tarih ve saatte, hangi işletim sistemi üzerinde derlendiği bilgisini de veriyor. Ancak kullanılan Python sürümünün ne olduğunu tespit etmek konusunda *version_info* biraz daha pratik görünüyor. Bu değişkenin bize *major*, *minor* ve *micro* gibi parametreler aracılığıyla sunduğu sayı değerli verileri işleçlerle birlikte kullanarak bu sayılar üzerinde aritmetik işlemler yapıp, kullanılan Python sürümünü kontrol edebiliyoruz.

version değişkeni bize bir karakter dizisi verdiği için, bu değişkenin değerini kullanarak herhangi bir aritmetik işlem yapamıyoruz. Mesela *version_info* değişkeniyle yukarıda yaptığımız büyüktür-küçüktür sorgulamasını *version* değişkeniyle tabii ki yapamayız.

Yukarıdaki örnekte seriler arası sürüm kontrolünü nasıl yapacağımızı gördük. Bunun için kullandığımız kod şuydu:

```
if sys.version_info.major < 3:
    ...
```

Burada kullanılan Python serisinin *3.x*'ten düşük olduğu durumları sorguladık. Peki aynı serinin farklı sürümlerini denetlemek istersek ne yapacağız? Mesela Python'ın *3.2* sürümünü sorgulamak istersek nasıl bir kod kullanacağız?

Bunun için şöyle bir şey yazabiliriz:

```
if sys.version_info.major == 3 and sys.version_info.minor == 2:
    ...
```

Gördüğünüz gibi burada *version_info* değişkeninin hem *major* hem de *minor* parametrelerini kullandık. Ayrıca hem ana sürüm, hem de alt sürüm için belli bir koşul talep ettiğimizden ötürü *and* adlı Bool işlecinden de yararlandık. Çünkü koşulun gerçekleşmesi, ana sürümün *3* **ve** alt sürümün *2* olmasına bağlı.

Yukarıdaki işlem için *version* değişkenini de kullanabilirdik. Dikkatlice bakın:

```
if "3.2" in sys.version:
    ...
```

Bildiğiniz gibi, *version* değişkeni Python'ın *3.x* sürümlerinde şuna benzer bir çıktı veriyor:

'3.3.2 (default, May 20 2013, 16:10:12) n[GCC 4.4.7 20120313 (Red Hat 4.4.7-3)]'

İşte biz burada *in* işlecini kullanarak, *version* değişkeninin verdiği karakter dizisi içinde '3.2' diye bir ifade aradık.

Bu konuyu daha iyi anlamak için kendi kendinize bazı denemeler yapmanızı tavsiye ederim. Ne kadar çok örnek kod yazarsanız, o kadar çok tecrübe kazanırsınız.

başa dön

Döngüler (Loops)

Şimdiye kadar öğrendiklerimiz sayesinde Python'la ufak tefek programlar yazabilecek düzeye geldik. Mesela öğrendiğimiz bilgiler yardımıyla bir önceki bölümde çok basit bir hesap makinesi yazabilmiştik. Yalnız o hesap makinesinde farkettiyseniz çok önemli bir eksiklik vardı. Hesap makinemizle hesap yaptıktan sonra programımız kapanıyor, yeni hesap yapabilmek için programı yeniden başlatmamız gerekiyordu.

Hesap makinesi programındaki sorun, örneğin, aşağıdaki program için de geçerlidir:

```
tuttuğum_sayı = 23

bilbakalım = int(input("Aklımdan bir sayı tuttum. Bil bakalım kaç tuttum? "))

if bilbakalım == tuttuğum_sayı:
    print("Tebrikler! Bildiniz...")

else:
    print("Ne yazık ki tuttuğum sayı bu değildi...")
```

Burada *tuttuğum_sayı* adlı bir değişken belirledik. Bu değişkenin değeri *23*. Kullanıcıdan tuttuğumuz sayıyı tahmin etmesini istiyoruz. Eğer kullanıcının verdiği cevap *tuttuğum_sayı* değişkeninin değeriyle aynıysa (yani *23* ise), ekrana 'Tebrikler!...' yazısı dökülecektir. Aksi halde 'Ne yazık ki...' cümlesi ekrana dökülecektir.

Bu program iyi, hoş, ama çok önemli bir eksiği var. Bu programı yalnızca bir kez kullanabiliyoruz. Yani kullanıcı yalnızca bir kez tahminde bulunabiliyor. Eğer kullanıcı bir kez daha tahminde bulunmak isterse programı yeniden çalıştırması gerekecek. Bunun hiç iyi bir yöntem olmadığı ortada. Halbuki yazdığımız bir program, ilk çalışmanın ardından kapanmasa, biz bu programı tekrar tekrar çalıştırabilsek, programımız sürekli olarak başa dönse ve program ancak biz istediğimizde kapansa ne iyi olurdu değil mi? Yani mesela yukarıdaki örnekte kullanıcı bir sayı tahmin ettikten sonra, eğer bu sayı bizim tuttuğumuz sayıyla aynı değilse, kullanıcıya tekrar tahmin etme fırsatı verebilsek çok hoş olurdu...

Yukarıda açıklamaya çalıştığımız süreç, yani bir sürecin tekrar tekrar devam etmesi Python'da 'döngü' (*loop*) olarak adlandırılır.

İşte bu bölümde, programlarımızın sürekli olarak çalışmasını nasıl sağlayabileceğimizi, yani programlarımızı bir döngü içine nasıl sokabileceğimizi öğreneceğiz.

Python'da programlarımızı tekrar tekrar çalıştırabilmek için döngü adı verilen bazı ifadelerden yararlanacağız.

Python'da iki tane döngü bulunur: while ve for

Dilerseniz işe while döngüsü ile başlayalım.

while Döngüsü

İngilizce bir kelime olan *while*, Türkçede '... iken, ... olduğu sürece' gibi anlamlara gelir. Python'da while bir döngüdür. Bir önceki bölümde söylediğimiz gibi, döngüler sayesinde programlarımızın sürekli olarak çalışmasını sağlayabiliriz.

Bu bölümde Python'da while döngüsünün ne olduğunu ve ne işe yaradığını anlamaya çalışacağız. Öncelikle while döngüsünün temellerini kavrayarak işe başlayalım.

Basit bir while döngüsü kabaca şuna benzer:

```
a = 1

while a == 1:
```

Burada *a* adlı bir değişken oluşturduk. Bu değişkenin değeri *1*. Bir sonraki satırda ise while a == 1: gibi bir ifade yazdık. En başta da söylediğimiz gibi *while* kelimesi, '... iken, olduğu sürece' gibi anlamlar taşıyor. Python programlama dilindeki anlamı da buna oldukça yakındır. Burada while a == 1 ifadesi programımıza şöyle bir anlam katıyor:

> *a* değişkeninin değeri *1* olduğu sürece...

Gördüğünüz gibi cümlemiz henüz eksik. Yani belli ki bunun bir de devamı olacak. Ayrıca while ifadesinin sonundaki *:* işaretinden anladığımız gibi, bundan sonra gelecek satır girintili yazılacak. Devam edelim:

```
a = 1

while a == 1:
    print("bilgisayar çıldırdı!")
```

Burada Python'a şu emri vermiş olduk:

> *a* değişkeninin değeri *1* olduğu sürece, ekrana 'bilgisayar çıldırdı!' yazısını dök!

Bu programı çalıştırdığımızda Python verdiğimiz emre sadakatle uyacak ve *a* değişkeninin değeri *1* olduğu müddetçe de bilgisayarımızın ekranına 'bilgisayar çıldırdı!' yazısını dökecektir. Programımızın içinde *a* değişkeninin değeri *1* olduğu ve bu değişkenin değerini değiştirecek herhangi bir şey bulunmadığı için Python hiç sıkılmadan ekrana 'bilgisayar çıldırdı!' yazısını basmaya devam edecektir. Eğer siz durdurmazsanız bu durum sonsuza kadar devam edebilir. Bu çılgınlığa bir son vermek için klavyenizde *Ctrl+C* veya *Ctrl+Z* tuşlarına basarak programı durmaya zorlayabilirsiniz.

Burada programımızı sonsuz bir döngüye sokmuş olduk (*infinite loop*). Esasında sonsuz döngüler genellikle bir program hatasına işaret eder. Yani çoğu durumda programcının arzu ettiği şey bu değildir. O yüzden doğru yaklaşım, döngüye soktuğumuz programlarımızı durduracak bir ölçüt belirlemektir. Yani öyle bir kod yazmalıyız ki, *a* değişkeninin *1* olan değeri bir noktadan sonra artık *1* olmasın ve böylece o noktaya ulaşıldığında programımız dursun. Kullanıcının *Ctrl+C* tuşlarına basarak programı durdurmak zorunda kalması pek hoş olmuyor. Gelin isterseniz bu soyut ifadeleri biraz somutlaştıralım.

Öncelikle şu satırı yazarak işe başlıyoruz:

```
a = 1
```

Burada normal bir şekilde *a* değişkenine *1* değerini atadık. Şimdi devam ediyoruz:

```
a = 1

while a < 10:
```

while ile verdiğimiz ilk örnekte while a == 1 gibi bir ifade kullanmıştık. Bu ifade;

a'nın değeri *1* olduğu müddetçe...

gibi bir anlama geliyordu.

while a < 10 ifadesi ise;

a'nın değeri *10*'dan küçük olduğu müddetçe...

anlamına gelir. İşte burada programımızın sonsuz döngüye girmesini engelleyecek bir ölçüt koymuş olduk. Buna göre, *a* değişkeninin şimdiki değeri *1*'dir. Biz, *a*'nın değeri *10*'dan küçük olduğu müddetçe bir işlem yapacağız. Devam edelim:

```
a = 1

while a < 10:
    print("bilgisayar yine çıldırdı!")
```

Ne oldu? İstediğimizi elde edemedik, değil mi? Programımız yine sonsuz döngüye girdi. Bu sonsuz döngüyü kırmak için *Ctrl+C* (veya *Ctrl+Z*)'ye basmamız gerekecek yine...

Sizce buradaki hata nereden kaynaklandı? Yani neyi eksik yaptık da programımız sonsuz döngüye girmekten kurtulamadı? Aslında bunun cevabı çok basit. Biz yukarıdaki kodları yazarak Python'a şu emri vermiş olduk:

a'nın değeri *10*'dan küçük olduğu müddetçe ekrana 'bilgisayar yine çıldırdı!' yazısını bas!

a değişkeninin değeri *1*. Yani *10*'dan küçük. Dolayısıyla Python'ın ekrana o çıktıyı basmasını engelleyecek herhangi bir şey yok...

Şimdi bu problemi nasıl aşacağımızı görelim:

```
a = 1

while a < 10:
    a += 1
    print("bilgisayar yine çıldırdı!")
```

Burada `a += 1` satırını ekledik kodlarımızın arasına. += işlecini anlatırken söylediğimiz gibi, bu satır, *a* değişkeninin değerine her defasında *1* ekliyor ve elde edilen sonucu tekrar *a* değişkenine atıyor. En sonunda *a*'nın değeri *10*'a ulaşınca da, Python ekrana 'bilgisayar yine çıldırdı!' cümlesini yazmayı bırakıyor. Çünkü `while` döngüsü içinde belirttiğimiz ölçüte göre, programımızın devam edebilmesi için *a* değişkeninin değerinin *10*'dan küçük olması gerekiyor. *a*'nın değeri *10*'a ulaştığı anda bu ölçüt bozulacaktır. Gelin isterseniz bu kodları Python'ın nasıl algıladığına bir bakalım:

1. Python öncelikle `a = 1` satırını görüyor ve *a*'nın değerini *1* yapıyor.
2. Daha sonra *a*'nın değeri *10*'dan küçük olduğu müddetçe... (`while a < 10`) satırını görüyor.
3. Ardından *a*'nın değerini, *1* artırıyor (`a += 1`) ve *a*'nın değeri *2* oluyor.
4. *a*'nın değeri (yani *2*) *10*'dan küçük olduğu için Python ekrana ilgili çıktıyı veriyor.
5. İlk döngüyü bitiren Python başa dönüyor ve *a*'nın değerinin *2* olduğunu görüyor.
6. *a*'nın değerini yine *1* artırıyor ve *a*'yı *3* yapıyor.
7. *a*'nın değeri hâlâ *10*'dan küçük olduğu için ekrana yine ilgili çıktıyı veriyor.
8. İkinci döngüyü de bitiren Python yine başa dönüyor ve *a*'nın değerinin *3* olduğunu görüyor.
9. Yukarıdaki adımları tekrar eden Python, *a*'nın değeri *9* olana kadar ilerlemeye devam ediyor.
10. *a*'nın değeri *9*'a ulaştığında Python *a*'nın değerini bir kez daha artırınca bu değer *10*'a ulaşıyor.
11. Python *a*'nın değerinin artık *10*'dan küçük olmadığını görüyor ve programdan çıkıyor.

Yukarıdaki kodları şöyle yazarsak belki durum daha anlaşılır olabilir:

```
a = 1

while a < 10:
    a += 1
    print(a)
```

Burada Python'un arkada ne işler çevirdiğini daha net görebiliyoruz. Kodlarımız içine eklediğimiz `while` döngüsü sayesinde Python her defasında *a* değişkeninin değerini kontrol ediyor ve bu değer *10*'dan küçük olduğu müddetçe *a* değişkeninin değerini *1* artırıp, yeni değeri ekrana basıyor. Bu değişkenin değeri *10*'a ulaştığında ise, bu değerin artık *10*'dan küçük olmadığını anlayıp bütün işlemleri durduruyor.

Gelin isterseniz bu `while` döngüsünü daha önce yazdığımız hesap makinemize uygulayalım:

```
giriş = """
(1) topla
(2) çıkar
(3) çarp
(4) böl
(5) karesini hesapla
(6) karekök hesapla
"""

print(giriş)

anahtar = 1

while anahtar == 1:
    soru = input("Yapmak istediğiniz işlemin numarasını girin: ")

    if soru == "q":
        print("çıkılıyor...")
        anahtar = 0

    elif soru == "1":
        sayı1 = int(input("Toplama işlemi için ilk sayıyı girin: "))
        sayı2 = int(input("Toplama işlemi için ikinci sayıyı girin: "))
        print(sayı1, "+", sayı2, "=", sayı1 + sayı2)

    elif soru == "2":
        sayı3 = int(input("Çıkarma işlemi için ilk sayıyı girin: "))
        sayı4 = int(input("Çıkarma işlemi için ikinci sayıyı girin: "))
        print(sayı3, "-", sayı4, "=", sayı3 - sayı4)

    elif soru == "3":
        sayı5 = int(input("Çarpma işlemi için ilk sayıyı girin: "))
        sayı6 = int(input("Çarpma işlemi için ikinci sayıyı girin: "))
        print(sayı5, "x", sayı6, "=", sayı5 * sayı6)

    elif soru == "4":
        sayı7 = int(input("Bölme işlemi için ilk sayıyı girin: "))
        sayı8 = int(input("Bölme işlemi için ikinci sayıyı girin: "))
        print(sayı7, "/", sayı8, "=", sayı7 / sayı8)

    elif soru == "5":
        sayı9 = int(input("Karesini hesaplamak istediğiniz sayıyı girin: "))
        print(sayı9, "sayısının karesi =", sayı9 ** 2)
```

```
    elif soru == "6":
        sayı10 = int(input("Karekökünü hesaplamak istediğiniz sayıyı girin: "))
        print(sayı10, "sayısının karekökü = ", sayı10 ** 0.5)

    else:
        print("Yanlış giriş.")
        print("Aşağıdaki seçeneklerden birini giriniz:", giriş)
```

Burada ilave olarak şu satırları görüyorsunuz:

```
anahtar = 1

while anahtar == 1:
    soru = input("Yapmak istediğiniz işlemin numarasını girin (Çıkmak için q): ")

    if soru == "q":
        print("çıkılıyor...")
        anahtar = 0
```

Bu kodlarda yaptığımız şey aslında çok basit. Öncelikle değeri *1* olan *anahtar* adlı bir değişken tanımladık. Bir alt satırda ise, programımızın sürekli olarak çalışmasını sağlayacak olan while döngümüzü yazıyoruz. Programımız, *anahtar* değişkeninin değeri *1* olduğu müddetçe çalışmaya devam edecek. Daha önce de dediğimiz gibi, eğer bu *anahtar* değişkeninin değerini programın bir noktasında değiştirmezsek programımız sonsuza kadar çalışmaya devam edecektir. Çünkü biz programımızı *anahtar* değişkeninin değeri *1* olduğu sürece çalışmaya ayarladık. İşte programımızın bu tür bir sonsuz döngüye girmesini önlemek için bir if bloğu oluşturuyoruz. Buna göre, eğer kullanıcı klavyede *q* tuşuna basarsa programımız önce *çıkılıyor...* çıktısı verecek, ardından da *anahtar* değişkeninin *1* olan değerini *0* yapacaktır. Böylece artık *anahtar*'ın değeri *1* olmayacağı için programımız çalışmaya son verecektir.

Buradaki mantığın ne kadar basit olduğunu görmenizi isterim. Önce bir değişken tanımlıyoruz, ardından bu değişkenin değeri aynı kaldığı müddetçe programımızı çalışmaya ayarlıyoruz. Bu döngüyü kırmak için de başta tanımladığımız o değişkene başka bir değer atıyoruz. Burada *anahtar* değişkenine atadığımız *1* ve *0* değerleri tamamen tesadüfidir. Yani siz bu değerleri istediğiniz gibi değiştirebilirsiniz. Mesela yukarıdaki kodları şöyle de yazabilirsiniz:

```
anahtar = "hoyda bre!"

#anahtar'ın değeri 'hoyda bre!' olduğu müddetçe aşağıdaki bloğu
#çalıştırmaya devam et.
while anahtar == "hoyda bre!":
    soru = input("Yapmak istediğiniz işlemin numarasını girin (Çıkmak için q): ")
```

```
    if soru == "q":
        print("çıkılıyor...")
        anahtar = "dur yolcu!"
        #anahtar'ın değeri artık 'hoyda bre!' değil, 'dur yolcu'
        #olduğu için döngüden çık ve böylece programı sona erdirmiş ol.
```

Gördüğünüz gibi, amaç herhangi bir değişkene herhangi bir değer atamak ve o değer aynı kaldığı müddetçe programın çalışmaya devam etmesini sağlamak. Kurduğumuz bu döngüyü kırmak için de o değişkene herhangi başka bir değer atamak...

Yukarıda verdiğimiz son örnekte önce *anahtar* adlı bir değişken atayıp, while döngüsünün işleyişini bu değişkenin değerine göre yapılandırdık. Ama aslında yukarıdaki kodları çok daha basit bir şekilde de yazabiliriz. Dikkatlice bakın:

```
while True:
    soru = input("Yapmak istediğiniz işlemin numarasını girin (Çıkmak için q): ")

    if soru == "q":
        print("çıkılıyor...")
        break
```

Bu yapıyı hesap makinemize uygulayalım:

```
giriş = """
(1) topla
(2) çıkar
(3) çarp
(4) böl
(5) karesini hesapla
(6) karekök hesapla
"""

print(giriş)

while True:
    soru = input("Yapmak istediğiniz işlemin numarasını girin: ")

    if soru == "q":
        print("çıkılıyor...")
        break

    elif soru == "1":
        sayı1 = int(input("Toplama işlemi için ilk sayıyı girin: "))
        sayı2 = int(input("Toplama işlemi için ikinci sayıyı girin: "))
```

```
    print(sayı1, "+", sayı2, "=", sayı1 + sayı2)

elif soru == "2":
    sayı3 = int(input("Çıkarma işlemi için ilk sayıyı girin: "))
    sayı4 = int(input("Çıkarma işlemi için ikinci sayıyı girin: "))
    print(sayı3, "-", sayı4, "=", sayı3 - sayı4)

elif soru == "3":
    sayı5 = int(input("Çarpma işlemi için ilk sayıyı girin: "))
    sayı6 = int(input("Çarpma işlemi için ikinci sayıyı girin: "))
    print(sayı5, "x", sayı6, "=", sayı5 * sayı6)

elif soru == "4":
    sayı7 = int(input("Bölme işlemi için ilk sayıyı girin: "))
    sayı8 = int(input("Bölme işlemi için ikinci sayıyı girin: "))
    print(sayı7, "/", sayı8, "=", sayı7 / sayı8)

elif soru == "5":
    sayı9 = int(input("Karesini hesaplamak istediğiniz sayıyı girin: "))
    print(sayı9, "sayısının karesi =", sayı9 ** 2)

elif soru == "6":
    sayı10 = int(input("Karekökünü hesaplamak istediğiniz sayıyı girin: "))
    print(sayı10, "sayısının karekökü = ", sayı10 ** 0.5)

else:
    print("Yanlış giriş.")
    print("Aşağıdaki seçeneklerden birini giriniz:", giriş)
```

Bu yapı sayesinde *anahtar* gibi bir değişken atama zorunluluğundan kurtulmuş olduk. Yukarıdaki kodların nasıl çalıştığını açıklayalım:

while True ifadesi şöyle bir anlama gelir:

> *True* olduğu müddetçe...

Peki ne *True* olduğu müddetçe? Burada neyin *True* olması gerektiğini belirtmediğimiz için, aslında bu kod parçası şu anlama geliyor:

> Aksi belirtilmediği sürece çalışmaya devam et!

Eğer yukarıdaki açıklamayı biraz bulanık bulduysanız şu örneği inceleyebilirsiniz:

```
while True:
    print("Bilgisayar çıldırdı!")
```

Bu kodları çalıştırdığınızda ekrana sürekli olarak *Bilgisayar çıldırdı!* çıktısı verilecektir. Bu döngüden çıkabilmek için *Ctrl+C* tuşlarına basmanız gerekiyor. Yukarıdaki kodların sonsuz döngüye girmesinin sorumlusu *while True* satırıdır. Çünkü burada biz Python'a;

> Aksi belirtilmediği sürece çalışmaya devam et!

emri veriyoruz. Python da bu emrimizi sadakatle yerine getiriyor. Böyle bir durumda sonsuz döngüyü engellemek için programımızın bir yerinde Python'a bu döngüden çıkmasını sağlayacak bir emir vermemiz gerekiyor. Biz hesap makinesi programımızda bu döngüyü şu şekilde kırdık:

```
if soru == "q":
    print("çıkılıyor...")
    break
```

Dikkat ederseniz burada break adlı yeni bir araç görüyoruz. Bu aracın tam olarak ne işe yaradığını ilerleyen sayfalarda inceleyeceğiz. Şimdilik yalnızca şunu bilelim: *break* kelimesi İngilizce'de 'kırmak, koparmak, bozmak' gibi anlamlara gelir. Bu aracın yukarıdaki görevi döngüyü 'kırmak'tır. Dolayısıyla kullanıcı klavyede *q* tuşuna bastığında, while True ifadesi ile çalışmaya başlayan döngü kırılacak ve programımız sona erecektir.

Bu yapıyı daha iyi anlayabilmek için şöyle basit bir örnek daha verelim:

```
#Aksi belirtilmediği sürece kullanıcıya
#aşağıdaki soruyu sormaya devam et!
while True:
    soru = input("Nasılsınız, iyi misiniz?")

    #Eğer kullanıcı 'q' tuşuna basarsa...
    if soru == "q":
        break #döngüyü kır ve programdan çık.
```

Görüyorsunuz, aslında mantık gayet basit:

> Bir döngü oluştur ve bu döngüden çıkmak istediğinde, programın bir yerinde bu döngüyü sona erdirecek bir koşul meydan getir.

Bu mantığı yukarıdaki örneğe şu şekilde uyguladık:

> *while True:* ifadesi yardımıyla bir döngü oluştur ve kullanıcı bu döngüden çıkmak istediğinde (yani *q* tuşuna bastığında), döngüyü kır ve programı sona erdir.

Gelin isterseniz bu konuyu daha net kavramak için bir örnek daha verelim:

```
tekrar = 1

while tekrar <= 3:
```

```
    tekrar += 1
    input("Nasılsınız, iyi misiniz?")
```

Burada programımız kullanıcıya üç kez 'Nasılsınız, iyi misiniz?' sorusunu soracak ve ardından kapanacaktır. Bu kodlarda while döngüsünü nasıl kullandığımıza dikkat edin. Aslında programın mantığı çok basit:

1. Öncelikle değeri *1* olan *tekrar* adlı bir değişken tanımlıyoruz.
2. Bu değişkenin değeri *3*'e eşit veya *3*'ten küçük olduğu müddetçe (while tekrar <= 3) değişkenin değerine *1* ekliyoruz (tekrar += 1).
3. Başka bir deyişle bool(tekrar <= 3) ifadesi *True* olduğu müddetçe değişkenin değerine *1* ekliyoruz.
4. *tekrar* değişkenine her *1* ekleyişimizde kullanıcıya 'Nasılsınız, iyi misiniz?' sorusunu soruyoruz (input("Nasılsınız, iyi misiniz?")).
5. *tekrar* değişkeninin değeri *3*'ü aştığında bool(tekrar <= 3) ifadesi artık *False* değeri verdiği için programımız sona eriyor.

Yukarıdaki uygulamada Python'ın alttan alta neler çevirdiğini daha iyi görmek için bu uygulamayı şöyle yazmayı deneyin:

```
tekrar = 1

while tekrar <= 3:
    print("tekrar: ", tekrar)
    tekrar += 1
    input("Nasılsınız, iyi misiniz?")
    print("bool değeri: ", bool(tekrar <= 3))
```

Daha önce de dediğimiz gibi, bir Python programının nasıl çalıştığını anlamanın en iyi yolu, program içinde uygun yerlere print() fonksiyonları yerleştirerek arka planda hangi kodların hangi çıktıları verdiğini izlemektir. İşte yukarıda da bu yöntemi kullandık. Yani *tekrar* değişkenininin değerini ve bool(tekrar <= 3) ifadesinin çıktısını ekrana yazdırarak arka tarafta neler olup bittiğini canlı canlı görme imkanına kavuştuk.

Yukarıdaki programı çalıştırdığımızda şuna benzer çıktılar görüyoruz:

```
tekrar:  1
Nasılsınız, iyi misiniz? evet
bool değeri:  True
tekrar:  2
Nasılsınız, iyi misiniz? evet
bool değeri:  True
tekrar:  3
Nasılsınız, iyi misiniz? evet
```

```
bool değeri:  False
```

Gördüğünüz gibi, *tekrar* değişkeninin değeri her döngüde *1* artıyor. tekrar <= 3 ifadesinin bool değeri, *tekrar* adlı değişkenin değeri *3*'ü aşana kadar hep *True* olacaktır. Bu değişkenin değeri *3*'ü aştığı anda tekrar <= 3 ifadesinin bool değeri *False*'a dönüyor ve böylece while döngüsü sona eriyor.

Peki size şöyle bir soru sorsam: Acaba while döngüsünü kullanarak *1*'den *100*'e kadar olan aralıktaki çift sayıları nasıl bulursunuz?

Çok basit:

```
a = 0

while a < 100:
    a += 1
    if a % 2 == 0:
        print(a)
```

Gördüğünüz gibi, while döngüsünün içine bir adet if bloğu yerleştirdik.

Yukarıdaki kodları şu şekilde Türkçeye çevirebiliriz:

> *a* değişkeninin değeri *100*'den küçük olduğu müddetçe *a* değişkeninin değerini *1* artır. Bu değişkenin değerini her artırışında yeni değerin *2*'ye tam bölünüp bölünmediğini kontrol et. Eğer a modülüs 2 değeri *0* ise (if a % 2 == 0), yani *a*'nın değeri bir çift sayı ise, bu değeri ekrana yazdır.

Gördüğünüz gibi, while döngüsü son derece kullanışlı bir araçtır. Üstelik kullanımı da son derece kolaydır. Bu döngüyle bol bol pratik yaparak bu döngüyü rahatça kullanabilecek duruma gelebilirsiniz.

En başta da söylediğimiz gibi, Python'da while dışında bir de for döngüsü vardır. En az while kadar önemli bir döngü olan for döngüsünün nasıl kullanıldığını anlamaya çalışalım şimdi de.

for Döngüsü

Etrafta yazılmış Python programlarının kaynak kodlarını incelediğinizde, içinde for döngüsü geçmeyen bir program kolay kolay bulamazsınız. Belki while döngüsünün kullanılmadığı programlar vardır. Ancak for döngüsü Python'da o kadar yaygındır ve o kadar geniş bir kullanım alanına sahiptir ki, hemen hemen bütün Python programları bu for döngüsünden en az bir kez yararlanır.

Peki nedir bu for döngüsü denen şey?

for da tıpkı while gibi bir döngüdür. Yani tıpkı while döngüsünde olduğu gibi, programlarımızın birden fazla sayıda çalışmasını sağlar. Ancak for döngüsü while döngüsüne göre biraz daha yeteneklidir. while döngüsü ile yapamayacağınız veya yaparken çok zorlanacağınız şeyleri for döngüsü yardımıyla çok kolay bir şekilde halledebilirsiniz.

Yalnız, söylediğimiz bu cümleden, for döngüsünün while döngüsüne bir alternatif olduğu sonucunu çıkarmayın. Evet, while ile yapabildiğiniz bir işlemi for ile de yapabilirsiniz çoğu zaman, ama bu döngülerin, belli vakalar için tek seçenek olduğu durumlar da vardır. Zira bu iki döngünün çalışma mantığı birbirinden farlıdır.

Şimdi gelelim for döngüsünün nasıl kullanılacağına...

Dikkatlice bakın:

```
tr_harfler = "şçöğüİı"

for harf in tr_harfler:
    print(harf)
```

Burada öncelikle *tr_harfler* adlı bir değişken tanımladık. Bu değişken Türkçeye özgü harfleri tutuyor. Daha sonra bir for döngüsü kurarak, *tr_harfler* adlı değişkenin her bir öğesini tek tek ekrana yazdırdık.

Peki bu for döngüsünü nasıl kurduk?

for döngülerinin söz dizimi şöyledir:

```
for değişken_adı in değişken:
    yapılacak_işlem
```

Bu söz dizimini Türkçe olarak şöyle ifade edebiliriz:

```
değişken içindeki herbir öğeyi değişken_adı olarak adlandır:
    ve bu öğelerle bir işlem yap.
```

Bu soyut yapıları kendi örneğimize uygulayarak durumu daha net anlamaya çalışalım:

```
tr_harfler adlı değişken içindeki herbir öğeyi harf olarak adlandır:
    ve harf olarak adlandırılan bu öğeleri ekrana yazdır.
```

Yukarıdaki örnekte bir for döngüsü yardımıyla *tr_harfler* adlı değişken içindeki herbir öğeyi ekrana yazdırdık. Esasında for döngüsünün yeteneklerini düşündüğümüzde bu örnek pek heyecan verici değil. Zira aynı işi aslında print() fonksiyonu ile de yapabilirdik:

```
tr_harfler = "şçöğüİı"
```

```
print(*tr_harfler, sep="\n")
```

Aslında bu işlemi while ile de yapmak mümkün (Bu kodlardaki, henüz öğrenmediğimiz kısmı şimdilik görmezden gelin):

```
tr_harfler = "şçöğüİı"
a = 0

while a < len(tr_harfler):
    print(tr_harfler[a], sep="\n")
    a += 1
```

while döngüsü kullanıldığında işi uzattığımızı görüyorsunuz. Dediğimiz gibi, for döngüsü while döngüsüne göre biraz daha yeteneklidir ve while ile yapması daha zor (veya uzun) olan işlemleri for döngüsü ile çok daha kolay bir şekilde yapabiliriz. Ayrıca for döngüsü ile while döngüsünün çalışma mantıkları birbirinden farklıdır. for döngüsü, üzerinde döngü kurulabilecek veri tiplerinin herbir öğesinin üzerinden tek tek geçer ve bu öğelerin herbiri üzerinde bir işlem yapar. while döngüsü ise herhangi bir ifadenin bool değerini kontrol eder ve bu değerin bool değeri *False* olana kadar, belirlenen işlemi yapmayı sürdürür.

Bu arada, biraz önce 'üzerinde döngü kurulabilecek veri tipleri' diye bir kavramdan söz ettik. Örneğin karakter dizileri, üzerinde döngü kurulabilecek bir veri tipidir. Ama sayılar öyle değildir. Yani sayılar üzerinde döngü kuramayız. Mesela:

```
>>> sayılar = 123456789
>>> for sayı in sayılar:
...     print(sayı)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

Buradaki hata mesajından da göreceğiniz gibi *int* (tam sayı) türündeki nesneler üzerinde döngü kuramıyoruz. Hata mesajında görünen *not iterable* (üzerinde döngü kurulamaz) ifadesiyle kastedilen de budur.

Gelin isterseniz for döngüsü ile bir örnek daha vererek durumu iyice anlamaya çalışalım:

```
sayılar = "123456789"

for sayı in sayılar:
    print(int(sayı) * 2)
```

Burada *sayılar* adlı değişkenin herbir öğesini *sayı* olarak adlandırdıktan sonra, int() fonksiyonu

yardımıyla bu öğeleri tek tek sayıya çevirdik ve herbir öğeyi *2* ile çarptık.

for döngüsünün mantığını az çok anlamış olmalısınız. Bu döngü bir değişken içindeki herbir öğeyi tek tek ele alıp, iki nokta üst üste işaretinden sonra yazdığımız kod bloğunu bu öğelere tek tek uyguluyor.

for kelimesi İngilizcede 'için' anlamına gelir. Döngünün yapısı içinde geçen *in* ifadesini de tanıyorsunuz. Biz bu ifadeyi 'Aitlik İşleçleri' konusunu işlerken de görmüştük. Hatırlarsanız *in* işleci bir öğenin bir veri tipi içinde bulunup bulunmadığını sorguluyordu. Mesela:

```
>>> a = "istihza.com"
>>> "h" in a

True
```

"h" öğesi *"istihza.com"* adlı karakter dizisi içinde geçtiği için `"h" in a` kodu *True* çıktısı veriyor. Bir de şuna bakın:

```
>>> "b" in a

False
```

"b" öğesi *"istihza.com"* karakter dizisi içinde bulunmuyor. Dolayısıyla `"b" in a` sorgulaması *False* çıktısı veriyor.

in kelimesi İngilizcede 'içinde' anlamına geliyor. Dolayısıyla `for falanca in filanca:` yazdığımızda aslında şöyle bir şey demiş oluyoruz:

> *filanca* içinde *falanca* adını verdiğimiz herbir öğe için...

Yani şu kod:

```
for s in "istihza":
    print(s)
```

Şu anlama geliyor:

> *"istihza"* karakter dizisi içinde *s* adını verdiğimiz herbir öğe için:
>> *s* öğesini ekrana basma işlemi gerçekleştir!

Ya da şu kod:

```
sayılar = "123456789"

for i in sayılar:
    if int(i) > 3:
```

```
        print(i)
```

Şu anlama geliyor:

sayılar değişkeni içinde *i* adını verdiğimiz herbir öğe için:
    eğer sayıya dönüştürülmüş *i* değeri *3*'ten büyükse:
        i öğesini ekrana basma işlemi gerçekleştir!

Yukarıdaki temsili kodların Türkçesi bozuk olsa da for döngüsünün çalışma mantığını anlamaya yardımcı olacağını zannediyorum. Ama yine de, eğer bu döngünün mantığını henüz kavrayamadıysanız hiç endişe etmeyin. Zira bu döngüyü oldukça sık bir biçimde kullanacağımız için, siz istemeseniz de bu döngü kafanızda yer etmiş olacak.

Bu for döngüsünü biraz daha iyi anlayabilmek için son bir örnek yapalım:

```
tr_harfler = "şçöğüİı"

parola = input("Parolanız: ")

for karakter in parola:
    if karakter in tr_harfler:
        print("parolada Türkçe karakter kullanılamaz")
```

Bu program, kullanıcıya bir parola soruyor. Eğer kullanıcının girdiği parola içinde Türkçe karakterlerden herhangi biri varsa kullanıcıyı Türkçe karakter kullanmaması konusunda uyarıyor. Buradaki for döngüsünü nasıl kurduğumuzu görüyorsunuz. Aslında burada şu Türkçe cümleyi Pythonca'ya çevirmiş olduk:

parola değişkeni içinde *karakter* adını verdiğimiz herbir öğe için:
    eğer *karakter* değişkeni *tr_harfler* adlı değişken içinde geçiyorsa:
        'parolada Türkçe karakter kullanılamaz' uyarısını göster!

Burada kullandığımız for döngüsü sayesinde kullanıcının girdiği *parola* adlı değişken içindeki bütün karakterlere tek tek bakıp, eğer bakılan karakter *tr_harfler* adlı değişken içinde geçiyorsa kullanıcıyı uyarıyoruz.

Aslında for döngüsüyle ilgili söyleyeceklerimiz bu kadar değil. Ama henüz bu döngüyle kullanılan önemli araçları tanımıyoruz. Gerçi zaten bu döngüyü bundan sonra sık sık kullandığımızı göreceksiniz.

Gelin isterseniz yeni bir konuya geçmeden önce döngülerle ilgili ufak bir örnek verelim:

Örneğin kullanıcıya bir parola belirletirken, belirlenecek parolanın *8* karakterden uzun, *3* karakterden kısa olmamasını sağlayalım:

```
while True:
    parola = input("Bir parola belirleyin: ")

    if not parola:
        print("parola bölümü boş geçilemez!")

    elif len(parola) > 8 or len(parola) < 3:
        print("parola 8 karakterden uzun 3 karakterden kısa olmamalı")

    else:
        print("Yeni parolanız", parola)
        break
```

Burada öncelikle, programınızın sürekli olarak çalışmasını sağlamak için bir while döngüsü oluşturduk. Buna göre, aksi belirtilmedikçe (while True) programımız çalışmaya devam edecek.

while döngüsünü kurduktan sonra kullanıcıya bir parola soruyoruz (parola = input("Bir parola belirleyin: "))

Eğer kullanıcı herhangi bir parola belirlemeden doğrudan *Enter* tuşuna basarsa, yani *parola* değişkeninin bool değeri *False* olursa (if not parola), kullanıcıya 'parola bölümü boş geçilemez!' uyarısı veriyoruz.

Eğer kullanıcı tarafından belirlenen parolanın uzunluğu *8* karakterden fazlaysa ya da *3* karakterden kısaysa, 'parola 8 karakterden uzun 3 karakterden kısa olmamalı' uyarısı veriyoruz.

Yukarıdaki koşullar harici durumlar için ise (else), belirlenen yeni parolayı kullanıcıya gösterip döngüden çıkıyoruz (break).

Bu arada, hatırlarsanız eval() fonksiyonunu anlatırken şöyle bir örnek vermiştik:

```
print("""
Basit bir hesap makinesi uygulaması.

İşleçler:

   +   toplama
   -   çıkarma
   *   çarpma
   /   bölme

Yapmak istediğiniz işlemi yazıp ENTER
tuşuna basın. (Örneğin 23 ve 46 sayılarını
çarpmak için 23 * 46 yazdıktan sonra
```

```
ENTER tuşuna basın.)
""")

veri = input("İşleminiz: ")
hesap = eval(veri)

print(hesap)
```

Bu programdaki eksiklikleri ve riskleri biliyorsunuz. Böyle bir program yazdığınızda, eval() fonksiyonunu kontrolsüz bir şekilde kullandığınız için önemli bir güvenlik açığına sebep olmuş oluyorsunuz. Gelin isterseniz bu derste öğrendiğimiz bilgileri de kullanarak yukarıdaki eval() fonksiyonu için basit bir kontrol mekanizması kuralım:

```
izinli_karakterler = "0123456789+-/*= "

print("""
Basit bir hesap makinesi uygulaması.

İşleçler:

   +   toplama
   -   çıkarma
   *   çarpma
   /   bölme

Yapmak istediğiniz işlemi yazıp ENTER
tuşuna basın. (Örneğin 23 ve 46 sayılarını
çarpmak için 23 * 46 yazdıktan sonra
ENTER tuşuna basın.)
""")

while True:
    veri = input("İşleminiz: ")
    if veri == "q":
        print("çıkılıyor...")
        break

    for s in veri:
        if s not in izinli_karakterler:
            print("Neyin peşindesin?!")
            quit()
        else:
            hesap = eval(veri)
```

```
    print(hesap)
```

Burada öncelikle programımızı bir while döngüsü içine aldık. Böylece programımızın ne zaman sona ereceğini kendimiz belirleyebileceğiz. Buna göre eğer kullanıcı klavyede ‘q’ tuşuna basarsa while döngüsü sona erecek.

Bu programda bizi özellikle ilgilendiren kısım şu:

```
izinli_karakterler = "0123456789+-/*= "

for s in veri:
    if s not in izinli_karakterler:
        print("Neyin peşindesin?!")
        quit()
    else:
        hesap = eval(veri)
```

Gördüğünüz gibi, ilk olarak *izinli_karakterler* adlı bir değişken tanımladık. Program içinde kullanılmasına izin verdiğimiz karakterleri bu değişken içine yazıyoruz. Buna göre kullanıcı yalnızca *0*, *1*, *2*, *3*, *4*, *5*, *6*, *7*, *8* ve *9* sayılarını, +, -, /, * ve = işleçlerini, ayrıca boşluk karakterini (‘ ‘) kullanabilecek.

Kullanıcının girdiği veri üzerinde bir for döngüsü kurarak, veri içindeki her bir karakterin *izinli_karakterler* değişkeni içinde yer alıp almadığını denetliyoruz. İzin verilen karakterler dışında herhangi bir karakterin girilmesi *Neyin peşindesin?!* çıktısının verilip programdan tamamen çıkılmasına (quit()) yol açacaktır.

Eğer kullanıcı izinli karakterleri kullanarak bir işlem gerçekleştirmişse hesap = eval(veri) kodu aracılığıyla, kullanıcının yaptığı işlemi eval() fonksiyonuna gönderiyoruz.

Böylece eval() fonksiyonunu daha güvenli bir hale getirebilmek için basit bir kontrol mekanizmasının nasıl kurulabileceğini görmüş olduk. Kurduğumuz kontrol mekanizmasının esası, kullanıcının girebileceği veri türlerini sınırlamaya dayanıyor. Böylece kullanıcı mesela şöyle tehlikeli bir komut giremiyor:

```
__import__("os").system("dir")
```

Çünkü bu komutu yazabilmesi için gereken karakterler *izinli_karakterler* değişkeni içinde tanımlı değil. Kullanıcı yalnızca basit bir hesap makinesinde kullanılabilecek olan sayıları ve işleçleri girebiliyor.

İlgili Araçlar

Elbette döngüler tek başlarına bir şey ifade etmezler. Döngülerle işe yarar kodlar yazabilmemiz için bazı araçlara ihtiyacımız var. İşte bu bölümde döngüleri daha verimli kullanmamızı sağlayacak bazı fonksiyon ve deyimlerden söz edeceğiz. İlk olarak range() adlı bir fonksiyondan bahsedelim.

range Fonksiyonu

range kelimesi İngilizcede 'aralık' anlamına gelir. Biz Python'da range() fonksiyonunu belli bir aralıkta bulunan sayıları göstermek için kullanıyoruz. Örneğin:

```
>>> for i in range(0, 10):
...     print(i)
...
0
1
2
3
4
5
6
7
8
9
```

Gördüğünüz gibi, range(0, 10) kodu sayesinde ve for döngüsünü de kullanarak, *0* ile *10* (*10* hariç) aralığındaki sayıları ekrana yazdırdık.

Yukarıdaki kodda range() fonksiyonuna *0* ve *10* olmak üzere iki adet parametre verdiğimizi görüyorsunuz. Burada *0* sayısı, aralıktaki ilk sayıyı, *10* sayısı ise aralıktaki son sayıyı gösteriyor. Yani range() fonksiyonunun formülü şöyledir:

```
range(ilk_sayı, son_sayı)
```

Bu arada, range(ilk_sayı, son_sayı) kodunun verdiği çıktıya ilk_sayının dahil olduğuna, ama son_sayının dahil olmadığına dikkat edin.

Eğer range() fonksiyonunun ilk parametresi *0* olacaksa, bu parametreyi belirtmesek de olur. Yani mesela *0*'dan *10*'a kadar olan sayıları listeleyeceksek range() fonksiyonunu şöyle yazmamız yeterli olacaktır:

```
>>> for i in range(10):
...     print(i)
```

range() fonksiyonunun *ilk_sayı* parametresi verilmediğinde Python ilk parametreyi *0* olarak alır. Yani range(10) gibi bir kodu Python range(0, 10) olarak algılar. Elbette, eğer aralıktaki ilk sayı

0'dan farklı olacaksa bu sayıyı açık açık belirtmek gerekir:

```
>>> for i in range(3, 20):
...     print(i)
```

Burada *3*'ten itibaren *20*'ye kadar olan sayılar ekrana dökülecektir.

Hatırlarsanız, biraz önce, kullanıcının *3* karakterden kısa, *8* karakterden uzun parola belirlemesini engelleyen bir uygulama yazmıştık. O uygulamayı range() fonksiyonunu kullanarak da yazabiliriz:

```
while True:
    parola = input("parola belirleyin: ")

    if not parola:
        print("parola bölümü boş geçilemez!")

    elif len(parola) in range(3, 8): #eğer parolanın uzunluğu 3 ile 8 karakter
        #aralığında ise...
        print("Yeni parolanız", parola)
        break

    else:
        print("parola 8 karakterden uzun 3 karakterden kısa olmamalı")
```

Bu fonksiyonu kullanarak bir döngünün kaç kez çalışacağını da belirleyebilirsiniz. Aşağıdaki kodları dikkatlice inceleyin:

```
for i in range(3):
    parola = input("parola belirleyin: ")
    if i == 2:
        print("parolayı 3 kez yanlış girdiniz.",
        "Lütfen 30 dakika sonra tekrar deneyin!")

    elif not parola:
        print("parola bölümü boş geçilemez!")

    elif len(parola) in range(3, 8):
        print("Yeni parolanız", parola)
        break

    else:
        print("parola 8 karakterden uzun 3 karakterden kısa olmamalı")
```

Burada if i == 2 kodu sayesinde for döngüsü içinde belirttiğimiz *i* adlı değişkenin değeri *2* olduğu

anda ‘parolayı 3 kez yanlış girdiniz...’ uyarısı gösterilecektir. Daha önce de birkaç yerde ifade ettiğimiz gibi, eğer yukarıdaki kodların çalışma mantığını anlamakta zorlanıyorsanız, programın uygun yerlerine print() fonksiyonu yerleştirerek arka planda Python’ın neler çevirdiğini daha net görebilirsiniz. Örneğin:

```
for i in range(3):
    print(i)
    parola = input("parola belirleyin: ")
    if i == 2:
        print("parolayı 3 kez yanlış girdiniz.",
        "Lütfen 30 dakika sonra tekrar deneyin!")

    elif not parola:
        print("parola bölümü boş geçilemez!")

    elif len(parola) in range(3, 8):
        print("Yeni parolanız", parola)
        break

    else:
        print("parola 8 karakterden uzun 3 karakterden kısa olmamalı")
```

Gördüğünüz gibi, *i* değişkeninin başlangıçtaki değeri *0*. Bu değer her döngüde *1* artıyor ve bu değişkenin değeri *2* olduğu anda if i == 2 bloğu devreye giriyor.

range() fonksiyonunun yetenekleri yukarı anlattıklarımızla sınırlı değildir. Bu fonksiyonun bazı başka maharetleri de bulunur. Hatırlarsanız yukarı bu fonksiyonun formülünü şöyle vermiştik:

```
range(ilk_sayı, son_sayı)
```

Buna göre range() fonksiyonu iki parametre alıyor. Ama aslında bu fonksiyonun üçüncü bir parametresi daha vardır. Buna göre formülümüzü güncelleyelim:

```
range(ilk_sayı, son_sayı, atlama_değeri)
```

Formüldeki son parametre olan *atlama_değeri*, aralıktaki sayıların kaçar kaçar ilerleyeceğini gösterir. Yani:

```
>>> for i in range(0, 10, 2):
...     print(i)
...

0
2
```

```
4
6
8
```

Gördüğünüz gibi, son parametre olarak verdiğimiz *2* sayısı sayesinde *0*'dan *10*'a kadar olan sayılar ikişer ikişer atlayarak ekrana dökülüyor.

Bu arada, bir şey dikkatinizi çekmiş olmalı:

range() fonksiyonu üç farklı parametre alan bir fonksiyon. Eğer ilk parametre *0* olacaksa bu parametreyi belirtmek zorunda olmadığımızı biliyoruz. Yani:

```
>>> range(10)
```

Python bu kodu range(0, 10) olarak algılayıp buna göre değerlendiriyor. Ancak eğer range() fonksiyonunda üçüncü parametreyi de kullanacaksak, yani range(0, 10, 2) gibi bir komut vereceksek, üç parametrenin tamamını da belirtmemiz gerekiyor. Eğer burada bütün parametreleri belirtmezsek Python hangi sayının hangi parametreye karşılık geldiğini anlayamaz. Yani mesela *0*'dan *10*'a kadar olan sayıları ikişer ikişer atlayarak ekrana dökmek için şöyle bir şey yazmaya çalıştığımızı düşünün:

```
>>> for i in range(10, 2):
...     print(i)
```

Burada Python ne yapmaya çalıştığınızı anlayamaz. Parantez içinde ilk değer olarak *10*, ikinci değer olarak ise *2* yazdığınız için, Python bu *10* sayısını başlangıç değeri; *2* sayısını ise bitiş değeri olarak algılayacaktır. Dolayısıyla da Python bu durumda sizin *10*'dan *2*'ye kadar olan sayıları listelemek istediğinizi zannedecek, range() fonksiyonuyla bu şekilde geriye doğru sayamayacağımız için de boş bir çıktı verecektir. Bu yüzden, Python'un şaşırmaması için yukarıdaki örneği şu şekilde yazmalıyız:

```
>>> for i in range(0, 10, 2):
...     print(i)
```

Kısacası, eğer range() fonksiyonunun kaçar kaçar sayacağını da belirtmek istiyorsak, parantez içinde, gerekli bütün parametreleri belirtmeliyiz.

Gördüğünüz gibi, range() fonksiyonunu kullanarak belirli bir aralıktaki sayıları alabiliyoruz. Peki bu sayıları tersten alabilir miyiz? Elbette:

```
>>> for i in range(10, 0, -1):
...     print(i)
...
10
9
8
```

```
7
6
5
4
3
2
1
```

Burada range() fonksiyonunu nasıl yazdığımıza çok dikkat edin. Sayıları tersten alacağımız için, ilk parametre 10, ikinci parametre ise *0*. Üçüncü parametre olarak ise eksi değerli bir sayı veriyoruz. Eğer sayıları hem tersten, hem de mesela 3'er 3'er atlayarak yazmak isterseniz şöyle bir komut verebilirsiniz:

```
>>> for i in range(10, 0, -3):
...     print(i)
...
10
7
4
1
```

Bu arada, etkileşimli kabukta range(10) gibi bir komut verdiğinizde *range(0, 10)* çıktısı aldığınızı görüyorsunuz. Bu çıktı, verdiğimiz komutun *0* ile *10* arası sayıları elde etmemizi sağlayacağını belirtiyor, ama bu sayıları o anda bize göstermiyor. Daha önce verdiğimiz örneklerden de anlaşılacağı gibi, *0-10* aralığındaki sayıları görebilmek için range(10) ifadesi üzerinde bir for döngüsü kurmamız gerekiyor. range(10) ifadesinin taşıdığı sayıları görebilmek için for döngüsü kurmak tek seçenek değildir. Bu işlem için yıldızlı parametrelerden de yararlanabiliriz. print() fonksiyonunu incelediğimiz derste yıldızlı parametrelerin nasıl kullanıldığını göstermiştik. Dilerseniz şimdi bu parametre tipini range() fonksiyonuna nasıl uygulayabileceğimizi görelim:

```
>>> print(*range(10))

0 1 2 3 4 5 6 7 8 9
```

print() fonksiyonunun *sep* parametresi yardımıyla bu çıktıyı istediğiniz gibi düzenleyebileceğinizi biliyorsunuz. Mesela çıktıdaki sayıları birbirlerinden virgülle ayırmak için şöyle bir komut verebiliyoruz:

```
>>> print(*range(10), sep=", ")

0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Böylece range() fonksiyonunu enine boyuna incelemiş ve bu fonksiyonun ne işe yaradığını, nasıl kullanılacağını anlamamızı sağlayan örnekler vermiş olduk. Artık başka bir konuyu geçebiliriz.

pass Deyimi

pass kelimesi İngilizcede 'geçmek, pas geçmek' gibi anlamlara gelir. Python'daki kullanımı da bu anlama oldukça yakındır. Biz bu deyimi Pyhon'da 'görmezden gel, hiçbir şey yapma' anlamında kullanacağız.

Dilerseniz pass deyimini tarif etmeye çalışmak yerine bu deyimi bir örnek üzerinde açıklamaya çalışalım.

Hatırlarsanız yukarıda şöyle bir örnek vermiştik:

```
while True:
    parola = input("parola belirleyin: ")

    if not parola:
        print("parola bölümü boş geçilemez!")

    elif len(parola) in range(3, 8): #eğer parolanın uzunluğu 3 ile 8 karakter
        #aralığında ise...
        print("Yeni parolanız", parola)
        break

    else:
        print("parola 8 karakterden uzun 3 karakterden kısa olmamalı")
```

Burada mesela eğer kullanıcı parolayı boş bırakırsa 'parola bölümü boş geçilemez!' uyarısı gösteriyoruz. Şimdi o if bloğunu şöyle yazdığımızı düşünün:

```
while True:
    parola = input("parola belirleyin: ")

    if not parola:
        pass

    elif len(parola) in range(3, 8): #eğer parolanın uzunluğu 3 ile 8 karakter
        #aralığında ise...
        print("Yeni parolanız", parola)
        break

    else:
        print("parola 8 karakterden uzun 3 karakterden kısa olmamalı")
```

Burada, eğer kullanıcı parolayı boş bırakırsa programımız hiçbir şey yapmadan yoluna devam edecektir. Yani burada pass deyimi yardımıyla programımıza şu emri vermiş oluyoruz:

Eğer kullanıcı parolayı boş geçerse görmezden gel. Hiçbir şey yapmadan yoluna devam et!

Başka bir örnek daha verelim:

```
while True:
    sayı = int(input("Bir sayı girin: "))

    if sayı == 0:
        break

    elif sayı < 0:
        pass

    else:
        print(sayı)
```

Burada eğer kullanıcı *0* sayısını girerse programımız sona erer (break deyimini biraz sonra inceleyeceğiz). Eğer kullanıcı *0*'dan küçük bir sayı girerse, yani kullanıcının girdiği sayı eksi değerli ise, pass deyimininin etkisiyle programımız hiçbir şey yapmadan yoluna devam eder. Bu koşulların dışındaki durumlarda ise programımız kullanıcının girdiği sayıları ekrana yazdıracaktır.

Yukarıda anlatılan durumların dışında, pass deyimini kodlarınız henüz taslak aşamasında olduğu zaman da kullanabilirsiniz. Örneğin, diyelim ki bir kod yazıyorsunuz. Programın gidişatına göre, bir noktada yapmanız gereken bir işlem var, ama henüz ne yapacağınıza karar vermediniz. Böyle bir durumda pass deyiminden yararlanabilirsiniz. Mesela birtakım if deyimleri yazmayı düşünüyor olun:

```
if .....:
    böyle yap

elif .....:
    şöyle yap

else:
    pass
```

Burada henüz else bloğunda ne yapılacağına karar vermemiş olduğunuz için, oraya bir pass koyarak durumu şimdilik geçiştiriyorsunuz. Program son haline gelene kadar oraya bir şeyler yazmış olacaksınız.

Sözün özü, pass deyimlerini, herhangi bir işlem yapılmasının gerekli olmadığı durumlar için kullanıyoruz. İlerde işe yarar programlar yazdığınızda, bu pass deyiminin göründüğünden daha faydalı bir araç olduğunu anlayacaksınız.

break Deyimi

Python'da break özel bir deyimdir. Bu deyim yardımıyla, devam eden bir süreci kesintiye uğratabiliriz. Bu deyimin kullanıldığı basit bir örnek verelim:

```
>>> while True:
...     parola = input("Lütfen bir parola belirleyiniz:")
...         if len(parola) < 5:
...             print("Parola 5 karakterden az olmamalı!")
...         else:
...             print("Parolanız belirlendi!")
...             break
```

Burada, eğer kullanıcının girdiği parolanın uzunluğu *5* karakterden azsa, *Parola 5 karakterden az olmamalı!* uyarısı gösterilecektir. Eğer kullanıcı *5* karakterden uzun bir parola belirlemişse, kendisine 'Parolanız belirlendi!' mesajını gösterip, break deyimi yardımıyla programdan çıkıyoruz.

Gördüğünüz gibi, break ifadesinin temel görevi bir döngüyü sona erdirmek. Buradan anlayacağımız gibi, break ifadesinin her zaman bir döngü içinde yer alması gerekiyor. Aksi halde Python bize şöyle bir hata verecektir:

```
SyntaxError: 'break' outside loop
```

Yani:

SözDizimiHatası: ``break`` döngü dışında ..

continue Deyimi

continue ilginç bir deyimdir. İsterseniz continue deyimini anlatmaya çalışmak yerine bununla ilgili bir örnek verelim:

```
while True:
    s = input("Bir sayı girin: ")
    if s == "iptal":
        break

    if len(s) <= 3:
        continue

    print("En fazla üç haneli bir sayı girebilirsiniz.")
```

Burada eğer kullanıcı klavyede *iptal* yazarsa programdan çıkılacaktır. Bunu;

```
if s == "iptal":
    break
```

satırıyla sağlamayı başardık.

Eğer kullanıcı tarafından girilen sayı üç haneli veya daha az haneli bir sayı ise, continue ifadesinin etkisiyle:

```
>>> print("En fazla üç haneli bir sayı girebilirsiniz.")
```

satırı es geçilecek ve döngünün en başına gidilecektir.

Eğer kullanıcının girdiği sayıdaki hane üçten fazlaysa ekrana:

```
En fazla üç haneli bir sayı girebilirsiniz.
```

cümlesi yazdırılacaktır.

Dolayısıyla buradan anladığımıza göre, continue deyiminin görevi kendisinden sonra gelen her şeyin es geçilip döngünün başına dönülmesini sağlamaktır. Bu bilgiye göre, yukarıdaki programda eğer kullanıcı, uzunluğu üç karakterden az bir sayı girerse continue deyiminin etkisiyle programımız döngünün en başına geri gidiyor. Ama eğer kullanıcı, uzunluğu üç karakterden fazla bir sayı girerse, ekrana 'En fazla üç haneli bir sayı girebilirsiniz,' cümlesinin yazdırıldığını görüyoruz.

Örnek Uygulamalar

Python programlama dilinde döngülerin neye benzediğini öğrendik. Bu bölünde ayrıca döngülerle birlikte kullanabileceğimiz başka araçları da tanıdık. Şimdi dilerseniz bu öğrendiklerimizi pekiştirmek için birkaç ufak çalışma yapalım.

Karakter Dizilerinin İçeriğini Karşılaştırma

Diyelim ki elinizde şöyle iki farklı metin var:

```
ilk_metin = "asdasfddgdhfjfdgdşfkgjdfklgşjdfklgjdfkghdfjghjklsdhajlsdhjkjhkhjjh"
ikinci_metin = "sdfsuıdoryeuıfsjkdfhdjklghjdfklruseldhfjlkdshfljskeeuf"
```

Siz burada, *ilk_metin* adlı değişken içinde bulunan, ama *ikinci_metin* adlı değişken içinde bulunmayan öğeleri ayıklamak istiyorsunuz. Yani bu iki metnin içeriğini karşılaştırıp, farklı öğeleri bulmayı amaçlıyorsunuz. Bu işlem için, bu bölümde öğrendiğimiz döngülerden ve daha önce öğrendiğimiz başka araçlardan yararlanabilirsiniz. Şimdi dikkatlice bakın:

```
ilk_metin = "asdasfddgdhfjfdgdşfkgjdfklgşjdfklgjdfkghdfjghjklsdhajlsdhjkjhkhjjh"
ikinci_metin = "sdfsuıdoryeuıfsjkdfhdjklghjdfklruseldhfjlkdshfljskeeuf"

for s in ilk_metin:
    if not s in ikinci_metin:
        print(s)
```

Bu kodları bir dosyaya kaydedip çalıştırdığımızda şu çıktıyı alıyoruz:

```
a
a
ş
ş
a
```

Demek ki *ilk_metin* adlı değişkende olup da *ikinci_metin* adlı değişkende olmayan öğeler bunlarmış...

Bu kodlarda anlayamayacağınız hiçbir şey yok. Ama dilerseniz biz yine de bu kodları tek tek inceleyelim.

İlk olarak değişkenlerimizi tanımladık:

```
ilk_metin = "asdasfddgdhfjfdgdşfkgjdfklgşjdfklgjdfkghdfjghjklsdhajlsdhjkjhkhjjh"
ikinci_metin = "sdfsuıdoryeuıfsjkdfhdjklghjdfklruseldhfjlkdshfljskeeuf"
```

Amacımız *ilk_metin*'de olan, ama *ikinci_metin*'de olmayan öğeleri görmek. Bunun için *ilk_metin*'deki öğeleri **tek tek** *ikinci_metin*'deki öğelerle karşılaştırmamız gerekiyor. Tahmin edebileceğiniz gibi, bir metnin bütün öğelerine tek tek bakabilmenin en iyi yolu for döngülerini kullanmaktır. O halde döngümüzü yazalım:

```
for s in ilk_metin: #ilk_metin'deki, 's' adını verdiğimiz bütün öğeler için
    if not s in ikinci_metin: #eğer 's' adlı bu öğe ikinci_metin'de yoksa
        print(s) #'s' adlı öğeyi ekrana bas
```

Gördüğünüz gibi, döngüleri (for), bool işleçlerini (*not*) ve aitlik işleçlerini (*in*) kullanarak, istediğimiz şeyi rahatlıkla yapabiliyoruz. Burada kullandığımız if deyimi, bir önceki satırda for döngüsü ile üzerinden geçtiğimiz öğeleri süzmemizi sağlıyor. Burada temel olarak şu üç işlemi yapıyoruz:

1. *ilk_metin* içindeki bütün öğelerin üzerinden geçiyoruz,
2. Bu öğeleri belli bir ölçüte göre süzüyoruz,
3. Ölçüte uyan öğeleri ekrana basıyoruz.

Elbette yukarıda yaptığımız işlemin tersini yapmak da mümkündür. Biz yukarıdaki kodlarda *ilk_metin*'de olan, ama *ikinci_metin*'de olmayan öğeleri süzdük. Eğer istersek *ikinci_metin*'de olan, ama *ilk_metin*'de olmayan öğeleri de süzebiliriz. Mantığımız yine aynı:

```
ilk_metin = "asdasfddgdhfjfdgdşfkgjdfklgşjdfklgjdfkghdfjghjklsdhajlsdhjkjhkhjjh"
ikinci_metin = "sdfsuıdoryeuıfsjkdfhdjklghjdfklruseldhfjlkdshfljskeeuf"

for s in ikinci_metin: #ikinci_metin'deki, 's' adını verdiğimiz bütün öğeler için
```

```
    if not s in ilk_metin: #eğer 's' adlı bu öğe ilk_metin'de yoksa
        print(s) #'s' adlı öğeyi ekrana bas
```

Bu da bize şu çıktıyı veriyor:

```
u
ı
o
r
y
e
u
ı
r
u
e
e
e
u
```

Gördüğünüz gibi, yaptığımız tek şey, *ilk_metin* ile *ikinci_metin*'in yerlerini değiştirmek oldu. Kullandığımız mantık ise değişmedi.

Bu arada, yukarıdaki çıktıda bizi rahatsız eden bir durum var. Çıktıda bazı harfler birbirini tekrar ediyor. Aslında temel olarak sadece şu harfler var:

```
u
ı
o
r
y
e
```

Ama metin içinde bazı harfler birden fazla sayıda geçtiği için, doğal olarak çıktıda da bu harfler birden fazla sayıda görünüyor. Ama tabii ki, eğer biz istersek farklı olan her harften yalnızca bir tanesini çıktıda görmeyi de tercih edebiliriz. Bunun için şöyle bir kod yazabiliriz:

```
ilk_metin = "asdasfddgdhfjfdgdşfkgjdfklgşjdfklgjdfkghdfjghjklsdhajlsdhjkjhkhjjh"
ikinci_metin = "sdfsuıdoryeuıfsjkdfhdjklghjdfklruseldhfjlkdshfljskeeuf"

fark = ""

for s in ikinci_metin:
    if not s in ilk_metin:
        if not s in fark:
```

```
            fark += s
print(fark)
```

Burada da anlayamayacağımız hiçbir şey yok. Bu kodlardaki bütün parçaları tanıyoruz. Herzamanki gibi öncelikle değişkenlerimizi tanımladık:

```
ilk_metin = "asdasfddgdhfjfdgdşfkgjdfklgşjdfklgjdfkghdfjghjklsdhajlsdhjkjhkhjjh"
ikinci_metin = "sdfsuıdoryeuıfsjkdfhdjklghjdfklruseldhfjlkdshfljskeeuf"
```

Daha sonra *fark* adlı boş bir karakter dizisi tanımlıyoruz. Metinler içindeki farklı karakter dizilerini *fark* adlı bu karakter dizisi içinde depolayacağız.

Ardından da for döngümüzü yazıyoruz:

```
for s in ikinci_metin:      # ikinci_metin'de 's' dediğimiz bütün öğeler için
    if not s in ilk_metin:  # eğer 's' ilk_metin'de yoksa
        if not s in fark:   # eğer 's' fark'ta da yoksa
            fark += s       # bu öğeyi fark değişkenine ekle
print(fark)                 # fark değişkenini ekrana bas
```

Uyguladığımız mantığın ne kadar basit olduğunu görüyorsunuz. Bu kodlarda basitçe şu işlemleri yapıyoruz:

1. *ikinci_metin* değişkeni içindeki bütün öğelerin üzerinden tek tek geç,
2. Eğer bu değişkendeki herhangi bir öğe *ilk_metin*'de ve *fark*'ta yoksa o öğeyi *fark*'a ekle.
3. Son olarak da *fark*'ı ekrana bas.

Bu kodlarda dikkatimizi çeken ve üzerinde durmamız gereken bazı noktalar var. Burada özellikle *fark* değişkenine öğe ekleme işlemini nasıl yaptığımıza dikkat edin.

Python programlama dilinde önceden oluşturduğumuz bir karakter dizisini başka bir karakter dizisi ile birleştirdiğimizde bu işlem ilk oluşturduğumuz karakter dizisini etkilemez. Yani:

```
>>> a = 'istihza'
>>> a + '.com'

'istihza.com'
```

Burada sanki *a* adlı özgün karakter dizisini değiştirmişiz ve 'istihza.com' değerini elde etmişiz gibi görünüyor. Ama aslında *a*'nın durumunda hiçbir değişiklik yok:

```
>>> a

'istihza'
```

Gördüğünüz gibi, *a* değişkeninin değeri hâlâ *'istihza'*. Bu durumun nedeni, birleştirme işlemlerinin

bir değiştirme işlemi olmamasıdır. Yani mesela iki karakter dizisini birleştirdiğinizde birleşen karakter dizileri üzerinde herhangi bir değişiklik olmaz. Bu durumda yapabileceğimiz tek şey, karakter dizisine eklemek istediğimiz öğeyi de içeren yeni bir karakter dizisi oluşturmaktır. Yani:

```
>>> a = 'istihza'
>>> a = a + '.com'
>>> print(a)

istihza.com
```

Burada sanki değeri *'istihza'* olan *a* adlı bir değişkene *'.com'* değerini eklemişiz gibi görünüyor, ama aslında biz burada *a* değişkenini yok edip, *'istihza.com'* değerini içeren, *a* adlı başka bir değişken tanımladık. Bu durumu nasıl teyit edeceğinizi biliyorsunuz:

```
>>> a = 'istihza'
>>> id(a)

15063200

>>> a = a + '.com'
>>> id(a)

15067960
```

Burada id() fonksiyonunu kullanarak karakter dizilerinin kimliklerini sorguladık. Gördüğünüz gibi, isimleri aynı da olsa, aslında ortada iki farklı *a* değişkeni var. Kimlik numaralarının farklı olmasından anladığımıza göre, ilk başta tanımladığımız *a* değişkeni ile a = a + '.com' satırıyla oluşturduğumuz *a* değişkeni birbirinden farklı.

Bu arada, eğer istersek yukarıdaki değer atama işlemini, önceki bölümlerde öğrendiğimiz değer atama işleçleri yardımıyla kısaltabileceğimizi de biliyorsunuz:

```
>>> a += '.com'
```

İşte *ilk_metin* ile *ikinci_metin* değişkenleri arasındaki farklı harfleri yalnızca birer kez yazdırmak için kullandığımız kodlarda da yukarıdaki işlemi yaptık:

```
ilk_metin = "asdasfddgdhfjfdgdşfkgjdfklgşjdfklgjdfkghdfjghjklsdhajlsdhjkjhkhjjh"
ikinci_metin = "sdfsuıdoryeuıfsjkdfhdjklghjdfklruseldhfjlkdshfljskeeuf"

fark = ''

for s in ikinci_metin:
    if not s in ilk_metin:
        if not s in fark:
```

```
            fark += s
print(fark)
```

Gördüğünüz gibi, önce boş bir *fark* değişkeni oluşturduk. Daha sonra bu değişkene for döngüsü içinde yeni değerler atayabilmek (daha doğrusu atarmış gibi yapmak) için fark += s gibi bir kod kullandık. Böylece *for* döngüsünün her dönüşünde *s* adını verdiğimiz herbir öğeyi tek tek *fark* değişkenine yolladık. Böylece program sonunda elimizde, farklı öğeleri yalnızca birer kez içeren *fark* adlı bir değişken olmuş oldu. Dediğimiz gibi, ilk başta tanımladığımız boş *fark* değişkeni ile, program sonunda farklı değerleri içeren *fark* değişkeni aslında aynı değil. Yani biz ilk *fark* değişkenine döngünün her dönüşünde yeni bir öğe eklemek yerine, döngünün her dönüşünde yeni bir *fark* değişkeni oluşturmuş oluyoruz. Ama programın sonunda sanki *fark* değişkenine her defasında yeni bir değer atamışız gibi görünüyor ve bu da bizim işimizi görmemize yetiyor...

Programın başındaki ve sonundaki *fark* değişkenlerinin aslında birbirinden farklı olduğunu teyit etmek için şu kodları kullanabilirsiniz:

```
ilk_metin = "asdasfddgdhfjfdgdşfkgjdfklgşjdfklgjdfkghdfjghjklsdhajlsdhjkjhkhjjh"
ikinci_metin = "sdfsuıdoryeuıfsjkdfhdjklghjdfklruseldhfjlkdshfljskeeuf"

fark = ""
print("fark'ın ilk tanımlandığı zamanki kimlik numarası: ", id(fark))

for s in ikinci_metin:
    if not s in ilk_metin:
        if not s in fark:
            fark += s

print("fark'ın program sonundaki kimlik numarası: ", id(fark))
```

Gördüğünüz gibi, gerçekten de ortada iki farklı *fark* değişkeni var. Bu durumu id() fonksiyonu yardımıyla doğrulayabiliyoruz.

Peki bu bilginin bize ne faydası var?

Şimdilik şu kadarını söyleyelim: Eğer o anda muhatap olduğunuz bir veri tipinin mizacını, huyunu-suyunu bilmezseniz yazdığınız programlarda çok kötü sürprizlerle karşılaşabilirsiniz. Birkaç bölüm sonra başka veri tiplerini de öğrendikten sonra bu durumu daha ayrıntılı bir şekilde inceleyeceğiz.

Bu arada, tahmin edebileceğiniz gibi yukarıdaki for döngüsünü şöyle de yazabilirdik:

```
for s in ikinci_metin:
    if not s in ilk_metin and not s in fark:
        fark += s
```

Burada iki farklı if deyimini iki farklı satırda yazmak yerine, bu deyimleri *and* işleci ile birbirine

bağladık.

Bu örnek ile ilgili söyleyeceklerimiz şimdilik bu kadar. Gelin biz şimdi isterseniz bilgilerimizi pekiştirmek için başka bir örnek daha yapalım.

Dosyaların İçeriğini Karşılaştırma

Bir önceki örnekte karakter dizilerinin içeriğini nasıl karşılaştırabileceğimizi gösteren bir örnek vermiştik. Şimdi de, gerçek hayatta karşınıza çıkması daha olası bir durum olması bakımından, dosyaların içeriğini nasıl karşılaştıracağımıza dair bir örnek verelim.

Esasında karakter dizilerinin içeriğini birbirleriyle nasıl karşılaştırıyorsak, dosyaların içeriğini de benzer şekilde karşılaştırabiliriz. Mesela içeriği şu olan *isimler1.txt* adlı bir dosyamız olduğunu varsayalım:

```
Ahmet
Mehmet
Sevgi
Sinan
Deniz
Ege
Efe
Ferhat
Fırat
Zeynep
Hazan
Mahmut
Celal
Cemal
Özhan
Özkan
```

Yine içeriği şu olan bir de *isimler2.txt* adlı başka bir dosya daha olduğunu düşünelim:

```
Gürsel
Mehmet
Sevgi
Sami
Deniz
Ege
Efe
Ferhat
Fırat
Tülay
```

```
Derya
Hazan
Mahmut
Tezcan
Cemal
Özhan
Özkan
Özcan
Dilek
```

Amacımız bu iki dosyanın içeriğini karşılaştırıp, farklı öğeleri ortaya sermek. Dediğimiz gibi, bir önceki örnekte izlediğimiz yolu burada da takip edebiliriz. Dikkatlice bakın:

```
d1 = open("a1.txt").readlines()
d2 = open("a2.txt").readlines()

for i in d2:
    if not i in d1:
        print(i)

d1.close()
d2.close()
```

Gerçekten de mantığın bir önceki örnekle tamamen aynı olduğunu görüyorsunuz. Biz henüz Python'da dosyaların nasıl işleneceğini öğrenmedik, ama daha önce gördüğümüz open() fonksiyonu yardımıyla en azından dosyaları açabilecek kadar biliyoruz dosya işlemlerinin nasıl yürütüleceğini...

Burada farklı olarak readlines() adlı bir metot görüyoruz. Biz burada bu metodun ayrıntılarına inmeyeceğiz, ama şimdilik dosya içeriğinin satırlar halinde okunmasını sağladığını bilelim yeter.

Eğer yukarıdaki kod biraz karışık göründüyse gözünüze, bu kodları şöyle yazmak biraz daha anlaşılır olabilir:

```
d1 = open("a1.txt") # dosyayı açıyoruz
d1 = d1.readlines() # satırları okuyoruz

d2 = open("a2.txt")
d2 = d2.readlines()

for i in d2:
    if not i in d1:
        print(i)

d1.close()
```

```
d2.close()
```

Bu arada, eğer çıktıda Türkçe karakterleri düzgün görüntüleyemiyorsanız open() fonksiyonunun *encoding* adlı bir parametresi vasıtasıyla içeriği *UTF-8* olarak kodlayabilirsiniz:

```
d1 = open("a1.txt", encoding="utf-8").readlines()
d2 = open("a2.txt", encoding="utf-8").readlines()

for i in d1:
    if not i in d2:
        print(i)

d1.close()
d2.close()
```

Bu şekilde Türkçe karakterleri düzgün bir şekilde görüntüleyebiliyor olmanız lazım.

Yukarıdaki örneklerde bir içerik karşılaştırması yapıp, **farklı** öğeleri ayıkladık. Aynı şekilde **benzer** öğeleri ayıklamak da mümkündür. Bu işlemin nasıl yapılacağını az çok tahmin ettiğinizi zannediyorum:

```
d1 = open("a1.txt", encoding="utf-8").readlines()
d2 = open("a2.txt", encoding="utf-8").readlines()

for i in d1:
    if i in d2:
        print(i)

d1.close()
d2.close()
```

Burada bir öncekinden farklı olarak if not i in d2 kodu yerine, doğal olarak, if i in d2 kodunu kullandığımıza dikkat edin.

Dosyalar üzerinde yaptığımız işlemleri tamamladıktan sonra close() metodu ile bunları kapatmayı unutmuyoruz:

```
d1.close()
d2.close()
```

Karakter Dizisindeki Karakterleri Sayma

Yukarıdaki örneklerde içerik karşılaştırmaya ilişkin birkaç örnek verdik. Şimdi yine bilgilerimizi pekiştirmek için başka bir konuya ilişkin örnekler verelim.

Mesela elimizde şöyle bir metin olduğunu varsayalım:

Bu programlama dili Guido Van Rossum adlı Hollandalı bir programcı tarafından 90'lı yılların başında geliştirilmeye başlanmıştır. Çoğu insan, isminin Python olmasına aldanarak, bu programlama dilinin, adını piton yılanından aldığını düşünür. Ancak zannedildiğinin aksine bu programlama dilinin adı piton yılanından gelmez. Guido Van Rossum bu programlama dilini, The Monty Python adlı bir İngiliz komedi grubunun, Monty Python's Flying Circus adlı gösterisinden esinlenerek adlandırmıştır. Ancak her ne kadar gerçek böyle olsa da, Python programlama dilinin pek çok yerde bir yılan figürü ile temsil edilmesi neredeyse bir gelenek halini almıştır.

Yapmamız gereken bir istatistik çalışması gereğince bu metinde her harfin kaç kez geçtiğini hesaplamanız gerekiyor.

Bunun için şöyle bir program yazabiliriz:

```
metin = """Bu programlama dili Guido Van Rossum adlı Hollandalı bir programcı
tarafından 90'lı yılların başında geliştirilmeye başlanmıştır. Çoğu insan,
isminin Python olmasına aldanarak, bu programlama dilinin, adını piton
yılanından aldığını düşünür. Ancak zannedildiğinin aksine bu programlama dilinin
adı piton yılanından gelmez. Guido Van Rossum bu programlama dilini, The Monty
Python adlı bir İngiliz komedi grubunun, Monty Python's Flying Circus adlı
gösterisinden esinlenerek adlandırmıştır. Ancak her ne kadar gerçek böyle olsa
da, Python programlama dilinin pek çok yerde bir yılan figürü ile temsil
edilmesi neredeyse bir gelenek halini almıştır."""

harf = input("Sorgulamak istediğiniz harf: ")

sayı = ''

for s in metin:
    if harf == s:
        sayı += harf

print(len(sayı))
```

Burada öncelikle metnimizi bir değişken olarak tanımladık. Ardından da kullanıcıya hangi harfi sorgulamak istediğini sorduk.

Bu kodlarda tanımladığımız *sayı* adlı değişken, sorgulanan harfi, metinde geçtiği sayıda içinde barındıracaktır. Yani mesela metin *5* tane *a* harfi varsa *sayı* değişkeninin değeri *aaaaa* olacaktır.

Sonraki satırlarda for döngümüzü tanımlıyoruz:

```
for s in metin:          # metin içinde 's' adını verdiğimiz herbir öğe için
    if harf == s:        # eğer kullanıcıdan gelen harf 's' ile aynıysa
        sayı += harf     # kullanıcıdan gelen bu harfi sayı değişkenine yolla
```

Dediğimiz gibi, *sayı* değişkeni, sorgulanan harfi, metinde geçtiği sayıda barındırıyor. Dolayısıyla bir harfin metinde kaç kez geçtiğini bulmak için *sayı* değişkeninin uzunluğunu yazdırmamız yeterli olacaktır:

```
print(len(sayı))
```

Dilerseniz yukarıdaki programı yazmak için daha farklı bir mantık da kullanabilirsiniz. Dikkatlice bakın:

```
metin = """Bu programlama dili Guido Van Rossum adlı Hollandalı bir programcı
tarafından 90'lı yılların başında geliştirilmeye başlanmıştır. Çoğu insan,
isminin Python olmasına aldanarak, bu programlama dilinin, adını piton
yılanından aldığını düşünür. Ancak zannedildiğinin aksine bu programlama dilinin
adı piton yılanından gelmez. Guido Van Rossum bu programlama dilini, The Monty
Python adlı bir İngiliz komedi grubunun, Monty Python's Flying Circus adlı
gösterisinden esinlenerek adlandırmıştır. Ancak her ne kadar gerçek böyle olsa
da, Python programlama dilinin pek çok yerde bir yılan figürü ile temsil
edilmesi neredeyse bir gelenek halini almıştır."""

harf = input("Sorgulamak istediğiniz harf: ")

sayı = 0

for s in metin:
    if harf == s:
        sayı += 1

print(sayı)
```

Burada *sayı* değişkeninin ilk değeri *0* olarak belirledik. Döngü içinde de, sorgulanan harfin metin içinde her geçişinde *sayı* değişkeninin değerini *1* sayı artırdık. Dolayısıyla sorgulanan harfin metinde kaç kez geçtiğini bulmak için *sayı* değişkeninin son değerini yazdırmamız yeterli oldu.

Dosya içindeki Karakterleri Sayma

Dilerseniz bir önceki örnekte kullandığımız metnin program içinde bir değişken değil de, mesela bir dosyadan okunan bir metin olduğunu varsayalım şimdi:

```
hakkında = open("hakkında.txt", encoding="utf-8")
```

```
harf = input("Sorgulamak istediğiniz harf: ")

sayı = 0

for karakter_dizisi in hakkında:
    for karakter in karakter_dizisi:
        if harf == karakter:
            sayı += 1
print(sayı)

hakkında.close()
```

Burada yaptığımız ilk iş elbette dosyamızı açmak oldu:

```
hakkında = open("a1.txt", encoding="utf-8")
```

Bu komutla, *hakkında.txt* adlı dosyayı *UTF-8* kodlaması ile açtık. Daha sonra kullanıcıya, sorgulamak istediği harfi soruyoruz:

```
harf = input("Sorgulamak istediğiniz harf: ")
```

Ardından da sorgulanan harfin dosyada kaç kez geçtiği bilgisini tutacak olan *sayı* adlı bir değişken tanımlıyoruz:

```
sayı = 0
```

Sıra geldi for döngümüzü tanımlamaya:

```
for karakter_dizisi in hakkında:
    for karakter in karakter_dizisi:
        if harf == karakter:
            sayı += 1
```

Bu döngüyü anlamakta bir miktar zorlanmış olabilirsiniz. Her zaman söylediğimiz gibi, Python'da bir kod parçasını anlamanın en iyi yöntemi, gerekli yerlere print() fonksiyonları yerleştirerek, programın verdiği çıktıları incelemektir:

```
for karakter_dizisi in hakkında:
    print(karakter_dizisi)
    #for karakter in karakter_dizisi:
    #    if harf == karakter:
    #        sayı += 1
```

Gördüğünüz gibi, ilk for döngüsünün hemen sonrasına bir print() fonksiyonu yerleştirerek bu döngünün verdiği çıktıları inceliyoruz. Bu arada, amacımıza hizmet etmeyen satırları da yorum içine

alarak etkisizleştirdiğimize dikkat edin.

Çıktıya baktığımız zaman, şöyle bir durumla karşılaşıyoruz:

Bu programlama dili Guido Van Rossum adlı Hollandalı bir programcı tarafından 90'lı yılların başında geliştirilmeye başlanmıştır. Çoğu insan, isminin Python olmasına aldanarak, bu programlama dilinin, adını piton yılanından aldığını düşünür. Ancak zannedildiğinin aksine bu programlama dilinin adı piton yılanından gelmez. Guido Van Rossum bu programlama dilini, The Monty Python adlı bir İngiliz komedi grubunun, Monty Python's Flying Circus adlı gösterisinden esinlenerek adlandırmıştır. Ancak her ne kadar gerçek böyle olsa da, Python programlama dilinin pek çok yerde bir yılan figürü ile temsil edilmesi neredeyse bir gelenek halini almıştır.

Burada herbir satır ayrı bir karakter dizisidir. Eğer herbir satırın ayrı bir karakter dizisi olduğunu daha net bir şekilde görmek istiyorsanız repr() adlı özel bir fonksiyondan yararlanabilirsiniz:

```
for karakter_dizisi in hakkında:
    print(repr(karakter_dizisi))
    #for karakter in karakter_dizisi:
    #    if harf == karakter:
    #        sayı += 1
```

Bu kodlar bu kez şöyle bir çıktı verir:

```
'Bu programlama dili Guido Van Rossum adlı Hollandalı bir programcı\n'
'tarafından 90'lı yılların başında geliştirilmeye başlanmıştır. Çoğu insan,\n'
'isminin Python olmasına aldanarak, bu programlama dilinin, adını piton\n'
'yılanından aldığını düşünür. Ancak zannedildiğinin aksine bu programlama dilinin\n'
'adı piton yılanından gelmez. Guido Van Rossum bu programlama dilini, The Monty\n'
'Python adlı bir İngiliz komedi grubunun, Monty Python's Flying Circus adlı\n'
'gösterisinden esinlenerek adlandırmıştır. Ancak her ne kadar gerçek böyle olsa\n'
'da, Python programlama dilinin pek çok yerde bir yılan figürü ile temsil\n'
'edilmesi neredeyse bir gelenek halini almıştır.'
```

Bu çıktıya çok dikkatlice bakın. repr() fonksiyonu sayesinde Python'ın alttan alta neler çevirdiğini

bariz bir biçimde görüyoruz. Karakter dizisinin başlangıç ve bitişini gösteren tırnak işaretleri ve *\n* kaçış dizilerinin görünür vaziyette olması sayesinde herbir satırın ayrı bir karakter dizisi olduğunu daha net bir şekilde görebiliyoruz.

Biz yazdığımız kodlarda, kullanıcıdan bir harf girmesini istiyoruz. Kullandığımız algoritma gereğince bu harfi metindeki karakter dizileri içinde geçen herbir karakterle tek tek karşılaştırmamız gerekiyor. input() metodu aracılığıyla kullanıcıdan tek bir karakter alıyoruz. Kullandığımız for döngüsü ise bize bir karakter yerine her satırda bir karakter dizisi veriyor. Dolayısıyla mesela kullanıcı 'a' harfini sorgulamışsa, ilk for döngüsü bu harfin karşısına *'Bu programlama dili Guido Van Rossum adlı Hollandalı bir programcın'* adlı karakter dizisini çıkaracaktır. Dolayısıyla bizim bir seviye daha alta inerek, ilk for döngüsünden elde edilen değişken üzerinde başka bir for döngüsü daha kurmamız gerekiyor. Bu yüzden şöyle bir kod yazıyoruz:

```
for karakter_dizisi in hakkında:
    for karakter in karakter_dizisi:
            ...
```

Böylece iç içe iki for döngüsü oluşturmuş oluyoruz. İsterseniz bu anlattığımız şeyleri daha net görmek için yine print() fonksiyonundan yararlanabilirsiniz:

```
hakkında = open("hakkında.txt", encoding="utf-8")

harf = input("Sorgulamak istediğiniz harf: ")

sayı = 0

for karakter_dizisi in hakkında:
    for karakter in karakter_dizisi:
        print(karakter)
#         if harf == karakter:
#             sayı += 1
#print(sayı)
```

karakter değişkenin değerini ekrana yazdırarak Python'ın alttan alta neler çevirdiğini daha net görebiliyoruz.

Kodların geri kalanında ise, kullanıcının sorguladığı harfin, for döngüsü ile üzerinden geçtiğimiz *karakter_dizisi* adlı değişken içindeki karakterlerle eşleşip eşleşmediğini denetliyoruz. Eğer eşleşiyorsa, her eşleşmede *sayı* değişkeninin derini *1* sayı artırıyoruz. Böylece en elimizde sorgulanan harfin metin içinde kaç kez geçtiği bilgisi olmuş oluyor.

Son olarak da, ilk başta açtığımız dosyayı kapatıyoruz:

```
hakkında.close()
```

Nihayet bir konunun daha sonuna ulaştık. Döngüler, ve döngülerle ilişkili araçları da epey ayrıntılı bir şekilde incelediğimize göre gönül rahatlığıyla bir sonraki konuya geçebiliriz.

başa dön

Hata Yakalama

Şimdiye kadar yazdığımız bütün programlar, dikkat ettiyseniz tek bir ortak varsayım üzerine kurulu. Buna göre biz, yazdığımız programın kullanıcı tarafından nasıl kullanılmasını istiyorsak, her zaman o şekilde kullanılacağını varsayıyoruz. Örneğin sayıları toplayan bir program yazdığımızda, kullanıcının her zaman sayı değerli bir veri gireceğini düşünüyoruz. Ancak bütün iyi niyetimize rağmen, yazdığımız programlarda işler her zaman beklediğimiz gibi gitmeyebilir. Örneğin, dediğimiz gibi, yazdığımız programı, kullanıcının bir sayı girmesi temeli üzerine kurgulamışsak, kullanıcının her zaman sayı değerli bir veri gireceğinden emin olamayız

Mesela şöyle bir program yazdığımızı düşünün:

```
veri1 = input("Karekökünü hesaplamak istediğiniz sayı: ")
karekök = int(veri1) ** 0.5

print(veri1, "sayısının karekökü: ", karekök)

veri2 = input("Karesini hesaplamak istediğiniz sayı: ")
kare = int(veri2) ** 2

print(veri2, "sayısının karesi: ", kare)
```

Bu kodlardaki sorunu anlamaya çalışmadan önce dilerseniz kodları şöyle bir inceleyelim.

Gördüğünüz gibi, burada kullanıcının gireceği sayılara göre karekök ve kare alma işlemleri yapıyoruz. Bu kodlarda gördüğümüz ** işleci yardımıyla bir sayının herhangi bir kuvvetini hesaplayabileceğimizi biliyorsunuz. Mesela 21^7'nin kaç ettiğini hesaplamak için ** işlecini kullanabiliyoruz:

```
>>> 21 ** 7

1801088541
```

Yine bildiğiniz gibi, bu işleçten, bir sayının karesini hesaplamak için de yararlanabiliyoruz. Çünkü neticede bir sayının karesi, o sayının 2. kuvvetidir:

```
>>> 12 ** 2

144
```

Aynı şekilde, eğer bir sayının, 0.5'inci kuvvetini hesaplarsak o sayının karekökünü bulmuş oluyoruz. (Bu bilgileri önceki konulardan hatırlıyor olmalısınız):

```
>>> 144 ** 0.5
```

```
12
```

Kodlarımızı incelediğimize göre, bu programdaki aksaklıkları irdelemeye başlayabiliriz.

Bu program, kullanıcı sayı değerli bir veri girdiği müddetçe sorunsuz bir şekilde çalışacaktır. Peki ya kullanıcı sayı değerli bir veri yerine başka bir şey girerse ne olur?

Örneğin kullanıcı yukarıdaki programa bir sayı yerine, (bilerek veya bilmeyerek) içinde harf barındıran bir veri girerse şuna benzer bir hata alır:

```
Traceback (most recent call last):
  File "deneme.py", line 2, in <module>
    karekök = int(veri1) ** 0.5
ValueError: invalid literal for int() with base 10: 'fds'
```

Yazdığınız programların bu tür hatalar vermesi normaldir. Ancak son kullanıcı açısından düşündüğümüzde, kullanıcının yukarıdaki gibi bir hata mesajı görmesi yerine, hatanın neden kaynaklandığını ya da neyi yanlış yaptığını daha açık bir şekilde ifade eden bir mesaj alması çok daha mantıklı olacaktır. Zira yukarıdaki hata mesajı programcılar açısından anlamlı olabilir, ancak son kullanıcı açısından büsbütün anlaşılmazdır!

Dediğimiz gibi, programınızın çalışma esnasında bu tür hatalar vermesi normal. Çünkü yapmaya çalıştığınız işlem, kullanıcının belli tipte bir veri girmesine bağlı. Burada sizin bir programcı olarak göreviniz, yazdığınız programın çalışma esnasında vermesi muhtemel hataları önceden kestirip, programınızda buna göre bazı önlemler almanızdır. İşte biz de bu bölümde bu önlemleri nasıl alacağımızı anlamaya çalışacağız.

Hata Türleri

Biz bu bölümde hatalardan bahsedeceğimizi söylemiştik. Ancak her şeyden önce ‘hata’ kavramının çok boyutlu olduğunu hatırlatmakta fayda var. Özellikle programcılık açısından hata kavramının ne anlama geldiğini biraz incelememiz gerekiyor.

Biz bu bölümde hataları üç farklı başlık altında ele alacağız:

1. Programcı Hataları (*Error*)
2. Program Kusurları (*Bug*)
3. İstisnalar (*Exception*)

Öncelikle programcı hatalarından bahsedelim.

Programcıdan kaynaklanan hatalar doğrudan doğruya programı yazan kişinin dikkatsizliğinden ötürü ortaya çıkan bariz hatalardır. Örneğin şu kod bir programcı hatası içerir:

```
>>> print "Merhaba Python!"
```

Bu kodu çalıştırdığınızda şöyle bir hata mesajı görürsünüz:

```
>>> print "Merhaba Python!"

File "<stdin>", line 1
  print "Merhaba Python!"
                ^
SyntaxError: invalid syntax
```

Bu hata mesajında bizi ilgilendiren kısım son cümlede yer alıyor: SyntaxError, yani Söz dizimi hatası.

Bu hatalar, programlama diline ilişkin bir özelliğin yanlış kullanımından veya en basit şekilde programcının yaptığı yazım hatalarından kaynaklanır. Programcının hataları genellikle SyntaxError şeklinde ortaya çıkar. Bu hatalar çoğunlukla programcı tarafından farkedilir ve program kullanıcıya ulaşmadan önce programcı tarafından düzeltilir. Bu tür hataların tespiti diğer hatalara kıyasla kolaydır. Çünkü bu tür hatalar programınızın çalışmasını engellediği için bunları farketmemek pek mümkün değildir...

Program kusurları, başka bir deyişle *bug*'lar ise çok daha karmaşıktır. Kusurlu programlar çoğu zaman herhangi bir hata vermeden çalışır. Ancak programın ürettiği çıktılar beklediğiniz gibi değildir. Örneğin yazdığınız programda bir formül hatası yapmış olabilirsiniz. Bu durumda programınız hiçbir şey yokmuş gibi çalışır, ancak formül hatalı olduğu için hesaplamaların sonuçları yanlıştır. Örneğin daha önceki derslerimizde yazdığımız şu program yukarıdaki gibi bir kusur içerir:

```
sayı1 = input("Toplama işlemi için ilk sayıyı girin: ")
sayı2 = input("Toplama işlemi için ikinci sayıyı girin: ")

print(sayı1, "+", sayı2, "=", sayı1 + sayı2)
```

Bu programda kullanıcı veri girdiği zaman, programımız toplama işlemi değil karakter dizisi birleştirme işlemi yapacaktır. Böyle bir program çalışma sırasında hata vermeyeceği için buradaki sorunu tespit etmek, özellikle büyük programlarda çok güçtür. Yani sizin düzgün çalıştığını zannettiğiniz program aslında gizliden gizliye bir *bug* barındırıyor olabilir.

Aynı şekilde, mesela eval() fonksiyonunun dikkatsizce kullanıldığı programlar da güvenlik açısından kusurludur. Yani bu tür programlar bir güvenlik kusuru (*security bug* veya *security flaw*) barındırır.

Dediğimiz gibi, program kusurları çok boyutlu olup, burada anlattığımızdan çok daha karmaşıktır.

Gelelim üçüncü kategori olan istisnalara (*exceptions*)...

İstisnalar, adından da az çok anlaşılacağı gibi, bir programın çalışması sırasında ortaya çıkan, normalden farklı, istisnai durumlardır. Örneğin şu programa bakalım:

```
ilk_sayı = input("ilk sayı: ")
ikinci_sayı = input("ikinci sayı: ")

ilk_sayı = int(ilk_sayı)
ikinci_sayı = int(ikinci_sayı)

print(ilk_sayı, "/", ikinci_sayı, "=", ilk_sayı / ikinci_sayı)
```

Burada ilk sayıyı ikinci sayıya bölen bir program yazdık. Bu program her türlü bölme işlemini yapabilir. Ama burada hesaba katmamız gereken iki şey var:

1. Kullanıcı sayı yerine, sayı değerli olmayan bir veri tipi girebilir. Mesela ilk sayıya karşılık *23*, ikinci sayıya karşılık 'fdsfd' gibi bir şey yazabilir.
2. Kullanıcı bir sayıyı *0*'a bölmeye çalışabilir. Mesela ilk sayıya karşılık *23*, ikinci sayıya karşılık *0* yazabilir.

İlk durumda programımız şöyle bir hata verir:

```
ilk sayı: 23
ikinci sayı: fdsfd
Traceback (most recent call last):
  File "deneme.py", line 5, in <module>
    ikinci_sayı = int(ikinci_sayı)
ValueError: invalid literal for int() with base 10: 'fdsfd'
```

Buradaki sorun, sayı değerli olmayan bir verinin, int() fonksiyonu aracılığıyla sayıya çevrilmeye çalışılıyor olması.

İkinci durumda ise programımız şöyle bir hata verir:

```
ilk sayı: 23
ikinci sayı: 0
Traceback (most recent call last):
  File "deneme.py", line 7, in <module>
    print(ilk_sayı, "/", ikinci_sayı, "=", ilk_sayı / ikinci_sayı)
ZeroDivisionError: division by zero
```

Buradaki sorun ise, bir sayının *0*'a bölünmeye çalışılıyor olması. Matematikte sayılar *0*'a bölünemez...

İşte bu iki örnekte gördüğümüz ValueError ve ZeroDivisionError birer istisnadır. Yani kullanıcıların, kendilerinden sayı beklenirken sayı değerli olmayan veri girmesi veya bir sayıyı 0'a bölmeye çalışması istisnai birer durumdur ve yazdığımız programların *exception* (istisna) üretmesine yol açar.

Böylece hata (*error*), kusur (*bug*) ve istisna (*exception*) arasındaki farkları şöyle bir gözden geçirmiş olduk. Yalnız burada şunu söylemekte yarar var: Bu üç kavram arasındaki fark belli belirsizdir. Yani bu kavramların çoğu yerde birbirlerinin yerine kullanıldığını da görebilirsiniz. Örneğin *exception* kavramı için Türkçe'de çoğu zaman 'hata' kelimesini kullanıyoruz. Zaten dikkat ederseniz bu bölümün başlığı da 'İstisna Yakalama' değil, 'Hata Yakalama'dır. Aynı şekilde, İngilizcede de bu kavramların çoğu yerde birbirleri yerine kullanıldığını görebilirsiniz. Dolayısıyla, konuya karşı özel bir ilginiz yoksa, hata, kusur ve istisna kavramlarını birbirinden ayırmak için kendinizi zorlamanıza gerek yok. Bu üç kavram çoğu zaman birbirinin yerine kullanılıyor da olsa, aslında aralarında bazı farklar olduğunu öğrenmişseniz bu bölüm amacına ulaşmış demektir.

Konuyla ilgili temel bilgileri edindiğimize göre asıl meseleye geçebiliriz...

try... except...

Bir önceki bölümde hatalardan ve hataları yakalamaktan söz ettik. Peki bu hataları nasıl yakalayacağız?

Python'da hata yakalama işlemleri için try... except... bloklarından yararlanılır. Hemen bir örnek verelim:

```
ilk_sayı    = input("ilk sayı: ")
ikinci_sayı = input("ikinci sayı: ")

try:
    sayı1 = int(ilk_sayı)
    sayı2 = int(ikinci_sayı)
    print(sayı1, "/", sayı2, "=", sayı1 / sayı2)
except ValueError:
    print("Lütfen sadece sayı girin!")
```

Biliyoruz ki, bir veriyi sayıya dönüştürmek istediğimizde eğer kullanıcı sayı değerli bir veri yerine harf değerli bir veri girerse programımız çöker. Dolayısıyla int(ilk_sayı) ve int(ikinci_sayı) kodları, kullanıcının gireceği veri türüne göre hata üretme potansiyeline sahiptir. O yüzden, burada hata vereceğini bildiğimiz o kodları try bloğu içine aldık.

Yine bildiğimiz gibi, veri dönüştürme işlemi sırasında kullanıcının uygun olmayan bir veri girmesi halinde üretilecek hata bir ValueError'dır. Dolayısıyla except bloğu içine yazacağımız hata türünün adı da ValueError olacaktır. O yüzden ValueError adlı hatayı yakalayabilmek için şu satırları yazdık:

```
except ValueError:
    print("Lütfen sadece sayı girin!")
```

Burada bu kodlarla Python'a şu emri vermiş olduk:

Eğer try bloğu içinde belirtilen işlemler sırasında bir ValueError ile karşılaşırsan bunu görmezden gel ve normal şartlar altında kullanıcıya göstereceğin hata mesajını gösterme. Onun yerine kullanıcıya Lütfen sadece sayı girin! uyarısını göster.

Yukarıda Türkçeye çevirdiğimiz emri Pythoncada nasıl ifade ettiğimize dikkat edin. Temel olarak şöyle bir yapıyla karşı karşıyayız:

```
try:
    hata verebileceğini bildiğimiz kodlar
except HataAdı:
    hata durumunda yapılacak işlem
```

Gelin isterseniz bir örnek daha verelim.

Hatırlarsanız bir sayının *0*'a bölünmesinin mümkün olmadığını, böyle bir durumda programımızın hata vereceğini söylemiştik. Bu durumu teyit etmek için etkileşimli kabukta şu kodu deneyebilirsiniz:

```
>>> 2 / 0
```

Bu kod şöyle bir hata mesajı verecektir:

```
>>> 2 / 0

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Daha önce de söylediğimiz gibi, bu hata mesajında bizi ilgilendiren kısım ZeroDivisionError. Demek ki bir sayı *0*'a bölündüğünde Python ZeroDivisionError veriyormuş. O halde şöyle bir kod yazabiliriz:

```
ilk_sayı    = input("ilk sayı: ")
ikinci_sayı = input("ikinci sayı: ")

try:
    sayı1 = int(ilk_sayı)
    sayı2 = int(ikinci_sayı)
    print(sayı1, "/", sayı2, "=", sayı1 / sayı2)
except ZeroDivisionError:
    print("Bir sayıyı 0'a bölemezsiniz!")
```

Gördüğünüz gibi, Python'ın ZeroDivisionError vereceğini bildiğimiz durumlara karşı bu hata türünü yakalama yoluna gidiyoruz. Böylece kullanıcıya anlamsız ve karmaşık hata mesajları göstermek ve daha da kötüsü, programımızın çökmesine sebep olmak yerine daha anlaşılır mesajlar üretiyoruz.

Yukarıdaki kodlarda özellikle bir nokta dikkatinizi çekmiş olmalı: Dikkat ederseniz yukarıdaki kodlar aslında bir değil iki farklı hata üretme potansiyeline sahip. Eğer kullanıcı sayı değerli veri yerine harf değerli bir veri girerse ValueError, eğer bir sayıyı *0*'a bölmeye çalışırsa da ZeroDivisionError hatası alıyoruz. Peki aynı kodlarda iki farklı hata türünü nasıl yakalayacağız?

Çok basit:

```
ilk_sayı    = input("ilk sayı: ")
ikinci_sayı = input("ikinci sayı: ")

try:
    sayı1 = int(ilk_sayı)
    sayı2 = int(ikinci_sayı)
    print(sayı1, "/", sayı2, "=", sayı1 / sayı2)
except ZeroDivisionError:
    print("Bir sayıyı 0'a bölemezsiniz!")
except ValueError:
    print("Lütfen sadece sayı girin!")
```

Gördüğünüz gibi çözüm gayet mantıklı. Birden fazla hata türü üreteceğini bildiğimiz kodları yine tek bir try bloğu içine alıyoruz. Hata türlerini ise ayrı except blokları içinde ele alıyoruz.

Bir program yazarken, en iyi yaklaşım, yukarıda yaptığımız gibi, her hata türü için kullanıcıya ayrı bir uyarı mesajı göstermektir. Böylece kullanıcılarımız bir hatayla karşılaştıklarında sorunu nasıl çözebilecekleri konusunda en azından bir fikir sahibi olabilirler.

Dediğimiz gibi, her hata için ayrı bir mesaj göstermek en iyisidir. Ama tabii dilerseniz hata türlerini gruplayıp hepsi için tek bir hata mesajı göstermeyi de tercih edebilirsiniz. Bunu nasıl yapacağımızı görelim:

```
ilk_sayı    = input("ilk sayı: ")
ikinci_sayı = input("ikinci sayı: ")

try:
    sayı1 = int(ilk_sayı)
    sayı2 = int(ikinci_sayı)
    print(sayı1, "/", sayı2, "=", sayı1 / sayı2)
except (ValueError, ZeroDivisionError):
    print("Bir hata oluştu!")
```

Gördüğünüz gibi, burada ValueError ve ZeroDivisionError adlı hata türlerini tek bir parantez içinde topladık. Burada dikkat edeceğimiz nokta, bu hata türlerini gruplarken bunları parantez içine almak ve birbirlerinden virgülle ayırmaktır.

Bu arada, gördüğünüz gibi yukarıdaki programlar sadece bir kez çalışıp kapanıyor. Ama biz bu

programları tekrar tekrar nasıl çalıştırabileceğimizi gayet iyi biliyoruz:

```
while True:
    ilk_sayı = input("ilk sayı (Programdan çıkmak için q tuşuna basın): ")

    if ilk_sayı == "q":
        break

    ikinci_sayı = input("ikinci sayı: ")

    try:
        sayı1 = int(ilk_sayı)
        sayı2 = int(ikinci_sayı)
        print(sayı1, "/", sayı2, "=", sayı1 / sayı2)
    except (ValueError, ZeroDivisionError):
        print("Bir hata oluştu!")
        print("Lütfen tekrar deneyin!")
```

Python'da hata yakalamanın en yaygın yolu yukarıda gösterdiğimiz gibi kodları try... except blokları içine almaktır. Programcılık maceranızın büyük bölümünde bu yapıyı kullanacaksınız. Ama bazen, karşı karşıya olduğunuz duruma veya ihtiyacınıza göre try... except bloklarının farklı varyasyonlarını kullanmanız gerekebilir. İşte şimdi biz de bu farklı varyasyonların neler olduğunu incelemeye çalışacağız.

try... except... as...

Bildiğiniz gibi, Python bir programın çalışması esnasında hata üretirken çıktıda hata türünün adıyla birlikte kısa bir hata açıklaması veriyor. Yani mesela şöyle bir çıktı üretiyor:

```
ValueError: invalid literal for int() with base 10: 'f'
```

Burada 'ValueError' hata türünün adı, 'invalid literal for int() with base 10: 'f'' ise hatanın açıklamasıdır. Eğer istersek, yazdığımız programda bu hata açıklamasına erişebiliriz. Dikkatlice bakın:

```
ilk_sayı    = input("ilk sayı: ")
ikinci_sayı = input("ikinci sayı: ")

try:
    sayı1 = int(ilk_sayı)
    sayı2 = int(ikinci_sayı)
    print(sayı1, "/", sayı2, "=", sayı1 / sayı2)
except ValueError as hata:
    print(hata)
```

Bu programı çalıştırıp sayı değerli olmayan bir veri girersek hata çıktısı şöyle olacaktır:

```
invalid literal for int() with base 10: 'f'
```

Gördüğünüz gibi, bu defa çıktıda hata türünün adı (ValueError) görünmüyor. Onun yerine sadece hata açıklaması var.

Diyelim ki kullanıcıya olası bir hata durumunda hem kendi yazdığınız hata mesajını, hem de özgün hata mesajını göstermek istiyorsunuz. İşte yukarıdaki yapı böyle durumlarda işe yarayabilir:

```
ilk_sayı    = input("ilk sayı: ")
ikinci_sayı = input("ikinci sayı: ")

try:
    sayı1 = int(ilk_sayı)
    sayı2 = int(ikinci_sayı)
    print(sayı1, "/", sayı2, "=", sayı1 / sayı2)
except ValueError as hata:
    print("Sadece sayı girin!")
    print("orijinal hata mesajı: ", hata)
```

Bu arada, biraz önce yaptığımız gibi, hata türlerini grupladığınızda da bu yöntemi kullanabilirsiniz:

```
ilk_sayı    = input("ilk sayı: ")
ikinci_sayı = input("ikinci sayı: ")

try:
    sayı1 = int(ilk_sayı)
    sayı2 = int(ikinci_sayı)
    print(sayı1, "/", sayı2, "=", sayı1 / sayı2)
except (ValueError, ZeroDivisionError) as hata:
    print("Bir hata oluştu!")
    print("orijinal hata mesajı: ", hata)
```

Burada except falancaHata as filanca yapısını kullanarak falancaHata'yı *filanca* olarak isimlendiriyor ve daha sonra bu ismi istediğimiz gibi kullanabiliyoruz. Böylece bütün hata türleri için hem kendi yazdığınız mesajı görüntüleyebiliyor, hem de özgün hata mesajını da çıktıya eklediğimiz için, kullanıcıya hata hakkında en azından bir fikir sahibi olma imkanı vermiş oluyoruz.

try... except... else...

Daha önce de dediğimiz gibi, Python'da hata yakalama işlemleri için çoğunlukla try... except... bloklarını bilmek yeterli olacaktır. İşlerimizin büyük kısmını sadece bu blokları kullanarak halledebiliriz. Ancak Python bize bu konuda, zaman zaman işimize yarayabilecek başka araçlar da

sunmaktadır. İşte try... except... else... blokları da bu araçlardan biridir. Bu bölümde kısaca bu blokların ne işe yaradığından söz edeceğiz.

Öncelikle try... except... else... bloğunun ne işe yaradığına bakalım. Esasında biz bu else deyimini daha önce de 'koşullu ifadeler' konusunu işlerken görmüştük. Buradaki kullanımı da zaten hemen hemen aynıdır. Diyelim ki elimizde şöyle bir şey var:

```
try:
    bölünen = int(input("bölünecek sayı: "))
    bölen = int(input("bölen sayı: "))
    print(bölünen/bölen)
except ValueError:
    print("hata!")
```

Burada eğer kullanıcı sayı yerine harf girerse ValueError hatası alırız. Bu hatayı except ValueError: ifadesiyle yakalıyoruz ve hata verildiğinde kullanıcıya bir mesaj göstererek programımızın çökmesini engelliyoruz. Ama biliyoruz ki, bu kodları çalıştırdığımızda Python'ın verebileceği tek hata ValueError değildir. Eğer kullanıcı bir sayıyı 0'a bölmeye çalışırsa Python ZeroDivisionError adlı hatayı verecektir. Dolayısıyla bu hatayı da yakalamak için şöyle bir şey yazabiliriz:

```
try:
    bölünen = int(input("bölünecek sayı: "))
    bölen = int(input("bölen sayı: "))
    print(bölünen/bölen)
except ValueError:
    print("Lütfen sadece sayı girin!")
except ZeroDivisionError:
    print("Bir sayıyı 0'a bölemezsiniz!")
```

Bu şekilde hem ValueError hatasını hem de ZeroDivisionError hatasını yakalamış oluruz. Bu kodların özelliği, except... bloklarının tek bir try... bloğunu temel almasıdır. Yani biz burada bütün kodlarımızı tek bir try... bloğu içine tıkıştırıyoruz. Bu blok içinde gerçekleşen hataları da daha sonra tek tek except... blokları yardımıyla yakalıyoruz. Ama eğer biz istersek bu kodlarda verilebilecek hataları gruplamayı da tercih edebiliriz:

```
try:
    bölünen = int(input("bölünecek sayı: "))
    bölen = int(input("bölen sayı: "))
except ValueError:
    print("Lütfen sadece sayı girin!")
else:
    try:
```

```
        print(bölünen/bölen)
    except ZeroDivisionError:
        print("Bir sayıyı 0'a bölemezsiniz!")
```

Burada yaptığımız şey şu: İlk try... except... bloğu yardımıyla öncelikle int(input()) fonksiyonu ile kullanıcıdan gelecek verinin sayı olup olmadığını denetliyoruz. Ardından bir else... bloğu açarak, bunun içinde ikinci try... except... bloğumuzu devreye sokuyoruz. Burada da bölme işlemini gerçekleştiriyoruz. Kullanıcının bölme işlemi sırasında *0* sayısını girmesi ihtimaline karşı da except ZeroDivisionError ifadesi yardımıyla olası hatayı göğüslüyoruz. Bu şekilde bir kodlamanın bize getireceği avantaj, hatalar üzerinde belli bir kontrol sağlamamıza yardımcı olmasıdır. Yukarıdaki kodlar sayesinde hatalara bir nevi 'teker teker gelin!' mesajı vermiş oluyoruz. Böylelikle her blok içinde sadece almayı beklediğimiz hatayı karşılıyoruz. Mesela yukarıda ilk try... bloğu içindeki dönüştürme işlemi yalnızca ValueError hatası verebilir. else: bloğundan sonraki try... bloğunda yer alan işlem ise ancak ZeroDivisionError verecektir. Biz yukarıda kullandığımız yapı sayesinde her bir hatayı tek tek ve yeri geldiğinde karşılıyoruz. Bu durumun aksine, bölümün ilk başında verdiğimiz try... except bloğunda hem ValueError hem de ZeroDivisionError hatalarının gerçekleşme ihtimali bulunuyor. Dolayısıyla biz orada bütün hataları tek bir try... bloğu içine sıkıştırmış oluyoruz. İşte else: bloğu bu sıkışıklığı gidermiş oluyor. Ancak sizi bir konuda uyarmak isterim: Bu yapı, her akla geldiğinde kullanılacak bir yapı değildir. Büyük programlarda bu tarz bir kullanım kodlarınızın darmadağın olmasına, kodlarınız üzerindeki denetimi tamamen kaybetmenize de yol açabilir. Sonunda da elinizde bölük pörçük bir kod yığını kalabilir. Zaten açıkça söylemek gerekirse try... except... else... yapısının çok geniş bir kullanım alanı yoktur. Bu yapı ancak çok nadir durumlarda kullanılmayı gerektirebilir. Dolayısıyla bu üçlü yapıyı hiç kullanmadan bir ömrü rahatlıkla geçirebilirsiniz.

try... except... finally...

try... except... else... yapılarının dışında, Python'ın bize sunduğu bir başka yapı da try... except... finally... yapılarıdır. Bunu şöyle kullanıyoruz:

```
try:
    ...bir takım işler...
except birHata:
    ...hata alınınca yapılacak işlemler...
finally:
    ...hata olsa da olmasa da yapılması gerekenler...
```

finally.. bloğunun en önemli özelliği, programın çalışması sırasında herhangi bir hata gerçekleşse de gerçekleşmese de işletilecek olmasıdır. Eğer yazdığınız programda mutlaka ama mutlaka işletilmesi gereken bir kısım varsa, o kısmı finally... bloğu içine yazabilirsiniz.

finally... bloğu özellikle dosya işlemlerinde işimize yarayabilir. Henüz Python'da dosyalarla nasıl

çalışacağımızı öğrenmedik, ama ben şimdilik size en azından dosyalarla çalışma prensibi hakkında bir şeyler söyleyeyim.

Genel olarak Python'da dosyalarla çalışabilmek için öncelikle bilgisayarda bulunan bir dosyayı okuma veya yazma kipinde açarız. Dosyayı açtıktan sonra bu dosyayla ihtiyacımız olan birtakım işlemler gerçekleştiririz. Dosyayla işimiz bittikten sonra ise dosyamızı mutlaka kapatmamız gerekir. Ancak eğer dosya üzerinde işlem yapılırken bir hata ile karşılaşılırsa dosyamızı kapatma işlemini gerçekleştirdiğimiz bölüme hiç ulaşılamayabilir. İşte finally... bloğu böyle bir durumda işimize yarayacaktır:

```
try:
    dosya = open("dosyaadı", "r")
    ...burada dosyayla bazı işlemler yapıyoruz...
    ...ve ansızın bir hata oluşuyor...
except IOError:
    print("bir hata oluştu!")
finally:
    dosya.close()
```

Burada finally... bloğu içine yazdığımız dosya.close() ifadesi dosyamızı güvenli bir şekilde kapatmaya yarıyor. Bu blok, yazdığımız program hata verse de vermese de işletilecektir.

raise

Bazen, yazdığımız bir programda, kullanıcının yaptığı bir işlem normal şartlar altında hata vermeyecek olsa bile biz ona 'Python tarzı' bir hata mesajı göstermek isteyebiliriz. Böyle bir durumda ihtiyacımız olan şey Python'ın bize sunduğu raise adlı deyimdir. Bu deyim yardımıyla duruma özgü hata mesajları üretebiliriz. Bir örnek verelim:

```
bölünen = int(input("bölünecek sayı: "))

if bölünen == 23:
    raise Exception("Bu programda 23 sayısını görmek istemiyorum!")

bölen = int(input("bölen sayı: "))
print(bölünen/bölen)
```

Burada eğer kullanıcı *23* sayısını girerse, kullanıcıya bir hata mesajı gösterilip programdan çıkılacaktır. Biz bu kodlarda Exception adlı genel hata mesajını kullandık. Burada Exception yerine her istediğimizi yazamayız. Yazabileceklerimiz ancak Python'da tanımlı hata mesajları olabilir. Örneğin NameError, TypeError, ZeroDivisionError, IOError, vb...

Bir örnek verelim:

```
tr_karakter = "şçğüöıİ"

parola = input("Parolanız: ")

for i in parola:
    if i in tr_karakter:
        raise TypeError("Parolada Türkçe karakter kullanılamaz!")
    else:
        pass

print("Parola kabul edildi!")
```

Bu kodlar çalıştırıldığında, eğer kullanıcı, içinde Türkçe karakter geçen bir parola yazarsa kendisine TypeError tipinde bir hata mesajı gösteriyoruz. Eğer kullanıcının parolası Türkçe karakter içermiyorsa hiçbir şey yapmadan geçiyoruz ve bir sonraki satırda kendisine 'Parola kabul edildi!' mesajını gösteriyoruz.

raise deyimini, bir hata mesajına ek olarak bir işlem yapmak istediğimizde de kullanabiliriz. Örneğin:

```
try:
    bölünen = int(input("bölünecek sayı: "))
    bölen = int(input("bölen sayı: "))
    print(bölünen/bölen)
except ZeroDivisionError:
    print("bir sayıyı 0'a bölemezsiniz")
    raise
```

Burada, eğer kullanıcı bir sayıyı *0*'a bölmeye çalışırsa, normal bir şekilde ZeroDivisionError hatası verilecek ve programdan çıkılacaktır. Ama bu hata mesajıyla birlikte kullanıcıya 'bir sayıyı 0'a bölemezsiniz,' uyarısını da gösterme imkanını elde edeceğiz. Yani burada except ZeroDivisionError bloğunu herhangi bir hatayı engellemek için değil, hataya ilave bilgi eklemek için kullanıyoruz. Bunu yapmamızı sağlayan şey tabii ki bu kodlar içinde görünen raise adlı deyimdir...

Bütün Hataları Yakalamak

Şimdiye kadar yaptığımız bütün örneklerde except... bloğunu bir hata mesajı adıyla birlikte kullandık. Yani örneklerimiz şuna benziyordu:

```
try:
    ....birtakım işler...
except ZeroDivisionError:
    ...hata mesajı...
```

Yukarıdaki kod yardımıyla sadece ZeroDivisionError adlı hatayı yakalayabiliriz. Eğer yazdığımız program başka bir hata daha veriyorsa, o hata mesajı yukarıdaki blokların kapsamı dışında kalacaktır. Ama eğer istersek yukarıdaki kodu şu şekilde yazarak olası bütün hataları yakalayabiliriz:

```
try:
    ....birtakım işler...
except:
    ...hata mesajı...
```

Gördüğünüz gibi, burada herhangi bir hata adı belirtmedik. Böylece Python, yazdığımız programda hangi hata oluşursa oluşsun hepsini yakalayabilecektir.

Bu yöntem gözünüze çok pratik görünmüş olabilir, ama aslında hiç de öyle sayılmaz. Hatta oldukça kötü bir yöntem olduğunu söyleyebiliriz bunun. Çünkü bu tarz bir kod yazımının bazı dezavantajları vardır. Örneğin bu şekilde bütün hata mesajlarını aynı kefeye koyarsak, programımızda ne tür bir hata oluşursa oluşsun, kullanıcıya hep aynı mesajı göstermek zorunda kalacağız. Bu da, herhangi bir hata durumunda kullanıcıyı ne yapması gerektiği konusunda doğru düzgün bilgilendiremeyeceğimiz anlamına geliyor. Yani kullanıcı bir hataya sebep olduğunda tersliğin nereden kaynaklandığını tam olarak kestiremeyecektir.

Ayrıca, eğer kendimiz bir program geliştirirken sürekli olarak bu tarz bir yazımı benimsersek, kendi kodlarımızdaki hataları da maskelemiş oluruz. Dolayısıyla, Python yukarıdaki geniş kapsamlı except... bloğu nedeniyle programımızdaki bütün hataları gizleyeceği için, programımızdaki potansiyel aksaklıkları görme imkanımız olmaz. Dolayısıyla bu tür bir yapıdan olabildiğince kaçınmakta fayda var. Ancak elbette böyle bir kod yazmanızı gerektiren bir durumla da karşılaşabilirsiniz. Örneğin:

```
try:
    birtakım kodlar
except ValueError:
    print("Yanlış değer")
except ZeroDivisionError:
    print("Sıfıra bölme hatası")
except:
    print("Beklenmeyen bir hata oluştu!")
```

Burada olası bütün hata türlerini yakaladıktan sonra, bunların dışında bizim o anda öngöremediğimiz bir hatanın oluşması ihtimaline karşı except: kodunu kullanarak kullanıcıya genel bir hata mesajı göstermeyi tercih edebiliriz. Böylece beklenmeyen bir hata meydana gelmesi durumunda da programımız çökmek yerine çalışmaya devam edebilecektir.

Örnek Uygulama

Hata yakalama konusunu bütün ayrıntılarıyla inceledik. Gelin şimdi isterseniz ufak bir örnek yapalım.

Hatırlarsanız bir kaç bölüm önce şöyle bir uygulama yazmıştık:

```
import sys

_2x_metni = """
Python'ın 2.x sürümlerinden birini kullanıyorsunuz.
Programı çalıştırabilmek için sisteminizde Python'ın
3.x sürümlerinden biri kurulu olmalı."""

_3x_metni = "Programa hoşgeldiniz."

if sys.version_info.major < 3:
    print(_2x_metni)
else:
    print(_3x_metni)
```

Bu programın ne iş yaptığını biliyorsunuz. Bu program yardımıyla, kullanıcılarımızın bilgisayarlarındaki Python sürümünü kontrol edip, programımızın kullanılan sürüme göre tepki vermesini sağlıyoruz.

Ancak burada çok ciddi bir problem var. Python'ın 2.7 öncesi sürümlerinde *sys* modülünün version_info() metodu farklı çıktılar verir. Mesela Python'ın 2.7 öncesi sürümlerinde version_info() metodunun *major*, *minor* veya *micro* gibi nitelikleri bulunmaz. Bu nitelikler Python programlama diline 2.7 sürümüyle birlikte geldi. Dolayısıyla yukarıdaki programı Python'ın 2.7 öncesi sürümlerinden biriyle çalıştıran kullanıcılarınız istediğiniz çıktıyı alamayacak, Python bu kullanıcalara şuna benzer bir hata mesajı göstererek programın çökmesine sebep olacaktır:

```
AttributeError: 'tuple' object has no attribute 'major'
```

Python'ın 2.7 öncesi sürümlerinin kurulu olduğu bilgisayarlarda da programınızın en azından çökmemesi ve makul bir çıktı verebilmesi için yukarıdaki kodlar şöyle yazabilirsiniz:

```
import sys

_2x_metni = """
Python'ın 2.x sürümlerinden birini kullanıyorsunuz.
Programı çalıştırabilmek için sisteminizde Python'ın
3.x sürümlerinden biri kurulu olmalı."""

_3x_metni = "Programa hoşgeldiniz."

try:
    if sys.version_info.major < 3:
        print(_2x_metni)
```

```
    else:
        print(_3x_metni)
except AttributeError:
    print(_2x_metni)
```

Gördüğünüz gibi, AttributeError adlı hatayı vereceğini bildiğimiz kısmı bir try... except bloğu içine aldık. Eğer programımız AttributeError hatasını veriyorsa, programımızın çalıştırıldığı sistem Python'ın 2.7 sürümünden daha düşük bir sürümü kullanıyor demektir. O yüzden kullanıcıya *_2x_metni*'ni gösteriyoruz.

Elbette yukarıdaki programı yazmanın çok daha düzgün yolları vardır. Ama biz hata yakalama yöntemlerinin buna benzer durumlarda da bir alternatif olarak kullanılabileceğini bilelim. Ayrıca, dediğimiz gibi, try... except blokları yukarıdaki sorunun çözümü için en uygun araçlar olmasa da, bazı durumlarda hatayı önlemenin makul tek yoludur.

başa dön

Karakter Dizileri

Buraya gelene kadar Python programlama diline ilişkin epey bilgi edindik. Artık yazdığımız programlarda input() fonksiyonu sayesinde kullanıcıyla iletişim kurabiliyor; if, elif, else deyimleri yardımıyla programlarımızın karar vermesini sağlayabiliyor; işleçler ve döngüler yoluyla programlarımızı istediğimiz sayıda çalıştırabiliyoruz. Eğer buraya kadar olan bölümleri dikkatlice takip ettiyseniz, şu ana kadar öğrendiklerinize dayanarak, Python'ı giriş düzeyinde bildiğinizi rahatlıkla iddia edebilirsiniz. Zira şimdiye kadar öğrendiklerinizi kullanarak ufak tefek de olsa işe yarar programlar yazabilecek durumdasınız.

Buraya kadar öğrendiğimiz bilgiler Python programlama dilinin temellerini oluşturuyordu. Temel Python bilgilerini edindiğimize göre, artık başlangıç-orta düzey arası konuları incelemeye başlayabileceğiz.

Bu bölümde, önceki derslerde üstünkörü bakıp geçtiğimiz bir konu olan karakter dizilerini çok daha derinlemesine ele alacağız. Python programlama dili içindeki önemi nedeniyle bu bölüm epey uzun olacak.

Aslında biz karakter dizisi kavramının ne olduğunu biliyoruz. Çok kaba bir şekilde ifade etmek gerekirse, karakter dizileri, adından da anlaşılacağı gibi, karakterlerin bir araya gelmesiyle oluşan bir dizidir. Karakter dizileri; tek, çift veya üç tırnak içinde gösterilen, öteki veri tiplerinden de bu tırnaklar aracılığıyla ayırt edilen özel bir veri tipidir. Teknik olarak ifade etmek gerekirse, bir nesneyi type() fonksiyonu yardımıyla sorguladığımızda, eğer *<class 'str'>* çıktısı alıyorsak bu nesne bir karakter dizisidir.

Her ne kadar ayrıntılarına girmemiş de olsak, dediğimiz gibi, biz karakter dizilerini daha ilk bölümlerden bu yana her fırsatta kullanıyoruz. Dolayısıyla bu veri tipinin ne olduğu konusunda bir sıkıntımız yok. Bu bölümde, şimdiye kadar karakter dizileri ile ilgili öğrendiğimiz şeylere ek olarak, karakter dizilerin metotlarından da söz edeceğiz.

Peki bu 'metot' denen şey de ne oluyor?

Kabaca ifade etmek gerekirse, metotlar Python'da nesnelerin niteliklerini değiştirmemizi, sorgulamamızı veya bu nesnelere yeni özellikler katmamızı sağlayan araçlardır. Metotlar sayesinde karakter dizilerini istediğimiz gibi eğip bükebileceğiz.

Elbette bu bölümde bahsedeceğimiz tek şey karakter dizilerinin metotları olmayacak. Bu bölümde aynı zamanda karakter dizilerinin yapısı ve özelliklerine dair söyleyeceklerimiz de olacak.

Python'da şimdiye kadar yapabildiğimiz şeylerin sizi tatmin etmekten uzak olduğunu, daha fazlasını yapabilmek için sabırsızlandığınızı tahmin edebiliyorum. O halde ne duruyoruz, hiç vakit kaybetmeden yola koyulalım.

Karakter Dizilerinin Öğelerine Erişmek

Python ile programlama yaparken karakter dizileri ile iki şekilde karşılaşabilirsiniz: Birincisi, bir karakter dizisini doğrudan kendiniz tanımlamış olabilirsiniz. İkincisi, karakter dizisi size başka bir kaynak aracılığıyla gelmiş olabilir (mesela input() fonksiyonu yardımıyla kullanıcıdan aldığınız bir veri).

Python'da kendi tanımladığınız ya da herhangi başka bir kaynaktan gelen karakter dizilerine erişmenin birkaç farklı yolu vardır. Örneğin:

```
>>> nesne = "karakter dizisi"
```

Burada değeri *"karakter dizisi"* olan *nesne* adlı bir değişken tanımladık. Yazdığımız programlarda bu değişkene erişmek için, değişkenin adını kullanmamız yeterlidir. Örneğin:

```
>>> print(nesne)
```

Bu komut bize karakter dizisinin tamamını verecektir.

Bir karakter dizisini yukarıda gördüğümüz gibi kendimiz tanımlayabiliriz. Bunun dışında, mesela input() fonksiyonuyla kullanıcıdan aldığımız verilerin de birer karakter dizisi olacağını biliyoruz:

```
veri = input("Herhangi bir şey: ")
```

Tıpkı kendi tanımladığımız karakter dizilerinde olduğu gibi, kullanıcıdan gelen karakter dizilerini de aşağıdaki komut yardımıyla ekranda görüntüleyebiliriz:

```
print(veri)
```

Bu komut da bize *veri* değişkeninin tuttuğu karakter dizisinin tamamını verecektir.

Ayrıca istersek bu karakter dizilerini bir for döngüsü içine alabilir, böylece bu dizinin öğelerine tek tek de erişebiliriz:

```
for karakter in nesne:
    print(karakter)
```

for döngüsüyle elde ettiğimiz bu etkiyi şu kodlar yardımıyla da elde edebileceğimizi gayet iyi biliyor olmalısınız:

```
print(*nesne, sep="\n")
```

Önceki derslerde verdiğimiz örneklerden de bildiğiniz gibi, karakter dizilerinin öğelerine yukarıdaki yöntemlerle tek tek erişebilmemiz sayesinde herhangi bir işlemi karakter dizilerinin bütün öğelerine bir çırpıda uygulayabiliyoruz. Mesela:

```
nesne = "123456789"
```

```
for n in nesne:
    print(int(n) * 2)
```

Burada *nesne* değişkeni içindeki sayı değerli karakter dizilerini *n* olarak adlandırdıktan sonra, *n* değişkenlerinin her birini tek tek *2* sayısı ile çarptık. Yani çarpma işlemini karakter dizisinin bütün öğelerine tek seferde uygulayabildik. Bu arada, yukarıdaki örnekte *nesne* değişkeninin her bir öğesini for döngüsü içinde int() fonksiyonu yardımıyla tam sayıya çevirdiğimizi görüyorsunuz. Daha önce de defalarca söylediğimiz gibi, Python'da o anda elinizde olan verinin tipini bilmeniz çok önemlidir. Eğer kendi yazdığınız veya mesela input() fonksiyonundan gelen bir verinin karakter dizisi olduğunu bilmezseniz yukarıdaki kodları şu şekilde yazma gafletine düşebilirsiniz:

```
nesne = "123456789"

for n in nesne:
    print(n * 2)
```

Bu kodlar çalıştırıldıktan sonra hiç beklemediğiniz sonuçlar verecektir:

```
11
22
33
44
55
66
77
88
99
```

Gördüğünüz gibi, aslında *nesne* içindeki öğeleri *2* ile çarpmak isterken, biz her bir öğeyi iki kez ekrana yazdırmış olduk. Çünkü bildiğiniz gibi karakter dizileri ile aritmetik işlemler yapamıyoruz. Eğer sayı değerli karakter dizileri arasında aritmetik işlem yapacaksak öncelikle bu karakter dizilerini sayıya çevirmemiz gerekir. Ayrıca gerçek bir program içinde yukarıdaki gibi bir durumun ne kadar yıkıcı sonuçlar doğuracabileceğini düşünün. Yukarıdaki program çalışma sırasında hiçbir hata vermeyeceği için, siz programınızın düzgün çalıştığını zannederek hayatınıza devam edeceksiniz. Ama belki de yukarıdaki sinsi hata yüzünden, programınızı kullanan bir şirket veri, zaman ve para kaybına uğrayacak.

Yukarıdaki örneklerde bir şey daha dikkatinizi çekmiş olmalı: Gördüğünüz gibi, karakter dizisinin öğelerine erişirken bu öğelerin tamamını elde ediyoruz. Mesela print(nesne) komutunu verdiğimizde veya *nesne* değişkenini bir döngü içine aldığımızda sonuç olarak elde ettiğimiz şey, ilgili karakter dizisinin tamamıdır. Yani aslında karakter dizisinin hangi öğesine erişeceğimizi seçemiyoruz. Peki ya biz bir karakter dizisinin öğelerinin tamamına değil de, sadece tek bir öğesine erişmek istersek ne yapacağız? Mesela yukarıdaki örnekte *nesne* adlı değişken içindeki sayıların tamamını değil de sadece tek bir öğesini (veya belli bir ölçüte göre yalnızca bazı öğelerini) *2* ile çarpmak istersek nasıl bir yol izleyeceğiz?

Python'da karakter dizilerinin içindeki öğelerin bir sırası vardır. Örneğin *"Python"* dediğimizde, bu karakter dizisinin ilk öğesi olan *"P"* karakterinin sırası *0*'dır. *"y"* karakteri ise *1.* sıradadır. Aynı şekilde devam edersek, *"t"* karakteri *2.*, *"h"* karakteri *3.*, *"o"* karakteri *4.*, *"n"* karakteri ise *5.* sırada yer alır.

Bu anlattığımız soyut durumu bir örnekle somutlaştırmaya çalışalım:

Dedik ki, *"Python"* gibi bir karakter dizisinin her bir öğesinin belli bir sırası vardır. İşte eğer biz bu karakter dizisinin bütün öğelerini değil de, sadece belli karakterlerini almak istersek, karakter dizisindeki öğelerin sahip olduğu bu sıradan yararlanacağız.

Diyelim ki *"Python"* karakter dizisinin ilk karakterini almak istiyoruz. Yani biz bu karakter dizisinin sadece *"P"* harfine ulaşmayı amaçlıyoruz.

Bu isteğimizi nasıl yerine getirebileceğimizi basit bir örnek üzerinde göstermeye çalışalım:

```
>>> kardiz = "Python"
```

Burada değeri *"Python"* olan *kardiz* adlı bir değişken tanımladık. Şimdi bu karakter dizisinin ilk öğesine erişeceğiz:

```
>>> kardiz[0]

'P'
```

Burada yaptığımız işleme çok dikkat edin. Karakter dizisinin istediğimiz bir öğesine ulaşmak için, ilgili öğenin sırasını köşeli parantezler içinde belirttik. Biz bu örnekte karakter dizisinin ilk öğesine ulaşmak istediğimiz için köşeli parantez içinde *0* sayısını kullandık.

Şimdi de, ilk verdiğimiz örnekteki *nesne* değişkeni içinde yer alan sayılar arasından sadece birini *2* ile çarpmak istediğimizi düşünelim:

```
>>> nesne = "123456789"
>>> int(nesne[1]) * 2

4
```

Burada da öncelikle *nesne* değişkeninin birinci sırasında yer alan öğeyi (dikkat: sıfırıncı sırada yer alan öğeyi değil!) elde etmek için köşeli parantezler içinde *1* sayısını kullandık. Daha sonra int() fonksiyonu yardımıyla bu karakter dizisini tam sayıya çevirdik, ki bununla aritmetik işlem yapabilelim... Son olarak da elimizdeki tam sayıyı *2* ile çarparak istediğimiz sonuca ulaştık.

Elbette yukarıdaki kodları şöyle de yazabilirdik:

```
>>> nesne = "123456789"
>>> sayı = int(nesne[1])
```

```
>>> sayı * 2

4
```

Belki farkındasınız, belki de değilsiniz, ama aslında şu noktada karakter dizilerinin çok önemli bir özelliği ile karşı karşıyayız. Gördüğünüz gibi, yukarıda bahsettiğimiz sıra kavramı sayesinde Python'da karakter dizilerinin bütün öğelerine tek tek ve herhangi bir sıra gözetmeksizin erişmemiz mümkün. Mesela yukarıdaki ilk örnekte kardiz[0] gibi bir yapı kullanarak karakter dizisinin sıfırıncı (yani ilk) öğesini, nesne[1] gibi bir yapı kullanarak da karakter dizisinin birinci (yani aslında ikinci) öğesini alabildik.

Bu yapının mantığını kavramak için şu örnekleri dikkatlice inceleyin:

```
>>> kardiz = "Python"

>>> kardiz[0]

'P'

>>> kardiz[1]

'y'

>>> kardiz[3]

'h'

>>> kardiz[5]

'n'

>>> kardiz[2]

't'

>>> kardiz[4]

'o'

>>> nesne = "123456789"

>>> nesne[0]

'1'
```

```
>>> nesne[1]
'2'
>>> nesne[2]
'3'
>>> nesne[3]
'4'
>>> nesne[4]
'5'
>>> nesne[5]
'6'
>>> nesne[6]
'7'
>>> nesne[7]
'8'
>>> nesne[8]
'9'
```

Burada şöyle bir formül yazabiliriz:

```
karakter_dizisi[öğe_sırası]
```

Bu formülü uygulayarak karakter dizilerinin her bir öğesine tek tek erişmemiz mümkün. Burada çok önemli bir noktaya daha dikkatinizi çekmek isterim. Yukarıdaki örneklerden de gördüğünüz gibi, Python'da öğe sıralaması *0*'dan başlıyor. Yani bir karakter dizisinin ilk öğesinin sırası *0* oluyor. Python programlama dilini özellikle yeni öğrenenlerin en sık yaptığı hatalardan biri de bir karakter dizisinin ilk öğesine ulaşmak için *1* sayısını kullanmalarıdır. Asla unutmayın, Python saymaya her zaman *0*'dan başlar. Dolayısıyla bir karakter dizisinin ilk öğesinin sırası *0*'dır. Eğer ilk öğeye ulaşayım derken *1* sayısını kullanırsanız ulaştığınız öğe ilk öğe değil, ikinci öğe olacaktır. Bu

ayrıntıyı gözden kaçırmamaya dikkat etmelisiniz.

Karakter dizilerinin öğelerine tek tek erişirken dikkat etmemiz gereken önemli noktalardan biri de, öğe sırası belirtirken, karakter dizisinin toplam uzunluğu dışına çıkmamaktır. Yani mesela *7* karakterlik bir karakter dizimiz varsa, bu karakter dizisinin son öğesinin sırası *6* olacaktır. Çünkü biliyorsunuz, Python saymaya *0*'dan başlıyor. Dolayısıyla ilk karakterin sırası *0* olacağı için, *7* karakterlik bir karakter dizisinde son öğenin sırası *6* olacaktır. Örneğin:

```
>>> kardiz = "istihza"
>>> len(kardiz)

7
```

Gördüğünüz gibi, *"istihza"* adlı karakter dizisinin uzunluğu *7*. Yani bu karakter dizisi içinde *7* adet karakter var. Bu karakter dizisini incelemeye devam edelim:

```
>>> kardiz[0]

'i'
```

Dediğimiz gibi, karakter dizisinin ilk öğesinin sırası *0*. Dolayısıyla son öğenin sırası *6* olacaktır:

```
>>> kardiz[6]

'a'
```

Bu durumu şöyle formüle edebiliriz:

```
>>> kardiz[len(kardiz)-1]
```

Yani;

> Bir karakter dizisinin uzunluğunun *1* eksiği, o karakter dizisinin son öğesini verir.

Yukarıdaki formülü eğer şöyle yazsaydık hata alırdık:

```
>>> kardiz[len(kardiz)]

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Çünkü `len(kardiz)` kodu bize karakter dizisinin uzunluğunu veriyor. Yani yukarıdaki *"istihza"* karakter dizisini göz önüne alırsak, `len(kardiz)` çıktısı *7* olacaktır. Dolayısıyla *"istihza"* karakter dizisinin son öğesine ulaşmak istersek bu değerin 1 eksiğini almamız gerekiyor. Yani `len(kardiz)-1`.

Şu ana kadar öğe sırası olarak hep artı değerli sayılar kullandık. Ancak istersek öğe sırası olarak eksi değerli sayıları da kullanabiliriz. Eğer bir karakter dizisine öğe sırası olarak eksi değerli bir sayı verirsek Python o karakter dizisini sondan başa doğru okumaya başlayacaktır. Yani:

```
>>> kardiz[-1]

'a'
```

Gördüğünüz gibi *-1* sayısı karakter dizisini tersten okuyup, sondan başa doğru ilk öğeyi veriyor. Dolayısıyla, yukarıda anlattığımız len(kardiz)-1 yönteminin yanısıra, *-1* sayısını kullanarak da karakter dizilerinin son karakterini elde edebiliyoruz. Bir de şuna bakalım:

```
>>> kardiz[-2]

'z'
```

Dediğimiz gibi, eksi değerli sayılar karakter dizisindeki karakterleri sondan başa doğru elde etmemizi sağlar. Dolayısıyla *-2* sayısı, karakter dizisinde sondan bir önceki karakteri verecektir.

Karakter dizilerinin öğelerine tek tek erişmek amacıyla öğe sırası belirtirken, karakter dizisinin toplam uzunluğu dışına çıkmamamız gerektiğini söylemiştik. Peki karakter dizisinin uzunluğunu aşan bir sayı verirsek ne olur? Ne olacağını yukarıdaki örneklerden birinde görmüştük aslında. Ama konunun öneminden dolayı bir kez daha tekrar edelim.

```
>>> kardiz = "istihza"
>>> kardiz[7]

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

...veya:

```
>>> kardiz[-8]

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Eğer karakter dizisinin uzunluğunu aşan bir sayı belirtirsek Python bize IndexError türünde bir hata mesajı verecektir.

Gördüğünüz gibi, kardiz[0], kardiz[1], kardiz[2], vb. komutlarla karakter dizisinin öğelerine erişebiliyoruz. Burada öğe sıralarını tek tek yazmak yerine range() fonksiyonunu kullanarak da öğelere tek tek erişebilirsiniz:

```
for i in range(7):
    print(kardiz[i])
```

Bu kodlarda, kardiz[0], kardiz[1], kardiz[2] şeklinde öğe sıralarını tek tek elle yazmak yerine, range(7) aralığındaki sayıları bir *for* döngüsüne alıyoruz. Böylece Python kardiz[öğe_sırası] gibi bir yapı içinde *öğe_sırası* yerine range(7) aralığındaki bütün sayıları (yani *0*, *1*, *2*, *3*, *4*, *5*, *6* sayılarını) tek tek uyguluyor.

Burada aklınıza hemen şöyle bir soru gelmiş olabilir:

> Biz kendi tanımladığımız karakter dizisinin uzunluğunun toplam *7* karakter olduğunu bildiğimiz için yukarıdaki örnekte range() fonksiyonunu range(7) şeklinde kullanabildik. Ama başka kaynaktan gelen bir karakter dizisinin uzunluğunu nasıl bileceğiz?

Aslında bu sorunun cevabı çok basit. Uzunluğunu bilmediğiniz karakter dizileri için range() fonksiyonuyla birlikte len() fonksiyonundan yararlanabilirsiniz. Nasıl mı? Hemen bir örnek verelim:

```
for karakter in range(len(kardiz)):
    print(kardiz[karakter])
```

Burada range() fonksiyonuna verdiğimiz *len(kardiz)* parametresine dikkatlice bakın. Biz *kardiz* adlı değişkenin tuttuğu karakter dizisinin *7* karakterden oluştuğunu biliyoruz. Ama eğer bu karakter dizisini biz belirlememişsek, karakter dizisinin tam olarak kaç karakterden oluşacağını bilemeyiz. Bu kodlarda len(kardiz) ifadesini kullanarak, sabit bir değer belirlemekten kaçınmış oluyoruz. Böylece, mesela kullanıcıdan aldığımız bir karakter dizisinin kaç karakterden oluştuğunu belirleme görevini Python'a bırakmış oluyoruz. Karakter dizisinin uzunluğu ne ise (len(kardiz)), Python range() fonksiyonuna o sayıyı parametre olarak kendisi atayacaktır.

Yukarıdaki durumu daha iyi anlayabilmek için bir örnek daha verelim. Diyelim ki kullanıcıya ismini sorup, kendisine şöyle bir çıktı vermek istiyorsunuz:

```
isminizin 1. harfi ...
isminizin 2. harfi ...
isminizin 3. harfi ...
...
```

Bunu yapabilmek için şöyle bir uygulama yazabilirsiniz:

```
isim = input("isminiz: ")

for i in range(len(isim)):
    print("isminizin {}. harfi: {}".format(i, isim[i]))
```

Gördüğünüz gibi, kullanıcının girdiği kelimenin uzunluğu kaç ise o sayı otomatik olarak range() fonksiyonuna atanıyor. Diyelim ki kullanıcı Ferhat ismini girmiş olsun. Bu kelimede toplam *6* karakter

var. Dolayısıyla Python for satırını şöyle yorumlayacaktır:

```
for i in range(6):
    ...
```

Python for döngüsünün ilk turunda şöyle bir işlem gerçekleştirir:

```
print("isminizin {}. harfi: {}".format(0, isim[0]))
```

İkinci turda ise şöyle bir işlem:

```
print("isminizin {}. harfi: {}".format(1, isim[1]))
```

Annenizin kızlık soyadının 0. harfi
[kaynak]

Bu döngü *6* sayısına gelene kadar devam eder. Burada *i* adlı değişkenin değerinin her döngüde nasıl değiştiğine dikkat edin. Python *i* adını verdiğimiz değişkene, for döngüsünün her turunda sırasıya *0*, *1*, *2*, *3*, *4* ve *5* sayılarını atayacağı için *isim* adlı değişkenin öğeleri isim[öğe_sırası] formülü sayesinde tek tek ekrana dökülecektir.

Yalnız bu kodların çıktısında iki nokta dikkatinizi çekmiş olmalı. Birincisi, *isminizin 0. harfi f* gibi bir çıktıyı kullanıcılarınız yadırgayabilir. Çünkü '*0.* harf' çok yapay duran bir ifade. Onun yerine ilk harfi '1. harf' olarak adlandırmamız çok daha mantıklı olacaktır. Bunun için kodlarınıza şu basit eklemeyi yapabilirsiniz:

```
isim = input("isminiz: ")

for i in range(len(isim)):
    print("isminizin {}. harfi: {}".format(i+1, isim[i]))
```

Burada ilk *i* değişkeninin değerini *1* sayı artırdık. Böylece *0* sayısı *1*'e, *1* sayısı *2*'ye, *2* sayısı *3*'e... dönüşmüş oldu. Bu şekilde kullanıcılarınıza çok daha doğal görünen bir çıktı verebilmiş oluyorsunuz. Eğer bu işlemi yapmazsanız, kullanıcılarınızın 'doğal görünmeyen' bir çıktı almalarının yanısıra, programınızın verdiği çıktı kimi durumlarda epey yanıltıcı da olabilir...

Karakter Dizilerini Dilimlemek

Bir önceki bölümde bir karakter dizisinin istediğimiz öğesini, o öğenin sırasını belirterek nasıl elde edebileceğimizi gördük. Bu bölümde de benzer bir şey yapacağız. Ama burada yapacağımız şey, bir önceki bölümde yaptığımız işleme göre biraz daha kapsamlı bir işlem olacak.

Bu bölümde karakter dizilerini 'dilimlemekten' söz edeceğiz. Peki 'dilimlemek' derken neyi kastediyoruz? Aslında burada gerçek anlamda 'karpuz gibi dilimlemekten' söz ediyoruz... Şu örnek, ne demek istediğimizi daha net ortaya koyacaktır:

```
>>> site = "www.istihza.com"
>>> site[4:11]

'istihza'

>>> site[12:16]

'com'

>>> site[0:3]

'www'
```

Gördüğünüz gibi, karakter dizisine köşeli parantez içinde bazı değerler vererek bu karakter dizisini dilim dilim ayırdık. Peki bunu nasıl yaptık? Yukarıdaki örneklerde şöyle bir yapı gözümüze çarpıyor:

```
karakter_dizisi[alınacak_ilk_öğenin_sırası:alınacak_son_öğenin_sırasının_bir_fazlası]
```

Bu formülü çok basit bir örneğe uygulayalım:

```
>>> karakter_dizisi = "istanbul"
>>> karakter_dizisi[0:3]

'ist'
```

Burada alacağımız ilk öğenin sıra numarası *0*. Yani *"istanbul"* karakter dizisindeki 'i' harfi. Alacağımız son öğenin sıra numarasının 1 fazlası ise *3*. Yani *2*. sıradaki 't' harfi. İşte `karakter_dizisi[0:3]` dediğimizde, Python *0*. öğe ile *3*. öğe arasında kalan bütün öğeleri bize verecektir. Bizim örneğimizde bu aralıktaki öğeler 'i', 's' ve 't' harfleri. Dolayısıyla Python bize 'istanbul' kelimesindeki 'ist' kısmını dilimleyip veriyor.

Bu bilgileri kullanarak şöyle bir uygulama yazalım:

```
site1 = "www.google.com"
site2 = "www.istihza.com"
site3 = "www.yahoo.com"
site4 = "www.gnu.org"
```

```
for isim in site1, site2, site3, site4:
    print("site: ", isim[4:-4])
```

Bu örnek Python'da dilimleme işlemlerinin yapısı ve özellikleri hakkında bize epey bilgi veriyor. Gördüğünüz gibi, hem artı hem de eksi değerli sayıları kullanabiliyoruz. Önceki bölümden hatırlayacağınız gibi, eğer verilen sayı eksi değerliyse Python karakter dizisini sağdan sola (yani sondan başa doğru) okuyacaktır. Yukarıdaki örnekte isim[4:-4] yapısını kullanarak, *site1*, *site2*, *site3*, *site4* adlı karakter dizilerini, ilk dört ve son dört karakterler hariç olacak şekilde dilimledik. Böylece elimizde ilk dört ve son dört karakter arasındaki bütün karakterler kalmış oldu. Yani *"google"*, *"istihza"*, *"yahoo"* ve *"gnu"*.

Bütün bu anlattıklarımızı daha iyi anlayabilmek için bir örnek daha verelim:

```
ata1 = "Akıllı bizi arayıp sormaz deli bacadan akar!"
ata2 = "Ağa güçlü olunca  kul suçlu olur!"
ata3 = "Avcı ne kadar hile bilirse ayı da o kadar yol bilir!"
ata4 = "Lafla pilav pişse deniz kadar yağ benden!"
ata5 = "Zenginin gönlü oluncaya kadar fukaranın canı çıkar!"
```

Burada beş adet atasözü verdik. Bizim görevimiz, bu atasözlerinin sonunda bulunan ünlem işaretlerini ortadan kaldırmak:

```
for ata in ata1, ata2, ata3, ata4, ata5:
    print(ata[0:-1])
```

Burada yaptığımız şey şu: *ata1*, *ata2*, *ata3*, *ata4* ve *ata5* adlı değişkenlerin her birini *ata* olarak adlandırdıktan sonra *ata* adlı değişkenin en başından en sonuna kadar olan kısmı dilimleyip aldık. Yani ata[0] ile ata[-1] arasında kalan bütün karakterleri elde etmiş olduk. Peki bu ünlem işaretlerini kaldırdıktan sonra bunların yerine birer nokta koymak istersek ne yapacağız?

O da çok basit bir işlem:

```
for ata in ata1, ata2, ata3, ata4, ata5:
    print(ata[0:-1] + ".")
```

Gördüğünüz gibi, son karakter olan ünlem işaretini attıktan sonra onun yerine bir nokta işareti koymak için yaptığımız tek şey, dilimlediğimiz karakter dizisine, artı işareti (+) yardımıyla bir . karakteri eklemekten ibarettir.

Böylece karakter dizilerini nasıl dilimleyeceğimizi öğrenmiş olduk. Bu konuyu kapatmadan önce dilimlemeye ilişkin bazı ayrıntılardan söz edelim. Diyelim ki elimizde şöyle bir karakter dizisi var:

```
>>> kardiz = "Sana Gül Bahçesi Vadetmedim"
```

Bu karakter dizisi içinden sadece 'Sana' kısmını dilimlemek için şöyle bir şey yazabileceğimizi

biliyorsunuz:

```
>>> kardiz[0:4]

'Sana'
```

Burada *0*. karakterden *4*. karaktere kadar olan kısmı dilimlemiş oluyoruz. Python bize bu tür durumlarda şöyle bir kolaylık sağlar: Eğer karakter dizisi içinden alınan ilk karakterin sırasını gösteren sayı *0* ise, bu sayıyı belirtmesek de olur. Yani kardiz[0:4] kodunu şöyle de yazabiliriz:

```
>>> kardiz[:4]

'Sana'
```

Gördüğünüz gibi, ilk sıra sayısını yazmazsak Python ilk sayıyı *0* kabul ediyor.

Şimdi de aynı karakter dizisi içindeki 'Vadetmedim' kısmını dilimlemeye çalışalım:

```
>>> kardiz[17:27]

'Vadetmedim'
```

Burada da *17*. karakter ile *27*. karakter arasında kalan bütün karakterleri dilimledik. Tıpkı, alacağımız ilk karakterin sırası *0* olduğunda bu sayıyı belirtmemize gerek olmadığı gibi, alacağımız son karakterin sırası karakter dizisinin sonuncu karakterine denk geliyorsa o sayıyı da yazmamıza gerek yok. Yani yukarıdaki kardiz[17:27] kodunu şöyle de yazabiliriz:

```
>>> kardiz[17:]

'Vadetmedim'
```

Python'daki bu dilimleme özelliğini kullanarak karakter dizilerini istediğiniz gibi eğip bükebilir, evirip çevirebilirsiniz.

Python'daki bu dilimleme yapısı ilk bakışta gözünüze biraz karmaşıkmış gibi görünebilir. Ama aslında hiç de öyle değildir. Bu yapının mantığını bir kez kavradıktan sonra kodlarınızı hatasız bir şekilde yazabilirsiniz.

Dilimleme yapısını daha iyi anlayabilmek için kendi kendinize bazı denemeler yapmanızı tavsiye ederim. Bu yapının nasıl çalıştığını anlamanın en iyi yolu bol bol örnek kod yazmaktır.

Karakter Dizilerini Ters Çevirmek

Eğer amacınız bir karakter dizisini ters çevirmek, yani karakter dizisi içindeki her bir öğeyi tersten yazdırmaksa biraz önce öğrendiğimiz dilimleme yöntemini kullanabilirsiniz. Dikkatlice bakın:

```
>>> kardiz[::-1]

'midemtedaV iseçhaB lüG anaS'
```

Gördüğünüz gibi, *"Sana Gül Bahçesi Vadetmedim"* adlı karakter dizisi içindeki bütün karakterler sondan başa doğru ekrana dizildi.

Aslında bu komutla Python'a şöyle bir emir vermiş oluyoruz:

> *kardiz* değişkeni içindeki bütün karakterleri, en son karakterden ilk karaktere kadar sondan başa doğru tek tek ekrana yazdır!

Bildiğiniz gibi, eğer almak istediğimiz karakter, dizi içindeki ilk karakterse bu karakterin dizi içindeki sırasını belirtmemize gerek yok. Aynı şekilde, eğer almak istediğimiz karakter, dizi içindeki son karakterse, bu karakterin de dizi içindeki sırasını belirtmemize gerek yok. İşte yukarıdaki örnekte bu kuraldan yararlandık.

Eğer bir karakter dizisinin tamamının değil de, sadece belli bir kısmının ters çevrilmiş halini elde etmek istiyorsanız elbette yapmanız gereken şey, almak istediğiniz ilk ve son karakterlerin sırasını parantez içinde belirtmek olacaktır. Mesela yukarıdaki karakter dizisinde sadece 'Gül' kelimesini ters çevirmek istersek şöyle bir şey yazabiliriz:

```
>>> kardiz[7:4:-1]

'lüG'
```

Yukarıdaki örnek, karakter dizisi dilimlemeye ilişkin olarak bize bazı başka ipuçları da veriyor. Gördüğünüz gibi, köşeli parantez içinde toplam üç adet parametre kullanabiliyoruz. Yani formülümüz şöyle:

```
kardiz[ilk_karakter:son_karakter:atlama_sayısı]
```

Bir örnek verelim:

```
>>> kardiz = "istanbul"
>>> kardiz[0:8:1]

'istanbul'
```

Burada *"istanbul"* adlı karakter dizisinin bütün öğelerini birer birer ekrana döktük. Bir de şuna bakalım:

```
>>> kardiz[0:8:2]

'itnu'
```

Burada ise *"istanbul"* adlı karakter dizisinin bütün öğelerini ikişer ikişer atlayarak ekrana döktük. Yani bir karakter yazıp bir karakter atladık (**i**s**t**a**n**b**u**l).

Python'ın kuralları gereğince yukarıdaki kodu şöyle yazabileceğimizi de biliyorsunuz:

```
>>> kardiz[::2]

'itnu'
```

Eğer karakter dizisini ters çevirmek istiyorsak, yukarıdaki örneği eksi değerli bir atlama sayısı ile yazmamız gerekir:

```
>>> kardiz = "istanbul"
>>> kardiz[::-1]

'lubnatsi'

>>> kardiz[::-2]

'lbas'
```

Dediğimiz gibi, yukarıdaki yöntemi kullanarak karakter dizilerini ters çevirebilirsiniz. Ama eğer isterseniz reversed() adlı bir fonksiyondan da yararlanabiliriz.

Gelelim bu fonksiyonun nasıl kullanılacağına... Önce şöyle bir deneme yapalım:

```
>>> reversed("Sana Gül Bahçesi Vadetmedim")

<reversed object at 0x00E8E250>
```

Gördüğünüz gibi, bu fonksiyonu düz bir şekilde kullandığımızda bize bir 'reversed' nesnesi vermekle yetiniyor. Buna benzer bir olguyla range() fonksiyonunda da karşılaşmıştık:

```
>>> range(10)

range(0, 10)
```

Hatırlarsanız, range(10) gibi bir komutun içeriğini görebilmek için bu komut üzerinde bir for döngüsü kurmamız gerekiyordu:

```
for i in range(10):
    print(i)
```

...veya:

```
print(*range(10))
```

Aynı durum reversed() fonksiyonu için de geçerlidir:

```
for i in reversed("Sana Gül Bahçesi Vadetmedim"):
    print(i, end="")
```

...veya:

```
print(*reversed("Sana Gül Bahçesi Vadetmedim"), sep="")
```

Dilimleme veya reversed() fonksiyonunu kullanma yöntemlerinden hangisi kolayınıza geliyorsa onu tercih edebilirsiniz.

Karakter Dizilerini Alfabe Sırasına Dizmek

Python'da karakter dizilerinin öğelerine tek tek ulaşma, öğeleri dilimleme ve ters çevirmenin yanısıra, bu öğeleri alfabe sırasına dizmek de mümkündür. Bunun için sorted() adlı bir fonksiyondan yararlanacağız:

```
>>> sorted("kitap")

['a', 'i', 'k', 'p', 't']
```

Nasıl input() fonksiyonu çıktı olarak bir karakter dizisi ve len() fonksiyonu bir sayı veriyorsa, sorted() fonksiyonu da bize çıktı olarak, birkaç bölüm sonra inceleyeceğimiz 'liste' adlı bir veri tipi verir.

Ama tabii eğer isterseniz bu çıktıyı alıştığınız biçimde alabilirsiniz:

```
print(*sorted("kitap"), sep="")
```

...veya:

```
for i in sorted("kitap"):
    print(i, end="")
```

Bir örnek daha verelim:

```
>>> sorted("elma")

['a', 'e', 'l', 'm']
```

Gördüğünüz gibi, sorted() fonksiyonunu kullanmak çok kolay, ama aslında bu fonksiyonun önemli bir problemi var. Dikkatlice bakın:

```
>>> sorted("çiçek")

['e', 'i', 'k', 'ç', 'ç']
```

Burada Türkçe bir karakter olan ‘ç’ harfinin düzgün sıralanamadığını görüyoruz. Bu sorun bütün Türkçe karakterler için geçerlidir.

Bu sorunu aşmak için şöyle bir yöntem deneyebilirsiniz:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, "Turkish_Turkey.1254") #Windows için
>>> locale.setlocale(locale.LC_ALL, "tr_TR") #GNU/Linux için
>>> sorted("çiçek", key=locale.strxfrm)

['ç', 'ç', 'e', 'i', 'k']
```

Burada locale adlı bir modülden yararlandık. locale de tıpkı sys, os ve keyword gibi bir modül olup, içinde pek çok değişken ve fonksiyon barındırır.

locale modülü bize belli bir dilin kendine has özelliklerine göre programlama yapma imkanı verir. Örneğin bu modülünün içinde yer alan fonksiyonlardan biri olan setlocale() fonksiyonunu kullanarak, programımızda öntanımlı dil ayarlarına uygun bir şekilde programlama yapma olanağı sağlarız.

Bu modülü ilerleyen derslerde daha ayrıntılı bir şekilde inceleyeceğiz. O yüzden locale modülünü bir kenara bırakıp yolumuza devam edelim.

Yukarıdaki örnekte Türkçe karakterleri doğru sıralayabilmek için sorted() fonksiyonunu nasıl kullandığımıza dikkat edin:

```
>>> sorted("çiçek", key=locale.strxfrm)
```

Burada sorted() metodunun *key* adlı özel bir parametresine *locale.strxfrm* değerini vererek Türkçeye duyarlı bir sıralama yapılmasını sağladık. Yukarıdaki yöntem pek çok durumda işinize yarar. Ancak bu yöntem tek bir yerde işe yaramaz. Dikkatlice bakın:

```
>>> sorted("afgdhkıi", key=locale.strxfrm)

['a', 'd', 'f', 'g', 'h', 'i', 'ı', 'k']
```

Gördüğünüz gibi, bu yöntem ‘i’ harfini ‘ı’ harfinden önce getiriyor. Halbuki Türk alfabesine göre bunun tersi olmalıydı. Buna benzer problemlerle İngiliz alfabesi dışındaki pek çok alfabede karşılaşırsınız. Dolayısıyla bu sadece Türkçeye özgü bir sorun değil.

Bu soruna karşı şöyle bir kod da yazabilirsiniz:

```
>>> harfler = "abcçdefgğhıijklmnoöprsştuüvyz"
>>> çevrim = {i: harfler.index(i) for i in harfler}
>>> sorted("afgdhkıi", key=çevrim.get)

['a', 'd', 'f', 'g', 'h', 'ı', 'i', 'k']
```

Gördüğünüz gibi burada ilk iş olarak Türk alfabesindeki bütün harfleri *harfler* adlı bir değişkene atadık. Daha sonra ise şöyle bir kod yazdık:

```
>>> çevrim = {i: harfler.index(i) for i in harfler}
```

Burada henüz öğrenmediğimiz bir yapı var, ama ne olup bittiğini daha iyi anlamak için bu *çevrim* değişkeninin içeriğini kontrol etmeyi deneyebilirsiniz:

```
>>> print(çevrim)

{'ğ': 8, 'ı': 10, 'v': 26, 'g': 7, 'ş': 22, 'a': 0, 'c': 2, 'b': 1, 'e': 5,
'd': 4, 'ç': 3, 'f': 6, 'i': 11, 'h': 9, 'k': 13, 'j': 12, 'm': 15, 'l': 14,
'o': 17, 'n': 16, 'p': 19, 's': 21, 'r': 20, 'u': 24, 't': 23, 'ö': 18,
'y': 27, 'z': 28, 'ü': 25}
```

Bu çıktıya dikkatlice bakarsanız, her bir harfin bir sayıya karşılık gelecek şekilde birbiriyle eşleştirildiğini göreceksiniz. Mesela 'ğ' harfi *8* ile, 'f' harfi *6* ile eşleşmiş. Yine dikkatlice bakarsanız, biraz önce bize sorun çıkaran 'ı' harfinin *10*, 'i' harfinin ise *11* ile eşleştiğini göreceksiniz. Evet, doğru tahmin ettiniz. Harfleri sayılarla eşleştirerek, Python'ın harfler yerine sayıları sıralamasını sağlayacağız. Bunu da yine *key* parametresini kullanarak yapıyoruz:

```
>>> sorted("afgdhkıi", key=çevrim.get)
```

Bu yapıyı daha iyi anlayabilmek için kendi kendinize bazı denemeler yapın. Eğer burada olan biteni anlamakta zorlanıyorsanız hiç endişe etmeyin. Bir-iki bölüm sonra bunları da kolayca anlayabilecek duruma geleceksiniz. Bizim burada bu bilgileri vermekteki amacımız, Python'ın Türkçe harflerle sıralama işlemini sorunsuz bir şekilde yapabileceğini göstermektir. Bu esnada bir-iki yeni bilgi kırıntısı da kapmanızı sağlayabildiysek kendimizi başarılı sayacağız.

Karakter Dizileri Üzerinde Değişiklik Yapmak

Bu kısımda karakter dizilerinin çok önemli bir özelliğinden söz edeceğiz. Konumuz karakter dizileri üzerinde değişiklik yapmak. İsterseniz neyle karşı karşıya olduğumuzu anlayabilmek için çok basit bir örnek verelim.

Elimizde şöyle bir karakter dizisi olduğunu düşünün:

```
>>> meyve = "elma"
```

Amacımız bu karakter dizisinin ilk harfini büyütmek olsun.

Bunun için dilimleme yönteminden yararlanabileceğimizi biliyorsunuz:

```
>>> "E" + meyve[1:]

'Elma'
```

Burada "E" harfi ile, *meyve* değişkeninin ilk harfi dışında kalan bütün harfleri birleştirdik.

Bir örnek daha verelim.

Elimizde şöyle dört adet internet sitesi adresi olsun:

```
site1 = "www.google.com"
site2 = "www.istihza.com"
site3 = "www.yahoo.com"
site4 = "www.gnu.org"
```

Bizim amacımız bu adreslerin her birinin baş tarafına *http://* ifadesini eklemek. Bunun için de yine karakter dizisi birleştirme işlemlerinden yararlanabiliriz. Dikkatlice inceleyin:

```
site1 = "www.google.com"
site2 = "www.istihza.com"
site3 = "www.yahoo.com"
site4 = "www.gnu.org"

for i in site1, site2, site3, site4:
    print("http://", i, sep="")
```

Eğer *www.* kısımlarını atmak isterseniz karakter dizisi birleştirme işlemleri ile birlikte dilimleme yöntemini de kullanmanız gerekir:

```
for i in site1, site2, site3, site4:
    print("http://", i[4:], sep="")
```

Belki farkındayız, belki de değiliz, ama aslında yukarıdaki örnekler karakter dizileri hakkında bize çok önemli bir bilgi veriyor. Dikkat ettiyseniz yukarıdaki örneklerde karakter dizileri üzerinde bir değişiklik yapmışız gibi görünüyor. Esasında öyle de denebilir. Ancak burada önemli bir ayrıntı var. Yukarıdaki örneklerde gördüğümüz değişiklikler kalıcı değildir. Yani aslında bu değişikliklerin orijinal karakter dizisi üzerinde hiçbir etkisi yoktur. Gelin isterseniz bunu teyit edelim:

```
>>> kardiz = "istihza"
>>> "İ" + kardiz[1:]

'İstihza'
```

Dediğimiz gibi, sanki burada *"istihza"* karakter dizisini *"İstihza"* karakter dizisine çevirmişiz gibi duruyor. Ama aslında öyle değil:

```
>>> print(kardiz)

istihza
```

Gördüğünüz gibi, *kardiz* değişkeninin orijinalinde hiçbir değişiklik yok. Ayrıca burada "İ" + kardiz[1:] satırı ile elde ettiğiniz sonuca tekrar ulaşmanızın imkanı yok. Bu değişiklik kaybolmuş durumda. Peki bunun sebebi nedir?

Bunun nedeni, karakter dizilerinin değiştirilemeyen (*immutable*) bir veri tipi olmasıdır. Python'da iki tür veri tipi bulunur: değiştirilemeyen veri tipleri (*immutable datatypes*) ve değiştirilebilen veri tipleri (*mutable datatypes*). Bizim şimdiye kadar gördüğümüz veri tipleri (sayılar ve karakter dizileri), değiştirilemeyen veri tipleridir. Henüz değiştirilebilen bir veri tipi görmedik. Ama birkaç bölüm sonra değiştirilebilen veri tiplerini de inceleyeceğiz.

Neyse... Dediğimiz gibi, karakter dizileri üzerinde yaptığımız değişikliklerin kalıcı olmamasını nedeni, karakter dizilerinin değiştirilemeyen bir veri tipi olmasıdır. Python'da bir karakter dizisini bir kez tanımladıktan sonra bu karakter dizisi üzerinde artık değişiklik yapamazsınız. Eğer bir karakter dizisi üzerinde değişiklik yapmanız gerekiyorsa, yapabileceğiniz tek şey o karakter dizisini yeniden tanımlamaktır. Mesela yukarıdaki örnekte *kardiz* değişkeninin tuttuğu karakter dizisini değiştirmek isterseniz şöyle bir kod yazabilirsiniz:

```
>>> kardiz = "İ" + kardiz[1:]
>>> print(kardiz)

İstihza
```

Burada yaptığımız şey *kardiz* değişkeninin değerini değiştirmek değildir. Biz burada aslında bambaşka bir *kardiz* değişkeni daha tanımlıyoruz. Yani ilk *kardiz* değişkeni ile sonraki *kardiz* değişkeni aynı şeyler değil. Bunu teyit etmek için önceki derslerimizde gördüğümüz id() fonksiyonundan yararlanabilirsiniz:

```
>>> kardiz = "istihza"
>>> id(kardiz)

3075853248

>>> kardiz = "İ" + kardiz[1:]
>>> id(kardiz)

3075853280
```

Gördüğünüz gibi, ilk *kardiz* değişkeni ile sonraki *kardiz* değişkeni farklı kimlik numaralarına sahip.

Yani bu iki değişken bellek içinde farklı adreslerde tutuluyor. Daha doğrusu, ikinci *kardiz*, ilk *kardiz*'i silip üzerine yazıyor.

Her ne kadar kardiz = "İ" + kardiz[1:] kodu *kardiz*'in değerini aslında değiştirmiyor olsa da, sanki *kardiz* değişkeninin tuttuğu karakter dizisi değişiyormuş gibi bir etki elde ediyoruz. Bu da bizi memnun etmeye yetiyor...

Yukarıdaki örnekte karakter dizisinin baş kısmı üzerinde değişiklik yaptık. Eğer karakter dizisinin ortasında kalan bir kısmı değiştirmek isterseniz de şöyle bir şey yazabilirsiniz:

```
>>> kardiz = "istihza"
>>> kardiz = kardiz[:3] + "İH" + kardiz[5:]
>>> kardiz

'istİHza'
```

Gördüğünüz gibi, yukarıdaki kodlarda karakter dizilerini dilimleyip birleştirerek, yani bir bakıma kesip biçerek istediğimiz çıktıyı elde ettik.

Mesela ilk örnekte *kardiz* değişkeninin ilk karakteri dışında kalan kısmını (kardiz[1:]) "İ" harfi ile birleştirdik ("İ" + kardiz[1:]").

İkinci örnekte ise *kardiz* değişkeninin ilk üç karakterine "İH" ifadesini ekledik ve sonra buna *kardiz* değişkeninin *5.* karakterinden sonraki kısmını ilave ettik.

Karakter dizileri üzerinde değişiklik yapmanızın hangi durumlarda gerekli olacağını gösteren bir örnek daha verip bu konuyu kapatalım.

Diyelim ki, bir kelime içindeki sesli ve sessiz harfleri birbirinden ayırmanız gereken bir program yazıyorsunuz. Yani mesela amacınız 'istanbul' kelimesi içinde geçen 'i', 'a' ve 'u' harflerini bir yerde, 's', 't', 'n', 'b' ve 'l' harflerini ise ayrı bir yerde toplamak. Bunun için şöyle bir program yazabilirsiniz:

```
sesli_harfler = "aeıioöuü"
sessiz_harfler = "bcçdfgğhjklmnprsştvyz"

sesliler = ""
sessizler = ""

kelime = "istanbul"

for i in kelime:
    if i in sesli_harfler:
        sesliler += i
    else:
```

```
        sessizler += i

print("sesli harfler: ", sesliler)
print("sessiz harfler: ", sessizler)
```

Burada öncelikle şu kodlar yardımıyla Türkçedeki sesli ve sessiz harfleri belirliyoruz:

```
sesli_harfler = "aeıioöuü"
sessiz_harfler = "bcçdfgğhjklmnprsştvyz"
```

Ardından da, sesli ve sessiz harflerini ayıklayacağımız kelimedeki sesli harfler ve sessiz harfler için boş birer karakter dizisi tanımlıyoruz:

```
sesliler = ""
sessizler = ""
```

Programımız içinde ilgili harfleri, o harfin ait olduğu değişkene atayacağız.

Kelimemiz *"istanbul"*:

```
kelime = "istanbul"
```

Şimdi bu kelime üzerinde bir for döngüsü kuruyoruz ve kelime içinde geçen herbir harfe tek tek bakıyoruz. Kelime içinde geçen harflerden, *sesli_harfler* değişkeninde tanımlı karakter dizisinde geçenleri *sesliler* adlı değişkene atıyoruz. Aksi durumda ise, yani kelime içinde geçen harflerden, *sessiz_harfler* değişkeninde tanımlı karakter dizisinde geçenleri, *sessizler* adlı değişkene gönderiyoruz:

```
for i in kelime:
    if i in sesli_harfler:
        sesliler += i
    else:
        sessizler += i
```

Bunun için for döngüsü içinde basit bir 'if-else' bloğu tanımladığımızı görüyorsunuz. Ayrıca bunu yaparken, *sesliler* ve *sessizler* adlı değişkenlere, for döngüsünün her bir dönüşünde yeni bir harf gönderip, bu değişkenleri, döngünün her dönüşünde yeni baştan tanımladığımıza dikkat edin. Çünkü, dediğimiz gibi, karakter dizileri değiştirilemeyen veri tipleridir. Bir karakter dizisi üzerinde değişiklik yapmak istiyorsak, o karakter dizisini baştan tanımlamamız gerekir.

Üç Önemli Fonksiyon

Karakter dizilerinin temel özellikleri hakkında söyleyeceklerimizin sonuna geldik sayılır. Biraz sonra karakter dizilerinin çok önemli bir parçası olan metotlardan söz edeceğiz. Ama isterseniz metotlara geçmeden önce, çok önemli üç fonksiyondan söz edelim. Bu fonksiyonlar sadece karakter dizileri ile

değil, başka veri tipleri ile çalışırken de işlerimizi bir hayli kolaylaştıracak.

dir()

İlk olarak dir() adlı özel bir fonksiyondan söz edeceğiz. Bu metot bize Python'daki bir nesnenin özellikleri hakkında bilgi edinme imkanı verecek. Mesela karakter dizilerinin bize hangi metotları sunduğunu görmek için bu fonksiyonu şöyle kullanabiliriz:

```
>>> dir(str)

['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'capitalize', 'center', 'count', 'encode', 'endswith',
'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha',
'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
```

İngilizcede 'karakter dizisi'nin karşılığının *string*, bu kelimenin kısaltmasının da 'str' olduğunu hatırlıyor olmalısınız. İşte dir() fonksiyonuna parametre olarak bu 'str' kelimesini verdiğimizde, Python bize karakter dizilerinin bütün metotlarını listeliyor.

Karakter dizileri dışında, şimdiye kadar öğrendiğimiz başka bir veri tipi de sayılar. Biz Python'da sayıların tam sayılar (*integer*), kayan noktalı sayılar (*float*) ve karmaşık sayılar (*complex*) olarak üçe ayrıldığını da biliyoruz. Örnek olması açısından dir() fonksiyonunu bir de sırasıyla, tam sayılar, kayan noktalı sayılar ve karmaşık sayılar üzerinde de uygulayalım:

```
>>> dir(int)

>>> dir(float)

>>> dir(complex)
```

Gördüğünüz gibi, dir() fonksiyonunu kullanmak için, metotlarını listelemek istediğimiz nesneyi alıp dir() fonksiyonuna parametre olarak veriyoruz. Örneğin yukarıda karakter dizileri için *str*; tam sayılar için *int*; kayan noktalı sayılar için *float*; karmaşık sayılar için ise *complex* parametrelerini kullandık.

dir() fonksiyonunu kullanabilmek için tek yöntemimiz, sorgulamak istediğimiz nesnenin adını kullanmak değil. Mesela karakter dizilerinin metotlarını sorgulamak için 'str' kelimesini

kullanabileceğimiz gibi, herhangi bir karakter dizisini de kullanabiliriz. Yani:

```
>>> dir("")
```

Burada dir() fonksiyonuna parametre olarak boş bir karakter dizisi verdik. Bu kodun dir(str) kodundan hiçbir farkı yoktur. Bu komut da bize karakter dizilerinin metotlarını listeler.

Aynı etkiyi dilersek şöyle de elde edebiliriz:

```
>>> a = "karakter"
>>> dir(a)
```

Karakter dizilerinin metotlarını listelemek için, siz hangi yöntem kolayınıza geliyorsa onu kullanabilirsiniz. Bu satırların yazarı genellikle şu yöntemi kullanıyor:

```
>>> dir("")
```

dir("") komutunun çıktısından da göreceğiniz gibi, karakter dizilerinin epey metodu var. Metot listesi içinde bizi ilgilendirenler başında veya sonunda _ işareti olmayanlar. Yani şunlar:

```
>>> for i in dir(""):
...     if "_" not in i[0]:
...         print(i)
...
```

Bu arada bu metotları listelemek için nasıl bir kod kullandığımıza dikkat edin:

```
for i in dir(""):
    if "_" not in i[0]:
        print(i)
```

Burada dir("") komutunun içerdiği her bir metoda tek tek bakıyoruz. Bu metotlar içinde, ilk harfi _ karakteri olmayan bütün metotları listeliyoruz. Böylece istediğimiz listeyi elde etmiş oluyoruz. İsterseniz ilgilendiğimiz metotların sayısını da çıktıya ekleyebiliriz:

```
sayaç = 0

for i in dir(""):
    if "_" not in i[0]:
        sayaç += 1
        print(i)

print("Toplam {} adet metot ile ilgileniyoruz.".format(sayaç))
```

Burada da, ilk karakteri _ olmayan her bir metot için *sayaç* değişkeninin değerini *1* artırıyoruz.

Böylece programın sonunda *sayaç* değişkeni ilgilendiğimiz metot sayısını göstermiş oluyor.

Eğer her metodun soluna, sıra numarasını da eklemek isterseniz elbette şöyle bir kod da yazabilirsiniz:

```
sayaç = 0

for i in dir(""):
    if "_" not in i[0]:
        sayaç += 1
        print(sayaç, i)

print("Toplam {} adet metot ile ilgileniyoruz.".format(sayaç))
```

Bu noktada bir parantez açalım. Yukarıdaki yöntemi kullanarak metotları numaralandırabilirsiniz. Ama aslında Python bize numaralandırma işlemleri için özel bir fonksiyon sunar. Şimdi isterseniz bu özel fonksiyonu inceleyelim.

enumerate()

Eğer yazdığınız bir programda numaralandırmaya ilişkin işlemler yapmanız gerekiyorsa Python'ın size sunduğu çok özel bir fonksiyondan yararlanabilirsiniz. Bu fonksiyonun adı enumerate().

Gelelim bu fonksiyonun nasıl kullanılacağına... Önce şöyle bir deneme yapalım:

```
>>> enumerate("istihza")

<enumerate object at 0x00E3BC88>
```

Tıpkı reversed() fonksiyonunun bir 'reversed' nesnesi vermesi gibi, bu fonksiyonun da bize yalnızca bir 'enumerate' nesnesi verdiğini görüyorsunuz.

reversed() fonksiyonunu kullanabilmek için şöyle bir kod yazmıştık:

```
>>> print(*reversed("istihza"))
```

enumerate() için de benzer bir şeyi deneyebiliriz:

```
>>> print(*enumerate("istihza"))
```

Burada şu çıktıyı aldık:

```
(0, 'i') (1, 's') (2, 't') (3, 'i') (4, 'h') (5, 'z') (6, 'a')
```

Enumerate kelimesi İngilizcede 'numaralamak, numaralandırmak' gibi anlamlara gelir. Dolayısıyla

enumerate() fonksiyonu, kendisine parametre olarak verilen değer hakkında bize iki farklı bilgi verir: Bir öğe ve bu öğeye ait bir sıra numarası. Yukarıdaki çıktıda gördüğünüz şey de işte her bir öğenin kendisi ve o öğeye ait bir sıra numarasıdır.

Yukarıdaki çıktıyı daha iyi anlayabilmek için bir for döngüsü kullanmak daha açıklayıcı olabilir:

```
>>> for i in enumerate("istihza"):
...     print(i)
...
(0, 'i')
(1, 's')
(2, 't')
(3, 'i')
(4, 'h')
(5, 'z')
(6, 'a')
```

Gördüğünüz gibi, gerçekten de bu fonksiyon bize bir öğe (mesela 'i' harfi) ve bu öğeye ait bir sıra numarası (mesela *0*) veriyor.

Hatırlarsanız, enumerate() fonksiyonunu öğrenmeden önce, dir("") komutundan elde ettiğimiz çıktıları şu şekilde numaralandırabileceğimizi söylemiştik:

```
sayaç = 0

for i in dir(""):
    if "_" not in i[0]:
        sayaç += 1
        print(sayaç, i)
```

Ama artık enumerate() fonksiyonunu öğrendiğimize göre, aynı işi çok daha verimli bir şekilde gerçekleştirebiliriz:

```
for sıra, metot in enumerate(dir("")):
    print(sıra, metot)
```

enumerate() metodunun verdiği her bir çıktının iki öğeli olduğunu biliyoruz (öğenin kendisi ve o öğenin sıra numarası). Yukarıdaki kodlar yardımıyla, bu öğelerin her birini ayrı bir değişkene (*sıra* ve *metot*) atamış oluyoruz. Böylece bu çıktıyı manipüle etmek bizim için daha kolay oluyor. Mesela bu özelliği kullanarak metot ve sıra numarasının yerlerini değiştirebiliriz:

```
>>> for sıra, metot in enumerate(dir("")):
...     print(metot, sıra)
...
```

```
__add__ 0
__class__ 1
__contains__ 2
__delattr__ 3
__doc__ 4
__eq__ 5
__format__ 6
__ge__ 7

(...)
```

Pratik olması açısından şöyle bir örnek daha verelim:

```
>>> for sıra, metot in enumerate(dir("")):
...     print(sıra, metot, len(metot))
...
0 __add__ 7
1 __class__ 9
2 __contains__ 12
3 __delattr__ 11
4 __doc__ 7
5 __eq__ 6

(...)
```

Burada, dir("") ile elde ettiğimiz metotların sırasını (*sıra*), bu metotların adlarını (*metot*) ve her bir metodun kaç karakterden oluştuğunu (len(metot)) gösteren bir çıktı elde ettik.

Bu arada, gördüğünüz gibi, enumerate() fonksiyonu numaralandırmaya *0*'dan başlıyor. Elbette eğer isterseniz bu fonksiyonun numaralandırmaya kaçtan başlayacağını kendiniz de belirleyebilirsiniz. Dikkatlice bakın:

```
>>> for sıra, harf in enumerate("istihza", 1):
...     print(sıra, harf)
...
1 i
2 s
3 t
4 i
5 h
6 z
7 a
```

Burada 'istihza' kelimesi içindeki harfleri numaralandırdık. Bunu yaparken de numaralandırmaya

1'den başladık. Bunun için enumerate() fonksiyonuna ikinci bir parametre verdiğimize dikkat edin.

enumerate() fonksiyonunu da incelediğimize göre önemli bir başka fonksiyondan daha söz edebiliriz.

help()

Python'la ilgili herhangi bir konuda yardıma ihtiyacınız olduğunda, internetten araştırma yaparak pek çok ayrıntılı belgeye ulaşabilirsiniz. Ama eğer herhangi bir nesne hakkında hızlı bir şekilde ve İngilizce olarak yardım almak isterseniz help() adlı özel bir fonksiyondan yararlanabilirsiniz.

Bu fonksiyonu iki farklı şekilde kullanıyoruz. Birinci yöntemde, etkileşimli kabuğa help() yazıp *Enter* düğmesine basıyoruz:

```
>>> help()

Welcome to Python 3.3!  This is the interactive help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.3/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help>
```

Gördüğünüz gibi, Python bu komutu verdiğimizde özel bir yardım ekranı açıyor bize. Bu ekranda >>> yerine *help>* ifadesinin olduğuna dikkat edin. Mesela dir() fonksiyonu hakkında bilgi almak için *help>* ifadesinden hemen sonra, hiç boşluk bırakmadan, şu komutu verebiliriz:

```
help> dir
```

Bu komut bize şu çıktıyı veriyor:

```
Help on built-in function dir in module builtins:

dir(...)
    dir([object]) -> list of strings
```

```
   If called without an argument, return the names in the current scope.
   Else, return an alphabetized list of names comprising (some of) the attributes
   of the given object, and of attributes reachable from it.
   If the object supplies a method named __dir__, it will be used; otherwise
   the default dir() logic is used and returns:
     for a module object: the module's attributes.
     for a class object:  its attributes, and recursively the attributes of its bases.
     for any other object: its attributes, its class's attributes, and
     recursively the attributes of its class's base classes.
```

Gördüğünüz gibi, dir() fonksiyonunun ne işe yaradığı ve nasıl kullanıldığı konusunda ayrıntılı bir bilgi ediniyoruz. Bu arada, hakkında bilgi almak istediğimiz fonksiyonu parantezsiz yazdığımıza dikkat edin.

Örnek olması açısından mesela bir de len() fonksiyonu hakkında bilgi edinelim:

```
help> len

Help on built-in function len in module builtins:

len(...)
   len(object) -> integer

   Return the number of items of a sequence or mapping.
```

'help' ekranından çıkmak için *Enter* düğmesine basabilir veya quit komutu verebilirsiniz.

En başta da dediğimiz gibi Python'da etkileşimli kabuk üzerinde İngilizce yardım almak için iki farklı yöntem kullanabiliyoruz. Bu yöntemlerden ilkini yukarıda anlattık. İkincisi ise doğrudan etkileşimli kabukta şu komutu kullanmaktır: (Mesela dir() fonksiyonu hakkında yardım alalım...)

```
>>> help(dir)

Help on built-in function dir in module builtins:

dir(...)
   dir([object]) -> list of strings

   If called without an argument, return the names in the current scope.
   Else, return an alphabetized list of names comprising (some of) the attributes
   of the given object, and of attributes reachable from it.
   If the object supplies a method named __dir__, it will be used; otherwise
   the default dir() logic is used and returns:
```

```
    for a module object: the module's attributes.
    for a class object:  its attributes, and recursively the attributes of its bases.
    for any other object: its attributes, its class's attributes, and
    recursively the attributes of its class's base classes.
```

Gördüğünüz gibi, 'help' ekranını açmadan, doğrudan etkileşimli kabuk üzerinden de help() fonksiyonunu herhangi bir fonksiyon gibi kullanıp, hakkında yardım almak istediğimiz nesneyi help() fonksiyonunun parantezleri içine parametre olarak yazabiliyoruz.

Böylece dir(), enumerate() ve help() adlı üç önemli fonksiyonu da geride bırakmış olduk. Dilerseniz şimdi karakter dizilerine dair birkaç ufak not düşelim.

Notlar

Hatırlarsanız döngüleri anlatırken şöyle bir örnek vermiştik:

```
tr_harfler = "şçöğüİı"
a = 0

while a < len(tr_harfler):
    print(tr_harfler[a], sep="\n")
    a += 1
```

Bu kodların for döngüsü ile yazılabilecek olan şu kodlara alternatif olduğundan söz etmiştik:

```
tr_harfler = "şçöğüİı"

for tr_harf in tr_harfler:
    print(tr_harf)
```

Yukarıdaki while örneğini verirken, henüz karakter dizilerinin öğelerine tek tek nasıl erişebileceğimizi öğrenmemiştik. Ama artık bu konuyu da öğrendiğimiz için yukarıdaki while döngüsünü rahatlıkla anlayabiliyoruz:

```
while a < len(tr_harfler):
    print(tr_harfler[a], sep="\n")
    a += 1
```

Burada yaptığımız şey şu: *a* değişkeninin değeri *tr_harfler* değişkeninin uzunluğundan (len(tr_harfler)) küçük olduğu müddetçe *a* değişkeninin değerini *1* sayı artırıp yine *a* değişkenine gönderiyoruz (a += 1).

while döngüsünün her dönüşünde de, *a* değişkeninin yeni değeri yardımıyla *tr_harfler* adlı karakter dizisinin öğelerine tek tek ve sırayla erişiyoruz (print(tr_hafler[a])).

Yine hatırlarsanız, önceki derslerimizde *sys* adlı bir modül içindeki *version* adlı bir değişkenden söz etmiştik. Bu değişken bize kullandığımız Python'ın sürümünü bir karakter dizisi olarak veriyordu:

```
>>> import sys
>>> sys.version
```

Buradan şu çıktıyı alıyoruz:

```
'3.3.2 (default, May 20 2013, 16:10:12) n[GCC 4.4.7 20120313 (Red Hat 4.4.7-3)]'
```

Bu çıktıda, kullandığımız Python sürümünün dışında başka birtakım bilgiler de var. İşte biz eğer istersek, bu bölümde öğrendiğimiz bilgileri kullanarak bu karakter dizisinin istediğimiz kısmını, mesela sadece sürüm bilgisini karakter dizisinin içinden dilimleyip alabiliriz:

```
>>> sys.version[:5]
```

```
'3.3.2'
```

Elbette, yukarıdaki karakter dizisini elde etmek için, kullanması ve yönetmesi daha kolay bir araç olan *version_info* değişkeninden de yararlanabilirdiniz:

```
>>> '{}.{}.{}'.format(sys.version_info.major, sys.version_info.minor,
sys.version_info.micro)
```

```
'3.3.2'
```

Ancak burada şöyle bir sorun olduğunu biliyorsunuz: Python'ın 2.7 öncesi sürümlerinde *version_info*'nun *major*, *minor* ve *micro* gibi nitelikleri yok. Dolayısıyla 2.7 öncesi sürümlerde *version_info*'yu kullanırken hata almamak için try... except bloklarından yararlanabileceğimizi görmüştük. Ancak *version_info*'yu bütün Python sürümlerinde güvenli bir şekilde kullanmanın başka bir yöntemi daha var. Dikkatlice bakın:

```
>>> major = sys.version_info[0]
>>> minor = sys.version_info[1]
>>> micro = sys.version_info[2]

>>> print(major, minor, micro, sep=".")
```

```
3.3.2
```

Bu yöntem bütün Python sürümlerinde çalışır. Dolayısıyla, farklı Python sürümlerinde çalışmasını tasarladığınız programlarınızda sürüm kontrolünü *sys.version_info*'nun *major*, *minor* veya *micro* nitelikleri ile yapmak yerine yukarıdaki yöntemle yapabilirsiniz:

```
if sys.version_info[1] < 3:
    print("Kullandığınız Python sürümü eski!")
```

Gördüğünüz gibi, karakter dizisi dilimleme işlemleri pek çok farklı kullanım alanına sahip. Programlama maceranız boyunca karakter dizilerinin bu özelliğinden bol bol yararlanacağınızdan hiç kuşkunuz olmasın.

başa dön

Karakter Dizilerinin Metotları

Geçen bölümde karakter dizilerinin genel özelliklerinden söz ettik. Bu ikinci bölümde ise karakter dizilerini biraz daha ayrıntılı bir şekilde incelemeye ve karakter dizilerinin yepyeni özelliklerini görmeye başlayacağız.

Hatırlarsanız, geçen bölümün en başında, metot diye bir şeyden söz edeceğimizi söylemiştik. Orada da kabaca tarif ettiğimiz gibi, metotlar Python'da nesnelerin niteliklerini değiştirmemizi, sorgulamamızı veya bu nesnelere yeni özellikler katmamızı sağlayan araçlardır. Metotlar sayesinde karakter dizilerini istediğimiz gibi eğip bükebileceğiz.

Geçen bölümün sonlarına doğru, bir karakter dizisinin hangi metotlara sahip olduğunu şu komut yardımıyla listeleyebileceğimizi öğrenmiştik:

```
>>> dir("")
```

Bu komutu verdiğinizde aldığınız çıktıdan da gördüğünüz gibi, karakter dizilerinin *40*'ın üzerinde metodu var. Dolayısıyla metot sayısının çokluğu gözünüzü korkutmuş olabilir. Ama aslında buna hiç lüzum yok. Çünkü programcılık maceranızda bu metotların bazılarını ya çok nadiren kullanacaksınız, ya da hiç kullanmayacaksınız. Çok kullanılan metotlar belli başlıdır. Elbette bütün metotlar hakkında fikir sahibi olmak gerekir. Zaten siz de göreceksiniz ki, bu metotlar kullandıkça aklınızda kalacak. Doğal olarak çok kullandığınız metotları daha kolay öğreneceksiniz. Eğer bir program yazarken hangi metodu kullanmanız gerektiğini veya kullanacağınız metodun ismini hatırlayamazsanız etkileşimli kabukta dir("") gibi bir komut verip çıkan sonucu incelemek pek zor olmasa gerek. Ayrıca hatırlayamadığınız bir metot olması durumunda dönüp bu sayfaları tekrar gözden geçirme imkanına da sahipsiniz. Unutmayın, bütün metotları ve bu metotların nasıl kullanıldığını ezbere bilmeniz zaten beklenmiyor. Metotları hatırlayamamanız gayet normal. Böyle bir durumda referans kitaplarına bakmak en doğal hakkınız.

replace()

Karakter dizisi metotları arasında inceleyeceğimiz ilk metot replace() metodu olacak. *replace* kelimesi Türkçede 'değiştirmek, yerine koymak' gibi anlamlar taşır. İşte bu metodun yerine getirdiği görev de tam olarak budur. Yani bu metodu kullanarak bir karakter dizisi içindeki karakterleri başka karakterlerle değiştirebileceğiz.

Peki bu metodu nasıl kullanacağız? Hemen bir örnek verelim:

```
>>> kardiz = "elma"
```

Burada *"elma"* değerini taşıyan *kardiz* adlı bir karakter dizisi tanımladık. Şimdi bu karakter dizisinin içinde geçen *"e"* harfini *"E"* ile değiştirelim. Dikkatlice bakın:

```
>>> kardiz.replace("e", "E")
```

```
'Elma'
```

Gördüğünüz gibi, replace() son derece yararlı ve kullanımı oldukça kolay bir metot. Bu arada bu ilk metodumuz sayesinde Python'daki metotların nasıl kullanılacağı konusunda da bilgi edinmiş olduk. Yukarıdaki örneklerin bize gösterdiği gibi şöyle bir formülle karşı karşıyayız:

```
karakter_dizisi.metot(parametre)
```

Metotlar karakter dizilerinden nokta ile ayrılır. Python'da bu yönteme 'noktalı gösterim' (*dot notation*) adı verilir.

Bu arada metotların görünüş ve kullanım olarak fonksiyonlara ne kadar benzediğine dikkat edin. Tıpkı fonksiyonlarda olduğu gibi, metotlar da birtakım parametreler alabiliyor.

Yukarıdaki örnekte, replace() metodunun iki farklı parametre aldığını görüyoruz. Bu metoda verdiğimiz ilk parametre değiştirmek istediğimiz karakter dizisini gösteriyor. İkinci parametre ise birinci parametrede belirlediğimiz karakter dizisinin yerine ne koyacağımızı belirtiyor. Yani replace() metodu şöyle bir formüle sahiptir:

```
karakter_dizisi.replace(eski_karakter_dizisi, yeni_karakter_dizisi)
```

Gelin isterseniz elimizin alışması için replace() metoduyla birkaç örnek daha verelim:

```
>>> kardiz = "memleket"
>>> kardiz.replace("ket", "KET")

'memleKET'
```

Burada gördüğünüz gibi, replace() metodu aynı anda birden fazla karakteri değiştirme yeteneğine de sahip.

replace() metodunun iki parametreden oluştuğunu, ilk parametrenin değiştirilecek karakter dizisini, ikinci parametrenin ise ilk karakter dizisinin yerine geçecek yeni karakter dizisini gösterdiğini söylemiştik. Aslında replace() metodu üçüncü bir parametre daha alır. Bu parametre ise bir karakter dizisi içindeki karakterlerin kaç tanesinin değiştirileceğini gösterir. Eğer bu parametreyi belirtmezsek replace() metodu ilgili karakterlerin tamamını değiştirir. Yani:

```
>>> kardiz = "memleket"

>>> kardiz.replace("e", "")

'mmlkt'
```

Gördüğünüz gibi, replace() metodunu iki parametre ile kullanıp üçüncü parametreyi

belirtmediğimizde, *"memleket"* kelimesi içindeki bütün *"e"* harfleri boş karakter dizisi ile değiştiriliyor (yani bir anlamda siliniyor).

Şimdi şu örneğe bakalım:

```
>>> kardiz.replace("e", "", 1)

'mmleket'
```

Burada replace() metodunu üçüncü bir parametre ile birlikte kullandık. Üçüncü parametre olarak *1* sayısını verdiğimiz için replace() metodu sadece tek bir *"e"* harfini sildi.

Bu üçüncü parametreyi, silmek istediğiniz harf sayısı kadar artırabilirsiniz. Mesela:

```
>>> kardiz.replace("e", "", 2)

'mmlket'

>>> kardiz.replace("e", "", 3)

'mmlkt'
```

Burada ilk örnekte üçüncü parametre olarak *2* sayısını kullandığımız için, 'replace' işleminden karakter dizisi içindeki *2* adet *"e"* harfi etkilendi. Üçüncü örnekte ise *"memleket"* adlı karakter dizisi içinde geçen üç adet *"e"* harfi değişiklikten etkilendi.

Karakter dizileri konusunun ilk bölümünde 'değiştirilebilirlik' (*mutability*) üzerine söylediğimiz şeylerin burada da geçerli olduğunu unutmayın. Orada da söylediğimiz gibi, karakter dizileri değiştirilemeyen veri tipleridir. Dolayısıyla eğer bir karakter dizisi üzerinde değişiklik yapmak istiyorsanız, o karakter dizisini baştan tanımlamalısınız. Örneğin:

```
>>> meyve = "elma"
>>> meyve = meyve.replace("e", "E")
>>> meyve

'Elma'
```

Böylece replace() metodunu incelemiş olduk. Sırada üç önemli metot var.

split(), rsplit(), splitlines()

Şimdi size şöyle bir soru sorduğumu düşünün: Acaba aşağıdaki karakter dizisinde yer alan bütün kelimelerin ilk harfini nasıl alırız?

```
>>> kardiz = "İstanbul Büyükşehir Belediyesi"
```

Yani diyorum ki burada *"İBB"* gibi bir çıktıyı nasıl elde ederiz?

Sadece bu karakter dizisi söz konusu ise, elbette karakter dizilerinin dilimlenme özelliğinden yararlanarak, *kardiz* değişkeni içindeki *"İ"*, *"B"*, ve *"B"* harflerini tek tek alabiliriz:

```
>>> print(kardiz[0], kardiz[9], kardiz[20], sep="")

İBB
```

Ancak bu yöntemin ne kadar kullanışsız olduğu ortada. Çünkü bu metot yalnızca *"İstanbul Büyükşehir Belediyesi"* adlı karakter dizisi için geçerlidir. Eğer karakter dizisi değişirse bu yöntem de çöpe gider. Bu soruna genel bir çözüm üretebilsek ne güzel olurdu, değil mi?

İşte Python'da bu sorunu çözmemizi sağlayacak çok güzel bir metot bulunur. Bu metodun adı split().

Bu metodun görevi karakter dizilerini belli noktalardan bölmektir. Zaten *split* kelimesi Türkçede 'bölmek, ayırmak' gibi anlamlara gelir. İşte bu metot, üzerine uygulandığı karakter dizilerini parçalarına ayırır. Örneğin:

```
>>> kardiz = "İstanbul Büyükşehir Belediyesi"
>>> kardiz.split()

['İstanbul', 'Büyükşehir', 'Belediyesi']
```

Gördüğünüz gibi bu metot sayesinde *"İstanbul Büyükşehir Belediyesi"* adlı karakter dizisini kelimelere bölmeyi başardık. Eğer bu çıktı üzerine bir for döngüsü uygularsak şöyle bir sonuç elde ederiz:

```
>>> for i in kardiz.split():
...     print(i)
...
İstanbul
Büyükşehir
Belediyesi
```

Artık bu bilgiyi kullanarak şöyle bir program yazabiliriz:

```
kardiz = input("Kısaltmasını öğrenmek istediğiniz kurum adını girin: ")

for i in kardiz.split():
    print(i[0], end="")
```

Burada kullanıcı hangi kurum adını girerse girsin, bu kurum adının her kelimesinin ilk harfi ekrana dökülecektir. Örneğin kullanıcı burada *"Türkiye Büyük Millet Meclisi"* ifadesini girmişse split() metodu öncelikle bu ifadeyi alıp şu şekle dönüştürür:

```
['Türkiye', 'Büyük', 'Millet', 'Meclisi']
```

Daha sonra biz bu çıktı üzerinde bir for döngüsü kurarsak bu kelime grubunun her bir öğesine tek tek müdahale etme imkanına erişiriz. Örneğin yukarıdaki programda bu kelime grubunun her bir öğesinin ilk harfini tek tek ekrana döktük ve *"TBMM"* çıktısını elde ettik.

Yukarıdaki örneklerde split() metodunu herhangi bir parametre içermeyecek şekilde kullandık. Yani metodun parantezleri içine herhangi bir şey eklemedik. split() metodunu bu şekilde parametresiz olarak kullandığımızda bu metot karakter dizilerini bölerken boşluk karakterini ölçüt alacaktır. Yani karakter dizisi içinde karşılaştığı her boşluk karakterinde bir bölme işlemi uygulayacaktır. Ama bazen istediğimiz şey, bir karakter dizisini boşluklardan bölmek değildir. Mesela şu örneğe bakalım:

```
>>> kardiz = "Bolvadin, Kilis, Siverek, İskenderun, İstanbul"
```

Eğer bu karakter dizisi üzerine split() metodunu parametresiz olarak uygularsak şöyle bir çıktı elde ederiz:

```
['Bolvadin,', 'Kilis,', 'Siverek,', 'İskenderun,', 'İstanbul']
```

split() metoduna herhangi bir parametre vermediğimiz için bu metot karakter dizisi içindeki kelimeleri boşluklardan böldü. Bu yüzden karakter dizisi içindeki virgül işaretleri de bölünen kelimeler içinde görünüyor:

```
>>> kardiz = kardiz.split()
>>> for i in kardiz:
...     print(i)
...
Bolvadin,
Kilis,
Siverek,
İskenderun,
İstanbul
```

Bu arada tıpkı replace() metodunu anlatırken gösterdiğimiz gibi, kardiz.split() ifadesini de yine *kardiz* adını taşıyan bir değişkene atadık. Böylece kardiz.split() komutu ile elde ettiğimiz değişiklik kaybolmamış oldu. Karakter dizilerinin değiştirilemeyen bir veri tipi olduğunu biliyorsunuz. Dolayısıyla yukarıdaki karakter dizisi üzerine split() metodunu uyguladığımızda aslında orijinal karakter dizisi üzerinde herhangi bir değişiklik yapmış olmuyoruz. Çıktıda görünen değişikliğin orijinal karakter dizisini etkileyebilmesi için eski karakter dizisini silip, yerine yeni değerleri yazmamız gerekiyor. Bunu da kardiz = kardiz.split() gibi bir komutla hallediyoruz.

Nerede kalmıştık? Gördüğünüz gibi split() metodu parametresiz olarak kullanıldığında karakter dizisini boşluklardan bölüyor. Ama yukarıdaki örnekte karakter dizisini boşluklardan değil de virgüllerden bölsek çok daha anlamlı bir çıktı elde edebiliriz.

Dikkatlice inceleyin:

```
>>> kardiz = "Bolvadin, Kilis, Siverek, İskenderun, İstanbul"
>>> kardiz = kardiz.split(",")
>>> print(kardiz)

['Bolvadin', ' Kilis', ' Siverek', ' İskenderun', ' İstanbul']

>>> for i in kardiz:
...     print(i)
...
Bolvadin
Kilis
Siverek
İskenderun
İstanbul
```

Gördüğünüz gibi, split() metodu tam da istediğimiz gibi, karakter dizisini bu kez boşluklardan değil virgüllerden böldü. Peki bunu nasıl başardı? Aslında bu sorunun cevabı gayet net bir şekilde görünüyor. Dikkat ederseniz yukarıdaki örnekte split() metoduna parametre olarak virgül karakter dizisini verdik. Yani şöyle bir şey yazdık:

```
kardiz.split(",")
```

Bu sayede split() metodu karakter dizisini virgüllerden bölmeyi başardı. Tahmin edebileceğiniz gibi, split() metoduna hangi parametreyi verirseniz bu metot ilgili karakter dizisini o karakterin geçtiği yerlerden bölecektir. Yani mesela siz bu metoda *"l"* parametresini verirseniz, bu metot da 'l' harfi geçen yerden karakter dizisini bölecektir:

```
>>> kardiz.split("l")

['Bo', 'vadin, Ki', 'is, Siverek, İskenderun, İstanbu', '']

>>> for i in kardiz.split("l"):
...     print(i)
...
Bo
vadin, Ki
is, Siverek, İskenderun, İstanbu
```

Eğer parametre olarak verdiğiniz değer karakter dizisi içinde hiç geçmiyorsa karakter dizisi üzerinde herhangi bir değişiklik yapılmaz:

```
>>> kardiz.split("z")
```

```
['Bolvadin, Kilis, Siverek, İskenderun, İstanbul']
```

Aynı şey, split() metodundan önce öğrendiğimiz replace() metodu için de geçerlidir. Yani eğer değiştirilmek istenen karakter, karakter dizisi içinde yer almıyorsa herhangi bir işlem yapılmaz.

split() metodu çoğunlukla, yukarıda anlattığımız şekilde parametresiz olarak veya tek parametre ile kullanılır. Ama aslında bu metot ikinci bir parametre daha alır. Bu ikinci parametre, karakter dizisinin kaç kez bölüneceğini belirler:

```
>>> kardiz = "Ankara Büyükşehir Belediyesi"

>>> kardiz.split(" ", 1)

['Ankara', 'Büyükşehir Belediyesi']

>>> kardiz.split(" ", 2)

['Ankara', 'Büyükşehir', 'Belediyesi']
```

Gördüğünüz gibi, ilk örnekte kullandığımız *1* sayısı sayesinde bölme işlemi karakter dizisi üzerine bir kez uygulandı. İkinci örnekte ise *2* sayısının etkisiyle karakter dizimiz iki kez bölme işlemine maruz kaldı.

Elbette, split() metodunun ikinci parametresini kullanabilmek için ilk parametreyi de mutlaka yazmanız gerekir. Aksi halde Python ne yapmaya çalıştığınızı anlayamaz:

```
>>> kardiz.split(2)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Gördüğünüz gibi, ilk parametreyi es geçip doğrudan ikinci parametreyi yazmaya çalıştığımızda Python parametre olarak verdiğimiz *2* sayısının bölme ölçütü olduğunu zannediyor. Yukarıdaki hatayı engellemek için bölme ölçütünü de açıkça belirtmemiz gerekir. Yukarıdaki örnekte bölme ölçütümüz bir adet boşluk karakteri idi. Bildiğiniz gibi, bölme ölçütü herhangi bir şey olabilir. Mesela virgül.

```
>>> arkadaşlar = "Ahmet, Mehmet, Kezban, Mualla, Süreyya, Veli"
>>> arkadaşlar.split(",", 3)

['Ahmet', ' Mehmet', ' Kezban', ' Mualla, Süreyya, Veli']
```

Burada da bölme ölçütü olarak virgül karakterini kullandık ve *arkadaşlar* adlı karakter dizisi üzerine *3* kez bölme işlemi uyguladık. İlk bölme işlemi *"Ahmet"* karakter dizisini; ikinci bölme işlemi

"Mehmet" karakter dizisini; üçüncü bölme işlemi ise *"Kezban"* karakter dizisini ayırdı. *arkadaşlar* adlı karakter dizisinin geri kalanını oluşturan *"Mualla, Süreyya, Veli"* kısmı ise herhangi bir bölme işlemine tabi tutulmadan tek parça olarak kaldı.

split() metoduyla son bir örnek verip yolumuza devam edelim.

Bildiğiniz gibi *sys* modülünün *version* değişkeni bize bir karakter dizisi veriyor:

'3.3.2 (default, May 20 2013, 16:10:12) n[GCC 4.4.7 20120313 (Red Hat 4.4.7-3)]'

Bu karakter dizisi içinden yalnızca sürüm kısmını ayıklamak için karakter dizilerinin dilimlenme özelliğinden yararlanabiliyoruz:

```
>>> sürüm = sys.version
>>> print(sürüm[:5])
```

3.3.2

Bu işlemin bir benzerini split() metoduyla da yapabiliriz. Dikkatlice inceleyin:

```
>>> sürüm = sys.version
>>> sürüm.split()

['3.3.0', '(v3.3.0:bd8afb90ebf2,', 'Sep', '29', '2012,', '10:55:48)',
 '[MSC', 'v.1600', '32', 'bit', '(Intel)]']
```

Gördüğünüz gibi, sys.version komutuna split() metodunu uyguladığımızda, üzerinde işlem yapması çok daha kolay olan bir veri tipi elde ediyoruz. Bu veri tipinin adı 'liste'. Önceki derslerimizde öğrendiğimiz dir() fonksiyonunun da liste adlı bu veri tipini verdiğini hatırlıyor olmalısınız. İlerleyen derslerde, tıpkı karakter dizileri ve sayılar adlı veri tipleri gibi, liste adlı veri tipini de bütün ayrıntılarıyla inceleyeceğiz. Şimdilik biz sadece **bazı durumlarda** liste veri tipinin karakter dizilerine kıyasla daha kullanışlı bir veri tipi olduğunu bilelim yeter.

Yukarıdaki örnekten de gördüğünüz gibi, sys.version komutunun çıktısını split() metodu yardımıyla boşluklardan bölerek bir liste elde ettik. Bu listenin ilk öğesi, kullandığımız Python serisinin sürüm numarasını verecektir:

```
>>> print(sürüm.split()[0])
```

3.3.2

Böylece split() metodunu öğrenmiş olduk. Gelelim rsplit() metoduna...

rsplit() metodu her yönüyle split() metoduna benzer. split() ile rsplit() arasındaki tek fark, split() metodunun karakter dizisini soldan sağa, rsplit() metodunun ise sağdan sola doğru okumasıdır. Şu

örnekleri dikkatlice inceleyerek bu iki metot arasındaki farkı bariz bir şekilde görebilirsiniz:

```
>>> kardiz.split(" ", 1)

['Ankara', 'Büyükşehir Belediyesi']

>>> kardiz.rsplit(" ", 1)

['Ankara Büyükşehir', 'Belediyesi']
```

Gördüğünüz gibi, split() metodu karakter dizisini soldan sağa doğru okuduğu için bölme işlemini *"Ankara"* karakter dizisine uyguladı. rsplit() metodu ise karakter dizisini sağdan sola soğru okuduğu için bölme işlemini *"Belediyesi"* adlı karakter dizisine uyguladı.

rsplit() metodunun pek yaygın kullanılan bir metot olmadığını belirterek splitlines() metoduna geçelim.

Bildiğiniz gibi, split() metodunu bir karakter dizisini kelime kelime ayırabilmek için kullanabiliyoruz. splitlines() metodunu ise bir karakter dizisini satır satır ayırmak için kullanabiliriz. Mesela elinizde uzun bir metin olduğunu ve amacınızın bu metin içindeki herbir satırı ayrı ayrı almak olduğunu düşünün. İşte splitlines() metoduyla bu amacınızı gerçekleştirebilirsiniz. Hemen bir örnek verelim:

```
metin = """Python programlama dili Guido Van Rossum adlı Hollandalı bir programcı
tarafından 90'lı yılların başında geliştirilmeye başlanmıştır. Çoğu insan, isminin
Python olmasına bakarak, bu programlama dilinin, adını piton yılanından aldığını
düşünür. Ancak zannedildiğinin aksine bu programlama dilinin adı piton yılanından
gelmez. Guido Van Rossum bu programlama dilini, The Monty Python adlı bir İngiliz
komedi grubunun, Monty Python's Flying Circus adlı gösterisinden esinlenerek
adlandırmıştır. Ancak her ne kadar gerçek böyle olsa da, Python programlama
dilinin pek çok yerde bir yılan figürü ile temsil edilmesi neredeyse bir gelenek
halini almıştır diyebiliriz."""

print(metin.splitlines())
```

Bu programı çalıştırdığınızda şöyle bir çıktı alırsınız:

```
['Python programlama dili Guido Van Rossum adlı Hollandalı bir programcı ',
"tarafından 90'lı yılların başında geliştirilmeye başlanmıştır. Çoğu insan,
isminin", 'Python olmasına bakarak, bu programlama dilinin, adını piton
yılanından aldığını ', 'düşünür. Ancak zannedildiğinin aksine bu programlama
dilinin adı piton yılanından ', 'gelmez. Guido Van Rossum bu programlama
dilini, The Monty Python adlı bir İngiliz ', "komedi grubunun, Monty Python's
Flying Circus adlı gösterisinden esinlenerek ", 'adlandırmıştır. Ancak her ne
```

```
kadar gerçek böyle olsa da, Python programlama ', 'dilinin pek çok yerde bir
yılan figürü ile temsil edilmesi neredeyse bir gelenek ', 'halini almıştır
diyebiliriz.']
```

Gördüğünüz gibi, metnimiz *Enter* tuşuna bastığımız noktalardan bölündü. Biz henüz bu çıktıyı nasıl değerlendireceğimizi ve bu çıktıdan nasıl yararlanacağımızı bilmiyoruz. Ayrıca şu anda bu çıktı gözünüze çok anlamlı görünmemiş olabilir. Ama 'Listeler' adlı konuyu öğrendiğimizde bu çıktı size çok daha anlamlı görünecek.

splitlines() metodu yukarıdaki gibi parametresiz olarak kullanılabileceği gibi, bir adet parametre ile de kullanılabilir. Bunu bir örnek üzerinde gösterelim:

```
metin = """Python programlama dili Guido Van Rossum adlı Hollandalı bir programcı
tarafından 90'lı yılların başında geliştirilmeye başlanmıştır. Çoğu insan, isminin
Python olmasına bakarak, bu programlama dilinin, adını piton yılanından aldığını
düşünür. Ancak zannedildiğinin aksine bu programlama dilinin adı piton yılanından
gelmez. Guido Van Rossum bu programlama dilini, The Monty Python adlı bir İngiliz
komedi grubunun, Monty Python's Flying Circus adlı gösterisinden esinlenerek
adlandırmıştır. Ancak her ne kadar gerçek böyle olsa da, Python programlama
dilinin pek çok yerde bir yılan figürü ile temsil edilmesi neredeyse bir gelenek
halini almıştır diyebiliriz."""

print(metin.splitlines(True))
```

Bu programı çalıştırdığımızda şuna benzer bir sonuç elde ederiz:

```
['Python programlama dili Guido Van Rossum adlı Hollandalı bir programcı \n',
"tarafından 90'lı yılların başında geliştirilmeye başlanmıştır. Çoğu insan,
isminin \n", 'Python olmasına bakarak, bu programlama dilinin, adını piton
yılanından aldığını \n', 'düşünür. Ancak zannedildiğinin aksine bu programlama
dilinin adı piton yılanından \n', 'gelmez. Guido Van Rossum bu programlama
dilini, The Monty Python adlı bir İngiliz \n', "komedi grubunun, Monty
Python's Flying Circus adlı gösterisinden esinlenerek \n", 'adlandırmıştır.
Ancak her ne kadar gerçek böyle olsa da, Python programlama \n', 'dilinin pek
çok yerde bir yılan figürü ile temsil edilmesi neredeyse bir gelenek \n',
'halini almıştır diyebiliriz.']
```

Gördüğünüz gibi, parametresiz kullanımda, program çıktısında satır başı karakterleri (*\n*) görünmüyor. Ama eğer splitlines() metoduna parametre olarak True verirsek program çıktısında satır başı karakterleri de görünüyor. Yazdığınız programlarda ihtiyacınıza göre splitlines() metodunu parametreli olarak veya parametresiz bir şekilde kullanabilirsiniz.

lower()

Mutlaka karşılaşmışsınızdır. Bazı programlarda kullanıcıdan istenen veriler büyük-küçük harfe duyarlıdır. Yani mesela kullanıcıdan bir parola isteniyorsa, kullanıcının bu parolayı büyük-küçük harfe dikkat ederek yazması gerekir. Bu programlar açısından, örneğin 'parola' ve 'Parola' aynı kelimeler değildir. Mesela kullanıcının parolası 'parola' ise, bu kullanıcı programa 'Parola' yazarak giremez.

Bazı başka programlarda ise bu durumun tam tersi söz konusudur. Yani büyük-küçük harfe duyarlı programların aksine bazı programlar da kullanıcıdan gelen verinin büyük harfli mi yoksa küçük harfli mi olduğunu önemsemez. Kullanıcı doğru kelimeyi büyük harfle de yazsa, küçük harfle de yazsa program istenen işlemi gerçekleştirir. Mesela Google'da yapılan aramalar bu mantık üzerine çalışır. Örneğin 'kitap' kelimesini Google'da aratıyorsanız, bu kelimeyi büyük harfle de yazsanız, küçük harfle de yazsanız Google size aynı sonuçları gösterecektir. Google açısından, aradığınız kelimeyi büyük ya da küçük harfle yazmanızın bir önemi yoktur.

Şimdi şöyle bir program yazdığımızı düşünün:

```
kişi = input("Aradığınız kişinin adı ve soyadı: ")

if kişi == "Ahmet Öz":
    print("email: aoz@hmail.com")
    print("tel  : 02121231212")
    print("şehir: istanbul")

elif kişi == "Mehmet Söz":
    print("email: msoz@zmail.com")
    print("tel  : 03121231212")
    print("şehir: ankara")

elif kişi == "Mahmut Göz":
    print("email: mgoz@jmail.com")
    print("tel  : 02161231212")
    print("şehir: istanbul")

else:
    print("Aradığınız kişi veritabanında yok!")
```

Bu programın doğru çalışabilmesi için kullanıcının, örneğin, Ahmet Öz adlı kişiyi ararken büyük-küçük harfe dikkat etmesi gerekir. Eğer kullanıcı Ahmet Öz yazarsa o kişiyle ilgili bilgileri alabilir, ama eğer mesela Ahmet öz yazarsa bilgileri alamaz. Peki acaba biz bu sorunun üstesinden nasıl gelebiliriz? Yani programımızın büyük-küçük harfe duyarlı olmamasını nasıl sağlayabiliriz?

Bu işi yapmanın iki yolu var: Birincisi if bloklarını her türlü ihtimali düşünerek yazabiliriz. Mesela:

```
if kişi == "Ahmet Öz" or "Ahmet öz" or "ahmet öz":
    ...
```

Ama burada bazı problemler var. Birincisi, kullanıcının kaç türlü veri girebileceğini kestiremeyebilirsiniz. İkincisi, kestirebilseniz bile, her kişi için olasılıkları girmeye çalışmak eziyetten başka bir şey değildir...

İşte burada imdadımıza lower() metodu yetişecek. Dikkatlice inceleyin:

```
kişi = input("Aradığınız kişinin adı ve soyadı: ")
kişi = kişi.lower()

if kişi == "ahmet öz":
    print("email: aoz@hmail.com")
    print("tel  : 02121231212")
    print("şehir: istanbul")

elif kişi == "mehmet söz":
    print("email: msoz@zmail.com")
    print("tel  : 03121231212")
    print("şehir: ankara")

elif kişi == "mahmut göz":
    print("email: mgoz@jmail.com")
    print("tel  : 02161231212")
    print("şehir: istanbul")

else:
    print("Aradığınız kişi veritabanında yok!")
```

Artık kullanıcı ‘ahmet öz’ de yazsa, ‘Ahmet Öz’ de yazsa, hatta ‘AhMeT öZ’ de yazsa programımız doğru çalışacaktır. Peki bu nasıl oluyor? Elbette lower() metodu sayesinde...

Yukarıdaki örneklerin de bize gösterdiği gibi, lower() metodu, karakter dizisindeki bütün harfleri küçük harfe çeviriyor. Örneğin:

```
>>> kardiz = "ELMA"
>>> kardiz.lower()

'elma'

>>> kardiz = "arMuT"
>>> kardiz.lower()

'armut'

>>> kardiz = "PYTHON PROGRAMLAMA"
```

```
>>> kardiz.lower()

'python programlama'
```

Eğer karakter dizisi zaten tamamen küçük harflerden oluşuyorsa bu metot hiçbir işlem yapmaz:

```
>>> kardiz = "elma"
>>> kardiz.lower()

'elma'
```

İşte verdiğimiz örnek programda da lower() metodunun bu özelliğinden yararlandık. Bu metot sayesinde, kullanıcı ne tür bir kelime girerse girsin, bu kelimeler her halükarda küçük harfe çevrileceği için, if blokları kullanıcıdan gelen veriyi yakalayabilecektir.

Gördüğünüz gibi, son derece kolay ve kullanışlı bir metot bu. Ama bu metodun bir problemi var. Şu örneği dikkatlice inceleyin:

```
>>> il = "İSTANBUL"
>>> print(il.lower())

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Python33\lib\encodings\cp857.py", line 19, in encode
    return codecs.charmap_encode(input,self.errors,encoding_map)[0]
UnicodeEncodeError: 'charmap' codec can't encode character '\u0307' in position
1: character maps to <undefined>
```

Buradaki problem 'İ' harfinden kaynaklanıyor. Python programlama dili bu harfi Türkçeye uygun bir şekilde küçültemediği için yukarıdaki hatayı alıyoruz. Yukarıdaki hatanın tam olarak ne anlama geldiğini birkaç bölüm sonra anlayacaksınız. Biz şimdilik sadece Python'ın 'İ' harfini Türkçeye uygun olarak küçültemediğini bilelim yeter.

Bir de şu örneğe bakalım:

```
>>> il = "ADIYAMAN"
>>> print(il.lower())

adiyaman
```

Gördüğünüz gibi, Python programlama dili 'I' harfini de düzgün küçültemiyor. 'I' harfinin küçük biçimi 'ı' olması gerekirken, bu metot 'I' harfini 'i' diye küçültüyor. Yani:

```
>>> "I".lower()
```

```
'i'
```

Peki bu durumda ne yapacağız? Elimiz kolumuz bağlı oturacak mıyız? Elbette hayır! Biz bu tür küçük sorunları aşabilecek kadar Python bilgisine sahibiz. 'İ' ve 'I' harfleri ile ilgili problemi, yalnızca mevcut bilgilerimizi kullanarak rahatlıkla çözebiliriz:

```
iller = "ISPARTA, ADIYAMAN, DİYARBAKIR, AYDIN, BALIKESİR, AĞRI"

iller = iller.replace("I", "ı").replace("İ", "i").lower()
print(iller)
```

Bu kodlarla yaptığımız şey çok basit:

1. İlk replace() metoduyla karakter dizisi içinde geçen bütün 'I' harflerini, 'ı' ile değiştiriyoruz.
2. İkinci replace() metoduyla karakter dizisi içinde geçen bütün 'İ' harflerini 'i' ile değiştiriyoruz.
3. Bu iki işlemin ardından karakter dizisi içinde geçen 'I' ve 'İ' harflerini küçültmüş olduk. Ancak öteki harfler henüz küçülmedi. O yüzden de karakter dizimiz üzerine bir de lower() metodunu uyguluyoruz. Böylece bütün harfler düzgün bir şekilde küçülmüş oluyor.
4. Bu kodlarda farklı metotları uç uca nasıl eklediğimize dikkat edin.

Bu örnek size şunu göstermiş olmalı: Aslında programlama dediğimiz şey gerçekten de çok basit parçaların uygun bir şekilde birleştirilmesinden ibaret. Tıpkı bir yap-bozun parçalarını birleştirmek gibi...

Ayrıca bu örnek sizi bir gerçekle daha tanıştırıyor: Gördüğünüz gibi, artık Python'da o kadar ilerlediniz ki Python'ın problemlerini tespit edip bu problemlere çözüm dahi üretebiliyorsunuz!

upper()

Bu metot biraz önce öğrendiğimiz lower() metodunun yaptığı işin tam tersini yapar. Hatırlarsanız lower() metodu yardımıyla karakter dizileri içindeki harfleri küçültüyorduk. upper() metodu ise bu harfleri büyütmemizi sağlar.

Örneğin:

```
>>> kardiz = "kalem"
>>> kardiz.upper()

'KALEM'
```

lower() metodunu anlatırken, kullanıcıdan gelen verileri belli bir düzene sokmak konusunda bu metodun oldukça faydalı olduğunu söylemiştik. Kullanıcıdan gelen verilerin lower() metodu

yardımıyla standart bir hale getirilmesi sayesinde, kullanıcının girdiği kelimelerin büyük-küçük harfli olmasının önemli olmadığı programlar yazabiliyoruz. Elbette eğer isterseniz kullanıcıdan gelen bütün verileri lower() metoduyla küçük harfe çevirmek yerine, upper() metoduyla büyük harfe çevirmeyi de tercih edebilirsiniz. Python programcıları genellikle kullanıcı verilerini standart bir hale getirmek için bütün harfleri küçültmeyi tercih eder, ama tabii ki sizin bunun tersini yapmak istemenizin önünde hiçbir engel yok.

Diyelim ki, şehirlere göre hava durumu bilgisi veren bir program yazmak istiyorsunuz. Bunun için şöyle bir kod yazarak işe başlayabilirsiniz:

```
şehir = input("Hava durumunu öğrenmek için bir şehir adı girin: ")

if şehir == "ADANA":
    print("parçalı bulutlu")

elif şehir == "ERZURUM":
    print("karla karışık yağmurlu")

elif şehir == "ANTAKYA":
    print("açık ve güneşli")

else:
    print("Girdiğiniz şehir veritabanında yok!")
```

Burada programımızın doğru çalışabilmesi, kullanıcının şehir adlarını büyük harfle girmesine bağlıdır. Örneğin programımız 'ADANA' cevabını kabul edecek, ama mesela 'Adana' cevabını kabul etmeyecektir. Bunu engellemek için lower() metodunu kullanabileceğimizi biliyoruz. Bu sorunu çözmek için aynı şekilde upper() metodunu da kullanabiliriz:

```
şehir = input("Hava durumunu öğrenmek için bir şehir adı girin: ")

şehir = şehir.upper()

if şehir == "ADANA":
    print("parçalı bulutlu")

elif şehir == "ERZURUM":
    print("karla karışık yağmurlu")

elif şehir == "ANTAKYA":
    print("açık ve güneşli")

else:
    print("Girdiğiniz şehir veritabanında yok!")
```

Burada yazdığımız şehir = şehir.upper() kodu sayesinde artık kullanıcı şehir adını büyük harfle de girse, küçük harfle de girse programımız düzgün çalışacaktır.

Hatırlarsanız lower() metodunu anlatırken bu metodun bazı Türkçe karakterlerle problemi olduğunu söylemiştik. Aynı sorun, tahmin edebileceğiniz gibi, upper() metodu için de geçerlidir.

Dikkatlice inceleyin:

```
>>> kardiz = "istanbul"
>>> kardiz.upper()

'ISTANBUL'
```

lower() metodu Türkçe'deki 'I' harfini 'i' şeklinde küçültüyordu. upper() metodu ise 'i' harfini yanlış olarak 'I' şeklinde büyütüyor. Elbette bu sorun da çözülemeyecek gibi değil. Burada da lower() metodu için uyguladığımız yöntemin bir benzerini uygulayacağız:

```
iller = "istanbul, izmir, siirt, mersin"

iller = iller.replace("i", "İ").upper()
print(iller)
```

Bu kodlarla, önce karakter dizisi içinde geçen 'i' harflerini 'İ' ile değiştiriyoruz. Böylece şöyle bir şey elde etmiş oluyoruz:

```
İstanbul, İzmİr, sİİrt, mersİn
```

Gördüğünüz gibi öteki harfler eski hallerinde kaldı. Öteki harfleri de büyütebilmek için karakter dizisine upper() metodunu uygulamamız yeterli olacaktır.

Bir sorunun daha üstesinden geldiğimize göre kendimizden emin bir şekilde bir sonraki metodumuzu incelemeye geçebiliriz.

islower(), isupper()

Yukarıda öğrendiğimiz lower() ve upper() adlı metotlar karakter dizileri üzerinde bazı değişiklikler yapmamıza yardımcı oluyor. Karakter dizileri üzerinde birtakım değişiklikler yapmamızı sağlayan bu tür metotlara 'değiştirici metotlar' adı verilir. Bu tür metotların dışında bir de 'sorgulayıcı metotlar'dan söz edebiliriz. Sorgulayıcı metotlar, değiştirici metotların aksine, bir karakter dizisi üzerinde değişiklik yapmamızı sağlamaz. Bu tür metotların görevi karakter dizilerinin durumunu sorgulamaktır. Sorgulayıcı metotlara örnek olarak islower() ve isupper() metotlarını verebiliriz.

Bildiğiniz gibi, lower() metodu bir karakter dizisini tamamen küçük harflerden oluşacak şekle getiriyordu. islower() metodu ise bir karakter dizisinin tamamen küçük harflerden oluşup

oluşmadığını sorguluyor.

Hemen bir örnek verelim:

```
>>> kardiz = "istihza"
>>> kardiz.islower()

True
```

"istihza" tamamen küçük harflerden oluşan bir karakter dizisi olduğu için islower() sorgusu *True* çıktısı veriyor. Bir de şuna bakalım:

```
>>> kardiz = "Ankara"
>>> kardiz.islower()

False
```

"Ankara" ise içinde bir adet büyük harf barındırdığı için islower() sorgusuna *False* cevabı veriyor.

Yazdığınız programlarda, örneğin, kullanıcıdan gelen verinin sadece küçük harflerden oluşmasını istiyorsanız bu metottan yararlanarak kullanıcıdan gelen verinin gerçekten tamamen küçük harflerden oluşup oluşmadığını denetleyebilirsiniz:

```
veri = input("Adınız: ")

if not veri.islower():
    print("Lütfen isminizi sadece küçük harflerle yazın")
```

isupper() metodu da islower() metodunun yaptığı işin tam tersini yapar. Bildiğiniz gibi, upper() metodu bir karakter dizisini tamamen büyük harflerden oluşacak şekle getiriyordu. isupper() metodu ise bir karakter dizisinin tamamen büyük harflerden oluşup olmadığını sorguluyor:

```
>>> kardiz = "İSTİHZA"
>>> kardiz.isupper()

True

>>> kardiz = "python"
>>> kardiz.isupper()

False
```

Tıpkı islower() metodunda olduğu gibi, isupper() metodunu da kullanıcıdan gelen verinin büyük harfli mi yoksa küçük harfli mi olduğunu denetlemek için kullanabilirsiniz.

Örneğin, internet kültüründe kullanıcıların forum ve e.posta listesi gibi yerlerde tamamı büyük harflerden oluşan kelimelerle yazması kaba bir davranış olarak kabul edilir. Kullanıcıların tamamı büyük harflerden oluşan kelimeler kullanmasını engellemek için yukarıdaki metotlardan yararlanabilirsiniz:

```
veri = input("mesajınız: ")
böl = veri.split()

for i in böl:
    if i.isupper():
        print("Tamamı büyük harflerden oluşan kelimeler kullanmayın!")
```

Burada kullanıcının girdiği mesaj içindeki her kelimeyi tek tek sorgulayabilmek için öncelikle split() metodu yardımıyla karakter dizisini parçalarına ayırdığımıza dikkat edin. böl = veri.split() satırının tam olarak ne işe yaradığını anlamak için bu programı bir de o satır olmadan çalıştırmayı deneyebilirsiniz.

islower() ve isupper() metotları programlamada sıklıkla kullanılan karakter dizisi metotlarından ikisidir. Dolayısıyla bu iki metodu iyi öğrenmek programlama maceranız sırasında işlerinizi epey kolaylaştıracaktır.

endswith()

Tıpkı isupper() ve islower() metotları gibi, endswith() metodu da sorgulayıcı metotlardan biridir. endswith() metodu karakter dizileri üzerinde herhangi bir değişiklik yapmamızı sağlamaz. Bu metodun görevi karakter dizisinin durumunu sorgulamaktır.

Bu metot yardımıyla bir karakter dizisinin hangi karakter dizisi ile bittiğini sorgulayabiliyoruz. Yani örneğin:

```
>>> kardiz = "istihza"
>>> kardiz.endswith("a")

True
```

Burada, değeri *"istihza"* olan *kardiz* adlı bir karakter dizisi tanımladık. Daha sonra da kardiz.endswith("a") ifadesiyle bu karakter dizisinin *"a"* karakteri ile bitip bitmediğini sorguladık. Gerçekten de *"istihza"* karakter dizisinin sonunda *"a"* karakteri bulunduğu için Python bize *True* cevabı verdi. Bir de şuna bakalım:

```
>>> kardiz.endswith("z")

False
```

Bu defa da *False* çıktısı aldık. Çünkü karakter dizimiz 'z' harfiyle bitmiyor.

Gelin isterseniz elimizi alıştırmak için bu metotla birkaç örnek daha yapalım:

```
d1 = "python.ogg"
d2 = "tkinter.mp3"
d3 = "pygtk.ogg"
d4 = "movie.avi"
d5 = "sarki.mp3"
d6 = "filanca.ogg"
d7 = "falanca.mp3"
d8 = "dosya.avi"
d9 = "perl.ogg"
d10 = "c.avi"
d11 = "c++.mp3"

for i in d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11:
    if i.endswith(".mp3"):
        print(i)
```

Bu örnekte, elimizde farklı uzantılara sahip bazı dosyalar olduğunu varsaydık ve bu dosya adlarının herbirini ayrı birer değişken içinde depoladık. Gördüğünüz gibi, dosya uzantıları *.ogg*, *.mp3* veya *.avi*. Bizim burada amacımız elimizdeki mp3 dosyalarını listelemek. Bu işlem için endswith() metodundan yararlanabiliyoruz. Burada yaptığımız şey şu:

Öncelikle *d1*, *d2*, *d3*, *d4*, *d5*, *d6*, *d7*, *d8*, *d9*, *d10* ve *d11* adlı değişkenleri bir for döngüsü içine alıyoruz ve bu değişkenlerinin herbirinin içeriğini tek tek kontrol ediyoruz (for i in d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11:). Ardından, eğer baktığımız bu değişkenlerin değerleri ".mp3" ifadesi ile bitiyorsa (if i.endswith(".mp3"):), ölçüte uyan bütün karakter dizilerini ekrana döküyoruz (print(i)).

Yukarıdaki örneği, dilerseniz, endswith() metodunu kullanmadan şöyle de yazabilirsiniz:

```
for i in d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11:
if i[-4:len(i)] == ".mp3":
    print(i)
```

Burada karakter dizilerinin dilimlenebilme özelliğinden yararlandık. Ancak gördüğünüz gibi, dilimlenecek kısmı ayarlamaya uğraşmak yerine endswith() metodunu kullanmak çok daha mantıklı ve kolay bir yöntemdir.

Yukarıdaki örnekte de gördüğünüz gibi, endswith() metodu özellikle dosya uzantılarına göre dosya türlerini tespit etmede oldukça işe yarar bir metottur.

startswith()

Bu metot, biraz önce gördüğümüz endswith() metodunun yaptığı işin tam tersini yapar. Hatırlarsanız endswith() metodu bir karakter dizisinin hangi karakter veya karakterlerle bittiğini denetliyordu. startswith() metodu ise bir karakter dizisinin hangi karakter veya karakterlerle başladığını denetler:

```
>>> kardiz = "python"
>>> kardiz.startswith("p")

True

>>> kardiz.startswith("a")

False
```

Gördüğünüz gibi, eğer karakter dizisi gerçekten belirtilen karakterle başlıyorsa Python *True* çıktısı, yok eğer belirtilen karakterle başlamıyorsa *False* çıktısı veriyor.

Bu metodun gerçek hayatta nasıl kullanılabileceğine dair bir örnek verelim:

```
d1 = "python.ogg"
d2 = "tkinter.mp3"
d3 = "pygtk.ogg"
d4 = "movie.avi"
d5 = "sarki.mp3"
d6 = "filanca.ogg"
d7 = "falanca.mp3"
d8 = "dosya.avi"
d9 = "perl.ogg"
d10 = "c.avi"
d11 = "c++.mp3"

for i in d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11:
    if i.startswith("p"):
        print(i)
```

Burada ‘p’ harfiyle başlayan bütün dosyaları listeledik. Elbette aynı etkiyi şu şekilde de elde edebilirsiniz:

```
for i in d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11:
    if i[0] == "p":
        print(i)
```

Sadece tek bir harfi sorguluyorsanız yukarıdaki yöntem de en az startswith() metodunu kullanmak

kadar pratiktir. Ama birden fazla karakteri sorguladığınız durumlarda elbette startswith() çok daha mantıklı bir tercih olacaktır:

```
for i in d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11:
    if i.startswith("py"):
        print(i)
```

Yukarıda yazdığımız kodu dilimleme tekniğinden yararlanarak yeniden yazmak isterseniz şöyle bir şeyler yapmanız gerekiyor:

```
for i in d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11:
    if i[:2] == "py":
        print(i)
```

Dediğim gibi, birden fazla karakteri sorguladığınız durumlarda, dilimlemek istediğiniz kısmın karakter dizisi içinde hangi aralığa denk geldiğini hesaplamaya uğraşmak yerine, daha kolay bir yöntem olan startswith() metodundan yararlanmayı tercih edebilirsiniz.

Böylece karakter dizilerinin 2. bölümünü de bitirmiş olduk. Sonraki bölümde yine karakter dizilerinin metotlarından söz etmeye devam edeceğiz.

başa dön

Karakter Dizilerinin Metotları (Devamı)

Karakter dizileri konusunun en başında söylediğimiz gibi, karakter dizileri metot yönünden bir hayli zengin bir veri tipidir. Bir önceki bölümde karakter dizileri metotlarının bir kısmını incelemiştik. Bu bölümde yine metotları incelemeye devam edeceğiz.

capitalize()

Hatırlarsanız, bir önceki bölümde öğrendiğimiz startswith() ve endswith() metotları karakter dizileri üzerinde herhangi bir değişiklik yapmıyordu. Bu iki metodun görevi, karakter dizilerini sorgulamamızı sağlamaktı. Şimdi göreceğimiz capitalize() metodu ise karakter dizileri üzerinde değişiklik yapmamızı sağlayacak. Dolayısıyla bu capitalize() metodu da 'değiştirici metotlar'dan biridir diyebiliriz.

Hatırlarsanız, upper() ve lower() metotları bir karakter dizisi içindeki bütün karakterleri etkiliyordu. Yani mesela upper() metodunu bir karakter dizisine uygularsak, o karakter dizisi içindeki bütün karakterler büyük harfe dönecektir. Aynı şekilde lower() metodu da bir karakter dizisi içindeki bütün karakterleri küçük harfe çevirir.

Şimdi göreceğimiz capitalize() metodu da upper() ve lower() metotlarına benzemekle birlikte onlardan biraz daha farklı davranır: capitalize() metodunun görevi karakter dizilerinin yalnızca ilk harfini büyütmektir. Örneğin:

```
>>> a = "python"
>>> a.capitalize()

'Python'
```

Bu metodu kullanırken dikkat etmemiz gereken bir nokta var: Bu metot bir karakter dizisinin yalnızca ilk harfini büyütür. Yani birden fazla kelimeden oluşan karakter dizilerine bu metodu uyguladığımızda bütün kelimelerin ilk harfi büyümez. Yalnızca ilk kelimenin ilk harfi büyür. Yani:

```
>>> a = "python programlama dili"
>>> a.capitalize()

'Python programlama dili'
```

"python programlama dili" üç kelimeden oluşan bir karakter dizisidir. Bu karakter dizisi üzerine capitalize() metodunu uyguladığımızda bu üç kelimenin tamamının ilk harfleri büyümüyor. Yalnızca ilk 'python' kelimesinin ilk harfi bu metottan etkileniyor.

Bu arada capitalize() metodunu kullanırken bir şey dikkatinizi çekmiş olmalı. Bu metodun da, tıpkı upper() ve lower() metotlarında olduğu gibi, Türkçe karakterlerden bazıları ile ufak bir problemi var. Mesela şu örneğe bir bakın:

```
>>> kardiz = "istanbul"
>>> kardiz.capitalize()

'Istanbul'
```

'istanbul' kelimesinin ilk harfi büyütüldüğünde 'İ' olması gerekirken 'I' oldu. Bildiğiniz gibi bu problem 'ş', 'ç', 'ö', 'ğ' ve 'ü' gibi öteki Türkçe karakterlerde karşımıza çıkmaz. Sadece 'i' ve 'İ' harfleri karakter dizisi metotlarında bize problem çıkaracaktır. Ama endişe etmemize hiç gerek yok. Bu sorunu da basit bir 'if-else' yapısıyla çözebilecek kadar Python bilgisine sahibiz:

```
kardiz = "istanbul büyükşehir belediyesi"

if kardiz.startswith("i"):
    kardiz = "İ" + kardiz[1:]

kardiz = kardiz.capitalize()

print(kardiz)
```

Burada yaptığımız şey şu: Eğer değişkenin tuttuğu karakter dizisi 'i' harfi ile başlıyorsa, "İ" + kardiz[1:] kodunu kullanarak karakter dizisinin ilk harfi dışında kalan kısmıyla 'İ' harfini birleştiriyoruz. Bu yapıyı daha iyi anlayabilmek için etkileşimli kabukta şu denemeleri yapabilirsiniz:

```
>>> kardiz = "istanbul"
>>> kardiz[1:]

'stanbul'
```

Gördüğünüz gibi, kardiz[1:] kodu bize karakter dizisinin ilk harfi hariç geri kalan kısmını veriyor. Bu yapıyı dilimleme konusundan hatırlıyor olmalısınız. İşte biz dilimleme tekniğinin bu özelliğinden yararlanarak, karakter dizisinin ilk harfini kesip, baş tarafa bir adet 'İ' harfi ekliyoruz:

```
>>> "İ" + kardiz[1:]

'İstanbul'
```

Hatırlarsanız karakter dizilerinin değiştirilemeyen bir veri tipi olduğunu söylemiştik. O yüzden, karakter dizisinin *"stanbul"* kısmını 'İ' harfiyle birleştirdikten sonra, bu değişikliğin kalıcı olabilmesi için kardiz = "İ" + kardiz[1:] kodu yardımıyla, yaptığımız değişikliği tekrar *kardiz* adlı bir değişkene atıyoruz.

Böylece;

```
if kardiz.startswith("i"):
    kardiz = "İ" + kardiz[1:]
```

kodlarının ne yaptığını anlamış olduk. Kodların geri kalanında ise şöyle bir kod bloğu görüyoruz:

```
kardiz = kardiz.capitalize()
```

Buna göre, hangi harfle başlarsa başlasın Python'ın standart capitalize() metodunu bu karakter dizisi üzerine uyguluyoruz.

Son olarak da print(kardiz) kodunu kullanarak yeni karakter dizisini ekrana yazdırıyoruz ve böylece capitalize() metodundaki Türkçe karakter sorununu kıvrak bir çalımla aşmış oluyoruz.

title()

Bu metot biraz önce öğrendiğimiz capitalize() metoduna benzer. Bildiğiniz gibi capitalize() metodu bir karakter dizisinin yalnızca ilk harfini büyütüyordu. title() metodu da karakter dizilerinin ilk harfini büyütür. Ama capitalize() metodundan farklı olarak bu metot, birden fazla kelimeden oluşan karakter dizilerinin her kelimesinin ilk harflerini büyütür.

Bunu bir örnek üzerinde anlatsak sanırım daha iyi olacak:

```
>>> a = "python programlama dili"
>>> a.capitalize()

'Python programlama dili'

>>> a.title()

'Python Programlama Dili'
```

capitalize() metodu ile title() metodu arasındaki fark bariz bir biçimde görünüyor. Dediğimiz gibi, capitalize() metodu yalnızca ilk kelimenin ilk harfini büyütmekle yetinirken, title() metodu karakter dizisi içindeki bütün kelimelerin ilk harflerini büyütüyor.

Tahmin edebileceğiniz gibi, capitalize() metodundaki Türkçe karakter problemi title() metodu için de geçerlidir. Yani:

```
>>> kardiz = "istanbul"
>>> kardiz.title()

'Istanbul'
```

```
>>> kardiz = "istanbul büyükşehir belediyesi"
>>> kardiz.title()

'Istanbul Büyükşehir Belediyesi'
```

Gördüğünüz gibi, burada da Python 'i' harfini düzgün büyütemedi. Ama tabii ki bu bizi durduramaz! Çözümümüz hazır:

```
kardiz = "istanbul"

if kardiz.startswith("i"):
    kardiz = "İ" + kardiz[1:]
    kardiz = kardiz.title()
else:
    kardiz = kardiz.title()

print(kardiz)
```

Bu kodların capitalize() metodunu anlatırken verdiğimiz koda ne kadar benzediğini görüyorsunuz. Bu iki kod hemen hemen birbirinin aynısı. Tek fark, en sondaki kardiz.capitalize() kodunun burada kardiz.title() olması ve if bloğu içine ek olarak kardiz = kardiz.title() satırını yazmış olmamız. kardiz.capitalize() kodunun neden kardiz.title() koduna dönüştüğünü açıklamaya gerek yok. Ama eğer kardiz = kardiz.title() kodunun ne işe yaradığını tam olarak anlamadıysanız o satırı silin ve *kardiz* değişkeninin değerini *"istanbul büyükşehir belediyesi"* yapın. Yani:

```
kardiz = "istanbul büyükşehir belediyesi"

if kardiz.startswith("i"):
    kardiz = "İ" + kardiz[1:]
else:
    kardiz = kardiz.title()

print(kardiz)
```

Bu kodları bu şekilde çalıştırırsanız şu çıktıyı alırsınız:

```
İstanbul büyükşehir belediyesi
```

Burada yalnızca ilk kelimenin ilk harfi büyüdü. Halbuki title() metodunun işleyişi gereğince karakter dizisi içindeki bütün kelimelerin ilk harflerinin büyümesi gerekiyordu. İşte o satır bütün kelimelerin ilk harflerinin büyümesini sağlıyor. Eğer bir kelimenin ilk harfi zaten büyükse title() metodu bu harfe dokunmaz, ama karakter dizisi içindeki öbür kelimelerin ilk harflerini yine de büyütür.

İşte yukarıda title() metodunun bu özelliğinden faydalanıyoruz. kardiz = "İ" + kardiz[1:] komutu karakter dizisinin ilk kelimesinin ilk harfini düzgün bir şekilde büyütüyor, ama geri kalan kelimelere hiçbir şey yapmıyor. kardiz = kardiz.title() komutu ise karakter dizisi içindeki geri kalan kelimelerin ilk harflerini büyütüyor. Böylece istediğimiz çıktıyı elde edebilmiş oluyoruz. Yalnız bu kodlarda bir şey dikkatinizi çekmiş olmalı. kardiz = kardiz.title() komutunu program içinde iki yerde kullandık. Programcılıktaki en önemli ilkelerden biri de mümkün olduğunca tekrardan kaçınmaktır. Eğer yazdığınız bir programda aynı kodları program boyunca tekrar tekrar yazıyorsanız muhtemelen bir yerde hata yapıyorsunuzdur. Öyle bir durumda yapmanız gereken şey kodlarınızı tekrar gözden geçirip, tekrar eden kodları nasıl azaltabileceğinizi düşünmektir. İşte burada da böyle bir tekrar söz konusu. Biz tekrara düşmekten kurtulmak için yukarıdaki kodları şöyle de yazabiliriz:

```
kardiz = "istanbul büyükşehir belediyesi"

if kardiz.startswith("i"):
    kardiz = "İ" + kardiz[1:]

kardiz = kardiz.title()

print(kardiz)
```

kardiz = kardiz.title() komutunu hem if bloğunda, hem de else bloğunda kullandığımız için, programımız her koşulda bu kodu zaten çalıştıracak. O yüzden bu satırı if bloğuna yazdıktan sonra bir de aynı şeyi else bloğu içine yazmak gereksiz. Onun yerine else bloğunu tamamen kaldırıp, o satırı if bloğunun çıkışına yerleştirebiliriz.

Eski kodlardaki mantık işleyişi şöyle idi:

1. *kardiz* adlı bir değişken tanımla
2. Eğer *kardiz* 'i' harfi ile başlıyorsa (if), *kardiz*'in ilk harfi hariç geri kalan kısmı ile 'İ' harfini birleştir.
3. Daha sonra *kardiz* değişkenine title() metodunu uygula.
4. Eğer *kardiz* 'i' harfi ile değil de başka bir harfle başlıyorsa (else), *kardiz* değişkenine title() metodunu uygula.
5. Son olarak *kardiz* değişkenini yazdır.

Tekrar eden kodları çıkardıktan sonra ise kodlarımızın mantık işleyişi şöyle oldu:

1. *kardiz* adlı bir değişken tanımla
2. Eğer *kardiz* 'i' harfi ile başlıyorsa (if), *kardiz*'in ilk harfi hariç geri kalan kısmı ile 'İ' harfini birleştir.
3. Daha sonra *kardiz* değişkenine title() metodunu uygula.
4. Son olarak *kardiz* değişkenini yazdır.

Gördüğünüz gibi, aynı sonuca daha kısa bir yoldan ulaşabiliyoruz.

Ama bir dakika! Burada bir sorun var!

Bu kodlar ‘i’ harfinin karakter dizisinin yalnızca en başında yer aldığı durumlarda düzgün çalışacaktır. Bu kodlar mesela şu karakter dizisini düzgün büyütemez:

```
on iki ada
```

Aynı şekilde bu kodlar şu karakter dizisini de büyütemez:

```
hükümet istifa!
```

Çünkü bu karakter dizilerinde ‘i’ harfi karakter dizisini oluşturan kelimelerin ilkinde yer almıyor. Bizim yazdığımız kod ise yalnızca ilk kelime düşünülerek yazılmış. Peki bu sorunun üstesinden nasıl geleceğiz?

Evet, doğru tahmin ettiniz. Bizi kurtaracak şey split() metodu ve basit bir for döngüsü. Dikkatlice bakın:

```
kardiz = "on iki ada"

for kelime in kardiz.split():
    if kelime.startswith("i"):
        kelime = "İ" + kelime[1:]

    kelime = kelime.title()

    print(kelime, end=" ")
```

Bu defa istediğimizi gerçekleştiren bir kod yazabildik. Bu kodlar, ‘i’ harfi karakter dizisini oluşturan kelimelerin hangisinde bulunursa bulunsun, karakter dizisini Türkçeye uygun bir şekilde büyütebilecektir.

Bir önceki kodlara göre, bu son kodlardaki tek farkın split() metodu ve for döngüsü olduğuna dikkat edin.

Bu kodları daha iyi anlayabilmek için etkileşimli kabukta kendi kendinize bazı deneme çalışmaları yapabilirsiniz:

```
>>> kardiz = "on iki ada"
>>> kardiz.split()

['on', 'iki', 'ada']

>>> for kelime in kardiz.split():
...     print(kelime[0])
```

```
...
o
i
a
```

Gördüğünüz gibi, split() metodu "on iki ada" adlı karakter dizisini kelimelerine ayırıyor. İşte biz de kelimelerine ayrılmış bu yapı üzerinde bir for döngüsü kurarak herbir öğenin ilk harfinin 'i' olup olmadığını kontrol edebiliyoruz.

swapcase()

swapcase() metodu da büyük-küçük harfle ilgili bir metottur. Bu metot bir karakter dizisi içindeki büyük harfleri küçük harfe; küçük harfleri de büyük harfe dönüştürür. Örneğin:

```
>>> kardiz = "python"
>>> kardiz.swapcase()

'PYTHON'

>>> kardiz = "PYTHON"
>>> kardiz.swapcase()

'python'

>>> kardiz = "Python"
>>> kardiz.swapcase()

'pYTHON'
```

Gördüğünüz gibi, bu metot aynen dediğimiz gibi işliyor. Yani küçük harfleri büyük harfe; büyük harfleri de küçük harfe dönüştürüyor.

Yine tahmin edebileceğiniz gibi, bu metodun da bazı Türkçe karakterlerle problemi var:

```
>>> kardiz = "istihza"
>>> kardiz.swapcase()

'ISTIHZA'
```

Bu sorunu da aşmak tabii ki bizim elimizde:

```
kardiz = "istanbul"

for i in kardiz:
```

```
    if i == 'İ':
        kardiz = kardiz.replace('İ', 'i')
    elif i == 'i':
        kardiz = kardiz.replace('i', 'İ')
    else:
        kardiz = kardiz.replace(i, i.swapcase())

print(kardiz)
```

Daha önceki örneklerde de olduğu gibi, bu kodlarda da ‘i’ ve ‘I’ harflerini tek tek kontrolden geçiriyoruz. Eğer bir karakter dizisi içinde bu iki harften biri varsa, bunların büyük harf veya küçük harf karşılıklarını elle yerine koyuyoruz. Bu karakterler dışında kalan karakterlere ise doğrudan swapcase() metodunu uygulayarak istediğimiz sonucu elde ediyoruz. Bu kodlarda kafanıza yatmayan yerler varsa, kodlar içinde kendinize göre bazı eklemeler çıkarmalar yaparak neyin ne işe yaradığını daha kolay anlayabilirsiniz.

casefold()

Bu metot işlev olarak lower() metoduna çok benzer. Hatta Türkçe açısından, bu metodun lower() metodundan hiçbir farkı yoktur. Ancak bazı başka dillerde, bu metot bazı harfler için lower() metodunun verdiğinden farklı bir çıktı verir. Örneğin Almancadaki ‘ß’ harfi bu duruma bir örnek olabilir:

```
>>> "ß".lower()
'ß'

>>> "ß".casefold()
'ss'
```

Gördüğünüz gibi, lower() ve casefold() metotları bu harfe farklı davranıyor.

Türkçedeki İ-i sorunu bu metot için de aynen geçerlidir.

strip(), lstrip(), rstrip()

Bu başlıkta birbiriyle bağlantılı üç adet karakter dizisi metodunu inceleyeceğiz. Bu metotlar strip(), lstrip() ve rstrip(). İlk olarak strip() metoduyla başlayalım.

Zaman zaman, içinde anlamsız ya da gereksiz karakterler barındıran metinleri bu anlamsız ve gereksiz karakterlerden temizlemeniz gereken durumlarla karşılaşabilirsiniz. Örneğin arkadaşınızdan gelen bir e.postada her satırın başında ve/veya sonunda > gibi bir karakter olabilir. Arkadaşınızdan gelen bu e.postayı kullanabilmek için öncelikle metin içindeki o > karakterlerini silmeniz gerekebilir. Hepimizin bildiği gibi, bu tür karakterleri elle temizlemeye kalkışmak son derece sıkıcı ve zaman

alıcı bir yöntemdir. Ama artık siz bir Python programcısı olduğunuza göre bu tür angaryaları Python'a devredebilirsiniz.

Yukarıda bahsettiğimiz duruma yönelik bir örnek vermeden önce dilerseniz strip() metoduyla ilgili çok basit örnekler vererek başlayalım işe:

```
>>> kardiz = " istihza "
```

Burada değeri *" istihza "* olan *kardiz* adlı bir karakter dizisi tanımladık. Dikkat ederseniz bu karakter dizisinin sağında ve solunda birer boşluk karakteri var. Bazı durumlarda kullanıcıdan ya da başka kaynaktan gelen karakter dizilerinde bu tür istenmeyen boşluklar olabilir. Ama sizin kullanıcıdan veya başka bir kaynaktan gelen o karakter dizisini düzgün kullanabilmeniz için öncelikle o karakter dizisinin sağında ve solunda bulunan boşluk karakterlerinden kurtulmanız gerekebilir. İşte böyle anlarda strip() metodu yardımınıza yetişecektir. Dikkatlice inceleyin:

```
>>> kardiz = " istihza "
>>> print(kardiz)

' istihza '

>>> kardiz.strip()

'istihza'
```

Gördüğünüz gibi, strip() metodunu kullanarak, karakter dizisinin orijinalinde bulunan sağlı sollu boşluk karakterlerini bir çırpıda ortadan kaldırdık.

strip() metodu yukarıdaki örnekte olduğu gibi parametresiz olarak kullanıldığında, bir karakter dizisinin sağında veya solunda bulunan belli başlı karakterleri kırpar. strip() metodunun öntanımlı olarak kırptığı karakterler şunlardır:

' '	boşluk karakteri
\t	sekme (TAB) oluşturan kaçış dizisi
\n	satır başına geçiren kaçış dizisi
\r	imleci aynı satırın başına döndüren kaçış dizisi
\v	düşey sekme oluşturan kaçış dizisi
\f	yeni bir sayfaya geçiren kaçış dizisi

Yani eğer strip() metoduna herhangi bir parametre vermezsek bu metot otomatik olarak karakter dizilerinin sağında ve solunda bulunan yukarıdaki karakterleri kırpacaktır. Ancak eğer biz istersek split() metoduna bir parametre vererek bu metodun istediğimiz herhangi başka bir karakteri kırpmasını da sağlayabiliriz. Örneğin:

```
>>> kardiz = "python"
>>> kardiz.strip("p")

'ython'
```

Burada strip() metoduna parametre olarak *"p"* karakter dizisini vererek, strip() metodunun, karakter dizisinin başında bulunan *"p"* karakterini ortadan kaldırmasını sağladık. Yalnız strip() metodunu kullanırken bir noktaya dikkat etmelisiniz. Bu metot bir karakter dizisinin hem başında, hem de sonunda bulunan karakterlerle ilgilenir. Mesela şu örneğe bakalım:

```
>>> kardiz = "kazak"
>>> kardiz.strip("k")

'aza'
```

Gördüğünüz gibi, strip() metoduna *"k"* parametresini vererek, *"kazak"* adlı karakter dizisinin hem başındaki hem de sonundaki *"k"* harflerini kırpmayı başardık. Eğer bu metoda verdiğiniz parametre karakter dizisinde geçmiyorsa, bu durumda strip() metodu herhangi bir işlem yapmaz. Ya da aradığınız karakter, karakter dizisinin yalnızca tek bir tarafında (mesela sadece başında veya sadece sonunda) geçiyorsa, strip() metodu, ilgili karakter hangi taraftaysa onu siler. Aranan karakterin bulunmadığı tarafla ilgilenmez.

strip() metodunu anlatmaya başlarken, içinde gereksiz yere < işaretlerinin geçtiği e.postalardan söz etmiş ve bu e.postalardaki o gereksiz karakterleri elle silmenin ne kadar da sıkıcı bir iş olduğunu söylemiştik. Eğer e.postalarınızda bu tip durumlarla sık sık karşılaşıyorsanız, gereksiz karakterleri silme görevini sizin yerinize Python yerine getirebilir. Şimdi şu kodları dikkatlice inceleyin:

```
metin = """
> Python programlama dili Guido Van Rossum adlı Hollandalı bir programcı tarafından
> 90'lı yılların başında geliştirilmeye başlanmıştır. Çoğu insan, isminin Python
> olmasına bakarak, bu programlama dilinin, adını piton yılanından aldığını düşünür.
> Ancak zannedildiğinin aksine bu programlama dilinin adı piton yılanından gelmez.
> Guido Van Rossum bu programlama dilini, The Monty Python adlı bir İngiliz komedi
> grubunun, Monty Python's Flying Circus adlı gösterisinden esinlenerek adlandırmıştır.
> Ancak her ne kadar gerçek böyle olsa da, Python programlama dilinin pek çok yerde
> bir yılan figürü ile temsil edilmesi neredeyse bir gelenek halini almıştır diyebiliriz.
"""

for i in metin.split():
    print(i.strip("> "), end=" ")
```

Bu programı çalıştırdığınızda şöyle bir çıktı elde edeceksiniz:

```
Python programlama dili Guido Van Rossum adlı Hollandalı bir programcı tarafından
```

```
90'lı yılların başında geliştirilmeye başlanmıştır. Çoğu insan, isminin Python
olmasına bakarak, bu programlama dilinin, adını piton yılanından aldığını düşünür.
Ancak zannedildiğinin aksine bu programlama dilinin adı piton yılanından gelmez.
Guido Van Rossum bu programlama dilini, The Monty Python adlı bir İngiliz komedi
grubunun, Monty Python's Flying Circus adlı gösterisinden esinlenerek adlandırmıştır.
Ancak her ne kadar gerçek böyle olsa da, Python programlama dilinin pek çok yerde
bir yılan figürü ile temsil edilmesi neredeyse bir gelenek halini almıştır diyebiliriz.
```

Gördüğünüz gibi, her satırın başında bulunan '> ' karakterlerini ufacık birkaç kod yardımıyla rahatlıkla temizledik. Burada strip() metoduyla birlikte split() metodunu da kullandığımızı görüyorsunuz. split() metodu ile önce *metin* adlı karakter dizisini parçaladık. Daha sonra da strip() metodu yardımıyla baş taraftaki istenmeyen karakterleri temizledik.

Yukarıdaki örnekte verdiğimiz metin, istenmeyen karakterleri yalnızca tek bir tarafta içeriyor. Ama elbette istenmeyen karakterler, karakter dizisinin ne tarafında olursa olsun strip() metodu bu karakterleri başarıyla kırpacaktır.

Bu bölümün başlığında strip() metodu ile birlikte lstrip() ve rstrip() adlı iki metodun daha adı geçiyordu. strip() metodunun ne işe yaradığını öğrendik. Peki bu lstrip() ve rstrip() metotları ne işe yarıyor?

lstrip() metodundan başlayalım anlatmaya...

strip() metodunu anlatırken, bu metodun bir karakter dizisinin sağında ve solunda bulunan istenmeyen karakterleri kırptığını söylemiştik. Ancak bazen, istediğimiz şey bu olmayabilir. Yani biz bir karakter dizisinin hem sağında, hem de solunda bulunan gereksiz karakterleri değil, yalnızca sağında veya yalnızca solunda bulunan gereksiz karakterleri kırpmak isteyebiliriz. Örneğin strip() metodunu anlatırken verdiğimiz *"kazak"* örneğini ele alalım. Şöyle bir komutun ne yapacağını biliyorsunuz:

```
>>> "kazak".strip("k")
```

Bu komut hem sol, hem de sağ taraftaki *"k"* karakterlerini kırpacaktır. Ama peki ya biz sadece sol taraftaki *"k"* karakterini atmak istersek ne olacak? İşte böyle bir durumda strip() metodundan değil, lstrip() metodundan faydalanacağız.

lstrip() metodu bir karakter dizisinin sol tarafındaki gereksiz karakterlerden kurtulmamızı sağlar. Mesela bu bilgiyi yukarıdaki örneğe uygulayalım:

```
>>> "kazak".lstrip("k")

'azak'
```

Gördüğünüz gibi, lstrip() metodu yalnızca sol baştaki *"k"* harfiyle ilgilendi. Sağ taraftaki *"k"* harfine ise dokunmadı. Eğer sol taraftaki karakteri değil de yalnızca sağ taraftaki karakteri uçurmak

istemeniz halinde ise rstrip() metodundan yararlanacaksınız:

```
>>> "kazak".rstrip("k")

'kaza'
```

Bu arada, yukarıdaki metotları doğrudan karakter dizileri üzerine uygulayabildiğimize de dikkat edin. Yani şu iki yöntem de uygun ve doğrudur:

```
>>> kardiz = "karakter dizisi"
>>> kardiz.metot_adı()
```

veya:

```
>>> "karakter dizisi".metot_adı()
```

join()

Hatırlarsanız şimdiye kadar öğrendiğimiz metotlar arasında split() adlı bir metot vardı. Bu metodun ne işe yaradığını ve nasıl kullanıldığını biliyorsunuz:

```
>>> kardiz = "Beşiktaş Jimnastik Kulübü"
>>> bölünmüş = kardiz.split()
>>> print(bölünmüş)

['Beşiktaş', 'Jimnastik', 'Kulübü']
```

Gördüğünüz gibi split() metodu bir karakter dizisini belli yerlerden bölerek parçalara ayırıyor. Bu noktada insanın aklına şöyle bir soru geliyor: Diyelim ki elimizde böyle bölünmüş bir karakter dizisi grubu var. Biz bu grup içindeki karakter dizilerini tekrar birleştirmek istersek ne yapacağız?

Şimdi şu kodlara çok dikkatlice bakın:

```
>>> " ".join(bölünmüş)

'Beşiktaş Jimnastik Kulübü'
```

Gördüğünüz gibi, *"Beşiktaş Jimnastik Kulübü"* adlı karakter dizisinin ilk halini tekrar elde ettik. Yani bu karakter dizisine ait, bölünmüş parçaları tekrar bir araya getirdik. Ancak bu işi yapan kod gözünüzüne biraz tuhaf ve anlaşılmaz görünmüş olabilir.

İlk başta dikkatimizi çeken şey, bu metodun öbür metotlara göre biraz daha farklı bir yapıya sahipmiş gibi görünmesi. Ama belki yukarıdaki örneği şöyle yazarsak bu örnek biraz daha anlaşılır gelebilir gözünüze:

```
>>> birleştirme_karakteri = " "
>>> birleştirme_karakteri.join(bölünmüş)
```

Burada da tıpkı öteki metotlarda olduğu gibi, join() metodunu bir karakter dizisi üzerine uyguladık. Bu karakter dizisi bir adet boşluk karakteri. Ayrıca gördüğünüz gibi join() metodu bir adet de parametre alıyor. Bu örnekte join() metoduna verdiğimiz parametre *bölünmüş* adlı değişken. Aslında şöyle bir düşününce yukarıdaki kodların sanki şöyle yazılması gerekiyormuş gibi gelebilir size:

```
>>> bölünmüş.join(birleştirme_karakteri)
```

Ama bu kullanım yanlıştır. Üstelik kodunuzu böyle yazarsanız Python size bir hata mesajı gösterecektir:

```
>>> bölünmüş.join(birleştirme_karakteri)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'join'
```

Buradaki hata mesajı bize şöyle diyor: 'liste nesnesinin *join* adlı bir niteliği yoktur!'. Bu cümledeki 'liste nesnesi' ifadesine özellikle dikkatinizi çekmek istiyorum. Biz şimdiye kadar iki tür nesne (ya da başka bir ifadeyle veri tipi) görmüştük. Bunlar karakter dizileri ve sayılardı. Burada karşımıza üçüncü bir nesne çıkıyor. Gördüğümüz kadarıyla bu yeni nesnenin adı 'liste'. (Liste adlı veri tipini birkaç bölüm sonra en ince ayrıntısına kadar inceleyeceğiz. Python'da böyle bir veri tipi olduğunu bilmemiz bizim için şimdilik yeterli.)

İşte yukarıdaki hatayı almamızın nedeni, aslında karakter dizilerine ait bir metot olan join() metodunu bir liste üzerinde uygulamaya çalışmamız. Böyle bir durumda da Python doğal olarak bizi 'liste nesnelerinin *join* adlı bir niteliği olmadığı' konusunda uyarıyor. Bütün bu anlattıklarımız bizi şu sonuca ulaştırıyor: Bir veri tipine ait metotlar doğal olarak yalnızca o veri tipi üzerinde kullanılabilir. Mesela yukarıdaki örnekte gördüğümüz gibi, bir karakter dizisi metodu olan join()'i başka bir veri tipine uygulamaya çalışırsak hata alırız.

Sonuç olarak, join() adlı metodu *bölünmüş* adlı değişkene uygulayamayacağımızı anlamış bulunuyoruz. O halde bu metotla birlikte kullanılmak üzere bir karakter dizisi bulmamız gerekiyor.

En başta da söylediğimiz gibi, join() metodunun görevi bölünmüş karakter dizisi gruplarını birleştirmektir. Bu metot görevini yerine getirirken, yani karakter dizisi gruplarını birleştirirken bir birleştirme karakterine ihtiyaç duyar. Bizim örneğimizde bu birleştirme karakteri bir adet boşluktur. Durumu daha iyi anlayabilmek için örneğimizi tekrar gözümünün önüne getirelim:

```
>>> kardiz = "Beşiktaş Jimnastik Kulübü"
>>> bölünmüş = kardiz.split()
>>> print(bölünmüş)
```

```
['Beşiktaş', 'Jimnastik', 'Kulübü']

>>> kardiz = " ".join(bölünmüş)
>>> print(kardiz)

Beşiktaş Jimnastik Kulübü
```

Gördüğünüz gibi, orijinal karakter dizisinin bölünmüş parçalarını, her bir parçanın arasında bir adet boşluk olacak şekilde yeniden birleştirdik. Elbette sadece boşluk karakteri kullanabileceğiz diye bir kaide yok. Mesela şu örneklere bakın:

```
>>> kardiz = "-".join(bölünmüş)

Beşiktaş-Jimnastik-Kulübü

>>> kardiz = "".join(bölünmüş)

BeşiktaşJimnastikKulübü
```

İlk örnekte, bölünmüş karakter dizilerini - işareti ile birleştirdik. İkinci örnekte ise bu karakter dizilerini birleştirmek için boş bir karakter dizisi kullandık. Yani parçaları birleştirirken arada boşluk olmamasını sağladık.

join() metodu ile bol bol pratik yaparak bu metodu hakkıyla öğrenmenizi tavsiye ederim. Zira programcılık maceranız boyunca en sık kullanacağınız karakter dizisi metotları listesinin en başlarında bu metot yer alır.

count()

Tıpkı daha önce öğrendiğimiz sorgulayıcı metotlar gibi, count() metodu da bir karakter dizisi üzerinde herhangi bir değişiklik yapmamızı sağlamaz. Bu metodun görevi bir karakter dizisi içinde belli bir karakterin kaç kez geçtiğini sorgulamaktır. Bununla ilgili hemen bir örnek verelim:

```
>>> şehir = "Kahramanmaraş"
>>> şehir.count("a")

5
```

Buradan anlıyoruz ki, *"Kahramanmaraş"* adlı karakter dizisi içinde toplam *5* adet *"a"* karakteri geçiyor.

count() metodu yaygın olarak yukarıdaki örnekte görüldüğü şekilde sadece tek bir parametre ile kullanılır. Ama aslında bu metot toplam 3 parametre alır. Şimdi şu örnekleri dikkatlice inceleyin:

```
>>> şehir = "adana"
>>> şehir.count("a")

3

>>> şehir.count("a", 1)

2

>>> şehir.count("a", 2)

2

>>> şehir.count("a", 3)

1

>>> şehir.count("a", 4)

1
```

İlk örnekte count() metodunu tek bir parametre ile birlikte kullandığımız için *"adana"* adlı karakter dizisi içindeki bütün *"a"* harflerinin toplam sayısı çıktı olarak verildi.

İkinci örnekte, ise count() metodununa ikinci bir parametre daha verdik. Bu ikinci parametre, count() metodunun bir karakteri saymaya başlarken karakter dizisinin kaçıncı sırasından başlayacağını gösteriyor. Bu örnekte ikinci parametre olarak *1* sayısını verdiğimiz için, Python saymaya *"adana"* karakter dizisinin *1*. sırasından başlayacak. Dolayısıyla *0*. sıradaki *"a"* harfi sayım işleminin dışında kalacağı için toplam *"a"* sayısı *4* değil *3* olarak görünecek. Gördüğünüz gibi, sonraki örneklerde de aynı mantığı takip ettiğimiz için aradığımız karakterin toplam sayısı örnekten örneğe farklılık gösteriyor.

Peki bu metodu gerçek programlarda ne amaçla kullanabilirsiniz? Bu metodu kullanarak, örneğin, kullanıcıyı aynı karakterden yalnızca bir adet girmeye zorlayabilirsiniz. Bunun için mesela şöyle bir yapı kullanabilirsiniz:

```
parola = input("parolanız: ")

kontrol = True

for i in parola:
    if parola.count(i) > 1:
        kontrol = False
```

```
if kontrol:
    print('Parolanız onaylandı!')
else:
    print('Parolanızda aynı harfi bir kez kullanabilirsiniz!')
```

Burada *kontrol* değişkeninin değerini *True* olarak belirledik. Eğer *parola* içindeki harflerden herhangi biri 1'den fazla geçiyorsa bu durumda *kontrol* değişkeninin değerini *False* yapıyoruz:

```
for i in parola:
    if parola.count(i) > 1:
        kontrol = False
```

Daha sonra da *kontrol* değişkeninin durumuna göre kullanıcıya parolanın onaylandığı veya onaylanmadığı bilgisini veriyoruz. Buna göre eğer *kontrol* değişkeninin değeri *True* ise şu çıktıyı veriyoruz:

```
Parolanız onaylandı!
```

Aksi halde şu çıktıyı veriyoruz:

```
Parolanızda aynı harfi bir kez kullanabilirsiniz!
```

Yukarıdakine benzer durumların dışında count() metodunu şöyle durumlarda da kullanabilirsiniz:

```
kelime = input("Herhangi bir kelime: ")

for harf in kelime:
    print("{} harfi {} kelimesinde {} kez geçiyor!".format(harf,
                                                          kelime,
                                                          kelime.count(harf)))
```

Burada amacımız kullanıcının girdiği bir kelime içindeki bütün harflerin o kelime içinde kaç kez geçtiğini bulmak. count() metodunu kullanarak bu işi çok kolay bir şekilde halledebiliyoruz. Kullanıcının mesela 'adana' kelimesini girdiğini varsayarsak yukarıdaki program şöyle bir çıktı verecektir:

```
a harfi adana kelimesinde 3 kez geçiyor!
d harfi adana kelimesinde 1 kez geçiyor!
a harfi adana kelimesinde 3 kez geçiyor!
n harfi adana kelimesinde 1 kez geçiyor!
a harfi adana kelimesinde 3 kez geçiyor!
```

Ancak burada şöyle bir problem var: 'adana' kelimesi içinde birden fazla geçen harfler (mesela 'a' harfi) çıktıda birkaç kez tekrarlanıyor. Yani mesela 'a' harfinin geçtiği her yerde programımız 'a' harfinin kelime içinde kaç kez geçtiğini rapor ediyor. İstediğiniz davranış bu olabilir. Ama bazı

durumlarda her harfin kelime içinde kaç kez geçtiği bilgisinin yalnızca bir kez raporlanmasını isteyebilirsiniz. Yani siz yukarıdaki gibi bir çıktı yerine şöyle bir çıktı elde etmek istiyor olabilirsiniz:

```
a harfi adana kelimesinde 3 kez geçiyor!
d harfi adana kelimesinde 1 kez geçiyor!
n harfi adana kelimesinde 1 kez geçiyor!
```

Böyle bir çıktı elde edebilmek için şöyle bir program yazabilirsiniz:

```
kelime = input("Herhangi bir kelime: ")
sayaç = ""

for harf in kelime:
    if harf not in sayaç:
        sayaç += harf

for harf in sayaç:
    print("{} harfi {} kelimesinde {} kez geçiyor!".format(harf,
                                                          kelime,
                                                          kelime.count(harf)))
```

Gelin isterseniz bu kodları şöyle bir inceleyelim.

Bu kodlarda öncelikle kullanıcıdan herhangi bir kelime girmesini istiyoruz.

Daha sonra *sayaç* adlı bir değişken tanımlıyoruz. Bu değişken, kullanıcının girdiği kelime içindeki harfleri tutacak. Bu değişken, *kelime* değişkeninden farklı olarak, kullanıcının girdiği sözcük içinde birden fazla geçen harflerden yalnızca tek bir örnek içerecek.

Değişkenimizi tanımladıktan sonra bir for döngüsü kuruyoruz. Bu döngüye dikkatlice bakın. Kullanıcının girdiği kelime içinde geçen harflerden her birini yalnızca bir kez alıp *sayaç* değişkenine gönderiyoruz. Böylece elimizde her harften sadece bir adet olmuş oluyor. Burada Python'ın arka planda neler çevirdiğini daha iyi anlayabilmek için isterseniz döngüden sonra şöyle bir satır ekleyerek *sayaç* değişkeninin içeriğini inceleyebilir, böylece burada kullandığımız for döngüsünün nasıl çalıştığını daha iyi görebilirsiniz:

```
print("sayaç içeriği: ", sayaç)
```

İlk döngümüz sayesinde, kullanıcının girdiği kelime içindeki her harfi teke indirerek, bu harfleri *sayaç* değişkeni içinde topladık. Şimdi yapmamız gereken şey, *sayaç* değişkenine gönderilen her bir harfin, *kelime* adlı değişken içinde kaç kez geçtiğini hesaplamak olmalı. Bunu da yine bir *for* döngüsü ile yapabiliriz:

```
for harf in sayaç:
```

```
    print("{} harfi {} kelimesinde {} kez geçiyor!".format(harf,
                                                          kelime,
                                                          kelime.count(harf)))
```

Burada yaptığımız şey şu: count() metodunu kullanarak, *sayaç* değişkeninin içindeki her bir harfin, *kelime* değişkeninin içinde kaç kez geçtiğini buluyoruz. Bu döngünün nasıl çalıştığını daha iyi anlayabilmek için, isterseniz bu döngüyü şu şekilde sadeleştirebilirsiniz:

```
for harf in sayaç:
    print(harf, kelime, kelime.count(harf))
```

Gördüğünüz gibi, *sayaç* değişkeni içindeki herbir harfin *kelime* adlı karakter dizisi içinde kaç kez geçtiğini tek tek sorguladık.

Yukarıdaki örneklerde count() metodunun iki farklı parametre aldığını gördük. Bu metot bunların dışında üçüncü bir parametre daha alır. Bu üçüncü parametre ikinci parametreyle ilişkilidir. Dilerseniz bu ilişkiyi bir örnek üzerinde görelim:

```
>>> kardiz = "python programlama dili"
>>> kardiz.count("a")

3

>>> kardiz.count("a", 15)

2
```

Bu örneklerden anladığımıza göre, *"python programlama dili"* adlı karakter dizisi içinde toplam *3* adet 'a' harfi var. Eğer bu karakter dizisi içindeki 'a' harflerini karakter dizisinin en başından itibaren değil de, *15.* karakterden itibaren saymaya başlarsak bu durumda *2* adet 'a' harfi buluyoruz. Şimdi de şu örneğe bakalım:

```
>>> kardiz.count("a", 15, 17)

1
```

Burada, *15.* karakter ile *17.* karakter arasında kalan 'a' harflerini saymış olduk. *15.* karakter ile *17.* karakter arasında toplam *1* adet 'a' harfi olduğu için de Python bize *1* sonucunu verdi. Bütün bu örneklerden sonra count() metoduna ilişkin olarak şöyle bir tespitte bulunabiliriz:

> count() metodu bir karakter dizisi içinde belli bir karakterin kaç kez geçtiğini sorgulamamızı sağlar. Örneğin bu metodu count("a") şeklinde kullanırsak Python bize karakter dizisi içindeki bütün "a" harflerinin sayısını verecektir. Eğer bu metoda 2. ve 3. parametreleri de verirsek, sorgulama işlemi karakter dizisinin belli bir kısmında gerçekleştirilecektir. Örneğin count("a", 4, 7) gibi bir kullanım, bize karakter dizisinin 4. ve 7. karakterleri arasında kalan "a"

harflerinin sayısını verecektir.

Böylece bir metodu daha ayrıntılı bir şekilde incelemiş olduk. Artık başka bir metot incelemeye geçebiliriz.

index(), rindex()

Bu bölümün başında karakter dizilerinin dilimlenme özelliğinden söz ederken, karakter dizisi içindeki her harfin bir sırası olduğunu söylemiştik. Örneğin *"python"* adlı karakter dizisinde 'p' harfinin sırası *0*'dır. Aynı şekilde 'n' harfinin sırası ise *5*'tir. Karakterlerin, bir karakter dizisi içinde hangi sırada bulunduğunu öğrenmek için index() adlı bir metottan yararlanabiliriz. Örneğin:

```
>>> kardiz = "python"
>>> kardiz.index("p")

0

>>> kardiz.index("n")

5
```

Eğer sırasını sorguladığımız karakter, o karakter dizisi içinde bulunmuyorsa, bu durumda Python bize bir hata mesajı gösterir:

```
>>> kardiz.index("z")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

Bu metodun özelliği, sorguladığımız karakterin, karakter dizisi içinde geçtiği ilk konumu vermesidir. Yani örneğin:

```
>>> kardiz = "adana"
>>> kardiz.index("a")

0
```

"adana" adlı karakter dizisi içinde *3* adet 'a' harfi var. Ancak biz index() metodu yardımıyla *"adana"* karakter dizisi içindeki 'a' harfinin konumunu sorgularsak, Python bize 'a' harfinin geçtiği ilk konumu, yani *0.* konumu, bildirecektir. Halbuki *"adana"* karakter dizisi içinde *2.* ve *4.* sıralarda da birer 'a' harfi var. Ancak index() metodu *0.* konumdaki 'a' harfini gördükten sonra karakter dizisinin geri kalanına bakmaz.

index() metodunu biz yukarıda tek bir parametre ile birlikte kullandık. Bu parametre, karakter dizisi içinde konumunu öğrenmek istediğimiz karakteri gösteriyor. Ama bu metot aslında toplam *3* parametre alır. Şu örnekleri dikkatlice inceleyelim:

```
>>> kardiz = "adana"
>>> kardiz.index("a")

0
```

Burada normal bir şekilde index() metodunu tek bir parametre ile birlikte kullandık. Böylece Python bize 'a' harfinin karakter dizisi içinde ilk olarak hangi sırada bulunduğunu gösterdi. Bir de şu örneğe bakalım:

```
>>> kardiz.index("a", 1)

2
```

Gördüğünüz gibi, bu defa index() metoduna ikinci bir parametre daha verdik. index() metodunun ikinci parametresi, Python'ın aramaya kaçıncı sıradan itibaren başlayacağını gösteriyor. Biz yukarıdaki örnekte Python'ın aramaya 1. sıradan itibaren başlamasını istedik. Bu yüzden Python 0. sıradaki "a" karakterini es geçti ve 2. sırada bulunan "a" karakterini gördü. Bir de şuna bakalım:

```
>>> kardiz.index("a", 3)
```

Bu defa Python'ın aramaya *3*. sıradan başlamasını istedik. Dolayısıyla Python *0*. ve *2*. sıralardaki 'a' harflerini görmezden gelip bize *4*. sıradaki 'a' harfinin sırasını bildirdi.

Gelelim index() metodunun *3*. parametresine... Dilerseniz *3*. parametrenin ne işe yaradığını bir örnek üzerinde gösterelim:

```
>>> kardiz = "adana"
>>> kardiz.index("a", 1, 3)

2
```

Hatırlarsanız, bundan önce count() adlı bir metot öğrenmiştik. O metot da toplam *3* parametre alıyordu. count() metodunda kullandığımız *2*. ve *3*. parametrelerin görevlerini hatırlıyor olmalısınız. İşte index() metodunun *2*. ve *3*. parametreleri de aynen count() metodundaki gibi çalışır. Yani Python'ın sorgulama işlemini hangi sıra aralıklarından gerçekleştireceğini gösterir. Mesela yukarıdaki örnekte biz "adana" karakter dizisinin *1*. ve *3*. sıraları arasındaki 'a' harflerini sorguladık. Yani yukarıdaki örnekte Python 'a' harfini aramaya *1*. konumdan başladı ve aramayı *3*. konumda kesti. Böylece *"adana"* karakter dizisinin *2*. sırasındaki 'a' harfinin konumunu bize bildirdi.

Gördüğünüz gibi, index() metodu bize aradığımız karakterin yalnızca ilk konumunu bildiriyor. Peki biz mesela *"adana"* karakter dizisi içindeki bütün 'a' harflerinin sırasını öğrenmek istersek ne

yapacağız?

Bu isteğimizi yerine getirmek için karakter dizisinin her bir sırasını tek tek kontrol etmemiz yeterli olacaktır. Yani şöyle bir şey yazmamız gerekiyor:

```
kardiz = "adana"

print(kardiz.index("a", 0))
print(kardiz.index("a", 1))
print(kardiz.index("a", 2))
print(kardiz.index("a", 3))
print(kardiz.index("a", 4))
```

Buradaki mantığı anladığınızı sanıyorum. Bildiğiniz gibi, index() metodunun ikinci parametresi sayesinde karakter dizisi içinde aradığımız bir karakteri hangi konumdan itibaren arayacağımızı belirleyebiliyoruz. Örneğin yukarıdaki kodlarda gördüğünüz ilk print() satırı 'a' karakterini *0.* konumdan itibaren arıyor ve gördüğü ilk 'a' harfinin konumunu raporluyor. İkinci print() satırı 'a' karakterini *1.* konumdan itibaren arıyor ve gördüğü ilk 'a' harfinin konumunu raporluyor. Bu süreç karakter dizisinin sonuna ulaşılıncaya kadar devam ediyor. Böylece karakter dizisi içinde geçen bütün 'a' harflerinin konumunu elde etmiş oluyoruz.

Elbette yukarıdaki kodları, sadece işin mantığını anlamanızı sağlamak için bu şekilde verdik. Tahmin edebileceğiniz gibi, yukarıdaki kod yazımı son derece verimsiz bir yoldur. Ayrıca gördüğünüz gibi, yukarıdaki kodlar sadece *5* karakter uzunluğundaki karakter dizileri için geçerlidir. Halbuki programlamada esas alınması gereken yöntem, kodlarınızı olabildiğince genel amaçlı tutup, farklı durumlarda da çalışabilmesini sağlamaktır. Dolayısıyla yukarıdaki mantığı şu şekilde kodlara dökmek çok daha akıllıca bir yol olacaktır:

```
kardiz = "adana"

for i in range(len(kardiz)):
    print(kardiz.index("a", i))
```

Gördüğünüz gibi, yukarıdaki kodlar yardımıyla, bir önceki verimsiz kodları hem kısalttık, hem de daha geniş kapsamlı bir hale getirdik. Hatta yukarıdaki kodları şöyle yazarsanız karakter dizisi ve bu karakter dizisi içinde aranacak karakteri kullanıcıdan da alabilirsiniz:

```
kardiz = input("Metin girin: ")
aranacak = input("Aradığınız harf: ")

for i in range(len(kardiz)):
    print(kardiz.index(aranacak, i)
```

Bu kodlarda bazı problemler dikkatinizi çekmiş olmalı. Mesela, aranan karakter dizisinin bulunduğu konumlar çıktıda tekrar ediyor. Örneğin, kullanıcının *"adana"* karakter dizisi içinde 'a' harfini

aramak istediğini varsayarsak programımız şöyle bir çıktı veriyor:

```
0
2
2
4
4
```

Burada *2* ve *4* sayılarının birden fazla geçtiğini görüyoruz. Bunu engellemek için şöyle bir kod yazabiliriz:

```
kardiz = input("Metin girin: ")
aranacak = input("Aradığınız harf: ")

for i in range(len(kardiz)):
    if i == kardiz.index(aranacak, i):
        print(i)
```

Bu kodlarla yaptığımız şey şu: Öncelikle karakter dizisinin uzunluğunu gösteren sayı aralığı üzerinde bir for döngüsü kuruyoruz. Kullanıcının burada yine *"adana"* karakter dizisini girdiğini varsayarsak, *"adana"* karakter dizisinin uzunluğu *5* olduğu için for döngümüz şöyle görünecektir:

```
for i in range(5):
    ...
```

Daha sonra for döngüsü içinde tanımladığımız *i* değişkeninin değerinin, karakter dizisi içinde aradığımız karakterin konumu ile eşleşip eşleşmediğini kontrol ediyoruz ve değeri eşleşen sayıları print() fonksiyonunu kullanarak ekrana döküyoruz.

Eğer bu kodlar ilk bakışta gözünüze anlaşılmaz göründüyse bu kodları bir de şu şekilde yazarak arka planda neler olup bittiğini daha net görebilirsiniz:

```
kardiz = input("Metin girin: ")
aranacak = input("Aradığınız harf: ")

for i in range(len(kardiz)):
    print("i'nin değeri: ", i)
    if i == kardiz.index(aranacak, i):
        print("%s. sırada 1 adet %s harfi bulunuyor" %(i, aranacak))
    else:
        print("%s. sırada %s harfi bulunmuyor" %(i, aranacak))
```

Gördüğünüz gibi index() metodu bir karakter dizisi içindeki karakterleri ararken karakter dizisini soldan sağa doğru okuyor. Python'da bu işlemin tersi de mümkündür. Yani isterseniz Python'ın, karakter dizisini soldan sağa doğru değil de, sağdan sola doğru okumasını da sağlayabilirsiniz. Bu iş

için rindex() adlı bir metottan yararlanacağız. Bu metot her yönden index() metoduyla aynıdır. index() ve rindex() metotlarının birbirinden tek farkı, index() metodunun karakter dizilerini soldan sağa, rindex() metodunun ise sağdan sola doğru okumasıdır. Hemen bir örnekle durumu açıklamaya çalışalım:

```
>>> kardiz = "adana"
>>> kardiz.index("a")

0

>>> kardiz.rindex("a")

4
```

Bu iki örnek, index() ve rindex() metotları arasındaki farkı gayet net bir şekilde ortaya koyuyor. index() metodu, karakter dizisini soldan sağa doğru okuduğu için *"adana"* karakter dizisinin 0. sırasındaki 'a' harfini yakaladı. rindex() metodu ise karakter dizisini sağdan sola doğru okuduğu için *"adana"* karakter dizisinin *4*. sırasındaki 'a' harfini yakaladı...

find, rfind()

find() ve rfind() metotları tamamen index() ve rindex() metotlarına benzer. find() ve rfind() metotlarının görevi de bir karakter dizisi içindeki bir karakterin konumunu sorgulamaktır:

```
>>> kardiz = "adana"
>>> kardiz.find("a")

0

>>> kardiz.rfind("a")

4
```

Peki index()/rindex() ve find()/rfind() metotları arasında ne fark var?

index() ve rindex() metotları karakter dizisi içindeki karakteri sorgularken, eğer o karakteri bulamazsa bir ValueError hatası verir:

```
>>> kardiz = "adana"
>>> kardiz.index("z")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
ValueError: substring not found
```

Ama find() ve rfind() metotları böyle bir durumda -1 çıktısı verir:

```
>>> kardiz = "adana"
>>> kardiz.find("z")

-1
```

Bu iki metot çifti arasındaki tek fark budur.

center()

Center kelimesi İngilizce'de 'orta, merkez, ortalamak' gibi anlamlara gelir. Bu anlama uygun olarak, center() metodunu karakter dizilerini ortalamak için kullanabilirsiniz. Örneğin:

```
for metot in dir(""):
    print(metot.center(15))
```

Gördüğünüz gibi center() metodu bir adet parametre alıyor. Bu parametre, karakter dizisine uygulanacak ortalama işleminin genişliğini gösteriyor. Bu parametrenin nasıl bir etki ortaya çıkardığını daha iyi anlayabilmek için isterseniz bir iki basit örnek verelim:

```
>>> kardiz = "python"
```

Burada *6* karakterlik bir karakter dizisi tanımladık. Şimdi dikkatlice bakın:

```
>>> kardiz.center(1)

'python'
```

Burada ise center() metoduna parametre olarak *1* sayısını verdik. Ancak bu parametre karakter dizimizinin uzunluğundan az olduğu için çıktı üzerinde herhangi bir etkisi olmadı. Bir de şuna bakalım:

```
>>> kardiz.center(10)

' python  '
```

Çıktıdaki tırnak işaretlerine bakarak, 'python' kelimesinin ortalandığını görebilirsiniz. Buradan şu sonucu çıkarıyoruz: center() metoduna verilen genişlik parametresi aslında bir karakter dizisinin toplam kaç karakterlik bir yer kaplayacağını gösteriyor. Mesela yukarıdaki örnekte bu metoda verdiğimiz *10* sayısı "python" adlı karakter dizisinin toplam *10* karakterlik bir yer kaplayacağını gösteriyor. Kaplanacak yere karakter dizisinin kendisi de dahildir. Yani *10* olarak belirttiğimiz boşluk adedinin *6*'sı 'python' kelimesinin kendisi tarafından işgal ediliyor. Geriye kalan *4* boşlukluk

mesafe ise karakter dizisinin sol ve sağ tarafına paylaştırılıyor.

center() metodunun karakter dizileri üzerindeki etkisini daha net olarak görmek için şöyle bir döngü kurabilirsiniz:

```
>>> for i in range(1, 20):
...     kardiz.center(i)
...
'python'
'python'
'python'
'python'
'python'
'python'
' python'
' python '
'  python '
'  python  '
'   python  '
'   python   '
'    python   '
'    python    '
'     python    '
'     python     '
'      python     '
'      python      '
'       python      '
```

Bu örnekte, karakter dizisinin her adımda nasıl ortaya doğru kaydığı açıkça görülüyor. Dikkat ederseniz çıktının ilk altı satırında karakter dizisinin konumu değişmiyor. Ancak center() metoduna verilen parametrenin değeri karakter dizisinin uzunluğunu aştığı anda karakter dizisi ortaya doğru ilermeye başlıyor.

center() metodu genellikle yukarıdaki gösterdiğimiz şekilde tek bir parametre ile birlikte kullanılır. Ancak bu metot aslında bir parametre daha alır. Şu örneği inceleyelim:

```
>>> kardiz = "elma"
>>> kardiz.center(10, "-")

'---elma---'
```

Gördüğünüz gibi, center() metoduna verdiğimiz *"-"* değeri sayesinde *"elma"* karakteri ortalanırken, sağ ve sol taraftaki boşluklara da *"-"* karakteri eklenmiş oldu.

rjust(), ljust()

Bu metotlar da tıpkı bir önceki center() metodu gibi karakter dizilerini hizalama vazifesi görür. rjust() metodu bir karakter dizisini sağa yaslarken, ljust() metodu karakter dizisini sola yaslar. Mesela şu iki kod parçasının çıktılarını inceleyin:

```
>>> for i in dir(""):
...     print(i.ljust(20))

>>> for i in dir(""):
...     print(i.rjust(20))
```

ljust() metodu bize özellikle karakter dizilerinin hizalama işlemlerinde yardımcı oluyor. Bu metot yardımıyla karakter dizilerimizi sola yaslayıp, sağ tarafına da istediğimiz karakterleri yerleştirebiliyoruz. Hemen bir örnek verelim:

```
>>> kardiz = "tel no"
>>> kardiz.ljust(10, ".")

'tel no....'
```

Burada olan şey şu: ljust() metodu, kendisine verilen *10* parametresinin etkisiyle *10* karakterlik bir alan oluşturuyor. Bu 10 karakterlik alanın içine önce *6* karakterlik yer kaplayan *"tel no"* ifadesini, geri kalan *4* karakterlik boşluğa ise *"."* karakterini yerleştiriyor. Eğer ljust() metoduna verilen sayı karakter dizisinin uzunluğundan az yer tutarsa, karakter dizisinin görünüşünde herhangi bir değişiklik olmayacaktır. Örneğin yukarıdaki örnekte karakter dizimizin uzunluğu *6*. Dolayısıyla kodumuzu şu şekilde yazarsak bir sonuç elde edemeyiz:

```
>>> kardiz.ljust(5, ".")

'tel no'
```

Gördüğünüz gibi, karakter dizisinde herhangi bir değişiklik olmadı. ljust() metoduna verdiğimiz *"."* karakterini görebilmemiz için, verdiğimiz sayı cinsli parametrenin en az karakter dizisinin boyunun bir fazlası olması gerekir:

```
>>> kardiz.ljust(7, ".")

'tel no.'
```

ljust() metoduyla ilgili basit bir örnek daha verelim:

```
>>> for i in "elma", "armut", "patlıcan":
...     i.ljust(10, ".")
```

```
...
'elma......'
'armut.....'
'patlıcan..'
```

Gördüğünüz gibi, bu metot karakter dizilerini şık bir biçimde sola hizalamamıza yardımcı oluyor.

rjust() metodu ise, ljust() metodunun yaptığı işin tam tersini yapar. Yani karakter dizilerini sola değil sağa yaslar:

```
>>> for i in "elma", "armut", "patlıcan":
...     i.rjust(10, ".")
...
'......elma'
'.....armut'
'..patlıcan'
```

ljust() ve rjust() metotları, kullanıcılarınıza göstereceğiniz çıktıların düzgün görünmesini sağlamak açısından oldukça faydalıdır.

zfill()

Bu metot kimi yerlerde işimizi epey kolaylaştırabilir. zfill() metodu yardımıyla karakter dizilerinin sol tarafına istediğimiz sayıda sıfır ekleyebiliriz:

```
>>> a = "12"
>>> a.zfill(3)

'012'
```

Bu metodu şöyle bir iş için kullanabilirsiniz:

```
>>> for i in range(11):
...     print(str(i).zfill(2))
00
01
02
03
04
05
06
07
08
09
```

```
10
```

Burada str() fonksiyonunu kullanarak, range() fonksiyonundan elde ettiğimiz sayıları birer karakter dizisine çevirdiğimize dikkat edin. Çünkü zfill() karakter dizilerinin bir metodudur. Sayıların değil...

partition(), rpartition()

Bu metot yardımıyla bir karakter dizisini belli bir ölçüte göre üçe bölüyoruz. Örneğin:

```
>>> a = "istanbul"
>>> a.partition("an")

('ist', 'an', 'bul')
```

Eğer partition() metoduna parantez içinde verdiğimiz ölçüt karakter dizisi içinde bulunmuyorsa şu sonuçla karşılaşırız:

```
>>> a = "istanbul"
>>> a.partition("h")

('istanbul', '', '')
```

Gelelim rpartition() metoduna... Bu metot da partition() metodu ile aynı işi yapar, ama yöntemi biraz farklıdır. partition() metodu karakter dizilerini soldan sağa doğru okur. rpartition() metodu ise sağdan sola doğru. Peki bu durumun ne gibi bir sonucu vardır? Hemen görelim:

```
>>> b = "istihza"
>>> b.partition("i")

('', 'i', 'stihza')
```

Gördüğünüz gibi, partition() metodu karakter dizisini ilk ‘i’ harfinden böldü. Şimdi aynı işlemi rpartition() metodu ile yapalım:

```
>>> b.rpartition("i")

('ist', 'i', 'hza')
```

rpartition() metodu ise, karakter dizisini sağdan sola doğru okuduğu için ilk ‘i’ harfinden değil, son ‘i’ harfinden böldü karakter dizisini.

partition() ve rpartition() metotları, ölçütün karakter dizisi içinde bulunmadığı durumlarda da farklı tepkiler verir:

```
>>> b.partition("g")

('istihza', '', '')

>>> b.rpartition("g")

('', '', 'istihza')
```

Gördüğünüz gibi, partition() metodu boş karakter dizilerini sağa doğru yaslarken, rpartition() metodu sola doğru yasladı.

encode()

Bu metot yardımıyla karakter dizilerimizi istediğimiz kodlama sistemine göre kodlayabiliriz. Python 3.x'te varsayılan karakter kodlaması *utf-8*'dir. Eğer istersek şu karakter dizisini *utf-8* yerine *cp1254* ile kodlayabiliriz:

```
>>> "çilek".encode("cp1254")
```

expandtabs()

Bu metot yardımıyla bir karakter dizisi içindeki sekme boşluklarını genişletebiliyoruz. Örneğin:

```
>>> a = "elma\tbir\tmeyvedir"
>>> a.expandtabs(10)

'elma bir meyvedir'
```

Böylece bir metot grubunu daha geride bırakmış olduk. Gördüğünüz gibi bazı metotlar sıklıkla kullanılabilme potansiyeli taşırken, bazı metotlar pek öyle sık kullanılacakmış gibi görünmüyor...

Sonraki bölümde metotları incelemeye devam edeceğiz.

başa dön

Karakter Dizilerinin Metotları (Devamı)

Karakter dizileri konusunun 4. bölümüne geldik. Bu bölümde de karakter dizilerinin metotlarını incelemeye devam edeceğiz.

str.maketrans(), translate()

Bu iki metot birbiriyle bağlantılı olduğu ve genellikle birlikte kullanıldığı için, bunları bir arada göreceğiz.

Dilerseniz bu iki metodun ne işe yaradığını anlatmaya çalışmak yerine bir örnek üzerinden bu metotların görevini anlamayı deneyelim.

Şöyle bir vaka hayal edin: Bildiğiniz gibi, internet üzerinde bazen Türkçe karakterleri kullanamıyoruz. Böyle durumlarda, elimizdeki bir metni, cümleyi veya kelimeyi Türkçe karakter içermeyecek bir hale getirmemiz gerekebiliyor. Örneğin şu cümleyi ele alalım:

> Bildiğiniz gibi, internet üzerinde bazen Türkçe karakterleri kullanamıyoruz.

İşte buna benzer bir cümleyi kimi zaman Türkçe karakterlerinden arındırmak zorunda kalabiliyoruz. Eğer elinizde Türkçe yazılmış bir metin varsa ve sizin amacınız bu metin içinde geçen Türkçeye özgü karakterleri noktasız benzerleriyle değiştirmek ise str.maketrans() ve translate() metotlarından yararlanabilirsiniz.

Örneğimiz şu cümle idi:

> Bildiğiniz gibi, internet üzerinde bazen Türkçe karakterleri kullanamıyoruz.

Amacımız bu cümleyi şu şekilde değiştirmek:

> Bildiginiz gibi, internet uzerinde bazen Turkce karakterleri kullanamiyoruz.

Bunun için şöyle bir kod yazabilirsiniz:

```
kaynak = "şçöğüıŞÇÖĞÜİ"
hedef  = "scoguiSCOGUI"

çeviri_tablosu = str.maketrans(kaynak, hedef)

metin = "Bildiğiniz gibi, internet üzerinde bazen Türkçe karakterleri kullanamıyoruz."

print(metin.translate(çeviri_tablosu))
```

Bu kodları çalıştırdığımızda şöyle bir çıktı elde ederiz:

```
Bildiginiz gibi, internet uzerinde bazen Turkce karakterleri kullanamiyoruz.
```

Gördüğünüz gibi, *"kaynak"* adlı karakter dizisi içinde belirttiğimiz bütün harfler *"hedef"* adlı karakter dizisi içindeki harflerle tek tek değiştirildi. Böylece Türkçeye özgü karakterleri ('şçöğüıŞÇÖĞÜİ') en yakın noktasız benzerleriyle ('scoguiSCOGUI') değiştirmiş olduk.

Peki yukarıda nasıl bir süreç işledi de biz istediğimiz sonucu elde edebildik. Dilerseniz yukarıdaki kodlara biraz daha yakından bakalım. Mesela *çeviri_tablosu* adlı değişkenin çıktısına bakarak str.maketrans() metodunun alttan alta neler karıştırdığını görelim:

```
kaynak = "şçöğüıŞÇÖĞÜİ"
hedef  = "scoguiSCOGUI"

çeviri_tablosu = str.maketrans(kaynak, hedef)

print(çeviri_tablosu)
```

Bu kodları çalıştırdığımızda şöyle bir çıktı alıyoruz:

```
{214: 79, 231: 99, 220: 85, 199: 67, 304: 73, 305: 105,
286: 71, 246: 111, 351: 115, 252: 117, 350: 83, 287: 103}
```

Bu çıktı size tamamen anlamsız görünmüş olabilir. Ama aslında son derece anlamlı ve bir o kadar da önemli bir çıktıdır bu. Gelin isterseniz bu çıktının yapısını biraz inceleyelim. (Buna benzer bir çıktıyı sorted() metodunu incelerken de görmüştük)

Gördüğünüz gibi, tamamen sayılardan oluşan bir çıktı bu. Burada birbirlerinden virgül ile ayrılmış sayı çiftleri görüyoruz. Bu sayı çiftlerini daha net görebilmek için bu çıktıyı derli toplu bir hale getirelim:

```
{214: 79,
 231: 99,
 220: 85,
 199: 67,
 304: 73,
 305: 105,
 286: 71,
 246: 111,
 351: 115,
 252: 117,
 350: 83,
 287: 103}
```

Bu şekilde sanırım çıktımız biraz daha anlam kazandı. Gördüğünüz gibi, iki nokta üst üste işaretinin solunda ve sağında bazı sayılar var. Tahmin edebileceğiniz gibi, soldaki sayılar sağdaki sayılarla ilişkili.

Peki bütün bu sayılar ne anlama geliyor ve bu sayılar arasında ne tür bir ilişki var?

Teknik olarak, bilgisayarların temelinde sayılar olduğunu duymuşsunuzdur. Bilgisayarınızda gördüğünüz her karakter aslında bir sayıya karşılık gelir. Zaten bilgisayarlar ‘a’, ‘b’, ‘c’, vb. kavramları anlayamaz. Bilgisayarların anlayabildiği tek şey sayılardır. Mesela siz klavyeden ‘a’ harfini girdiğinizde bilgisayar bunu *97* olarak algılar. Ya da siz ‘i’ harfi girdiğinizde, bilgisayarın gördüğü tek şey *105* sayısıdır... Bu durumu Python’daki chr() adlı özel bir fonksiyon yardımıyla teyit edebiliriz. Dikkatlice inceleyin:

```
>>> chr(97)

'a'

>>> chr(105)

'i'

>>> chr(65)

'A'
```

Gördüğünüz gibi, gerçekten de her sayı bir karaktere karşılık geliyor. İsterseniz bir de yukarıdaki sayı grubundaki sayıları denetleyelim:

```
for i in 214, 231, 220, 199, 304, 305, 286, 246, 351, 252, 350, 287:
    print(i, chr(i))
```

Bu kodları çalıştırdığımızda şu çıktıyı elde ediyoruz:

```
214 Ö
231 ç
220 Ü
199 Ç
304 İ
305 ı
286 Ğ
246 ö
351 ş
252 ü
350 Ş
287 ğ
```

Bu çıktı sayesinde bazı şeyler zihninizde yavaş yavaş açıklığa kavuşuyor olmalı. Bu çıktı mesela *214* sayısının 'Ö' harfine, *220* sayısının 'Ü' harfine, *305* sayısının da 'ı' harfine karşılık geldiğini gösteriyor.

Burada iki nokta işaretinin sol tarafında kalan sayıların karakter karşılıklarını gördük. Bir de iki nokta işaretinin sağ tarafında kalan sayılara bakalım:

```
for i in 79, 99, 85, 67, 73, 105, 71, 111, 115, 117, 83, 103:
    print(i, chr(i))
```

Bu da şu çıktıyı verdi:

```
79  O
99  c
85  U
67  C
73  I
105 i
71  G
111 o
115 s
117 u
83  S
103 g
```

Burada da mesela *79* sayısının 'O' harfine, *85* sayısının 'U' harfine, *105* sayısının da 'i' harfine karşılık geldiğini görüyoruz.

Yukarıdaki ve yukarıdan bir önceki kodların çıktılarını bir araya getirirseniz şöyle bir durumla karşı karşıya olduğunuzu görürsünüz:

```
Ö  O
ç  c
Ü  U
Ç  C
İ  I
ı  i
Ğ  G
ö  o
ş  s
ü  u
Ş  S
ğ  g
```

Bütün bu söylediklerimizden şu sonuç çıkıyor:

çeviri_tablosu = str.maketrans(kaynak, hedef) satırı, *kaynak* ve *hedef* olarak adlandırdığımız karakter dizilerini birleştirip, bu değişkenler içindeki herbir karakteri birbiriyle eşleştiriyor. Yani aşağıdaki gibi bir işlem yapıyor:

```
çeviri_tablosu = {"Ö": "O",
                  "ç": "c",
                  "Ü": "U",
                  "Ç": "C",
                  "İ": "I",
                  "ı": "i",
                  "Ğ": "G",
                  "ö": "o",
                  "ş": "s",
                  "ü": "u",
                  "Ş": "S",
                  "ğ": "g"}
```

Burada *çeviri_tablosu* değişkeni içinde gösterdiğimiz biçimin Python'daki adı 'sözlük'tür. Sözlükler de tıpkı karakter dizileri gibi bir veri tipidir. Bunları da birkaç bölüm sonra ayrıntılı bir biçimde inceleyeceğiz. Biz burada, bazı şeyleri anlamamızı kolaylaştıracağı için sözlük adlı veri tipini oldukça genel bir biçimde sizlere tanıttık. Dediğim gibi, bu veri tipinin ayrıntılarını daha sonra inceleyeceğiz, ama yine de şu noktada sözlükleri kenarından köşesinden de olsa tanımamız bizim için faydalı olacaktır.

Dediğim gibi, yukarıda *çeviri_tablosu* adıyla gösterdiğimiz şey bir sözlüktür. Bu sözlüğün nasıl çalıştığını görmek için şöyle bir kod yazalım:

```
çeviri_tablosu = {"Ö": "O",
                  "ç": "c",
                  "Ü": "U",
                  "Ç": "C",
                  "İ": "I",
                  "ı": "i",
                  "Ğ": "G",
                  "ö": "o",
                  "ş": "s",
                  "ü": "u",
                  "Ş": "S",
                  "ğ": "g"}

print(çeviri_tablosu["Ö"])
```

Bu kodları bir dosyaya kaydedip çalıştırırsanız şöyle bir çıktı alırsınız:

```
O
```

Gördüğünüz gibi, sözlük içinde geçen *"Ö"* adlı öğeyi parantez içinde belirttiğimiz zaman, Python bize bu öğenin karşısındaki değeri veriyor. Sözlük içinde *"Ö"* öğesinin karşılığı *"O"* harfi olduğu için de çıktımız *"O"* oluyor. Bir de şunlara bakalım:

```
çeviri_tablosu = {"Ö": "O",
                  "ç": "c",
                  "Ü": "U",
                  "Ç": "C",
                  "İ": "I",
                  "ı": "i",
                  "Ğ": "G",
                  "ö": "o",
                  "ş": "s",
                  "ü": "u",
                  "Ş": "S",
                  "ğ": "g"}

print(çeviri_tablosu["Ö"])
print(çeviri_tablosu["ç"])
print(çeviri_tablosu["Ü"])
print(çeviri_tablosu["Ç"])
print(çeviri_tablosu["İ"])
print(çeviri_tablosu["ı"])
print(çeviri_tablosu["Ğ"])
print(çeviri_tablosu["ö"])
print(çeviri_tablosu["Ş"])
print(çeviri_tablosu["ğ"])
```

Bu kodları çalıştırdığımızda ise şöyle bir çıktı alıyoruz:

```
O
c
U
C
I
i
G
o
S
g
```

Gördüğünüz gibi, sözlük içinde iki nokta üst üste işaretinin sol tarafında görünen öğeleri parantez

içinde yazarak, iki nokta üst üste işaretinin sağ tarafındaki değerleri elde edebiliyoruz.

Bütün bu anlattıklarımızdan sonra şu satırları gayet iyi anlamış olmalısınız:

```
kaynak = "şçöğüıŞÇÖĞÜİ"
hedef  = "scoguiSCOGUI"

çeviri_tablosu = str.maketrans(kaynak, hedef)
```

Burada Python, *kaynak* ve *hedef* adlı değişkenler içindeki karakter dizilerini birer birer eşleştirerek bize bir sözlük veriyor. Bu sözlükte:

```
"ş" harfi "s" harfine;
"ç" harfi "c" harfine;
"ö" harfi "o" harfine;
"ğ" harfi "g" harfine;
"ü" harfi "u" harfine;
"ı" harfi "i" harfine;
"Ş" harfi "S" harfine;
"Ç" harfi "C" harfine;
"Ö" harfi "O" harfine;
"Ğ" harfi "G" harfine;
"Ü" harfi "U" harfine;
"İ" harfi "I" harfine
```

karşılık geliyor...

Kodların geri kalanında ise şu satırları görmüştük:

```
metin = "Bildiğiniz gibi, internet üzerinde bazen Türkçe karakterleri kullanamıyoruz."

print(metin.translate(çeviri_tablosu))
```

Burada da orijinal metnimizi tanımladıktan sonra translate() adlı metot yardımıyla, çeviri tablosundaki öğe eşleşmesi doğrultusunda metnimizi tercüme ediyoruz. Bu kodlarda metin.translate(çeviri_tablosu) satırının yaptığı tek şey *çeviri_tablosu* adlı sözlükteki eşleşme kriterlerini *metin* adlı karakter dizisine uygulamaktan ibarettir.

Karakter dizilerinin bu maketrans() adlı metodu kullanım olarak gözünüze öteki metotlardan farklı görünmüş olabilir. Daha açık bir dille ifade etmek gerekirse, bu metodu bir karakter dizisi üzerine değil de *str* üzerine uyguluyor olmamız, yani str.maketrans() yazıyor olmamız sizi şaşırtmış olabilir. Eğer anlamanızı kolaylaştıracaksa;

```
çeviri_tablosu = str.maketrans(kaynak, hedef)
```

satırını şu şekilde de yazabilirsiniz:

```
çeviri_tablosu = ''.maketrans(kaynak, hedef)
```

Yani maketrans() metodunu boş bir karakter dizisi üzerine de uygulayabilirsiniz. Neticede maketrans() karakter dizilerinin bir metodudur. Bu metot hangi karakter dizisi üzerine uygulandığıyla değil, parametre olarak hangi değerleri aldığıyla (bizim örneğimizde *kaynak* ve *hedef*) ilgilenir. Dolayısıyla bu metodu ilgili-ilgisiz her türlü karakter dizisine uygulayabilirsiniz:

```
çeviri_tablosu = 'mahmut'.maketrans(kaynak, hedef)
çeviri_tablosu = 'zalim dünya!'.maketrans(kaynak, hedef)
```

Ama tabii dikkat dağıtmamak açısından en uygun hareket, bu karakter dizisini *str* üzerine uygulamak olacaktır:

```
çeviri_tablosu = str.maketrans(kaynak, hedef)
```

Bu küçük ayrıntıya da dikkati çektiğimize göre yolumuza devam edebiliriz...

Yukarıda verdiğimiz örnek vasıtasıyla str.maketrans() ve translate() adlı metotları epey ayrıntılı bir şekilde incelemiş olduk. Dilerseniz pratik olması açısından bir örnek daha verelim:

istihza.com sitemizin forum üyelerinden Barbaros Akkurt http://www.istihza.com/forum/viewtopic.php?f=25&t=63 adresinde şöyle bir problemden bahsediyor:

> Ben on parmak Türkçe F klavye kullanıyorum. Bunun için, bazı tuş kombinasyonları ile veya sistem tepsisi üzerindeki klavye simgesine tıklayarak Türkçe Q - Türkçe F değişimi yapıyorum. Bazen bunu yapmayı unutuyorum ve bir metne bakarak yazıyorsam gözüm ekranda olmuyor. Bir paragrafı yazıp bitirdikten sonra ekranda bir karakter salatası görünce çok bozuluyorum.

İşte böyle bir durumda yukarıdaki iki metodu kullanarak o karakter salatasını düzeltebilirsiniz. Karakter salatamız şu olsun:

> Bfjflrk öa kdhsı yteua idjslyd bdcusldvdj ks?

Buna göre kodlarımızı yazmaya başlayabiliriz. Öncelikle metnimizi tanımlayalım:

```
metin = "Bfjflrk öa kdhsı yteua idjslyd bdcusldvdj ks?"
```

Şimdi de sırasıyla q ve f klavye düzenlerini birer karakter dizisi haline getirelim:

```
q_klavye_düzeni = "qwertyuıopğüasdfghjklşi,zxcvbnmöç."
f_klavye_düzeni = "fgğıodrnhpqwuieaütkmlyşxjövcçzsb.,"
```

Burada amacımız yanlışlıkla q klavye düzeninde yazıldığı için karman çorman bir hale gelmiş metni

düzgün bir şekilde f klavye düzenine dönüştürmek. Yani burada çıkış noktamız (kaynağımız) *q_klavye_düzeni* iken, varış noktamız (hedefimiz) *f_klavye_düzeni*. Buna göre çeviri tablomuzu oluşturabiliriz:

```
çeviri_tablosu = str.maketrans(q_klavye_düzeni, f_klavye_düzeni)
```

Tıpkı bir önceki örnekte olduğu gibi, burada da *çeviri_tablosu* adlı değişkeni print() fonksiyonunu kullanarak yazdırırsanız şöyle bir çıktıyla karşılaşırsınız:

```
{231: 46,
 287: 113,
 44 : 120,
 46 : 44,
 305: 110,
 246: 98,
 351: 121,
 97 : 117,
 98 : 231,
 99 : 118,
 100: 101,
 101: 287,
 102: 97,
 103: 252,
 104: 116,
 105: 351,
 106: 107,
 107: 109,
 108: 108,
 109: 115,
 110: 122,
 111: 104,
 112: 112,
 113: 102,
 114: 305,
 115: 105,
 116: 111,
 117: 114,
 118: 99,
 119: 103,
 120: 246,
 121: 100,
 122: 106,
 252: 119}
```

Tahmin edebileceğiniz gibi, bu sözlükte iki nokta üst üste işaretinin solundaki sayılar *q_klavye_düzeni* adlı değişken içindeki karakterleri; sağındaki sayılar ise *f_klavye_düzeni* adlı değişken içindeki karakterleri temsil ediyor.

Son olarak translate() metodunu yardımıyla sözlükteki öğe eşleşmesini *metin* adlı değişkenin üzerine uyguluyoruz:

```
print(metin.translate(çeviri_tablosu))
```

Kodları topluca görelim:

```
metin = "Bfjflrk öa kdhsı yteua idjslyd bdcusldvdj ks?"

q_klavye_düzeni = "qwertyuıopğüasdfghjklşi,zxcvbnmöç."
f_klavye_düzeni = "fgğıodrnhpqwuieaütkmlyşxjövcçzsb.,"

çeviri_tablosu = str.maketrans(q_klavye_düzeni, f_klavye_düzeni)

print(metin.translate(çeviri_tablosu))
```

Ne elde ettiniz?

Yukarıdaki iki örnekte de gördüğümüz gibi, str.maketrans() metodu kaynak ve hedef karakter dizilerini alıp bunları birleştirerek bize bir sözlük veri tipinde bir nesne veriyor. Yani tıpkı input() fonksiyonunun bize bir karakter dizisi verdiği gibi, str.maketrans() metodu da bize bir sözlük veriyor.

Eğer isterseniz, sözlüğü str.maketrans() metoduna oluşturtmak yerine, kendiniz de bir sözlük oluşturarak str.maketrans() metoduna parametre olarak atayabilirsiniz. Örneğin:

```
metin = "Bfjflrk öa kdhsı yteua idjslyd bdcusldvdj ks?"

sözlük = {"q": "f",
        "w": "g",
        "e": "ğ",
        "r": "ı",
        "t": "o",
        "y": "d",
        "u": "r",
        "ı": "n",
        "o": "h",
        "p": "p",
        "ğ": "q",
        "ü": "w",
```

```
        "a": "u",
        "s": "i",
        "d": "e",
        "f": "a",
        "g": "ü",
        "h": "t",
        "j": "k",
        "k": "m",
        "l": "l",
        "ş": "y",
        "i": "ş",
        ",": "x",
        "z": "j",
        "x": "ö",
        "c": "v",
        "v": "c",
        "b": "ç",
        "n": "z",
        "m": "s",
        "ö": "b",
        "ç": ".",
        ".": ","}

çeviri_tablosu = str.maketrans(sözlük)
print(metin.translate(çeviri_tablosu))
```

Burada birbiriyle eşleşecek karakterleri kendimiz yazıp bir sözlük oluşturduk ve bunu parametre olarak doğrudan str.maketrans() metoduna verdik. Bu kodlarda kaynak ve hedef diye iki ayrı karakter dizisi tanımlamak yerine tek bir sözlük oluşturduğumuz için, str.maketrans() metodunu iki parametreyle değil, tek parametreyle kullandığımıza dikkat edin. Ayrıca sözlüğü nasıl oluşturduğumuzu da dikkatlice inceleyin.

Sözlükteki öğe çiftlerini böyle alt alta yazmamızın nedeni zorunluluk değil, bir tercihtir. İstersek bu sözlüğü şöyle de tanımlayabilirdik:

```
sözlük = {"q": "f", "w": "g", "e": "ğ", "r": "ı", "t": "o", "y": "d", "u": "r",
        "ı": "n", "o": "h", "p": "p", "ğ": "q", "ü": "w", "a": "u", "s": "i",
        "d": "e", "f": "a", "g": "ü", "h": "t", "j": "k", "k": "m", "l": "l",
        "ş": "y", "i": "ş", ",": "x", "z": "j", "x": "ö", "c": "v", "v": "c",
        "b": "ç", "n": "z", "m": "s", "ö": "b", "ç": ".", ".": ","}
```

Burada da öğe çiftlerini yan yana yazdık. Bu iki yöntemden hangisi size daha okunaklı geliyorsa onu tercih edebilirsiniz.

Şimdi size bir soru sormama izin verin. Acaba aşağıdaki metin içinde geçen bütün sesli harfleri silin desem, nasıl bir kod yazarsınız?

> Bu programlama dili Guido Van Rossum adlı Hollandalı bir programcı tarafından 90'lı yılların başında geliştirilmeye başlanmıştır. Çoğu insan, isminin Python olmasına bakarak, bu programlama dilinin, adını piton yılanından aldığını düşünür. Ancak zannedildiğinin aksine bu programlama dilinin adı piton yılanından gelmez. Guido Van Rossum bu programlama dilini, The Monty Python adlı bir İngiliz komedi grubunun, Monty Python's Flying Circus adlı gösterisinden esinlenerek adlandırmıştır. Ancak her ne kadar gerçek böyle olsa da, Python programlama dilinin pek çok yerde bir yılan figürü ile temsil edilmesi neredeyse bir gelenek halini almıştır diyebiliriz.

Aklınıza ilk olarak şöyle bir kod yazmak gelebilir:

```
metin = """Bu programlama dili Guido Van Rossum adlı Hollandalı bir
programcı tarafından 90'lı yılların başında geliştirilmeye başlanmıştır.
Çoğu insan, isminin Python olmasına bakarak, bu programlama dilinin, adını
piton yılanından aldığını düşünür. Ancak zannedildiğinin aksine bu
programlama dilinin adı piton yılanından gelmez. Guido Van Rossum bu
programlama dilini, The Monty Python adlı bir İngiliz komedi grubunun, Monty
Python's Flying Circus adlı gösterisinden esinlenerek adlandırmıştır. Ancak
her ne kadar gerçek böyle olsa da, Python programlama dilinin pek çok yerde
bir yılan figürü ile temsil edilmesi neredeyse bir gelenek halini almıştır
diyebiliriz."""

sesli_harfler = "aeıioöuüAEIİOÖUÜ"

yeni_metin = ""

for i in metin:
    if not i in sesli_harfler:
        yeni_metin += i

print(yeni_metin)
```

Burada öncelikle *metin* adlı bir değişken tanımlayarak metnimizi bu değişken içine yerleştirdik. Ardından da Türkçedeki sesli harfleri içeren bir karakter dizisi tanımladık.

Daha sonra da *yeni_metin* adlı boş bir karakter dizisi oluşturduk. Bu karakter dizisi, orijinal metnin, sesli harfler ayıklandıktan sonraki halini barındıracak. Biliyorsunuz, karakter dizileri değiştirilemeyen (*immutable*) bir veri tipidir. Dolayısıyla bir karakter dizisi içinde yaptığımız değişiklikleri koruyabilmek için bu değişiklikleri başka bir değişken içinde tutmamız gerekiyor.

Bu kodların ardından bir *for* döngüsü tanımlıyoruz. Buna göre, metin içinde geçen her bir karaktere tek tek bakıyoruz (`for i in metin:`) ve bu karakterler arasında, *sesli_harfler* değişkeni içinde

geçmeyenleri, yani bütün sessiz harfleri (if not i in sesli_harfler:) tek tek *yeni_metin* adlı değişkene yolluyoruz (yeni_metin += i)

Son olarak da *yeni_metin* adlı karakter dizisini ekrana basıyoruz. Böylece orijinal metin içindeki bütün sesli harfleri ayıklamış oluyoruz.

Yukarıdaki, gayet doğru ve geçerli bir yöntemdir. Böyle bir kod yazmanızın hiçbir sakıncası yok. Ama eğer isterseniz aynı işi str.maketrans() ve translate() metotları yardımıyla da halledebilirsiniz:

```
metin = """Bu programlama dili Guido Van Rossum adlı Hollandalı bir
programcı tarafından 90'lı yılların başında geliştirilmeye başlanmıştır.
Çoğu insan, isminin Python olmasına bakarak, bu programlama dilinin, adını
piton yılanından aldığını düşünür. Ancak zannedildiğinin aksine bu
programlama dilinin adı piton yılanından gelmez. Guido Van Rossum bu
programlama dilini, The Monty Python adlı bir İngiliz komedi grubunun, Monty
Python's Flying Circus adlı gösterisinden esinlenerek adlandırmıştır. Ancak
her ne kadar gerçek böyle olsa da, Python programlama dilinin pek çok yerde
bir yılan figürü ile temsil edilmesi neredeyse bir gelenek halini almıştır
diyebiliriz."""

silinecek = "aeıioöuüAEIİOÖUÜ"

çeviri_tablosu = str.maketrans('', '', silinecek)

print(metin.translate(çeviri_tablosu))
```

Burada da öncelikle metnimizi bir karakter dizisi içine yerleştirdik. Daha sonra da şu kodu yazdık:

```
silinecek = "aeıioöuüAEIİOÖUÜ"
```

Bu kodlar yardımıyla, metin içinden çıkarmak istediğimiz harfleri tek tek belirledik.

Ardından str.maketrans() fonksiyonumuzu yazarak çeviri tablosunu oluşturduk. Burada ilk iki parametrenin boş birer karakter dizisi olduğuna dikkat ediyoruz. İlk iki parametreyi bu şekilde yazmamızın nedeni şu: Biz orijinal metin içindeki herhangi bir şeyi değiştirmek istemiyoruz. Bizim amacımız orijinal metin içindeki sesli harfleri silmek. Tabii o iki parametreyi yazmasak da olmaz. O yüzden o iki parametrenin yerine birer tane boş karakter dizisi yerleştiriyoruz.

Bu noktada *çeviri_tablosu* adlı değişkeni yazdırarak neler olup bittiğini daha net görebilirsiniz:

```
{214: None,
 97 : None,
 101: None,
 65 : None,
```

```
105: None,
111: None,
304: None,
305: None,
220: None,
117: None,
246: None,
73 : None,
79 : None,
252: None,
85 : None,
69 : None}
```

Gördüğünüz gibi, *silinecek* adlı değişken içindeki bütün karakterler None değeriyle eşleşiyor... None 'hiç, sıfır, yokluk' gibi anlamlara gelir. Dolayısıyla Python, iki nokta üst üste işaretinin sol tarafındaki karakterlerle karşılaştığında bunların yerine birer adet 'yokluk' koyuyor! Yani sonuç olarak bu karakterleri metinden silmiş oluyor...

Bu kodlarda iki nokta üst üste işaretinin solundaki karakterlerin None ile eşleşmesini sağlayan şey, str.maketrans() metoduna verdiğimiz üçüncü parametredir. Eğer o parametreyi yazmazsak, yani kodlarımızı şu şekle getirirsek *çeviri_tablosu* değişkeninin çıktısı farklı olacaktır:

```
metin = """Bu programlama dili Guido Van Rossum adlı Hollandalı bir
programcı tarafından 90'lı yılların başında geliştirilmeye başlanmıştır.
Çoğu insan, isminin Python olmasına bakarak, bu programlama dilinin, adını
piton yılanından aldığını düşünür. Ancak zannedildiğinin aksine bu
programlama dilinin adı piton yılanından gelmez. Guido Van Rossum bu
programlama dilini, The Monty Python adlı bir İngiliz komedi grubunun, Monty
Python's Flying Circus adlı gösterisinden esinlenerek adlandırmıştır. Ancak
her ne kadar gerçek böyle olsa da, Python programlama dilinin pek çok yerde
bir yılan figürü ile temsil edilmesi neredeyse bir gelenek halini almıştır
diyebiliriz."""

silinecek = "aeıioöuüAEIİOÖUÜ"

çeviri_tablosu = str.maketrans('', '')

print(çeviri_tablosu)
```

Bu kodları çalıştırdığımızda şöyle bir çıktı alırız:

```
{}
```

Gördüğünüz gibi, elde ettiğimiz şey boş bir sözlüktür. Sözlük boş olduğu, yani değiştirilecek herhangi

bir karakter olmadığı için bu kodlar orijinal metin üzerinde herhangi bir değişiklik yapmaz.

İsterseniz üçüncü parametrenin ne işe yaradığını ve nasıl çalıştığını daha iyi anlayabilmek için daha basit bir örnek verelim:

```
metin = "Cem Yılmaz"

kaynak = "CY"
hedef  = "cy"
silinecek = "eıa "

çeviri_tablosu = str.maketrans(kaynak, hedef, silinecek)

print(metin.translate(çeviri_tablosu))
```

Burada ‘C’ ve ‘Y’ harflerini sırasıyla ‘c’ ve ‘y’ harfleriyle eşleştirdik. Bu nedenle orijinal metin içindeki ‘C’ ve ‘Y’ harfleri yerlerini sırasıyla ‘c’ ve ‘y’ harflerine bıraktı. Silinecek karakterler olarak ise ‘e’, ‘ı’ ve boşluk karakterlerini seçtik. Böylece ‘Cem Yılmaz’ adlı orijinal metin içindeki boşluk karakteri de silinerek, bu metin ‘cmylmz’ karakter dizisine dönüştü.

isalpha()

Bu metot yardımıyla bir karakter dizisinin ‘alfabetik’ olup olmadığını denetleyeceğiz. Peki ‘alfabetik’ ne demek?

Eğer bir karakter dizisi içinde yalnızca alfabe harfleri (‘a’, ‘b’, ‘c’ gibi...) varsa o karakter dizisi için ‘alfabetik’ diyoruz. Bir örnekle bunu doğrulayalım:

```
>>> a = "kezban"
>>> a.isalpha()

True
```

Ama:

```
>>> b = "k3zb6n"
>>> b.isalpha()

False
```

isdigit()

Bu metot da isalpha() metoduna benzer. Bunun yardımıyla bir karakter dizisinin sayısal olup olmadığını denetleyebiliriz. Sayılardan oluşan karakter dizilerine ‘sayı değerli karakter dizileri’ adı

verilir. Örneğin şu bir ‘sayı değerli karakter dizisi’dir:

```
>>> a = "12345"
```

Metodumuz yardımıyla bunu doğrulayabiliriz:

```
>>> a.isdigit()

True
```

Ama şu karakter dizisi sayısal değildir:

```
>>> b = "123445b"
```

Hemen kontrol edelim:

```
>>> b.isdigit()

False
```

isalnum()

Bu metot, bir karakter dizisinin ‘alfanümerik’ olup olmadığını denetlememizi sağlar. Peki ‘alfanümerik’ nedir?

Daha önce bahsettiğimiz metotlardan hatırlayacaksınız:

Alfabetik karakter dizileri, alfabe harflerinden oluşan karakter dizileridir.

Sayısal karakter dizileri, sayılardan oluşan karakter dizileridir.

Alfanümerik karakter dizileri ise bunun birleşimidir. Yani sayı ve harflerden oluşan karakter dizilerine alfanümerik karakter dizileri adı verilir. Örneğin şu karakter dizisi alfanümerik bir karakter dizisidir:

```
>>> a = "123abc"
```

İsterseniz hemen bu yeni metodumuz yardımıyla bunu doğrulayalım:

```
>>> a.isalnum()

True
```

Eğer denetleme sonucunda *True* alıyorsak, o karakter dizisi alfanümeriktir. Bir de şuna bakalım:

```
>>> b = "123abc>"
```

```
>>> b.isalnum()

False
```

b değişkeninin tuttuğu karakter dizisinde alfanümerik karakterlerin yanısıra (*"123abc"*), alfanümerik olmayan bir karakter dizisi de bulunduğu için (*">"*), b.isalnum() şeklinde gösterdiğimiz denetlemenin sonucu *False* (yanlış) olarak görünecektir.

Dolayısıyla, bir karakter dizisi içinde en az bir adet alfanümerik olmayan bir karakter dizisi bulunursa (bizim örneğimizde "<"), o karakter dizisi alfanümerik olmayacaktır.

isdecimal()

Bu metot yardımıyla bir karakter dizisinin ondalık sayı cinsinden olup olmadığını denetliyoruz. Mesela aşağıdaki örnek ondalık sayı cinsinden bir karakter dizisidir:

```
>>> a = "123"
>>> a.isdecimal()

True
```

Ama şu ise kayan noktalı (*floating-point*) sayı cinsinden bir karakter dizisidir:

```
>>> a = "123.3"
>>> a.isdecimal()

False
```

Dolayısıyla a.isdecimal() komutu *False* çıktısı verir...

isidentifier()

Identifier kelimesi Türkçede 'tanımlayıcı' anlamına gelir. Python'da değişkenler, fonksiyon ve modül adlarına 'tanımlayıcı' denir. İşte başlıkta gördüğümüz isidentifier() metodu, neyin tanımlayıcı olup neyin tanımlayıcı olamayacağını denetlememizi sağlar. Hatırlarsanız değişkenler konusundan bahsederken, değişken adı belirlemenin bazı kuralları olduğunu söylemiştik. Buna göre, örneğin, değişken adları bir sayı ile başlayamıyordu. Dolayısıyla şöyle bir değişken adı belirleyemiyoruz:

```
>>> 1a = 12
```

Dediğimiz gibi, değişkenler birer tanımlayıcıdır. Dolayısıyla bir değişken adının geçerli olup olmadığını isidentifier() metodu yardımıyla denetleyebiliriz:

```
>>> "1a".isidentifier()
```

```
False
```

Demek ki "1a" ifadesini herhangi bir tanımlayıcı adı olarak kullanamıyoruz. Yani bu ada sahip bir değişken, fonksiyon adı veya modül adı oluşturamıyoruz. Ama mesela "liste1" ifadesi geçerli bir tanımlayıcıdır. Hemen denetleyelim:

```
>>> "liste1".isidentifier()

True
```

isnumeric()

Bu metot bir karakter dizisinin nümerik olup olmadığını denetler. Yani bu metot yardımıyla bir karakter dizisinin sayı değerli olup olmadığını denetleyebiliriz:

```
>>> "12".isnumeric()

True

>>> "dasd".isnumeric()

False
```

isspace()

Bu metot yardımıyla bir karakter dizisinin tamamen boşluklardan oluşup oluşmadığını denetleyebiliriz. Eğer karakter dizimiz boşluklardan oluşuyorsa bu metot *True* çıktısı verecek, ama eğer karakter dizimizin içinde bir tane bile boşluk harici karakter varsa bu metot *False* çıktısı verecektir:

```
>>> a = " "
>>> a.isspace()

True

>>> a = "          "
>>> a.isspace()

True

>>> a = "" #karakter dizimiz tamamen boş. İçinde boşluk karakteri bile yok...
>>> a.isspace()
```

```
False

>>> a = "fd"
>>> a.isspace()

False
```

isprintable()

Hatırlarsanız önceki derslerimizde \n, \t, \r ve buna benzer karakterlerden söz etmiştik. Örneğin \n karakterinin 'satır başı' anlamına geldiğini ve bu karakterin görevinin karakter dizisini bir alt satıra almak olduğunu söylemiştik. Örnek verelim:

```
>>> print("birinci satır\nikinci satır")

birinci satır
ikinci satır
```

Bu örnekte \n karakterinin öteki karakterlerden farklı olduğunu görüyorsunuz. Mesela *"b"* karakteri komut çıktısında görünüyor. Ama \n karakteri çıktıda görünmüyor. \n karakteri elbette yukarıdaki kodlar içinde belli bir işleve sahip. Ancak karakter dizisindeki öteki karakterlerden farklı olarak \n karakteri ekranda görünmüyor. İşte Python'da bunun gibi, ekranda görünmeyen karakterlere 'basılmayan karakterler' (*non-printing characters*) adı verilir. 'b', 'c', 'z', 'x', '=', '?', '!' ve benzeri karakterler ise 'basılabilen karakterler' (*printable characters*) olarak adlandırılır. İşte başlıkta gördüğünüz isprintable() metodu da karakterlerin bu yönünü sorgular. Yani bir karakterin basılabilen bir karakter mi yoksa basılmayan bir karakter mi olduğunu söyler bize. Örneğin:

```
>>> karakter = "a"
>>> karakter.isprintable()

True
```

Demek ki *"a"* karakteri basılabilen bir karaktermiş. Bir de şuna bakalım:

```
>>> karakter = "\n"
>>> karakter.isprintable()

False
```

Demek ki \n karakteri gerçekten de basılamayan bir karaktermiş.

Basılamayan karakterlerin listesini görmek için http://www.asciitable.com/ adresini ziyaret edebilirsiniz. Listedeki ilk *33* karakter (*0*'dan başlayarak *33*'e kadar olan karakterler) ve listedeki

127. karakter basılamayan karakterlerdir.

başa dön

Karakter Dizilerini Biçimlendirmek

Bu bölüme gelinceye kadar, Python'da karakter dizilerinin biçimlendirilmesine ilişkin epey söz söyledik. Ancak bu konu ile ilgili bilgilerimiz hem çok dağınık, hem de çok yüzeysel. İşte bu bölümde amacımız, daha önce farklı yerlerde dile getirdiğimiz bu önemli konuya ait bilgi kırıntılarını bir araya toplayıp, karakter dizisi biçimlendirme konusunu, Python bilgimiz elverdiği ölçüde ayrıntılı bir şekilde ele almak olacak.

Şu ana kadar yaptığımız örneklere bakarak, programlama maceranız boyunca karakter dizileriyle bol bol haşır neşir olacağınızı anlamış olmalısınız. Bundan sonra yazdığınız programlarda da karakter dizilerinin size pek çok farklı biçimlerde geldiğine tanık olacaksınız. Farklı farklı biçimlerde elinize ulaşan bu karakter dizilerini, muhtemelen, sadece alt alta ve rastgele bir şekilde ekrana yazdırmakla yetinmeyeceksiniz. Bu karakter dizilerini, yazdığınız programlarda kullanabilmek için, programınıza uygun şekillerde biçimlendirmeniz gerekecek. Dilerseniz neden bahsettiğimizi daha net bir şekilde anlatabilmek için çok basit bir örnek verelim.

Diyelim ki, yazdığınız bir programda kullanmak üzere, kullanıcıdan isim bilgisi almanız gerekiyor. Programınızın işleyişi gereğince, eğer isim *5* karakterden küçükse ismin tamamı görüntülenecek, ama eğer isim *5* karakterden büyükse *5* karakteri aşan kısım yerine üç nokta işareti koyulacak. Yani eğer isim *Fırat* ise bu ismin tamamı görüntülenecek. Ama eğer isim mesela *Abdullah* ise, o zaman bu isim *Abdul...* şeklinde görüntülenecek.

Bu amaca ulaşmak için ilk denememizi yapalım:

```
isim = input("isminiz: ")

if len(isim) <= 5:
    print(isim[:5])
else:
    print(isim[:5], "...")
```

Buradan elde ettiğimiz çıktı ihtiyacımızı kısmen karşılıyor. Ama çıktı tam istediğimiz gibi değil. Çünkü normalde isme bitişik olması gereken üç nokta işareti, isimden bir boşluk ile ayrılmış. Yani biz şöyle bir çıktı isterken:

```
Abdul...
```

Şöyle bir çıktı elde ediyoruz:

```
Abdul ...
```

Bu sorunu şu şekilde halledebiliriz:

```
isim = input("isminiz: ")
```

```
if len(isim) <= 5:
    print(isim[:5])
else:
    print(isim[:5] + "...")
```

veya:

```
isim = input("isminiz: ")

if len(isim) <= 5:
    print(isim[:5])
else:
    print(isim[:5], "...", sep="")
```

Yukarıdaki gibi basit durumlarda klasik karakter dizisi birleştirme yöntemlerini kullanarak işinizi halledebilirsiniz. Ama daha karmaşık durumlarda, farklı kaynaklardan gelen karakter dizilerini ihtiyaçlarınıza göre bir araya getirmek, karakter dizisi birleştirme yöntemleri ile pek mümkün olmayacak veya çok zor olacaktır.

Mesela şöyle bir durum düşünün:

Yazdığınız programda kullanıcıya bir parola soruyorsunuz. Amacınız bu parolanın, programınızda belirlediğiniz ölçütlere uyup uymadığını tespit etmek. Eğer kullanıcı tarafından belirlenen parola uygunsa ona şu çıktıyı göstermek istiyorsunuz (parolanın *b5tY6g* olduğunu varsayalım):

Girdiğiniz parola (b5tY6g) kurallara uygun bir paroladır!

Bu çıktıyı elde etmek için şöyle bir kod yazabilirsiniz:

```
parola = input("parola: ")

print("Girdiğiniz parola (" + parola + ") kurallara uygun bir paroladır!")
```

Gördüğünüz gibi, sadece karakter dizisi birleştirme yöntemlerini kullanarak istediğimiz çıktıyı elde ettik, ama farkettiyseniz bu defa işler biraz da olsa zorlaştı.

Bir de uzun ve karmaşık bir metnin içine dışarıdan değerler yerleştirmeniz gereken şöyle bir metinle karşı karşıya olduğunuzu düşünün:

```
Sayın .....

.... tarihinde yapmış olduğunuz, ..... hakkındaki başvurunuz incelemeye alınmıştır.

Size .... işgünü içinde cevap verilecektir.
```

```
Saygılarımızla,

.....
```

Böyle bir metin içine dışarıdan değer yerleştirmek için karakter dizisi birleştirme yöntemlerine başvurmak işinizi epey zorlaştıracaktır.

İşte klasik karakter dizisi birleştirme işlemlerinin yetersiz kaldığı veya işleri büsbütün zorlaştırdığı bu tür durumlarda Python'ın size sunduğu 'karakter dizisi biçimlendirme' araçlarından yararlanabilirsiniz.

Bunun için biz bu bölümde iki farklı yöntemden söz edeceğiz:

1. % işareti ile biçimlendirme
2. format() metodu ile biçimlendirme.

% işareti ile biçimlendirme, karakter dizisi biçimlendirmenin eski yöntemidir. Bu yöntem ağırlıklı olarak Python'ın 3.x sürümlerinden önce kullanılıyordu. Ama Python'ın 3.x sürümlerinde de bu yöntemi kullanma imkanımız var. Her ne kadar bu yöntem Python3'te geçerliliğini korusa da muhtemelen ileride dilden tamamen kaldırılacak. Ancak hem etrafta bu yöntemle yazılmış eski programlar olması, hem de bu yöntemin halen geçerliliğini koruması nedeniyle bu yöntemi (kendimiz kullanmayacak bile olsak) mutlaka öğrenmemiz gerekiyor.

format() metodu ise Python'ın 3.x sürümleri ile dile dahil olan bir özelliktir. Python'ın 2.x sürümlerinde bu metodu kullanamazsınız. Dilin geleceğinde bu metot olduğu için, yeni yazılan kodlarda format() metodunu kullanmak daha akıllıca olacaktır.

Biz bu sayfalarda yukarıda adını andığımız her iki yöntemi de inceleyeceğiz. İlk olarak % işareti ile biçimlendirmeden söz edelim.

% İşareti ile Biçimlendirme (Eski Yöntem)

Daha önce de söylediğimiz gibi, Python programlama dilinin 3.x sürümlerinden önce, bir karakter dizisini biçimlendirebilmek için % işaretinden yararlanıyorduk. Bununla ilgili basit bir örnek verelim:

```
parola = input("parola: ")

print("Girdiğiniz parola (%s) kurallara uygun bir paroladır!" %parola)
```

Bu programı çalıştırıp parola girdiğinizde, yazdığınız parola çıktıda parantez içinde görünecektir.

Yukarıdaki yapıyı incelediğimizde iki nokta gözümüze çarpıyor:

1. İlk olarak, karakter dizisinin içinde bir *%* işareti ve buna bitişik olarak yazılmış bir *s* harfi görüyoruz.
2. İkincisi, karakter dizisinin dışında *%parola* gibi bir ifade daha var.

Rahatlıkla tahmin edebileceğiniz gibi, bu ifadeler birbiriyle doğrudan bağlantılıdır. Dilerseniz bu yapıyı açıklamaya geçmeden önce bir örnek daha verelim. Bu örnek sayesinde benim açıklamama gerek kalmadan karakter dizisi biçimlendirme mantığını derhal kavrayacağınızı zannediyorum:

```
print("%s ve %s iyi bir ikilidir!" %("Python", "Django"))
```

Dediğim gibi, bu basit örnek karakter dizilerinin nasıl biçimlendirildiğini gayet açık bir şekilde gösteriyor. Dilerseniz yapıyı şöyle bir inceleyelim:

1. Python'da *%s* yapısı, karakter dizisi içinde bir yer tutma vazifesi görür.

2. *%s* yapısı bir anlamda değişkenlere benzer. Tıpkı değişkenlerde olduğu gibi, *%s* yapısının değeri değişebilir.

3. Bir karakter dizisi içindeki her *%s* ifadesi için, karakter dizisi dışında bu ifadeye karşılık gelen bir değer olmalıdır. Python, karakter dizisi içinde geçen her *%s* ifadesinin yerine, karakter dizisi dışındaki her bir değeri tek tek yerleştirir. Bizim örneğimizde karakter dizisi içindeki ilk *%s* ifadesinin karakter dizisi dışındaki karşılığı *"Python"*; karakter dizisi içindeki ikinci *%s* ifadesinin karakter dizisi dışındaki karşılığı ise *"Django"*'dur.

4. Eğer karakter dizisi içindeki *%s* işaretlerinin sayısı ile karakter dizisi dışında bu işaretlere karşılık gelen değerlerin sayısı birbirini tutmazsa Python bize bir hata mesajı gösterecektir. Mesela:

```
>>> print("Benim adım %s, soyadım %s" %"istihza")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: not enough arguments for format string
```

Gördüğünüz gibi bu kodlar hata verdi. Çünkü karakter dizisi içindeki iki adet *%s* ifadesine karşılık, karakter dizisinin dışında tek bir değer var (*"istihza"*). Halbuki bizim şöyle bir kod yazmamız gerekiyordu:

```
>>> isim = "istihza"
>>> print("%s adlı kişinin mekanı www.%s.com adresidir." %(isim, isim))
```

Bu defa herhangi bir hata mesajı almadık. Çünkü bu kodlarda, olması gerektiği gibi, karakter dizisi içindeki iki adet *%s* ifadesine karşılık, dışarıda da iki adet değer var.

Eğer karakter dizisi içinde tek bir *%s* ifadesi varsa, karakter dizisi dışında buna karşılık gelen değeri gösterirken, bu değeri parantez içine almamıza gerek yok. Ama eğer karakter dizisi içinde birden

fazla *%s* işareti varsa, bunlara karşılık gelen değerleri parantez içinde gösteriyoruz. Mesela yukarıdaki parola örneğinde, karakter dizisinin içinde tek bir *%s* ifadesi var. Dolayısıyla karakter dizisi dışında bu ifadeye karşılık gelen *parola* değişkenini parantez içine almıyoruz. Ama *"Python"* ve *"Django"* örneğinde karakter dizisi içinde iki adet *%s* ifadesi yer aldığı için, karakter dizisi dışında bu ifadelere karşılık gelen *"Python"* ve *"Django"* kelimelerini parantez içinde gösteriyoruz.

Bütün bu anlattıklarımızı sindirebilmek için dilerseniz bir örnek verelim:

```
kardiz = "istihza"

for sıra, karakter in enumerate(kardiz, 1):
    print("%s. karakter: '%s'" %(sıra, karakter))
```

Gördüğünüz gibi, *"istihza"* adlı karakter dizisi içindeki her bir harfin sırasını ve harfin kendisini uygun bir düzen içinde ekrana yazdırdık. karakter sırasının ve karakterin kendisinin cümle içinde geleceği yerleri *%s* işaretleri ile gösteriyoruz. Python da her bir değeri, ilgili konumlara tek tek yerleştiriyor.

Hatırlarsanız önceki derslerimizde basit bir hesap makinesi örneği vermiştik. İşte şimdi öğrendiklerimizi o programa uygularsak karakter dizisi biçimlendiricileri üzerine epey pratik yapmış oluruz:

```
giriş = """
    (1) topla
    (2) çıkar
    (3) çarp
    (4) böl
    (5) karesini hesapla
    (6) karekök hesapla
    """
print(giriş)

a = 1

while a == 1:
    soru = input("Yapmak istediğiniz işlemin numarasını girin (Çıkmak için q): ")

    if soru == "q":
        print("çıkılıyor...")
        a = 0

    elif soru == "1":
        sayı1 = int(input("Toplama işlemi için ilk sayıyı girin: "))
        sayı2 = int(input("Toplama işlemi için ikinci sayıyı girin: "))
```

```
    #İlk %s'ye karşılık gelen değer   : sayı1
    #İkinci %s'ye karşılık gelen değer: sayı2
    #Üçüncü %s'ye karşılık gelen değer: sayı1 + sayı2
    print("%s + %s = %s" %(sayı1, sayı2, sayı1 + sayı2))

elif soru == "2":
    sayı3 = int(input("Çıkarma işlemi için ilk sayıyı girin: "))
    sayı4 = int(input("Çıkarma işlemi için ikinci sayıyı girin: "))
    print("%s - %s = %s" %(sayı3, sayı4, sayı3 - sayı4))

elif soru == "3":
    sayı5 = int(input("Çarpma işlemi için ilk sayıyı girin: "))
    sayı6 = int(input("Çarpma işlemi için ikinci sayıyı girin: "))
    print("%s x %s = %s" %(sayı5, sayı6, sayı5 * sayı6))

elif soru == "4":
    sayı7 = int(input("Bölme işlemi için ilk sayıyı girin: "))
    sayı8 = int(input("Bölme işlemi için ikinci sayıyı girin: "))
    print("%s / %s = %s" %(sayı7, sayı8, sayı7 / sayı8))

elif soru == "5":
    sayı9 = int(input("Karesini hesaplamak istediğiniz sayıyı girin: "))

    #İlk %s'ye karşılık gelen değer   : sayı9
    #İkinci %s'ye karşılık gelen değer: sayı9 ** 2
    print("%s sayısının karesi = %s" %(sayı9, sayı9 ** 2))

elif soru == "6":
    sayı10 = int(input("Karekökünü hesaplamak istediğiniz sayıyı girin: "))
    print("%s sayısının karekökü = %s" %(sayı10, sayı10 ** 0.5))

else:
    print("Yanlış giriş.")
    print("Aşağıdaki seçeneklerden birini giriniz:", giriş)
```

Bu arada, gördüğünüz gibi, Python'da biçim düzenleyici olarak kullanılan simge aynı zamanda 'yüzde' (*%*) anlamına da geliyor. O halde size şöyle bir soru sorayım: Acaba *0*'dan *100*'e kadar olan sayıların başına birer yüzde işareti koyarak bu sayıları nasıl gösterirsiniz? *%0*, *%1*, *%10*, *%15*, gibi... Önce şöyle bir şey deneyelim:

```
>>> for i in range(100):
...     print("%s" %i)
...
```

Bu kodlar tabii ki sadece *0*'dan *100*'e kadar olan sayıları ekrana dökmekle yetinecektir. Sayıların başında *%* işaretini göremeyeceğiz.

Bir de şöyle bir şey deneyelim:

```
>>> for i in range(100):
...     print("%%s" %i)
...
Traceback (most recent call last):
   File "<stdin>", line 2, in <module>
TypeError: not all arguments converted during string formatting
```

Bu defa da hata mesajı aldık. Halbuki doğru cevap şu olmalıydı:

```
>>> for i in range(100):
...     print("%%%s" %i)
...
```

Burada *%* işaretini arka arkaya iki kez kullanarak bir adet *%* işareti elde ettik. Daha sonra da normal bir şekilde *%s* biçimini kullandık. Yani üç adet '%' işaretini yan yana getirmiş olduk.

Bütün bu örneklerden sonra, karakter dizisi biçimlendiricilerinin işimizi ne kadar kolaylaştırdığını görmüş olmalısınız. İstediğimiz etkiyi elde etmek için karakter dizisi biçimlendiricilerini kullanmak, karakter dizilerini birleştirme işlemlerinden yararlanmaya göre çok daha esnek bir yöntemdir. Hatta bazı durumlarda karakter dizisi biçimlendiricilerini kullanmak makul tek yöntemdir.

Yukarıda verdiğimiz örnekler, *%s* ile biçimlendirme konusunun en temel yönlerini gösteriyor. Ama aslında bu aracı kullanarak çok daha karmaşık biçimlendirme işlemleri de yapabiliriz.

Yani yukarıdaki örneklerde *%s* yapısını en basit şekilde mesela şöyle kullandık:

```
>>> print("Karakter dizilerinin toplam %s adet metodu vardır" %len(dir(str)))
```

Ama eğer istersek bundan daha karmaşık biçimlendirme işlemleri de gerçekleştirebiliriz. Şu örneğe bakın:

```
>>> for i in dir(str):
...     print("%15s" %i)
```

Gördüğünüz gibi *%* ile *s* işaretleri arasına bir sayı yerleştirdik. Bu sayı, biçimlendirilecek karakter dizisinin toplam kaç karakterlik yer kaplayacağını gösteriyor. Durumu daha net görebilmeniz için şöyle bir örnek verelim:

```
>>> print("|%15s|" %"istihza")

|        istihza|
```

Karakter dizisinin başına ve sonuna eklediğimiz '|' işaretleri sayesinde karakter dizisinin nasıl ve ne şekilde hizalandığını daha belirgin bir şekilde görebiliyoruz. Aslında yukarıdaki örneğin yaptığı iş size hiç yabancı değil. Aynı etkiyi, karakter dizisi metotlarından rjust() ile de yapabileceğimizi biliyorsunuz:

```
>>> print("istihza".rjust(15))
```

Aynen yukarıdaki çıktıyı rjust() metodunu kullanarak elde etmek için ise şöyle bir şey yazabilirsiniz:

```
>>> print("|%s|" %"istihza".rjust(15))

|       istihza|
```

Yukarıdaki örnekte *"istihza"* karakter dizisini sağa doğru yasladık. Sola yaslamak için ise negatif sayılardan yararlanabilirsiniz:

```
>>> print("|%-15s|" %"istihza")

|istihza        |
```

Tıpkı biraz önce verdiğimiz örnekteki gibi, aynı etkiyi ljust() metoduyla da elde edebilirsiniz:

```
>>> print("|%s|" %"istihza".ljust(15))

|istihza        |
```

Gördüğünüz gibi, *%s* yapısını farklı şekillerde kullanarak epey karmaşık çıktılar elde edebiliyoruz. Ama aslında karakter dizisi biçimlendiricilerini kullanarak yapabileceklerimiz bunlarla da sınırlı değildir. Mesela size şöyle bir soru sorduğumu düşünün: Acaba aşağıdaki içeriğe sahip bir *HTML* şablonunu nasıl elde edebiliriz?

```
<html>
   <head>
      <title> {{ sayfa başlığı }} </title>
   </head>

   <body>
      <h1> {{ birinci seviye başlık }} </h1>
      <p>Web sitemize hoşgeldiniz! Konumuz: {{ konu }}</p>
   </body>
</html>
```

Burada bütün değişkenler tek bir değere sahip olacak. Örneğin değişkenimiz *Python Programlama Dili* ise yukarıdaki şablon şöyle bir *HTML* sayfası üretecek:

```
<html>
   <head>
      <title> Python Programlama Dili </title>
   </head>

   <body>
      <h1> Python Programlama Dili </h1>
      <p>Web sitemize hoşgeldiniz! Konumuz: Python Programlama Dili</p>
   </body>
</html>
```

Aklınıza ilk olarak şöyle bir çözüm gelmiş olabilir:

```
sayfa = """
<html>
   <head>
      <title> %s </title>
   </head>

   <body>
      <h1> %s </h1>
      <p>Web sitemize hoşgeldiniz! Konumuz: %s</p>
   </body>
</html>
"""

print(sayfa % ("Python Programlama Dili",
            "Python Programlama Dili",
            "Python Programlama Dili"))
```

Bu gayet makul ve doğru bir çözümdür. Ancak gördüğünüz gibi yukarıdaki kodlarda bizi rahatsız eden bir nokta var. Bu kodlarda aynı karakter dizisini (*"Python Programlama Dili"*) üç kez tekrar ediyoruz. En baştan beri söylediğimiz gibi, kod yazarken tekrarlardan olabildiğince kaçınmaya çalışmamız programımızın performansını artıracaktır. Burada da tekrardan kaçınmak amacıyla şöyle bir kod yazmayı tercih edebiliriz. Dikkatlice inceleyin:

```
sayfa = """
<html>
   <head>
      <title> %(dil)s </title>
   </head>

   <body>
      <h1> %(dil)s </h1>
```

```
    <p>Web sitemize hoşgeldiniz! Konumuz: %(dil)s</p>
  </body>
</html>
"""

print(sayfa % {"dil": "Python Programlama Dili"})
```

Gördüğünüz gibi, yukarıdaki kodlar bizi aynı karakter dizisini tekrar tekrar yazma zahmetinden kurtardı. Peki ama nasıl? Gelin isterseniz bu yapıyı daha iyi anlayabilmek için daha basit bir örnek verelim:

```
print("depoda %(miktar)s kilo %(ürün)s kaldı" %{"miktar": 25,
                                                "ürün": "elma"})
```

Burada şöyle bir yapıyla karşı karşıyayız:

```
"%(değişken_adı)s" % {"değişken_adı": "değişken_değeri"})
```

{"değişken_adı": "değişken_değeri"} yapısıyla önceki derslerimizde karşılaşmıştınız. Dolayısıyla bu yapının temel olarak ne işe yaradığını biliyorsunuz. Hatta bu yapının adının 'sözlük' olduğunu da öğrenmiştiniz. İşte burada, sözlük adlı veri tipinden yararlanarak değişken adları ile değişken değerlerini eşleştirdik. Böylece aynı şeyleri tekrar tekrar yazmamıza gerek kalmadı. Ayrıca yukarıdaki örnekte değerleri sırasına göre değil, ismine göre çağırdığımız için, karakter dizisi içindeki değerlerin sırasını takip etme zahmetinden de kurtulmuş olduk.

Böylece % yapısının tüm temel ayrıntılarını öğrenmiş olduk. Artık % işaretinin başka yönlerini incelemeye başlayabiliriz.

Biçimlendirme Karakterleri

Biraz önce, Python'da eski usul karakter dizisi biçimlendirme yöntemi olan % işareti üzerine en temel bilgileri edindik. Buraya kadar öğrendiklerimiz, yazdığımız programlarda genellikle yolumuzu yordamımızı bulmamıza yetecektir. Ama isterseniz şimdi karakter dizisi biçimlendirme konusunu biraz daha derinlemesine ele alalım. Mesela Python'daki biçimlendirme karakterlerinin neler olduğunu inceleyelim.

s

Önceki örneklerden de gördüğünüz gibi, Python'da biçim düzenleme işlemleri için *%s* adlı bir yapıdan faydalanıyoruz. Bu yapıyı şöyle bir masaya yatırdığımızda aslında bu yapının iki parçadan oluştuğunu görebiliriz. Bu parçalar % ve *s* karakterleridir. Burada gördüğümüz parçalardan % sabit, *s* ise değişkendir. Yani % sabit değerini bazı harflerle birlikte kullanarak, farklı karakter dizisi biçimlendirme işlemleri gerçekleştirebiliriz.

Biz önceki sayfalarda verdiğimiz örneklerde bu simgeyi *s* harfiyle birlikte kullandık. Örneğin:

```
>>> print("Benim adım %s" %"istihza")
```

Bu kodlardaki *s* karakteri İngilizce *string*, yani 'karakter dizisi' ifadesinin kısaltmasıdır. Esasında en yaygın çift de budur. Yani etraftaki Python programlarında yaygın olarak *%s* yapısını görürüz. Ancak Python'da *%* biçim düzenleyicisiyle birlikte kullanılabilecek tek karakter *s* değildir. Daha önce de dediğimiz gibi, *s* karakteri *string*, yani 'karakter dizisi' ifadesinin kısaltmasıdır. Yani aslında *%s* yapısı Python'da özel olarak karakter dizilerini temsil eder.

Peki bu ne demek oluyor?

Bir karakter dizisi içinde *%s* yapısını kullandığımızda, dışarıda buna karşılık gelen değerin de bir karakter dizisi veya karakter dizisine çevrilebilecek bir değer olması gerekir. Python'da her şey bir karakter dizisi olarak temsil edilebilir. Dolayısıyla bütün işlemlerinizde *%* işaretini *s* karakteri ile birlikte kullanabilirsiniz. Ama bazı özel durumlarda *%* işaretini *s* dışında başka harflerle birlikte kullanmanız da gerekebilir.

Biz *%* yapısı ile ilgili verdiğimiz ilk örneklerde bu yapının *s* karakteri ile birlikte kullanılışını gösteren pek çok örnek verdiğimiz için *%* ile *s* birlikteliği üzerinde daha fazla durmayacağız. Bunun yerine, *%* ile birlikte kullanılan öteki karakterleri inceleyeceğiz. O halde yola koyulalım.

d

Bir önceki başlıkta gördüğümüz *s* harfi nasıl karakter dizilerini temsil ediyorsa, *d* harfi de sayıları temsil eder. İsterseniz küçük bir örnekle açıklamaya çalışalım durumu:

```
>>> print("Şubat ayı bu yıl %d gün çekiyor" %28)

Şubat ayı bu yıl 28 gün çekiyor.
```

Gördüğünüz gibi, *%* işaretiyle birlikte bu defa *s* yerine *d* harfini kullandık. Buna uygun olarak da dış tarafta *28* sayısını kullandık. Peki yukarıdaki ifadeyi şöyle de yazamaz mıydık?

```
>>> print("Şubat ayı bu yıl %s gün çekiyor" %28)
```

Elbette yazabilirdik. Bu kod da bize doğru çıktı verecektir. Çünkü daha önce de dediğimiz gibi, *s* harfi karakter dizilerini ve karakter dizisine çevrilebilen değerleri temsil eder. Python'da sayılar karakter dizisine çevrilebildiği için *%s* gibi bir yapıyı hata almadan kullanabiliyoruz. Ama mesela şöyle bir şey yazamayız:

```
>>> print("Şubat ayı bu yıl %d gün çekiyor" %"yirmi sekiz")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: %d format: a number is required, not str
```

Gördüğünüz gibi bu defa hata aldık. Çünkü *d* harfi yalnızca sayı değerleri temsil edebilir. Bu harfle

birlikte karakter dizilerini kullanamayız.

Doğrusunu söylemek gerekirse, *d* harfi aslında tam sayı (*integer*) değerleri temsil eder. Eğer bu harfin kullanıldığı bir karakter dizisinde değer olarak mesela bir kayan noktalı sayı (*float*) verirsek, bu değer tamsayıya çevrilecektir. Bunun ne demek olduğunu hemen bir örnekle görelim:

```
>>> print("%d" %13.5)

13
```

Gördüğünüz gibi, *%d* ifadesi, *13.5* sayısının ondalık kısmını çıktıda göstermiyor. Çünkü *d* harfi sadece tamsayıları temsil etme işlevi görüyor.

Burada şöyle bir soru aklınıza gelmiş olabilir: 'Acaba *%d* ifadesi ile hiç uğraşmasak, bunun yerine her yerde *%s* ifadesini kullansak olmaz mı?'.

Çoğu zaman olur, ama mesela şöyle bir durum düşünün: Yazdığınız programda kullanıcıdan sadece tam sayı girmesini istiyor olabilirsiniz. Yani mesela kullanıcının ondalık sayı girmesi halinde, siz bu sayının sadece tam sayı kısmını almak istiyor olabilirsiniz. Örneğin kullanıcı *23.8* gibi bir sayı girmişse, siz bu sayıda ihtiyacınız olan *23* kısmını almak isteyebilirsiniz. İşte bu *%d* işaretinden yararlanarak, kullanıcının girdiği ondalık sayının sadece tam sayı kısmını çekebilirsiniz:

```
sayı = input("sayı: ")

print("%d" %float(sayı))
```

Elbette Python'da bir ondalık sayının sadece taban kısmını almanın başka yöntemleri de vardır. Ama yukarıda verdiğimiz örnek bir ondalık sayının sadece tabanını almanın gayet basit ve etkili bir yoludur.

%s yapısını anlatırken gösterdiğimiz ileri düzey biçimlendirme tekniklerini *%d* ile de kullanabilirsiniz. Örneğin:

```
>>> print("|%7d|" %23)

|     23|

>>> print("|%-7d|" %23)

|23     |
```

veya:

```
>>> print("%(sayı)d" % {"sayı": 23})

23
```

%s yapısına ek olarak, sayının kaplayacağı alandaki boşluklara birer *0* da yerleştirebilirsiniz:

```
>>> print("%05d" %23)

00023
```

...veya:

```
>>> print("%.5d" %23)

00023
```

Hatta hem sayının kaplayacağı boşluk miktarını hem de bu boşlukların ne kadarının *0* ile doldurulacağını da belirleyebilirsiniz:

```
>>> print("%10.5d" %23)

     00023
```

Burada *23* sayısının toplam *10* boşlukluk bir yer kaplamasını ve bu *10* adet boşluğun *5* tanesinin içine *0* sayılarının ve *23* sayısının sığdırılmasını istedik.

Bir de şuna bakalım:

```
>>> print("%010.d" %23)

0000000023
```

Burada ise *23* sayısının toplam *10* boşlukluk bir yer kaplamasını ve bu *10* adet boşluğa *23* sayısı yerleştirildikten sonra arta kalan kısmın *0* sayıları ile doldurulmasını istedik.

Bu arada, son örnekte yaptığımız şeyi, daha önce öğrendiğimiz zfill() metoduyla da yapabileceğimizi biliyorsunuz:

```
>>> "23".zfill(10)

'0000000023'
```

Yukarıdaki kullanımlar ilk bakışta gözünüze karışık görünmüş olabilir. Ama eğer yeterince pratik yaparsanız, aslında bu biçimlerin hiç de o kadar karmaşık olmadığını anlarsınız. İsterseniz bu biçimlerle neler yapabileceğimizi şöyle bir kısaca tarif edelim:

d harfi, *%* işaretiyle birlikte kullanıldığında sayıları temsil eder. Bu iki karakterin en temel kullanımı şöyledir:

```
>>> "%d" %10
```

```
'10'
```

d harfi ile *%* işareti arasına bir pozitif veya negatif sayı getirerek, temsil edilecek sayının toplam kaç boşluktan oluşan bir alan içine yerleştirileceğini belirleyebiliyoruz:

```
>>> "%5d" %10

'   10'
```

Burada *10* sayısını toplam *5* boşlukluk bir alan içine yerleştirdik. Gördüğünüz gibi, bir pozitif sayı kullandığımızda, sayımız kendisine ayrılan alan içinde sağa yaslanıyor. Eğer bu sayıyı sola yaslamak istersek negatif sayılardan yararlanabiliriz:

```
>>> "%-5d" %10

'10   '
```

Eğer sağa yasladığımız bir sayının sol tarafını sıfırla doldurmak istersek, hizalama miktarını belirtmek için kullandığımız sayının soluna bir sıfır ekleyebiliriz:

```
>>> "%05d" %10

'00010'
```

Aynı etkiyi şu şekilde de elde edebilirsiniz:

```
>>> "%.5d" %10

'00010'
```

Eğer nokta işaretinden önce bir sayı belirtirseniz, karakter dizisi o belirttiğiniz sayı kadar sağa yaslanacaktır. Yani:

```
>>> "%10.5d" %10

'    00010'
```

... veya sola:

```
>>> "%-10.5d" %10

'00010    '
```

Her iki şekilde de, karakter dizisini toplam *10* boşluktan oluşan bir alan içine yerleştirmiş olduk. Bu toplam alanın *5* boşlukluk kısmı sayının kendisi ve sayının soluna gelecek *0*'lar arasında paylaştırıldı.

Gördüğünüz gibi, biçimlendirme mantığının aslında o kadar da korkulacak bir yanı yok. Kendi kendinize yukarıdakilere benzer örnekler yaparak bu yapıyı daha iyi bir şekilde anlamaya çalışabilirsiniz.

i

Bu harf de *integer*, yani 'tam sayı' kelimesinin kısaltmasıdır. Kullanım ve işlev olarak, *d* harfinden hiç bir farkı yoktur.

o

Bu harf *octal* (sekizli) kelimesinin kısaltmasıdır. Adından da anlaşılacağı gibi, sekizli düzendeki sayıları temsil eder. Dolayısıyla bu harfi kullanarak onlu düzendeki bir sayıyı sekizli düzendeki karşılığına dönüştürebilirsiniz. Örneğin:

```
>>> print("%i sayısının sekizli düzendeki karşılığı %o sayısıdır." %(10, 10))

10 sayısının sekizli düzendeki karşılığı 12 sayısıdır.
```

Not

%d yapısını anlatırken gösterdiğimiz ileri düzey biçimlendirme tekniklerinin tamamını *%o* ile de kullanabilirsiniz.

x

Bu harf *hexadecimal*, yani onaltılı düzendeki sayıları temsil eder. Dolayısıyla bu harfi kullanarak onlu düzendeki bir sayıyı onaltılı düzendeki karşılığına çevirebilirsiniz:

```
>>> print("%i sayısının onaltılı düzendeki karşılığı %x sayısıdır." %(20, 20))

20 sayısının onaltılı düzendeki karşılığı 14 sayısıdır.
```

Buradaki 'x' küçük harf olarak kullanıldığında, onaltılı düzende harfle gösterilen sayılar da küçük harfle temsil edilecektir:

```
>>> print("%i sayısının onaltılı düzendeki karşılığı %x sayısıdır." %(10, 10))

10 sayısının onaltılı düzendeki karşılığı a sayısıdır.
```

Not

%d yapısını anlatırken gösterdiğimiz ileri düzey biçimlendirme tekniklerinin tamamını *%x* ile de

kullanabilirsiniz.

X

Bu da tıpkı *x* harfinde olduğu gibi, onaltılı düzendeki sayıları temsil eder. Ancak bunun farkı, harfle gösterilen onaltılı sayıları büyük harfle temsil etmesidir:

```
>>> print("%i sayısının onaltılı düzendeki karşılığı %X sayısıdır." %(10, 10))

10 sayısının onaltılı düzendeki karşılığı A sayısıdır.
```

Not

%d yapısını anlatırken gösterdiğimiz ileri düzey biçimlendirme tekniklerinin tamamını *%X* ile de kullanabilirsiniz.

f

Python'da karakter dizilerini biçimlendirirken *s* harfinden sonra en çok kullanılan harf *f* harfidir. Bu harf İngilizce'deki *float*, yani 'kayan noktalı sayı' kelimesinin kısaltmasıdır. Adından da anlaşılacağı gibi, karakter dizileri içindeki kayan noktalı sayıları temsil etmek için kullanılır.

```
>>> print("Dolar %f TL olmuş..." %1.4710)

Dolar 1.471000 TL olmuş...
```

Bu çıktı sizi biraz şaşırtmış olabilir. Çünkü gördüğünüz gibi, çıktıda bizim eklemediğimiz haneler var.

Python'da bir karakter dizisi içindeki sayıyı *%f* yapısı ile kayan noktalı sayıya çevirdiğimizde noktadan sonra öntanımlı olarak *6* hane yer alacaktır. Yani mesela:

```
>>> print("%f" %10)

10.000000
```

Gördüğünüz gibi, gerçekten de *10* tam sayısı *%f* yapısı ile kayan noktalı sayıya dönüştürüldüğünde noktadan sonra *6* adet sıfıra sahip oluyor.

Başka bir örnek daha verelim:

```
>>> print("%f"%23.6)
```

```
23.600000
```

Bu örnek, *%f* yapısının, kayan noktalı sayıların noktadan sonraki hane sayısını da *6*'ya tamamladığını gösteriyor. Ama elbette biz istersek, daha önce öğrendiğimiz teknikleri kullanarak, noktadan sonra kaç hane olacağını belirleyebiliriz:

```
>>> print("%.2f" % 10)

10.00
```

%f yapısında, *%* ile *f* arasına *.2* gibi bir ifade yerleştirerek noktadan sonra *2* hane olmasını sağladık.

Not

Daha önce gösterdiğimiz ileri düzey biçimlendirme tekniklerini *%f* ile de kullanabilirsiniz.

c

Bu harf de Python'daki önemli karakter dizisi biçimlendiricilerinden biridir. Bu harf tek bir karakteri temsil eder:

```
>>> print("%c" %"a")

a
```

Ama:

```
>>> print("%c" %"istihza")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: %c requires int or char
```

Gördüğünüz gibi, *c* harfi sadece tek bir karakteri kabul ediyor. Karakter sayısı birden fazla olduğunda bu komut hata veriyor.

c harfinin bir başka özelliği de ASCII tablosunda sayılara karşılık gelen karakterleri de gösterebilmesidir:

```
>>> print("%c" %65)

A
```

ASCII tablosunda 65 sayısı ‘A’ harfine karşılık geldiği için yukarıdaki komutun çıktısı ‘A’ harfini gösteriyor. Eğer isterseniz *c* harfini kullanarak bütün ASCII tablosunu ekrana dökebilirsiniz:

```
>>> for i in range(128):
...     print("%s ==> %c" %(i, i))
```

Not

%s yapısını anlatırken gösterdiğimiz ileri düzey biçimlendirme tekniklerinin tamamını *%c* ile de kullanabilirsiniz.

Böylece Python’da *%* işareti kullanarak nasıl biçimlendirme yapabileceğimizi öğrenmiş olduk. Dilerseniz pratik olması açısından, karakter dizisi biçimlendiricilerinin kullanımını gösteren bir örnek vererek bu bölümü noktalayalım.

Dikkatlice inceleyin:

```
for sıra, karakter in enumerate(dir(str)):
    if sıra % 3 == 0:
        print("\n", end="")
    print("%-20s" %karakter, end="")
```

Burada, gördüğünüz gibi, karakter dizisi metotlarını bir tablo görünümü içinde ekrana yazdırdık. Şu satırlar yardımıyla tablodaki sütun sayısını *3* olarak belirledik:

```
if sıra % 3 == 0:
    print("\n", end="")
```

Burada modülüs işlecini nasıl kullandığımıza çok dikkat edin. *sıra* değişkeninin değerini *3*‘e böldüğümüzde kalan değerin *0* olduğu her sayıda satır başına geçiyoruz. Böylece her *3*. sütunda bir satır aşağı geçilmiş oluyor.

Bununla ilgili bir örnek daha verelim:

```
for i in range(20):
    print("%5d%5o%5x" %(i, i, i))
```

Burada *0*‘dan *20*‘ye kadar olan sayıların onlu ve onaltılı düzendeki karşılıklarını bir tablo görünümü içinde ekrana çıktı verdik. Bu arada, eğer isterseniz yukarıdaki kodları şöyle de yazabileceğinizi biliyorsunuz:

```
for i in range(20):
    print("%(deger)5d%(deger)5o%(deger)5x" %({"deger": i}))
```

Burada da, tablomuzu biçimlendirmek için ‘sözlük’ adını verdiğimiz yapıdan yararlandık.

format() Metodu ile Biçimlendirme (Yeni Yöntem)

En başta da söylediğimiz gibi, % işaretini kullanarak karakter dizisi biçimlendirme eskide kalmış bir yöntemdir. Bu yöntem ağırlıklı olarak Python'ın 2.x sürümlerinde kullanılıyordu. Her ne kadar bu yöntemi Python'ın 3.x sürümlerinde de kullanmak mümkün olsa da yeni yazılan kodlarda bu yöntem yerine biraz sonra göreceğimiz format() metodunu kullanmak çok daha akıllıca olacaktır. Çünkü muhtemelen % ile biçimlendirme yöntemi, ileriki bir Python sürümünde dilden tamamen kaldırılacak. Bu yüzden bu eski metoda fazla bel bağlamamak gerekiyor.

Daha önceki derslerimizde verdiğimiz örnekler sayesinde format() metodunun temel olarak nasıl kullanılacağını biliyoruz. Ama isterseniz biz yine de bütünlük açısından format() metodunun temel kullanımını burada tekrar ele alalım.

format() metodunu en basit şekilde şöyle kullanıyoruz:

```
>>> print("{} ve {} iyi bir ikilidir!".format("Django", "Python"))

Django ve Python iyi bir ikilidir!
```

Gördüğünüz gibi, eski yöntemdeki % işaretine karşılık, yeni yöntemde *{}* işaretini kullanıyoruz.

Çok basit bir örnek daha verelim:

```
isim = input("İsminiz: ")
print("Merhaba {}. Nasılsın?".format(isim))
```

Elbette bu örneği şu şekilde de yazabilirdik:

```
isim = input("İsminiz: ")
print("Merhaba", isim + ".", "Nasılsın?")
```

Burada format() metodunu ve biçim düzenleyicileri hiç kullanmadan, sadece karakter dizilerini birleştirerek istediğimiz çıktıyı elde ettik. Ama siz de görüyorsunuz; karakter dizilerini birleştirmekle uğraşacağımıza format() metodunu kullanmak hem daha pratiktir, hem de bu şekilde yazdığımız kodlar daha okunaklı olur.

Yukarıdaki örnekte format() metodunu tek bir parametre ile birlikte kullandık (*isim*). Bu parametre (tıpkı eski % işaretinde olduğu gibi), karakter dizisi içindeki *{}* işaretine karşılık geliyor.

Bu konuyu daha iyi anlayabilmek için bir örnek daha verelim:

```
kalkış      = input("Kalkış yeri: ")
varış       = input("Varış yeri: ")
isim_soyisim = input("İsim ve soyisim: ")
bilet_sayısı = input("Bilet sayısı: ")
```

```
print("""{} noktasından {} noktasına, 14:30 hareket saatli
sefer için {} adına {} adet bilet ayrılmıştır!""".format(kalkış,
                                                         varış,
                                                         isim_soyisim,
                                                         bilet_sayısı))
```

Gördüğünüz gibi, *{}* işaretleri karakter dizisi içinde bir 'yer tutma' görevi görüyor. Tutulan bu yerlere nelerin geleceğini format() metodunun parametreleri vasıtasıyla belirliyoruz.

Elbette eğer isterseniz yukarıdaki örneği şu şekilde de yazabilirsiniz:

```
kalkış      = input("Kalkış yeri: ")
varış       = input("Varış yeri: ")
isim_soyisim = input("İsim ve soyisim: ")
bilet_sayısı = input("Bilet sayısı: ")

metin = "{} noktasından {} noktasına, 14:30 hareket saatli \
sefer için {} adına {} adet bilet ayrılmıştır!"

print(metin.format(kalkış, varış, isim_soyisim, bilet_sayısı))
```

Ancak yaygın olarak kullanılan yöntem, karakter dizisini herhangi bir değişkene atamadan, doğrudan format() metoduna bağlamaktır. Elbette hangi yöntem kolayınıza geliyorsa onu tercih etmekte özgürsünüz. Ama özellikle biçimlendirilecek karakter dizisinin çok uzun olduğu durumlarda, yukarıdaki gibi, karakter dizisini önce bir değişkene atayıp, sonra da bu değişken üzerine format() metodunu uygulamak daha mantıklı olabilir.

Verdiğimiz bu örneği, her zaman olduğu gibi, format() metoduna başvurmadan yazmak da mümkündür:

```
kalkış      = input("Kalkış yeri: ")
varış       = input("Varış yeri: ")
isim_soyisim = input("İsim ve soyisim: ")
bilet_sayısı = input("Bilet sayısı: ")

print(kalkış, "noktasından", varış, "noktasına, 14:30 hareket saatli \
sefer için", isim_soyisim, "adına", bilet_sayısı, "adet bilet ayrılmıştır!")
```

Tıpkı daha önce verdiğimiz örnekte olduğu gibi, burada da format() metodunu kullanmak karakter dizilerini birleştirme yöntemine göre daha mantıklı ve kolay görünüyor. Ayrıca bir karakter dizisi karmaşıklaştıkça bu karakter dizisini sadece karakter dizisi birleştirme yöntemleriyle biçimlendirmeye çalışmak bir süre sonra tam bir eziyet halini alabilir. O yüzden, 'Ben format() metodunu öğrenmesem de olur,' diye düşünmeyin sakın. Mesela şöyle bir programı format() metodu

kullanmadan yazmaya çalışmak hiç akıl kârı değildir:

```
kodlama  = "utf-8"
site_adı = "Python Programlama Dili"
dosya    = open("deneme.html", "w", encoding=kodlama)
içerik   = """
<html>

<head>
   <meta http-equiv="Content-Type" content="text/html; charset={}" />
   <title>{}</title>
</head>

<body>
   <h1>istihza.com web sitesine hoş geldiniz!</h1>
   <p><b>{}</b> için bir Türkçe belgelendirme projesi...</p>
</body>

</html>
"""

print(içerik.format(kodlama, site_adı, site_adı), file=dosya)

dosya.close()
```

Burada şu satırın bir kısmı hariç bütün kodları anlayabilecek düzeydesiniz:

```
dosya  = open("deneme.html", "w", encoding=kodlama)
```

Bu kodlarla, *deneme.html* adlı bir dosya oluşturduğumuzu biliyorsunuz. Daha önceki derslerimizde birkaç kez gördüğümüz open() fonksiyonu Python'da dosya oluşturmamıza imkan veriyor. Bu fonksiyon içinde kullandığımız üç parametrenin ilk ikisi size tanıdık gelecektir. İlk parametre dosyanın adını, ikinci parametre ise bu dosyanın hangi kipte açılacağını gösteriyor. Burada kullandığımız *"w"* parametresi *deneme.html* adlı dosyanın yazma kipinde açılacağını gösteriyor. Bu fonksiyona atadığımız *encoding* parametresi ise oluşturulacak dosyanın kodlama biçimini gösteriyor. Bu da Türkçe karakterlerin dosyada düzgün görüntülenebilmesi açısından önem taşıyor.

Küme parantezlerini, yukarıdaki örneklerde görüldüğü şekilde içi boş olarak kullanabilirsiniz. Böyle bir durumda Python, karakter dizisi içindeki küme parantezleriyle, karakter dizisi dışındaki değerleri teker teker ve sırasıyla eşleştirecektir. Ama isterseniz küme parantezleri içine birer sayı yazarak, karakter dizisi dışındaki değerlerin hangi sırayla kullanılacağını belirleyebilirsiniz. Örneğin:

```
>>> "{0} {1}".format("Fırat", "Özgül")
```

```
'Fırat Özgül'
```

Küme parantezleri içinde sayı kullanabilme imkanı sayesinde değerlerin sırasını istediğiniz gibi düzenleyebilirsiniz:

```
>>> "{1} {0}".format("Fırat", "Özgül")

'Özgül Fırat'
```

Hatta bu özellik sayesinde değerleri bir kez yazıp, birden fazla sayıda tekrar edebilirsiniz:

```
>>> "{0} {1} ({1} {0})".format("Fırat", "Özgül")

'Fırat Özgül (Özgül Fırat)'
```

Dolayısıyla, *{}* işaretleri içinde öğelerin sırasını da belirterek, biraz önce verdiğimiz *HTML* sayfası örneğini şu şekilde yazabilirsiniz:

```
kodlama  = "utf-8"
site_adı = "Python Programlama Dili"
dosya    = open("deneme.html", "w", encoding=kodlama)
içerik   = """
<html>

<head>
   <meta http-equiv="Content-Type" content="text/html; charset={0}" />
   <title>{1}</title>
</head>

<body>
   <h1>istihza.com web sitesine hoş geldiniz!</h1>
   <p><b>{1}</b> için bir Türkçe belgelendirme projesi...</p>
</body>

</html>
"""

print(içerik.format(kodlama, site_adı), file=dosya)

dosya.close()
```

Gördüğünüz gibi, öğelerin sıra numarasını belirtmemiz sayesinde, karakter dizisi içinde iki kez ihtiyaç duyduğumuz *site_adı* adlı değişkeni format() metodu içinde iki kez yazmak zorunda kalmadık.

Yukarıdaki örnekler bize, format() metodunun parametrelerine sıra numarasına göre erişebileceğimizi gösteriyor. Biz aynı zamanda bu metodun parametrelerine isme göre de erişebiliriz. Çok basit bir örnek:

```
print("{dil} dersleri".format(dil="python"))
```

Bu yöntemi kullanarak, aynı değişkeni birkaç farklı yerde kullanabilirsiniz:

```
sayfa = """
<html>

<head>
   <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
   <title>{konu}</title>
</head>

<body>
   <h1>istihza.com web sitesine hoş geldiniz!</h1>
   <p><b>{konu}</b> için bir Türkçe belgelendirme projesi...</p>
</body>

</html>
"""

print(sayfa.format(konu="Python Programlama Dili"))
```

format() metodunun yetenekleri yukarıda gösterdiğimiz şeylerle sınırlı değildir. Tıpkı eski biçimlendirme yönteminde olduğu gibi, *{}* işaretleri arasında bazı sayılar kullanarak, karakter dizileri üzerinde hizalama işlemleri de yapabiliriz.

Dikkatlice bakın:

```
>>> print("{:>15}".format("istihza"))

       istihza
```

Bu gösterim gözünüze oldukça yabancı ve karışık gelmiş olabilir. Ama aslında hiç de öyle anlaşılmaz bir yanı yoktur bu kodların. Gördüğünüz gibi, burada öncelikle *:* adlı bir işaretten yararlanıyoruz. Bu işaretin ardından > adlı başka bir işaret görüyoruz. Son olarak da *15* sayısını kullanıyoruz.

: işareti, bir biçimlendirme işlemi yapacağımızı gösteriyor. > işareti ise bu biçimlendirmenin bir hizalama işlemini olacağını haber veriyor. En sondaki *15* sayısı ise bu hizalama işleminin 15 karakterlik bir alan ile ilgili olduğunu söylüyor. Bu şekilde karakter dizisini *15* karakterlik bir alan içine yerleştirip karakter dizisini sağa yasladık. Yukarıdaki çıktıyı daha iyi anlayabilmek için kodları şöyle de yazabilirsiniz:

```
>>> print("|{:>15}|".format("istihza"))

|       istihza|
```

Gördüğünüz gibi, karakter dizimiz, kendisine ayrılan *15* karakterlik alan içinde sağa yaslanmış vaziyette duruyor.

Eğer aynı karakter dizisini sola yaslamak isterseniz şöyle bir şey yazabilirsiniz:

```
>>> print("|{:<15}|".format("istihza"))

|istihza        |
```

Bu defa < adlı işaretten yararlandığımıza dikkat edin.

Yukarıdaki yöntemi kullanarak, karakter dizilerini sola veya sağa yaslamanın yanısıra, kendilerine ayrılan alan içinde ortalayabilirsiniz de:

```
>>> print("|{:^15}|".format("istihza"))

|    istihza    |
```

Gördüğünüz gibi, python3 ile gelen format() metodunu hizalama işlemleri için kullanırken üç farklı işaretten yararlanıyoruz:

>	sağa yaslama
<	sola yaslama
^	ortalama

Yukarıdaki işaretler, yaptıkları işi çağrıştırdıkları için, bunları akılda tutmak çok zor olmasa gerek. Mesela örnek olması açısından, eski biçimlendirme yönteminin son kısmında verdiğimiz şu örneği:

```
for sıra, karakter in enumerate(dir(str)):
    if sıra % 3 == 0:
        print("\n", end="")
    print("%-20s" %karakter, end="")
```

... bir de yeni format() metoduyla yazalım:

```
for sıra, karakter in enumerate(dir(str)):
    if sıra % 3 == 0:
        print("\n", end="")
    print("{:<20}".format(karakter), end="")
```

Bu örneği inceleyerek, eski ile yeni yöntem arasında nelerin değiştiğini, neyin neye karşılık geldiğini

görebilirsiniz.

Biçimlendirme Karakterleri

Hatırlarsanız Python2'de geçerli olan eski biçimlendirme yönteminde % karakteri ile bazı harfleri birlikte kullanarak karakter dizileri üzerinde biçimlendirme ve dönüştürme işlemleri yapabiliyorduk. Aynı şey Python3 ile birlikte gelen bu format() metodu için de geçerlidir. Yani benzer harfleri kullanarak format() metodu ile de karakter dizileri üzerinde biçimlendirme ve dönüştürme işlemleri yapabiliriz.

format() metodu ile birlikte şu harfleri kullanabiliyoruz:

s

Bu harf karakter dizilerini temsil eder.

Yalnız bu biçimlendirici karakterlerin {} işaretleri içindeki kullanımı ilk bakışta gözünüze biraz karışık gelebilir:

```
>>> print("{:s}".format("karakter dizisi"))

karakter dizisi
```

Bu arada, harfleri {} yapısının içinde nasıl kullandığımıza dikkat edin. Gördüğünüz gibi biçimlendirme karakterini kullanırken, karakterin sol tarafına bir adet : işareti de yerleştiriyoruz. Bir örnek verelim:

```
print("{:s} ve {:s} iyi bir ikilidir!".format("Python", "Django"))
```

Yalnız, *s* harfi karakter dizilerini temsil ettiği için, {} işaretleri arasında bu harfi kullandığımızda, format() metodunun alabileceği parametreyi karakter dizisiyle sınırlandırmış oluruz. Dolayısıyla bu harfi kullandıktan sonra format() metodu içinde sadece karakter dizilerini kullanabiliriz. Eğer sayı kullanırsak Python bize bir hata mesajı gösterecektir:

```
>>> print("{:s}".format(12))

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Unknown format code 's' for object of type 'int'
```

Bu yüzden, eğer amacınız format() metoduna parametre olarak karakter dizisi vermekse, {} işaretleri içinde herhangi bir harf kullanmamak daha akıllıca olabilir:

```
print("{} ve {} iyi bir ikilidir!".format("Python", "Django"))
```

c

Bu harf *0* ile *256* arası sayıların ASCII tablosundaki karşılıklarını temsil eder:

```
>>> print("{:c}".format(65))

A
```

d

Bu harf sayıları temsil eder:

```
>>> print("{:d}".format(65))

65
```

Eğer sayı dışında bir değer kullanırsanız Python size bir hata mesajı gösterir:

```
>>> print("{:d}".format("65"))

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Unknown format code 'd' for object of type 'str'
```

o

Bu harf onlu düzendeki sayıları sekizli düzendeki karşılıklarına çevirir:

```
>>> print("{:o}".format(65))

101
```

x

Bu harf onlu düzendeki sayıları onaltılı düzendeki karşılıklarına çevirir:

```
>>> print("{:x}".format(65))

41
```

X

Tıpkı *x* harfinde olduğu gibi, bu harf de onlu düzendeki sayıları onaltılı düzendeki karşılıklarına çevirir:

```
>>> "{:X}".format(65)

'41'
```

Peki *x* ile *X* harfi arasında ne fark var? Fark şudur: *x*; onaltılı düzende harfle gösterilen sayıları küçük harf şeklinde temsil eder. *X* işare bu sayıları büyük harf şeklinde temsil eder. Bu ikisi arasındaki farkı daha net görmek için şöyle bir kod yazabilirsiniz:

```
>>> for i in range(20):
...     print("{:x}{:10X}".format(i, i))
...
0          0
1          1
2          2
3          3
4          4
5          5
6          6
7          7
8          8
9          9
a          A
b          B
c          C
d          D
e          E
f          F
10        10
11        11
12        12
13        13
```

Gördüğünüz gibi gerçekten de *x* harfi onaltılı düzende harflerle gösterilen sayıları küçük harf olarak; *X* harfi ise büyük harf olarak temsil ediyor.

b

Bu işaret, onlu düzendeki sayıları ikili düzendeki karşılıklarına çevirir:

```
>>> "{:b}.format(2)

'10'
```

f

Bu işaret, eski biçimlendirme yöntemini anlatırken gösterdiğimiz *f* işaretiyle benzer bir işleve sahiptir:

```
print("{:.2f}".format(50))
```

,

: işaretini , işareti (basamak ayracı) ile birlikte kullanarak, sayıları basamaklarına ayırabilirsiniz:

```
>>> "{:,}.format(1234567890)

'1,234,567,890'
```

Böylece Python'da karakter dizisi biçimlendirmenin hem eski hem de yeni yöntemini, şu ana kadarki Python bilgimiz elverdiği ölçüde ayrıntılı bir şekilde incelemiş olduk. Buradaki bilgileri kullanarak bol bol örnek yapmak bu konuyu daha iyi anlamanıza yardımcı olacaktır.

başa dön

Listeler ve Demetler

Bu bölüme gelene kadar yalnızca iki farklı veri tipi görmüştük. Bunlardan biri karakter dizileri, öteki ise sayılardı. Ancak tabii ki Python'daki veri tipleri yalnızca bu ikisiyle sınırlı değildir. Python'da karakter dizileri ve sayıların dışında, başka amaçlara hizmet eden, başka veri tipleri de vardır. İşte biz bu bölümde iki farklı veri tipi daha öğreneceğiz. Bu bölümde ele alacağımız veri tiplerinin adı 'liste' (*list*) ve 'demet' (*tuple*).

Bu bölümde birer veri tipi olarak listeler ve demetlerden söz etmenin yanısıra liste ve demetlerin metotlarından da bahsedeceğiz. Listelerle demetleri öğrendikten sonra Python'daki hareket imkanınızın bir hayli genişlediğine tanık olacaksınız.

Python programlama diline yeni başlayan biri, karakter dizilerini öğrendikten sonra bu dilde her şeyi karakter dizileri yardımıyla halledebileceğini zannedebilir. O yüzden yeni bir veri tipi ile karşılaştığında (örneğin listeler veya demetler), bu yeni veri tipi ona anlamsız ve gereksizmiş gibi görünebilir. Aslında daha önce de söylediğimiz gibi, bir programlama dilini yeni öğrenenlerin genel sorunudur bu. Öğrenci bir programlama dilini oluşturan minik parçaları öğrenirken, öğrencinin zihni bu parçaların ne işine yarayacağı konusunda şüpheyle dolar. Sanki gereksiz şeylerle vakit kaybediyormuş gibi hissedebilir. En önemli ve en büyük programların, bu minik parçaların sistematik bir şekilde birleştirilmesiyle ortaya çıkacak olması öğrencinin kafasına yatmayabilir. Halbuki en karmaşık programların bile kaynak kodlarını incelediğinizde göreceksiniz karakter dizileri, listeler, demetler, sayılar ve buna benzer başka veri tiplerinden ibarettir. Nasıl en lezzetli yemekler birkaç basit malzemenin bir araya gelmesi ile ortaya çıkıyorsa, en abidevi programlar da ilk bakışta birbiriyle ilgisiz görünen çok basit parçaların incelikli bir şekilde birleştirilmesinden oluşur.

O halde bu noktada, Python programlama diline yeni başlayan hemen herkesin sorduğu o soruyu soralım kendimize: 'Neden farklı veri tipleri var? Bu veri tiplerinin hepsine gerçekten ihtiyacım olacak mı?'

Bu soruyu başka bir soruyla cevaplamaya çalışalım: 'Acaba neden farklı giysi tipleri var? Neden kot pantolon, kumaş pantolon, tişört, gömlek ve buna benzer ayrımlara ihtiyaç duyuyoruz?' Bu sorunun cevabı çok basit: 'Çünkü farklı durumlara farklı giysi türleri uygundur!'

Örneğin ev taşıyacaksanız, herhalde kumaş pantolon ve gömlek giymezsiniz üzerinize. Buna benzer bir şekilde iş görüşmesine giderken de kot pantolon ve tişört doğru bir tercih olmayabilir. İşte buna benzer sebeplerden, programlama dillerinde de belli durumlarda belli veri tiplerini kullanmanız gerekir. Örneğin bir durumda karakter dizilerini kullanmak uygunken, başka bir durumda listeleri veya demetleri kullanmak daha mantıklı olabilir. Zira her veri tipinin kendine has güçlü ve zayıf yanları vardır. Veri tiplerini ve bunların ayrıntılarını öğrendikçe, hangi veri tipinin hangi sorun için daha kullanışlı olduğunu kestirebilecek duruma geleceğinizden hiç kuşkunuz olmasın.

Biz bu bölümde listeleri ve demetleri olabildiğince ayrıntılı bir şekilde inceleyeceğiz. O yüzden bu veri tiplerini incelerken konuyu birkaç farklı bölüme ayıracağız.

Listeleri ve demetleri incelemeye listelerden başlayalım...

Listeler

Giriş bölümünde de değindiğimiz gibi, listeler Python'daki veri tiplerinden biridir. Tıpkı karakter dizileri ve sayılar gibi...

Liste Tanımlamak

Listeleri tanımaya, bu veri tipini nasıl tanımlayacağımızı öğrenerek başlayalım.

Hatırlarsanız bir karakter dizisi tanımlayabilmek için şöyle bir yol izliyorduk:

```
>>> kardiz = "karakter dizisi"
```

Yani herhangi bir öğeyi karakter dizisi olarak tanımlayabilmek için yapmamız gereken tek şey o öğeyi tırnak içine almaktı. Herhangi bir öğeyi (tek, çift veya üç) tırnak içine aldığımızda karakter dizimizi tanımlamış oluyoruz. Liste tanımlamak için de buna benzer bir şey yapıyoruz. Dikkatlice bakın:

```
>>> liste = ["öğe1", "öğe2", "öğe3"]
```

Gördüğünüz gibi, liste tanımlamak da son derece kolay. Bir liste elde etmek için, öğeleri birbirinden virgülle ayırıp, bunların hepsini köşeli parantezler içine alıyoruz.

Karakter dizilerini anlatırken, herhangi bir nesnenin karakter dizisi olup olmadığından emin olmak için type() fonksiyonundan yararlanabileceğimizi söylemiştik. Eğer bir nesne type() fonksiyonuna *<class 'str'>* cevabı veriyorsa o nesne bir karakter dizisidir. Listeler için de buna benzer bir sorgulama yapabiliriz:

```
>>> liste = ["öğe1", "öğe2", "öğe3"]
>>> type(liste)

<class 'list'>
```

Bu çıktıdan anlıyoruz ki, liste veri tipi type() fonksiyonuna *<class 'list'>* cevabı veriyor. Dolayısıyla, eğer bir nesne type() fonksiyonuna *<class 'list'>* cevabı veriyorsa o nesnenin bir liste olduğunu rahatlıkla söyleyebiliriz.

Yukarıda tanımladığımız *liste* adlı listeye baktığımızda dikkatimizi bir şey çekiyor olmalı. Bu listeye şöyle bir baktığımızda, aslında bu listenin, içinde üç adet karakter dizisi barındırdığını görüyoruz. Gerçekten de listeler, bir veya daha fazla veri tipini içinde barındıran kapsayıcı bir veri tipidir. Mesela şu listeye bir bakalım:

```
>>> liste = ["Ahmet", "Mehmet", 23, 65, 3.2]
```

Gördüğünüz gibi, liste içinde hem karakter dizileri (*"Ahmet"*, *"Mehmet"*), hem de sayılar (*23*, *65*, *3.2*) var.

Dahası, listeler içlerinde başka listeleri de barındırabilir:

```
>>> liste = ["Ali", "Veli", ["Ayşe", "Nazan", "Zeynep"], 34, 65, 33, 5.6]
```

Bu *liste* adlı değişkenin tipini sorgularsak şöyle bir çıktı alacağımızı biliyorsunuz:

```
>>> type(liste)

<class 'list'>
```

Bir de şunu deneyelim:

```
for öğe in liste:
    print("{} adlı öğenin veri tipi: {}".format(öğe, type(öğe)))
```

Bu kodları çalıştırdığımızda da şöyle bir çıktı alıyoruz:

```
Ali adlı öğenin veri tipi: <class 'str'>
Veli adlı öğenin veri tipi: <class 'str'>
['Ayşe', 'Nazan', 'Zeynep'] adlı öğenin veri tipi: <class 'list'>
34 adlı öğenin veri tipi: <class 'int'>
65 adlı öğenin veri tipi: <class 'int'>
33 adlı öğenin veri tipi: <class 'int'>
5.6 adlı öğenin veri tipi: <class 'float'>
```

Bu kodlar bize şunu gösteriyor: farklı öğeleri bir araya getirip bunları köşeli parantezler içine alırsak 'liste' adlı veri tipini oluşturmuş oluyoruz. Bu listenin öğeleri farklı veri tiplerine ait olabilir. Yukarıdaki kodların da gösterdiği gibi, liste içinde yer alan *"Ali"* ve *"Veli"* öğeleri birer karakter dizisi, *['Ayşe', 'Nazan', 'Zeynep']* adlı öğe bir liste, *34*, *65* ve *33* öğeleri birer tam sayı, *5.6* öğesi ise bir kayan noktalı sayıdır. İşte farklı veri tiplerine ait bu öğelerin hepsi bir araya gelerek liste denen veri tipini oluşturuyor. Yukarıdaki örnekten de gördüğünüz gibi, bir listenin içinde başka bir liste de yer alabiliyor. Örneğin burada listemizin öğelerinden biri, *['Ayşe', 'Nazan', 'Zeynep']* adlı başka bir listedir.

Hatırlarsanız karakter dizilerinin belirleyici özelliği tırnak işaretleri idi. Yukarıdaki örneklerden de gördüğünüz gibi listelerin belirleyici özelliği de köşeli parantezlerdir. Mesela:

```
>>> karakter = ""
```

Bu boş bir karakter dizisidir. Şu ise boş bir liste:

```
>>> liste = []
```

Tıpkı karakter dizilerinde olduğu gibi, listelerle de iki şekilde karşılaşabilirsiniz:

1. Listeyi kendiniz tanımlamış olabilirsiniz.

2. Liste size başka bir kaynaktan gelmiş olabilir.

Yukarıdaki örneklerde bir listeyi kendimizin nasıl tanımlayacağımızı öğrendik. Peki listeler bize başka hangi kaynaktan gelebilir?

Hatırlarsanız karakter dizilerinin metotlarını sıralamak için dir() adlı bir fonksiyondan yararlanmıştık.

Mesela karakter dizilerinin bize hangi metotları sunduğunu görmek için bu fonksiyonu şöyle kullanmıştık:

```
>>> dir(str)
```

Bu komut bize şu çıktıyı vermişti:

```
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'capitalize', 'center', 'count', 'encode', 'endswith',
'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha',
'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
```

Artık bu çıktı size çok daha anlamlı geliyor olmalı. Gördüğünüz gibi çıktımız köşeli parantezler arasında yer alıyor. Yani aslında yukarıdaki çıktı bir liste. Dilerseniz bunu nasıl teyit edebileceğinizi biliyorsunuz:

```
>>> komut = dir(str)
>>> type(komut)

<class 'list'>
```

Gördüğünüz gibi, tıpkı input() fonksiyonundan gelen verinin bir karakter dizisi olması gibi, dir() fonksiyonundan gelen veri tipi de bir listedir.

dir() fonksiyonu dışında, başka bir şeyin daha bize liste verdiğini biliyoruz. Bu şey, karakter dizilerinin split() adlı metodudur:

```
>>> kardiz = "İstanbul Büyükşehir Belediyesi"
>>> kardiz.split()
```

```
['İstanbul', 'Büyükşehir', 'Belediyesi']
```

Görüyorsunuz, split() metodunun çıktısı da köşeli parantezler içinde yer alıyor. Demek ki bu çıktı da bir listedir.

Peki bir fonksiyonun bize karakter dizisi mi, liste mi yoksa başka bir veri tipi mi verdiğini bilmenin ne faydası var?

Her zaman söylediğimiz gibi, Python'da o anda elinizde olan verinin tipini bilmeniz son derece önemlidir. Aksi halde o veriyi nasıl evirip çevireceğinizi, o veriyle neler yapabileceğinizi bilemezsiniz. Mesela 'İstanbul Büyükşehir Belediyesi' ifadesini ele alalım. Bu ifadeyle ilgili size şöyle bir soru sorduğumu düşünün: 'Acaba bu ifadenin ilk harfini nasıl alırız?'

Eğer bu ifade size input() fonksiyonundan gelmişse, yani bir karakter dizisiyse uygulayacağınız yöntem farklı, split() metoduyla gelmişse, yani liste ise uygulayacağınız yöntem farklı olacaktır.

Eğer bu ifade bir karakter dizisi ise ilk harfi şu şekilde alabilirsiniz:

```
>>> kardiz = "İstanbul Büyükşehir Belediyesi"
>>> kardiz[0]

'İ'
```

Ama eğer bu ifade bir liste ise yukarıdaki yöntem size farklı bir sonuç verir:

```
>>> liste = kardiz.split()
>>> liste[0]

'İstanbul'
```

Çünkü *"İstanbul Büyükşehir Belediyesi"* adlı karakter dizisinin ilk öğesi *"İ"* karakteridir, ama *['İstanbul', 'Büyükşehir', 'Belediyesi']* adlı listenin ilk öğesi *"İ"* karakteri değil, *"İstanbul"* kelimesidir.

Gördüğünüz gibi, bir nesnenin hangi veri tipine ait olduğunu bilmek o nesneyle neleri nasıl yapabileceğimizi doğrudan etkiliyor. O yüzden programlama çalışmalarınız esnasında veri tiplerine karşı her zaman uyanık olmalısınız.

Not

Python'da bir nesnenin hangi veri tipine ait olduğunu bilmenin neden bu kadar önemli olduğunu gerçek bir örnek üzerinde görmek isterseniz istihza.com/forum/viewtopic.php?f=43&t=62 adresindeki tartışmayı inceleyebilirsiniz.

Her ne kadar karakter dizileri ve listeler iki farklı veri tipi olsa ve bu iki veri tipinin birbirinden çok farklı yönleri ve yetenekleri olsa da, bu iki veri tipi arasında önemli benzerlikler de vardır. Örneğin karakter dizilerini işlerken öğrendiğimiz pek çok fonksiyonu listelerle birlikte de kullanabilirsiniz. Mesela karakter dizilerini incelerken öğrendiğimiz len() fonksiyonu listelerin boyutunu hesaplamada da kullanılabilir:

```
>>> diller = ["İngilizce", "Fransızca", "Türkçe", "İtalyanca", "İspanyolca"]
>>> len(diller)

5
```

Karakter dizileri karakterlerden oluşan bir veri tipi olduğu için len() fonksiyonu karakter dizisi içindeki karakterlerin sayısını veriyor. Listeler ise başka veri tiplerini bir araya toplayan bir veri tipi olduğu için len() fonksiyonu liste içindeki veri tiplerinin sayısını söylüyor.

len() fonksiyonu dışında, range() fonksiyonuyla listeleri de birlikte kullanabilirsiniz. Mesela herhangi bir kaynaktan size şunlar gibi iki öğeli listeler geliyor olabilir:

```
[0, 10]
[6, 60]
[12, 54]
[67, 99]
```

Bu iki öğeli listeleri tek bir liste içinde topladığımızı düşünürsek şöyle bir kod yazabiliriz:

```
sayılar = [[0, 10], [6, 60], [12, 54], [67, 99]]

for i in sayılar:
    print(*range(*i))
```

Eğer ilk bakışta bu kod gözünüze anlaşılmaz göründüyse bu kodu parçalara ayırarak inceleyebilirsiniz.

Burada öncelikle bir for döngüsü oluşturduk. Bu sayede *sayılar* adlı listedeki öğelerin üzerinden tek tek geçebileceğiz. Eğer döngü içinde sadece öğeleri ekrana yazdırıyor olsaydık şöyle bir kodumuz olacaktı:

```
for i in sayılar:
    print(i)
```

Bu kod bize şöyle bir çıktı verecektir:

```
[0, 10]
[6, 60]
[12, 54]
```

```
[67, 99]
```

range() fonksiyonunun nasıl kullanıldığını hatırlıyorsunuz. Yukarıdaki listelerde görünen ilk sayılar range() fonksiyonunun ilk parametresi, ikinci sayılar ise ikinci parametresi olacak. Yani her döngüde şöyle bir şey elde etmemiz gerekiyor:

```
range(0, 10)
range(6, 60)
range(12, 54)
range(67, 99)
```

Aslında kodlarımızı şöyle yazarak yukarıdaki çıktıyı elde edebilirdik:

```
sayılar = [[0, 10], [6, 60], [12, 54], [67, 99]]

for i in sayılar:
    print(range(i[0], i[1]))
```

Yukarıdaki açıklamalarda gördüğünüz gibi, *i* değişkeninin çıktısı ikişer öğeli bir liste oluyor. İşte burada yaptığımız şey, bu ikişer öğeli listelerin ilk öğesini (i[0]) range() fonksiyonunun ilk parametresi, ikinci öğesini (i[1]) ise range() fonksiyonunun ikinci parametresi olarak atamaktan ibaret. Ancak ilk derslerimizden hatırlayacağınız gibi, bunu yapmanın daha kısa bir yolu var. Bildiğiniz gibi, öğelerden oluşan dizileri ayrıştırmak için yıldız işaretinden yararlanabiliyoruz. Dolayısıyla yukarıdaki kodları şöyle yazmak daha pratik olabilir:

```
sayılar = [[0, 10], [6, 60], [12, 54], [67, 99]]

for i in sayılar:
    print(range(*i))
```

Gördüğünüz gibi, *i* değişkeninin soluna bir yıldız ekleyerek bu değişken içindeki değerleri ayrıştırdık ve şöyle bir çıktı elde ettik:

```
range(0, 10)
range(6, 60)
range(12, 54)
range(67, 99)
```

Hatırlarsanız, range(0, 10) gibi bir kod yazdığımızda Python bize *0* ile *10* arasındaki sayıları doğrudan göstermiyordu. Aralıktaki sayıları görmek için range() fonksiyonunun çıktısını bir döngü içine almalıyız:

```
for i in range(0, 10):
    print(i)
```

range(0, 10) çıktısını görmek için döngü kurmak yerine yine yıldız işaretinden yararlanabiliyoruz. Örneğin:

```
>>> print(*range(0, 10))

0 1 2 3 4 5 6 7 8 9
```

Aynı şeyi yukarıdaki kodlara da uygularsak şöyle bir şey elde ederiz:

```
sayılar = [[0, 10], [6, 60], [12, 54], [67, 99]]

for i in sayılar:
    print(*range(*i))
```

Gördüğünüz gibi, yıldız işaretini hem *i* değişkenine, hem de range() fonksiyonuna ayrı ayrı uygulayarak istediğimiz sonucu elde ettik.

Bu arada, yukarıdaki örnek bize listeler hakkında önemli bir bilgi de verdi. Karakter dizilerinin öğelerine erişmek için nasıl kardiz[öğe_sırası] gibi bir formülden yararlanıyorsak, listelerin öğelerine erişmek için de aynı şekilde liste[öğe_sırası] gibi bir formülden yararlanabiliyoruz.

Listelerin öğelerine nasıl ulaşacağımızın ayrıntılarını biraz sonra göreceğiz. Ama biz şimdi listelere ilişkin önemli bir fonksiyonu inceleyerek yolumuza devam edelim.

list() Fonksiyonu

Yukarıdaki örneklerden de gördüğünüz gibi liste oluşturmak için öğeleri belirleyip bunları köşeli parantezler içine almamız yeterli oluyor. Bu yöntemin dışında, liste oluşturmanın bir yöntemi daha bulunur. Mesela elimizde şöyle bir karakter dizisi olduğunu düşünelim:

```
>>> alfabe = "abcçdefgğhıijklmnoöprsştuüvyz"
```

Sorumuz şu olsun: ‘Acaba bu karakter dizisini listeye nasıl çeviririz?’

Karakter dizilerini anlatırken split() adlı bir metottan söz etmiştik. Bu metot karakter dizilerini belli bir ölçüte göre bölmemizi sağlıyordu. split() metoduyla elde edilen verinin bir liste olduğunu biliyorsunuz. Örneğin:

```
>>> isimler = "ahmet mehmet cem"

>>> isimler.split()

['ahmet', 'mehmet', 'cem']
```

Ancak split() metodunun bir karakter dizisini bölüp bize bir liste verebilmesi için karakter dizisinin

belli bir ölçüte göre bölünebilir durumda olması gerekiyor. Mesela yukarıdaki *isimler* adlı karakter dizisi belli bir ölçüte göre bölünebilir durumdadır. Neden? Çünkü karakter dizisi içindeki her parça arasında bir boşluk karakteri var. Dolayısıyla split() metodu bu karakter dizisini boşluklardan bölebiliyor. Aynı şey şu karakter dizisi için de geçerlidir:

```
>>> isimler = "elma, armut, çilek"
```

Bu karakter dizisini oluşturan her bir parça arasında bir adet virgül ve bir adet boşluk karakteri var. Dolayısıyla biz bu karakter dizisini split() metodunu kullanarak "virgül + boşluk karakteri" ölçütüne göre bölebiliriz:

```
>>> isimler.split(", ")

['elma', 'armut', 'çilek']
```

Ancak bölümün başında tanımladığımız *alfabe* adlı karakter dizisi biraz farklıdır:

```
>>> alfabe = "abcçdefgğhıijklmnoöprsştuüvyz"
```

Gördüğünüz gibi, bu karakter dizisi tek bir parçadan oluşuyor. Dolayısıyla bu karakter dizisini öğelerine bölmemizi sağlayacak bir ölçüt yok. Yani bu karakter dizisini şu şekilde bölemeyiz:

```
>>> alfabe.split()

['abcçdefgğhıijklmnoöprsştuüvyz']
```

Elbette bu karakter dizisini isterseniz farklı şekillerde bölebilirsiniz. Mesela:

```
>>> alfabe.split("i")

['abcçdefgğhı', 'jklmnoöprsştuüvyz']
```

Gördüğünüz gibi, biz burada *alfabe* karakter dizisini "i" harfinden bölebildik. Ama istediğimiz şey bu değil. Biz aslında şöyle bir çıktı elde etmek istiyoruz:

```
['a', 'b', 'c', 'ç', 'd', 'e', 'f', 'g', 'ğ', 'h', 'ı', 'i', 'j',
 'k', 'l', 'm', 'n', 'o', 'ö', 'p', 'r', 's', 'ş', 't', 'u', 'ü',
 'v', 'y', 'z']
```

Yani bizim amacımız, *alfabe* karakter dizisi içindeki her bir öğeyi birbirinden ayırmak. İşte Türk alfabesindeki harflerden oluşan bu karakter dizisini, list() adlı bir fonksiyondan yararlanarak istediğimiz şekilde bölebiliriz:

```
>>> harf_listesi = list(alfabe)
>>> print(harf_listesi)
```

```
['a', 'b', 'c', 'ç', 'd', 'e', 'f', 'g', 'ğ', 'h', 'ı', 'i', 'j',
 'k', 'l', 'm', 'n', 'o', 'ö', 'p', 'r', 's', 'ş', 't', 'u', 'ü',
 'v', 'y', 'z']
```

Böylece list() fonksiyonu yardımıyla bu karakter dizisini tek hamlede listeye çevirmiş olduk.

Peki bir karakter dizisini neden listeye çevirme ihtiyacı duyarız? Şu anda listelerle ilgili pek çok şeyi henüz bilmediğimiz için ilk bakışta bu çevirme işlemi gözünüze gereksizmiş gibi görünebilir, ama ilerleyen zamanda sizin de göreceğiniz gibi, bazı durumlarda listeleri manipüle etmek karakter dizilerini manipüle etmeye kıyasla çok daha kolaydır. O yüzden kimi zaman karakter dizilerini listeye çevirmek durumunda kalabilirsiniz.

list() fonksiyonunun yaptığı işi, daha önce öğrendiğimiz str(), int() ve float() fonksiyonlarının yaptığı işle kıyaslayabilirsiniz. list() fonksiyonu da tıpkı str(), int() ve float() fonksiyonları gibi bir dönüştürme fonksiyonudur. Örneğin int() fonksiyonunu kullanarak sayı değerli karakter dizilerini sayıya dönüştürebiliyoruz:

```
>>> k = "123"
>>> int(k)

123
```

Bu dönüştürme işlemi sayesinde sayılar üzerinde aritmetik işlem yapma imkanımız olabiliyor. İşte list() fonksiyonu da buna benzer bir amaca hizmet eder. Mesela input() fonksiyonundan gelen bir karakter dizisi ile toplama çıkarma yapabilmek için nasıl bu karakter dizisini önce sayıya dönüştürmemiz gerekiyorsa, bazı durumlarda bu karakter dizisini (veya başka veri tiplerini) listeye çevirmemiz de gerekebilir. Böyle bir durumda list() fonksiyonunu kullanarak farklı veri tiplerini rahatlıkla listeye çevirebiliriz.

Yukarıdaki işlevlerinin dışında, list() fonksiyonu boş bir liste oluşturmak için de kullanılabilir:

```
>>> li = list()
>>> print(li)

[]
```

Yukarıdaki kodlardan gördüğünüz gibi, boş bir liste oluşturmak için liste = [] koduna alternatif olarak list() fonksiyonundan da yararlanabilirsiniz.

list() fonksiyonunun önemli bir görevi de range() fonksiyonunun, sayı aralığını ekrana basmasını sağlamaktır. Bildiğiniz gibi, range() fonksiyonu tek başına bir sayı aralığını ekrana dökmez. Bu fonksiyon bize yalnızca şöyle bir çıktı verir:

```
>>> range(10)

range(0, 10)
```

Bu sayı aralığını ekrana dökmek için range() fonksiyonu üzerinde bir for döngüsü kurmamız gerekir:

```
>>> for i in range(10):
...     print(i)
...
0
1
2
3
4
5
6
7
8
9
```

Bu bölümde verdiğimiz örneklerde aynı işi şöyle de yapabileceğimizi öğrenmiştik:

```
>>> print(*range(10))

0 1 2 3 4 5 6 7 8 9
```

Bu görevi yerine getirmenin üçüncü bir yolu da list() fonksiyonunu kullanmaktır:

```
>>> list(range(10))

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Aslında burada yaptığımız şey range(10) ifadesini bir listeye dönüştürmekten ibarettir. Burada *range* türünde bir veriyi *list* türünde bir veriye dönüştürüyoruz:

```
>>> type(range(10))

<class 'range'>

>>> li = list(range(10))
>>> type(li)

<class 'list'>
```

Gördüğünüz gibi, yukarıdaki üç yöntem de aralıktaki sayıları ekrana döküyor. Yalnız dikkat ederseniz bu üç yöntemin çıktıları aslında görünüş olarak birbirlerinden ince farklarla ayrılıyor. Yazdığınız programda nasıl bir çıktıya ihtiyacınız olduğuna bağlı olarak yukarıdaki yöntemlerden herhangi birini tercih edebilirsiniz.

Böylece Python'da listelerin ne olduğunu ve bu veri tipinin nasıl oluşturulacağını öğrenmiş olduk. O halde bir adım daha atarak listelerin başka özelliklerine değinelim.

Listelerin Öğelerine Erişmek

Tıpkı karakter dizilerinde olduğu gibi, listelerde de her öğenin bir sırası vardır. Hatırlarsanız karakter dizilerinin öğelerine şu şekilde ulaşıyorduk:

```
>>> kardiz = "python"
>>> kardiz[0]

'p'
```

Bu bölümdeki birkaç örnekte de gördüğünüz gibi, listelerin öğelerine ulaşırken de aynı yöntemi kullanabiliyoruz:

```
>>> meyveler = ["elma", "armut", "çilek", "kiraz"]
>>> meyveler[0]

'elma'
```

Yalnız yöntem aynı olsa da yukarıdaki iki çıktı arasında bazı farklar olduğunu da gözden kaçırmayın. Bir karakter dizisinin *0.* öğesini aldığımızda o karakter dizisinin ilk karakterini almış oluyoruz. Bir listenin *0.* öğesini aldığımızda ise o listenin ilk öğesini almış oluyoruz.

Sayma yöntemi olarak ise karakter dizileri ve listelerde aynı mantık geçerli. Hem listelerde hem de karakter dizilerinde Python saymaya *0*'dan başlıyor. Yani karakter dizilerinde olduğu gibi, listelerde de ilk öğenin sırası *0*.

Eğer bu listenin öğelerinin hepsine tek tek ulaşmak isterseniz for döngüsünden yararlanabilirsiniz:

```
meyveler = ["elma", "armut", "çilek", "kiraz"]

for meyve in meyveler:
    print(meyve)
```

Bu listedeki öğeleri numaralandırmak da mümkün:

```
meyveler = ["elma", "armut", "çilek", "kiraz"]
```

```
for öğe_sırası in range(len(meyveler)):
    print("{}. {}".format(öğe_sırası, meyveler[öğe_sırası]))
```

...veya enumerate() fonksiyonunu kullanarak şöyle bir şey de yazabiliriz:

```
for sıra, öğe in enumerate(meyveler, 1):
    print("{}. {}".format(sıra, öğe))
```

Dediğimiz gibi, liste öğelerine ulaşmak için kullandığımız yöntem, karakter dizilerinin öğelerine ulaşmak için kullandığımız yöntemle aynı. Aslında karakter dizileri ile listeler arasındaki benzerlik bununla sınırlı değildir. Benzerlikleri birkaç örnek üzerinde gösterelim:

```
>>> meyveler = ["elma", "armut", "çilek", "kiraz"]
>>> meyveler[-1]

'kiraz'
```

Karakter dizilerinde olduğu gibi, öğe sırasını eksi değerli bir sayı yaptığımızda liste öğeleri sondan başa doğru okunuyor. Dolayısıyla meyveler[-1] komutu bize *meyveler* adlı listenin son öğesini veriyor.

```
>>> meyveler[0:2]

['elma', 'armut']
```

Karakter dizileri konusunu işlerken öğrendiğimiz dilimleme yöntemi listeler için de aynen geçerlidir. Orada öğrendiğimiz dilimleme kurallarını listelere de uygulayabiliyoruz. Örneğin liste öğelerini ters çevirmek için şöyle bir kod yazabiliyoruz:

```
>>> meyveler[::-1]

['kiraz', 'çilek', 'armut', 'elma']
```

Bu bölümün başında da söylediğimiz gibi, liste adlı veri tipi, içinde başka bir liste de barındırabilir. Buna şöyle bir örnek vermiştik:

```
>>> liste = ["Ali", "Veli", ["Ayşe", "Nazan", "Zeynep"], 34, 65, 33, 5.6]
```

Bu listedeki öğeler şunlardır:

```
Ali
Veli
['Ayşe', 'Nazan', 'Zeynep']
34
65
```

```
33
5.6
```

Gördüğünüz gibi, bu liste içinde *['Ayşe', 'Nazan', 'Zeynep']* gibi bir liste daha var. Bu liste ana listenin öğelerinden biridir ve bu da öteki öğeler gibi tek öğelik bir yer kaplar. Yani:

```
>>> len(liste)

7
```

Bu çıktıdan anlıyoruz ki, listemiz toplam *7* öğeden oluşuyor. Listenin *2.* sırasında yer alan listenin kendisi üç öğeden oluştuğu halde bu öğe ana liste içinde sadece tek öğelik bir yer kaplıyor. Yani *2.* sıradaki listenin öğeleri tek tek sayılmıyor. Peki böyle bir liste içindeki gömülü listenin öğelerini elde etmek istersek ne yapacağız? Yani mesela içe geçmiş listenin tamamını değil de, örneğin sadece *"Ayşe"* öğesini almak istersek ne yapmamız gerekiyor? Dikkatlice bakın:

```
>>> liste[2][0]

'Ayşe'
```

"Nazan" öğesini almak için:

```
>>> liste[2][1]

'Nazan'
```

"Zeynep" öğesini almak için:

```
>>> liste[2][2]

'Zeynep'
```

Gördüğünüz gibi, iç içe geçmiş listelerin öğelerini almak oldukça basit. Yapmamız gereken tek şey, gömülü listenin önce ana listedeki konumunu, ardından da almak istediğimiz öğenin gömülü listedeki konumunu belirtmektir.

İstersek gömülü listeyi ayrı bir liste olarak da alabiliriz:

```
>>> yeni_liste = liste[2]
>>> yeni_liste

['Ayşe', 'Nazan', 'Zeynep']
```

Böylece bu listenin öğelerine normal bir şekilde ulaşabiliriz:

```
>>> yeni_liste[0]
```

```
'Ayşe'

>>> yeni_liste[1]

'Nazan'

>>> yeni_liste[2]

'Zeynep'
```

Eğer bir listenin öğelerine erişmeye çalışırken, varolmayan bir sıra sayısı belirtirseniz Python size bir hata mesajı gösterecektir:

```
>>> liste = range(10)
>>> print(len(liste))

10
```

Burada range() fonksiyonundan yararlanarak *10* öğeli bir liste tanımladık. Bu listenin son öğesinin şu formüle göre bulunabileceğini karakter dizileri konusundan hatırlıyor olmalısınız:

```
>>> liste[len(liste)-1]

9
```

Demek ki bu listenin son öğesi *9* sayısı imiş... Bir de şunu deneyelim:

```
>>> liste[10]

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: range object index out of range
```

Gördüğünüz gibi, listemizde *10.* öğe diye bir şey olmadığı için Python bize *IndexError* tipinde bir hata mesajı gösteriyor. Çünkü bu listenin son öğesinin sırası len(liste)-1, yani *9*'dur.

Listelerin Öğelerini Değiştirmek

Hatırlarsanız karakter dizilerinden söz ederken bunların değiştirilemez (*immutable*) bir veri tipi olduğunu söylemiştik. Bu özellikten ötürü, bir karakter dizisi üzerinde değişiklik yapmak istediğimizde o karakter dizisini yeniden oluşturuyoruz. Örneğin:

```
>>> kardiz = "istihza"
>>> kardiz = "İ" + kardiz[1:]
```

```
>>> kardiz

'İstihza'
```

Listeler ise değiştirilebilen (*mutable*) bir veri tipidir. Dolayısıyla listeler üzerinde doğrudan değişiklik yapabiliriz. Bir liste üzerinde değişiklik yapabilmek için o listeyi yeniden tanımlamamıza gerek yok. Şu örneği dikkatlice inceleyin:

```
>>> renkler = ["kırmızı", "sarı", "mavi", "yeşil", "beyaz"]
>>> print(renkler)

['kırmızı', 'sarı', 'mavi', 'yeşil', 'beyaz']

>>> renkler[0] = "siyah"
>>> print(renkler)

['siyah', 'sarı', 'mavi', 'yeşil', 'beyaz']
```

Liste öğelerini nasıl değiştirdiğimize çok dikkat edin. Yukarıdaki örnekte *renkler* adlı listenin *0.* öğesini değiştirmek istiyoruz. Bunun için şöyle bir formül kullandık:

```
renkler[öğe_sırası] = yeni_öğe
```

Örnek olması açısından, aynı listenin 2. sırasındaki *"mavi"* adlı öğeyi *"mor"* yapalım bir de:

```
>>> renkler[2] = "mor"
>>> print(renkler)

['siyah', 'sarı', 'mor', 'yeşil', 'beyaz']
```

Gördüğünüz gibi, listeler üzerinde değişiklik yapmak son derece kolay. Sırf bu özellik bile, neden bazı durumlarda listelerin karakter dizileri yerine tercih edilebileceğini gösterecek güçtedir.

Liste öğelerini değiştirmeye çalışırken, eğer var olmayan bir sıra numarasına atıfta bulunursanız Python size IndexError tipinde bir hata mesajı gösterecektir:

```
>>> renkler[10] = "pembe"

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

Sıra numaralarını kullanarak listeler üzerinde daha ilginç işlemler de yapabilirsiniz. Mesela şu örneğe bakın:

```
>>> liste = [1, 2, 3]
>>> liste[0:len(liste)] = 5, 6, 7
>>> print(liste)

[5, 6, 7]
```

Burada *liste* adlı listenin bütün öğelerini bir çırpıda değiştirdik. Peki bunu nasıl yaptık?

Yukarıdaki örneği şu şekilde yazarsak biraz daha açıklayıcı olabilir:

```
>>> liste[0:3] = 5, 6, 7
```

Bu kodlarla yaptığımız şey, listenin *0.* ve *3.* öğesi arasında kalan bütün öğelerin yerine *5*, *6* ve *7* öğelerini yerleştirmekten ibarettir.

Karakter dizilerinden hatırlayacağınız gibi, eğer sıra numarası bir karakter dizisinin ilk öğesine karşılık geliyorsa o sıra numarasını belirtmeyebiliriz. Aynı şekilde eğer sıra numarası bir karakter dizisinin son öğesine karşılık geliyorsa o sıra numarasını da belirtmeyebiliriz. Bu kural listeler için de geçerlidir. Dolayısıyla yukarıdaki örneği şöyle de yazabilirdik:

```
>>> liste[:] = 5, 6, 7
```

Sıra numaralarını kullanarak gerçekten son derece enteresan işlemler yapabilirsiniz. Sıra numaraları ile neler yapabileceğinizi görmek için kendi kendinize ve hayal gücünüzü zorlayarak bazı denemeler yapmanızı tavsiye ederim.

Listeye Öğe Eklemek

Listeler büyüyüp küçülebilen bir veri tipidir. Yani Python'da bir listeye istediğiniz kadar öğe ekleyebilirsiniz. Diyelim ki elimizde şöyle bir liste var:

```
>>> liste = [2, 4, 5, 7]
```

Bu listeye yeni bir öğe ekleyebilmek için şöyle bir kod yazabiliriz:

```
>>> liste + [8]

[2, 4, 5, 7, 8]
```

Bu örnek, bize listeler hakkında önemli bir bilgi veriyor. Python'da + işareti kullanarak bir listeye öğe ekleyecekseniz, eklediğiniz öğenin de liste olması gerekiyor. Mesela bir listeye doğrudan karakter dizilerini veya sayıları ekleyemezsiniz:

```
>>> liste + 8

Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list

>>> liste + "8"

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "str") to list
```

Listelere + işareti ile ekleyeceğiniz öğelerin de bir liste olması gerekiyor. Aksi halde Python bize bir hata mesajı gösteriyor.

Listeleri Birleştirmek

Bazı durumlarda elinize farklı kaynaklardan farklı listeler gelebilir. Böyle bir durumda bu farklı listeleri tek bir liste halinde birleştirmeniz gerekebilir. Tıpkı karakter dizilerinde olduğu gibi, listelerde de birleştirme işlemleri için + işlecinden yararlanabilirsiniz.

Diyelim ki elimizde şöyle iki adet liste var:

```
>>> derlenen_diller = ["C", "C++", "C#", "Java"]
>>> yorumlanan_diller = ["Python", "Perl", "Ruby"]
```

Bu iki farklı listeyi tek bir liste haline getirmek için şöyle bir kod yazabiliriz:

```
>>> programlama_dilleri = derlenen_diller + yorumlanan_diller

['C', 'C++', 'C#', 'Java', 'Python', 'Perl', 'Ruby']
```

Bu işlemin sonucunu görelim:

```
>>> print(programlama_dilleri)
```

Gördüğünüz gibi, *derlenen_diller* ve *yorumlanan_diller* adlı listelerin öğelerini *programlama_dilleri* adlı tek bir liste içinde topladık.

Programcılık maceranız boyunca listeleri birleştirmenizi gerektiren pek çok farklı durumla karşılaşabilirsiniz. Örneğin şöyle bir durum düşünün: Diyelim ki kullanıcı tarafından girilen sayıların ortalamasını hesaplayan bir program yazmak istiyorsunuz. Bunun için şöyle bir kod yazabilirsiniz:

```
sayılar = 0

for i in range(10):
    sayılar += int(input("not: "))
```

```
print(sayılar/10)
```

Bu program kullanıcının *10* adet sayı girmesine izin verip, program çıkışında, girilen sayıların ortalamasını verecektir.

Peki girilen sayıların ortalaması ile birlikte, hangi sayıların girildiğini de göstermek isterseniz nasıl bir kod yazarsınız?

Eğer böyle bir şeyi karakter dizileri ile yazmaya kalkışırsanız epey eziyet çekersiniz. Ama şöyle bir kod yardımıyla istediğiniz şeyi basit bir şekilde elde edebilirsiniz:

```
sayılar = 0
notlar = []

for i in range(10):
    veri = int(input("{}. not: ".format(i+1)))
    sayılar += veri
    notlar += [veri]

print("Girdiğiniz notlar: ", *notlar)
print("Not ortalamanız: ", sayılar/10)
```

Burada kullanıcıdan gelen verileri her döngüde tek tek *notlar* adlı listeye gönderiyoruz. Böylece programın sonunda, kullanıcıdan gelen veriler bir liste halinde elimizde bulunmuş oluyor.

Bu arada, yukarıdaki kodlarda dikkatinizi bir şey çekmiş olmalı. Kullanıcıdan gelen verileri *notlar* adlı listeye gönderirken şöyle bir kod yazdık:

```
notlar += [veri]
```

Buradaki [veri] ifadesine dikkat edin. Bu kod yardımıyla kullanıcıdan gelen *veri* adlı değişkeni liste haline getiriyoruz. Bu yöntem bizim için yeni bir şey. Peki neden burada list() fonksiyonundan yararlanmadık?

Bunu anlamak için list() fonksiyonunun çalışma mantığını anlamamız gerekiyor.

Elinizde şöyle bir karakter dizisi olduğunu düşünün:

```
>>> alfabe = "abcçdefgğhıijklmnoöprsştuüvyz"
```

Diyelim ki siz bu karakter dizisindeki bütün öğeleri tek tek bir listeye atmak istiyorsunuz. Bu iş için list() fonksiyonunu kullanabileceğimizi daha önce söylemiştik:

```
>>> liste = list(alfabe)
```

Peki list() fonksiyonu bu karakter dizisinin öğelerini listeye atarken nasıl bir yöntem izliyor?

Aslında list() fonksiyonunun yaptığı iş şuna eşdeğerdir:

```
liste = []
alfabe = "abcçdefgğhıijklmnoöprsştuüvyz"

for harf in alfabe:
    liste += harf

print(liste)
```

list() fonksiyonu da tam olarak böyle çalışır. Yani bir karakter dizisi üzerinde döngü kurarak, o karakter dizisinin her bir öğesini tek tek bir listeye atar.

for döngülerini işlerken, bu döngünün sayılar üzerinde çalışmayacağını söylemiştik. Çünkü sayılar, karakter dizilerinin aksine, üzerinde döngü kurulabilen bir veri tipi değildir. Bunu bir örnek üzerinde tekrar görelim:

```
>>> for i in 12345:
...     print(i)
...

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

Gördüğünüz gibi, *12345* sayısı üzerinde döngü kuramıyoruz. Aynı hata mesajını list() fonksiyonunda da görürsünüz:

```
>>> list(12345)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

Dediğimiz gibi, tıpkı for döngüsünde olduğu gibi, list() fonksiyonu da ancak, üzerinde döngü kurulabilen nesneler üzerinde çalışabilir. Mesela:

```
>>> list("12345")

['1', '2', '3', '4', '5']
```

Bu bilgilerin ışığında, yukarıda yazdığımız kodların şu şekilde yazılması halinde Python'ın bize hata mesajı göstereceğini söyleyebiliriz:

```
for i in range(10):
    veri = int(input("{}. not: ".format(i+1)))
    notlar = list(veri)

print("Girdiğiniz notlar: ", *notlar)
```

Kullanıcıdan gelen *veri* değerini int() fonksiyonuyla sayıya dönüştürdüğümüz için ve sayılar da üzerinde döngü kurulabilen bir veri tipi olmadığı için list() fonksiyonuna parametre olarak atanamaz.

Peki kullanıcıdan gelen *veri* değerini sayıya dönüştürmeden, karakter dizisi biçiminde list() fonksiyonuna parametre olarak verirsek ne olur? Bu durumda list() fonksiyonu çalışır, ama istediğimiz gibi bir sonuç vermez. Şu kodları dikkatlice inceleyin:

```
notlar = []

for i in range(10):
    veri = input("{}. not: ".format(i+1))
    notlar += list(veri)

print("Girdiğiniz notlar: ", *notlar)
```

Bu kodları çalıştırdığınızda, tek haneli sayılar düzgün bir şekilde listeye eklenir, ancak çift ve daha fazla haneli sayılar ise listeye parça parça eklenir. Örneğin *234* sayısını girdiğinizde listeye *2*, *3* ve *4* sayıları tek tek eklenir. Çünkü, yukarıda da dediğim gibi, list() fonksiyonu, aslında karakter dizileri üzerine bir for döngüsü kurar. Yani:

```
>>> for i in "234":
...     print(i)

2
3
4
```

Dolayısıyla listeye *234* sayısı bir bütün olarak değil de, parça parça eklendiği için istediğiniz sonucu alamamış olursunuz.

Peki bu sorunun üstesinden nasıl geleceğiz? Aslında bu sorunun çözümü çok basittir. Eğer bir verinin listeye parça parça değil de, bir bütün olarak eklenmesini istiyorsanız *[]* işaretlerinden yararlanabilirsiniz. Tıpkı şu örnekte olduğu gibi:

```
liste = []

while True:
    sayı = input("Bir sayı girin: (çıkmak için q) ")
```

```
    if sayı == "q":
        break

    sayı = int(sayı)

    if sayı not in liste:
        liste +=  [sayı]
        print(liste)
    else:
        print("Bu sayıyı daha önce girdiniz!")
```

Gördüğünüz gibi, kullanıcı tarafından aynı verinin birden fazla girilmesini önlemek için de listelerden yararlanabiliyoruz.

Yalnız burada şunu söyleyelim: Gerçek programlarda listelere öğe eklemek veya listeleri birleştirmek gibi işlemler için yukarıdaki gibi + işlecinden yararlanmayacağız. Yukarıda gösterdiğimiz yöntem de doğru olmakla birlikte, bu iş için genellikle liste metotlarından yararlanılır. Bu metotları birazdan göreceğiz.

Listeden Öğe Çıkarmak

Bir listeden öğe silmek için *del* adlı ifadeden yararlanabilirsiniz. Örneğin:

```
>>> liste = [1, 5, 3, 2, 9]
>>> del liste[-1]
>>> liste

[1, 5, 3, 2]
```

Listeleri Silmek

Python'da listeleri tamamen silmek de mümkündür. Örneğin:

```
>>> liste = [1, 5, 3, 2, 9]
>>> del liste
>>> liste

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'liste' is not defined
```

Listeleri Kopyalamak

Diyelim ki, yazdığınız bir programda, varolan bir listeyi kopyalamak, yani aynı listeden bir tane daha üretmek istiyorsunuz. Mesela elimizde şöyle bir liste olsun:

```
>>> li1 = ["elma", "armut", "erik"]
```

Amacımız bu listeden bir tane daha oluşturmak. İlk olarak aklınıza şöyle bir yöntem gelmiş olabilir:

```
>>> li2 = li1
```

Böylece elimizde aynı öğelere sahip iki liste olmuş oldu:

```
>>> print(li1)

["elma", "armut", "erik"]

>>> print(li2)

["elma", "armut", "erik"]
```

Şimdi ilk listemiz olan *li1* üzerinde bir değişiklik yapalım. Mesela bu listenin *"elma"* olan ilk öğesini *"karpuz"* olarak değiştirelim:

```
>>> li1[0] = "karpuz"
>>> print(li1)

["karpuz", "armut", "erik"]
```

Gördüğünüz gibi, *li1* adlı listenin ilk öğesini başarıyla değiştirdik. Şimdi şu noktada, *li2* adlı öbür listemizin durumunu kontrol edelim:

```
>>> print(li2)

["karpuz", "armut", "erik"]
```

O da ne! Biz biraz önce *li1* üzerinde değişiklik yapmıştık, ama görünüşe göre bu değişiklikten *li2* de etkilenmiş. Muhtemelen beklediğiniz şey bu değildi. Yani siz *li2* listesinin içeriğinin aynı kalıp, değişiklikten yalnızca *li1* listesinin etkilenmesini istiyordunuz. Biraz sonra bu isteğinizi nasıl yerine getirebileceğinizi göstereceğiz. Ama önce dilerseniz, bir liste üzerindeki değişiklikten öteki listenin de neden etkilendiğini anlamaya çalışalım.

Hatırlarsanız, listelerin değiştirilebilir (*mutable*) bir veri tipi olduğunu söylemiştik. Listeler bu özellikleriyle karakter dizilerinden ayrılıyor. Zira biraz önce *li1* ve *li2* üzerinde yaptığımız işlemin bir benzerini karakter dizileri ile yaparsak farklı bir sonuç alırız. Dikkatlice bakın:

```
>>> a = "elma"
```

Burada, değeri *"elma"* olan *a* adlı bir karakter dizisi tanımladık. Şimdi bu karakter dizisini kopyalayalım:

```
>>> b = a

>>> a

'elma'

>>> b

'elma'
```

Böylece aynı değere sahip iki farklı karakter dizimiz olmuş oldu. Şimdi *a* adlı karakter dizisi üzerinde değişiklik yapalım:

```
>>> a = "E" + a[1:]

>>> a

'Elma'
```

Peki bu değişiklikten öbür karakter dizisi etkilendi mi?

```
>>> b

'elma'
```

Gördüğünüz gibi, bu değişiklik öteki karakter dizisini etkilememiş. Bunun sebebi karakter dizilerinin değiştirilemeyen (*immutable*) bir veri tipi olmasıdır.

Gelin isterseniz bu olgunun derinlerine inelim biraz...

Yukarıda *a* ve *b* adlı iki değişken var. Bunların kimliklerini kontrol edelim:

```
>>> id(a)

15182784

>>> id(b)

15181184
```

Gördüğünüz gibi, bu iki değişken farklı kimlik numaralarına sahip. Bu durumu şu şekilde de teyit edebileceğimizi biliyorsunuz:

```
>>> id(a) == id(b)

False
```

Demek ki gerçekten de id(a) ile id(b) birbirinden farklıymış.

Bu sonuç bize, bu iki karakter dizisinin bellekte farklı konumlarda saklandığını gösteriyor. Dolayısıyla Python, bir karakter dizisini kopyaladığımızda bellekte ikinci bir nesne daha oluşturuyor. Bu nedenle birbirinden kopyalanan karakter dizilerinin biri üzerinde yapılan değişiklik öbürünü etkilemiyor. Ama listelerde (ve değiştirilebilir bütün veri tiplerinde) durum farklı. Şimdi şu örneklere dikkatlice bakın:

```
>>> liste1 = ["ahmet", "mehmet", "özlem"]
```

Bu listeyi kopyalayalım:

```
>>> liste2 = liste1
```

Elimizde aynı öğelere sahip iki liste var:

```
>>> liste1

['ahmet', 'mehmet', 'özlem']

>>> liste2

['ahmet', 'mehmet', 'özlem']
```

Bu listelerin kimlik numaralarını kontrol edelim:

```
>>> id(liste1)

14901376

>>> id(liste2)

14901376

>>> id(liste1) == id(liste2)

True
```

Liste Üreteçleri (List Comprehensions)

Şimdi Python'daki listelere ilişkin çok önemli bir konuya değineceğiz. Bu konunun adı 'liste

üreteçleri'. İngilizce'de buna "*List Comprehension*" adı veriliyor.

Adından da anlaşılacağı gibi, liste üreteçlerinin görevi liste üretmektir. Basit bir örnek ile liste üreteçleri konusuna giriş yapalım:

```
liste = [i for i in range(1000)]
```

Burada 0'dan 1000'e kadar olan sayıları tek satırda bir liste haline getirdik. Bu kodların söz dizimine çok dikkat edin. Aslında yukarıdaki kod şu şekilde de yazılabilir:

```
liste = []

for i in range(1000):
    liste += [i]
```

Burada önce *liste* adlı boş bir liste tanımladık. Daha sonra 0 ile 1000 aralığında bütün sayıları bu boş listeye teker teker gönderdik. Böylece elimizde 0'dan 1000'e kadar olan sayıları tutan bir liste olmuş oldu. Aynı iş için liste üreteçlerini kullandığımızda ise bu etkiyi çok daha kısa bir yoldan halletmiş oluyoruz. Liste üreteçlerini kullandığımız kodu tekrar önümüze alalım:

```
liste = [i for i in range(1000)]
```

Gördüğünüz gibi, burada önceden boş bir liste tanımlamamıza gerek kalmadı. Ayrıca bu kodlarda for döngüsünün parantezler içine alınarak nasıl sadeleştirildiğine de dikkatinizi çekmek isterim. Şu kod:

```
for i in range(1000):
    liste += [i]
```

Liste üreteçlerini kullandığımızda şu koda dönüşüyor:

```
[i for i in range(1000)]
```

Pek çok durumda liste üreteçleri öbür seçeneklere kıyasla bir alternatif olma işlevi görür. Yani liste üreteçleri ile elde edeceğiniz sonucu başka araçlarla da elde edebilirsiniz. Mesela yukarıdaki kodların yaptığı işlevi yerine getirmek için başka bir seçenek olarak list() fonksiyonundan da yararlanabileceğimizi biliyorsunuz:

```
liste = list(range(1000))
```

Bu basit örneklerde liste üreteçlerini kullanmanın erdemi pek gözde çarpmıyor. Ama bazı durumlarda liste üreteçleri öteki alternatiflere kıyasla çok daha pratik bir çözüm sunar. Böyle durumlarda başka seçeneklere başvurup yolunuzu uzatmak yerine liste üreteçlerini kullanarak işinizi kısa yoldan halledebilirsiniz.

Örneğin 0 ile 1000 arasındaki çift sayıları listelemek için liste üreteçlerini kullanmak, alternatiflerine göre daha makul bir tercih olabilir:

```
liste = [i for i in range(1000) if i % 2 == 0]
```

Aynı işi for döngüsü ile yapmak için şöyle bir kod yazmamız gerekir:

```
liste = []
for i in range(1000):
    if i % 2 == 0:
        liste += [i]
```

Gördüğünüz gibi, liste üreteçleri bize aynı işi daha kısa bir yoldan halletme imkanı tanıyor. Bu arada for döngüsünün ve bu döngü içinde yer alan *if* deyiminin liste üreteçleri içinde nasıl göründüğüne dikkat ediyoruz.

Liste üreteçleri ile ilgili bir örnek daha verelim. Mesela elinizde şöyle bir liste olduğunu düşünün:

```
liste = [[1, 2, 3],
         [4, 5, 6],
         [7, 8, 9],
         [10, 11, 12]]
```

Burada iç içe geçmiş 4 adet liste var. Bu listenin bütün öğelerini tek bir listeye nasıl alabiliriz? Yani şöyle bir çıktıyı nasıl elde ederiz?

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

for döngülerini kullanarak şöyle bir kod yazabiliriz:

```
liste = [[1, 2, 3],
         [4, 5, 6],
         [7, 8, 9],
         [10, 11, 12]]

tümü = []

for i in liste:
    for z in i:
        tümü += [z]

print(tümü)
```

Liste üreteçleri ise daha kısa bir çözüm sunar:

```
liste = [[1, 2, 3],
         [4, 5, 6],
         [7, 8, 9],
```

```
      [10, 11, 12]]

tümü = [z for i in liste for z in i]
print(tümü)
```

Bu liste üreteci gerçekten de bize kısa bir çözüm sunuyor, ama bu tip iç içe geçmiş for döngülerinden oluşan liste üreteçlerinde bazen okunaklılık sorunu ortaya çıkabilir. Yani bu tür iç içe geçmiş for döngülerinden oluşan liste üreteçlerini anlamak, alternatif yöntemlere göre daha zor olabilir.

Bazı durumlarda ise liste üreteçleri bir sorunun çözümü için tek makul yol olabilir. Diyelim ki bir X.O.X Oyunu (*Tic Tac Toe*) yazıyorsunuz. Bu oyunda oyuncular oyun tahtası üzerine X veya O işaretlerinden birini yerleştirecek. Oyuncunun bu oyunu kazanabilmesi için, X veya O işaretlerinden birisinin oyun tahtası üzerinde belli konumlarda bulunması gerekiyor. Yani mesela X işaretinin oyunu kazanabilmesi için bu işaretin oyun tahtası üzerinde şu şekilde bir dizilime sahip olması gerekir:

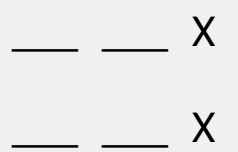
```
O  X  O

___ X  O

___ X ___
```

Bu dizilime göre oyunu X işareti kazanır. Peki X işaretinin, oyunu kazanmasını sağlayacak bu dizilime ulaştığını nasıl tespit edeceksiniz?

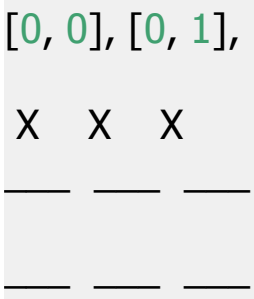
Bunun için öncelikle oyun tahtası üzerinde hangi dizilim şekillerinin galibiyeti getireceğini gösteren bir liste hazırlayabilirsiniz. Mesela yukarıdaki gibi 3x3 boyutundaki bir oyun tahtasında X işaretinin oyunu kazanabilmesi için şu dizilimlerden herhangi birine sahip olması gerekir:

```
[0, 0], [1, 0], [2, 0]

X ___ ___

X ___ ___

X ___ ___


[0, 1], [1, 1], [2, 1]

___ X ___

___ X ___

___ X ___
```

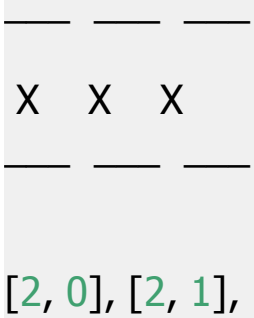
[0, 2], [1, 2], [2, 2]



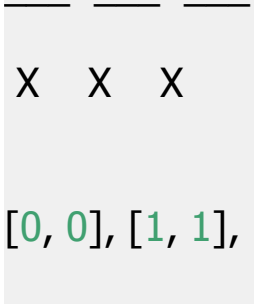
[0, 0], [0, 1], [0, 2]



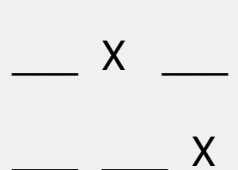
[1, 0], [1, 1], [1, 2]



[2, 0], [2, 1], [2, 2]



[0, 0], [1, 1], [2, 2]



```
[0, 2], [1, 1], [2, 0]

___ ___ X

___ X ___

X ___ ___
```

Aynı dizilimler O işareti için de geçerlidir. Dolayısıyla bu kazanma ölçütlerini şöyle bir liste içinde toplayabilirsiniz:

```
kazanma_ölçütleri = [[[0, 0], [1, 0], [2, 0]],
                     [[0, 1], [1, 1], [2, 1]],
                     [[0, 2], [1, 2], [2, 2]],
                     [[0, 0], [0, 1], [0, 2]],
                     [[1, 0], [1, 1], [1, 2]],
                     [[2, 0], [2, 1], [2, 2]],
                     [[0, 0], [1, 1], [2, 2]],
                     [[0, 2], [1, 1], [2, 0]]]
```

Oyun sırasında X veya O işaretlerinin aldığı konumu bu kazanma ölçütleri ile karşılaştırarak oyunu kimin kazandığını tespit edebilirsiniz. Yani *kazanma_ölçütleri* adlı liste içindeki, iç içe geçmiş listelerden herhangi biri ile oyunun herhangi bir aşamasında tamamen eşleşen işaret, oyunu kazanmış demektir.

Bir sonraki bölümde bu bahsettiğimiz X.O.X Oyununu yazacağız. O zaman bu sürecin nasıl işlediğini daha ayrıntılı bir şekilde inceleyeceğiz. Şimdilik yukarıdaki durumu temsil eden basit bir örnek vererek liste üreteçlerinin kullanımını incelemeye devam edelim.

Örneğin elinizde, yukarıda bahsettiğimiz kazanma ölçütlerini temsil eden şöyle bir liste olduğunu düşünün:

```
liste1 = [[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9],
        [10, 11, 12],
        [13, 14, 15],
        [16, 17, 18],
        [19, 20, 21],
        [22, 23, 24],
        [25, 26, 27],
        [28, 29, 30],
        [31, 32, 33]]
```

Bir de şöyle bir liste:

```
liste2 = [1, 27, 88, 98, 50, 9, 28, 45, 54, 66, 61, 23, 10, 33,
        22, 12, 6, 99, 63, 26, 87, 25, 77, 5, 16, 93, 99, 44,
        59, 69, 34, 10, 60, 92, 61, 44, 5, 3, 23, 99, 79, 51,
        89, 63, 53, 31, 76, 41, 49, 10, 88, 63, 55, 43, 40, 71,
        16, 49, 78, 41, 35, 97, 33, 76, 25, 81, 15, 99, 64, 20,
        33, 6, 89, 81, 44, 53, 59, 75, 27, 15, 64, 36, 72, 78,
        34, 36, 20, 41, 41, 75, 56, 30, 86, 46, 9, 42, 21, 64,
        26, 52, 77, 65, 64, 12, 38, 1, 35, 20, 73, 71, 37, 35,
        72, 38, 100, 52, 16, 49, 79]
```

Burada amacınız *liste1* içinde yer alan iç içe geçmiş listelerden hangisinin *liste2* içindeki sayıların alt kümesi olduğunu, yani *liste2* içindeki sayıların, *liste1* içindeki üçlü listelerden hangisiyle birebir eşleştiğini bulmak. Bunun için şöyle bir kod yazabiliriz:

```
for i in liste1:
    ortak = [z for z in i if z in liste2]
    if len(ortak) == len(i):
        print(i)
```

Bu kodlar ilk bakışta gözünüze çok karmaşık gelmiş olabilir. Ama aslında hiç de karmaşık değildir bu kodlar. Şimdi bu kodları Türkçe'ye çevirelim:

1. satır: *liste1* adlı listedeki her bir öğeye *i* adını verelim

2. satır: *i* içindeki, *liste2*'de de yer alan her bir öğeye de *z* adını verelim ve bunları *ortak* adlı bir listede toplayalım.

3. satır: eğer *ortak* adlı listenin uzunluğu *i* değişkeninin uzunluğu ile aynıysa

4. satır: *i*'yi ekrana basalım ve böylece alt kümeyi bulmuş olalım.

Eğer bu satırları anlamakta zorluk çekiyorsanız okumaya devam edin. Biraz sonra vereceğimiz örnek programda da bu kodları görecek ve bu kodların ne işe yaradığını orada daha iyi anlayacaksınız.

Örnek Program: X.O.X Oyunu

Şu ana kadar Python programlama dili hakkında epey bilgi edindik. Buraya kadar öğrendiklerimizi kullanarak işe yarar programlar yazabiliyoruz. Belki farkındasınız, belki de değilsiniz, ama özellikle listeler konusunu öğrenmemiz bize çok şey kazandırdı.

Bir önceki bölümde, bir X.O.X Oyunu yazacağımızdan söz etmiş ve bu oyunun Python'la nasıl yazılabileceğine dair bazı ipuçları da vermiştik. İşte bu bölümde, Python programlama dilinde şimdiye kadar öğrendiklerimizi kullanarak bu oyunu yazacağız.

Yazacağımız oyunun İngilizce adı *Tic Tac Toe*. Bu oyunun ne olduğunu ve kurallarını bir önceki bölümde kabataslak bir şekilde vermiştik. Eğer isterseniz oyun kurallarına wikipedia.org/wiki/Çocuk_oyunları#X_O_X_OYUNU adresinden de bakabilirsiniz.

Oyunu ve kurallarını bildiğinizi varsayarak kodlamaya başlayalım.

Burada ilk yapmamız gereken şey, üzerinde oyun oynanacak tahtayı çizmek olmalı. Amacımız şöyle bir görüntü elde etmek:

```
___ ___ ___

___ ___ ___

___ ___ ___
```

Bu tahtada oyuncu soldan sağa ve yukarıdan aşağıya doğru iki adet konum bilgisi girecek ve oyunu oynayan kişinin gireceği bu konumlara "X" ve "O" harfleri işaretlenecek.

Böyle bir görüntü oluşturmak için pek çok farklı yöntem kullanılabilir. Ama oyuncunun her konum bilgisi girişinde, X veya O işaretini tahta üzerinde göstereceğimiz için tahta üzerinde oyun boyunca sürekli birtakım değişiklikler olacak. Bildiğiniz gibi karakter dizileri, üzerinde değişiklik yapmaya müsait bir veri tipi değil. Böyle bir görev için listeler daha uygun bir araç olacaktır. O yüzden tahtayı oluşturmada listeleri kullanmayı tercih edeceğiz.

```
tahta = [["___","___","___"],
        ["___","___","___"],
        ["___","___","___"]]
```

Gördüğünüz gibi, burada iç içe geçmiş üç adet listeden oluşan bir liste var. print(tahta) komutunu kullanarak bu listeyi ekrana yazdırırsanız listenin yapısı daha belirgin bir şekilde ortaya çıkacaktır:

```
[['___', '___', '___'], ['___', '___', '___'], ['___', '___', '___']]
```

Oyun tahtasını oluşturduğumuza göre, şimdi yapmamız gereken şey bu oyun tahtasını düzgün bir şekilde oyuncuya göstermek olmalı. Dediğimiz gibi, oyuncu şöyle bir çıktı görmeli:

```
___ ___ ___

___ ___ ___

___ ___ ___
```

Bu görüntüyü elde etmek için şu kodları yazıyoruz:

```
print("\n"*15)
```

```
for i in tahta:
    print("\t".expandtabs(30), *i, end="\n"*2)
```

Bu kodlarda bilmediğiniz hiçbir şey yok. Burada gördüğünüz her şeyi önceki derslerde öğrenmiştiniz.

Yukarıdaki kodları yazarken tamamen, elde etmek istediğimiz görüntüye odaklanıyoruz. Mesela print("\\n"*15) kodunu yazmamızın nedeni, oyun tahtası için ekranda boş bir alan oluşturmak. Bu etkiyi elde etmek için 15 adet yeni satır karakteri bastık ekrana. Bu kodla elde edilen etkiyi daha iyi görebilmek için bu kodu programdan çıkarmayı deneyebilirsiniz.

Alttaki satırda ise bir for döngüsü tanımladık. Bu döngünün amacı *tahta* adlı listedeki "__" öğelerini düzgün bir şekilde oyuncuya gösterebilmek. Oyun tahtasının, ekranı (yaklaşık olarak da olsa) ortalamasını istiyoruz. O yüzden, tahta öğelerine soldan girinti verebilmek için print() fonksiyonunun ilk parametresini "\t".expandtabs(30) şeklinde yazdık. Karakter dizilerinin expandtabs() adlı metodunu önceki derslerimizden hatırlıyor olmalısınız. Bu metodu kullanarak sekme (TAB) karakterlerini genişletebiliyorduk. Burada da "\t" karakterini bu metot yardımıyla genişleterek liste öğelerini sol baştan girintiledik.

print() fonksiyonunun ikinci parametresi ise *i. Bu parametrenin ne iş yaptığını anlamak için şöyle bir kod yazalım:

```
tahta = [["___", "___", "___"],
        ["___", "___", "___"],
        ["___", "___", "___"]]

for i in tahta:
    print(i)
```

Bu kodları çalıştırdığımızda şöyle bir çıktı elde ederiz:

```
['___', '___', '___']
['___', '___', '___']
['___', '___', '___']
```

Gördüğünüz gibi, iç içe geçmiş üç adet listeden oluşan *tahta* adlı liste içindeki bu iç listeler ekrana döküldü. Bir de şuna bakın:

```
tahta = [["___", "___", "___"],
        ["___", "___", "___"],
        ["___", "___", "___"]]

for i in tahta:
    print(*i)
```

Bu kodlar çalıştırıldığında şu çıktıyı verir:

```
___ ___ ___
___ ___ ___
___ ___ ___
```

Bu defa liste yapısını değil, listeyi oluşturan öğelerin kendisini görüyoruz. Yıldız işaretinin, birlikte kullanıldığı öğeler üzerinde nasıl bir etkiye sahip olduğunu yine önceki derslerimizden hatırlıyorsunuz. Mesela şu örneğe bakın:

```
kardiz = "istihza"

for i in kardiz:
    print(i, end=" ")
print()
```

Bu kodlar şu çıktıyı veriyor:

```
i s t i h z a
```

Aynı çıktıyı basitçe şu şekilde de elde edebileceğimizi biliyorsunuz:

```
kardiz = "istihza"
print(*kardiz)
```

İşte oyun tahtasını ekrana dökmek için kullandığımız kodda da benzer bir şey yaptık. Yıldız işareti yardımıyla, *tahta* adlı listeyi oluşturan iç içe geçmiş listeleri liste dışına çıkarıp düzgün bir şekilde kullanıcıya gösterdik.

`print()` fonksiyonu içindeki son parametremiz şu: `end="\n"*2`

Bu parametrenin ne işe yaradığını kolaylıkla anlayabildiğinizi zannediyorum. Bu parametre de istediğimiz çıktıyı elde etmeye yönelik bir çabadan ibarettir. *tahta* adlı liste içindeki iç içe geçmiş listelerin her birinin sonuna ikişer adet "\n" karakteri yerleştirerek, çıktıdaki satırlar arasında yeterli miktarda aralık bıraktık. Eğer oyun tahtasındaki satırların biraz daha aralıklı olmasını isterseniz bu parametredeki 2 çarpanını artırabilirsiniz. Mesela: `end="\n"*3`

Şimdi yapmamız gereken şey, oyundaki kazanma ölçütlerini belirlemek. Hatırlarsanız bu konuya bir önceki bölümde değinmiştik. O yüzden aşağıda söyleyeceklerimizin bir bölümüne zaten aşinasınız. Burada önceden söylediğimiz bazı şeylerin yeniden üzerinden geçeceğiz.

Dediğim gibi, kodların bu bölümünde, hangi durumda oyununun biteceğini ve kazananın kim olacağını tespit edebilmemiz gerekiyor. Mesela oyun sırasında şöyle bir görüntü ortaya çıkarsa hemen oyunu durdurup "O KAZANDI!" gibi bir çıktı verebilmemiz lazım:

```
O   O   O

___  X   X
```

```
___ ___ ___
```

Veya şöyle bir durumda "X KAZANDI!" diyebilmeliyiz:

```
X  O  ___

X  O  O

X  ___ ___
```

Yukarıdaki iki örnek üzerinden düşünecek olursak, herhangi bir işaretin şu konumlarda bulunması o işaretin kazandığını gösteriyor:

```
yukarıdan aşağıya 0; soldan sağa 0
yukarıdan aşağıya 1; soldan sağa 0
yukarıdan aşağıya 2; soldan sağa 0
```

veya:

```
yukarıdan aşağıya 0; soldan sağa 0
yukarıdan aşağıya 0; soldan sağa 1
yukarıdan aşağıya 0; soldan sağa 2
```

İşte bizim yapmamız gereken şey, bir işaretin oyun tahtası üzerinde hangi konumlarda bulunması halinde oyunun biteceğini tespit etmek. Yukarıdaki örnekleri göz önüne alarak bunun için şöyle bir liste hazırlayabiliriz:

```
kazanma_ölçütleri = [[[0, 0], [1, 0], [2, 0]],
                     [[0, 0], [0, 1], [0, 2]]]
```

Burada iki adet listeden oluşan, *kazanma_ölçütleri* adlı bir listemiz var. Liste içinde, her biri üçer öğeden oluşan şu listeleri görüyoruz:

```
[[0, 0], [1, 0], [2, 0]]
[[0, 0], [0, 1], [0, 2]]
```

Bu listeler de kendi içinde ikişer öğeli bazı listelerden oluşuyor. Mesela ilk liste içinde şu listeler var:

```
[0, 0], [1, 0], [2, 0]
```

İkinci liste içinde ise şu listeler:

```
[0, 0], [0, 1], [0, 2]
```

Burada her bir liste içindeki ilk sayı oyun tahtasında yukarıdan aşağıya doğru olan düzlemi; ikinci sayı ise soldan sağa doğru olan düzlemi gösteriyor.

Tabii ki oyun içindeki tek kazanma ölçütü bu ikisi olmayacak. Öteki kazanma ölçütlerini de tek tek tanımlamalıyız:

```
kazanma_ölçütleri = [[[0, 0], [1, 0], [2, 0]],
                     [[0, 1], [1, 1], [2, 1]],
                     [[0, 2], [1, 2], [2, 2]],
                     [[0, 0], [0, 1], [0, 2]],
                     [[1, 0], [1, 1], [1, 2]],
                     [[2, 0], [2, 1], [2, 2]],
                     [[0, 0], [1, 1], [2, 2]],
                     [[0, 2], [1, 1], [2, 0]]]
```

İşte X veya O işaretleri *kazanma_ölçütleri* adlı listede belirtilen koordinatlarda bulunduğunda, ilgili işaretin oyunu kazandığını ilan edip oyundan çıkabileceğiz.

Yukarıdaki açıklamalardan da anlayacağınız gibi, X ve O işaretlerinin oyun tahtasındaki konumu, oyunun gidişatı açısından önem taşıyor. O yüzden şu şekilde iki farklı liste daha tanımlamamızda fayda var:

```
x_durumu = []
o_durumu = []
```

Bu değişkenler sırasıyla X işaretinin ve O işaretinin oyun içinde aldıkları konumları kaydedecek. Bu konumlarla, bir önceki adımda tanımladığımız kazanma ölçütlerini karşılaştırarak oyunu kimin kazandığını tespit edebileceğiz.

Gördüğünüz gibi, oyunda iki farklı işaret var: X ve O. Dolayısıyla oynama sırası sürekli olarak bu iki işaret arasında değişmeli. Mesela oyuna 0 işareti ile başlanacaksa, 0 işaretinin yerleştirilmesinden sonra sıranın X işaretine geçmesi gerekiyor. X işareti de yerleştirildikten sonra sıra tekrar 0 işaretine geçmeli ve oyun süresince bu böyle devam edebilmeli.

Bu sürekliliği sağlamak için şöyle bir kod yazabiliriz:

```
sıra = 1

while True:
    if sıra % 2 == 0:
        işaret = "X".center(3)
    else:
        işaret = "O".center(3)

    sıra += 1
```

```
    print()
    print("İŞARET: {}\n".format(işaret))
```

Burada sayıların tek veya çift olma özelliğinden yararlanarak X ve Y işaretleri arasında geçiş yaptık. Önce *sıra* adlı bir değişken tanımlayıp bunun değerini 1 olarak belirledik. *while* döngüsünde ise bu değişkenin değerini her defasında 1 artırdık. Eğer sayının değeri çiftse işaret X; tekse O olacak. Bu arada X ve O adlı karakter dizilerini, center() metodu yardımıyla ortaladığımıza dikkat edin.

Yukarıdaki kodları bu şekilde çalıştırdığınızda X ve Y harflerinin çok hızlı bir şekilde ekrandan geçtiğini göreceksiniz. Eğer ekranda son hız akıp giden bu verileri yavaşlatmak ve neler olup bittiğini daha net görmek isterseniz yukarıdaki kodları şöyle yazabilirsiniz:

```
from time import sleep

sıra = 1

while True:
    if sıra % 2 == 0:
        işaret = "X".center(3)
    else:
        işaret = "O".center(3)
    sıra += 1

    print()
    print("İŞARET: {}\n".format(işaret))
    sleep(0.3)
```

Bu kodlarda henüz öğrenmediğimiz parçalar var. Ama şimdilik bu bilmediğiniz parçalara değil, sonuca odaklanın. Burada yaptığımız şey, *while* döngüsü içinde her bir print() fonksiyonu arasına 0.3 saniyelik duraklamalar eklemek. Böylece programımın akışı yavaşlamış oluyor. Biz de *işaret* değişkeninin her döngüde bir X, bir O oluşunu daha net bir şekilde görebiliyoruz.

Not

Asıl program içinde X ve O karakterlerinin geçişini özellikle yavaşlatmamıza gerek kalmayacak. Programın ilerleyen satırlarında input() fonksiyonu yardımıyla kullanıcıdan veri girişi isteyeceğimiz için X ve O'ların akışı zaten doğal olarak duraklamış olacak.

while döngümüzü yazmaya devam edelim:

```
x = input("yukarıdan aşağıya [1, 2, 3]: ".ljust(30))
if x == "q":
```

```
    break

y = input("soldan sağa [1, 2, 3]: ".ljust(30))
if y == "q":
    break

x = int(x)-1
y = int(y)-1
```

Burada X veya O işaretlerini tahta üzerinde uygun yerlere yerleştirebilmek için kullanıcının konum bilgisi girmesini istiyoruz. *x* değişkeni yukarıdan aşağıya doğru olan düzlemdeki konumu, *y* değişkeni ise soldan sağa doğru olan düzlemdeki konumu depolayacak. Oyunda kullanıcının girebileceği değerler 1, 2 veya 3 olacak. Mesela oyuncu O işareti için yukarıdan aşağıya 1; soldan sağa 2 değerini girmişse şöyle bir görüntü elde edeceğiz:

```
___  O  ___

___ ___ ___

___ ___ ___
```

Burada ljust() metotlarını, kullanıcıya gösterilecek verinin düzgün bir şekilde hizalanması amacıyla kullandık.

Eğer kullanıcı *x* veya *y* değişkenlerinden herhangi birine "q" cevabı verirse oyundan çıkıyoruz.

Yukarıdaki kodların son iki satırında ise kullanıcıdan gelen karakter dizilerini birer sayıya dönüştürüyoruz. Bu arada, bildiğiniz gibi Python saymaya 0'dan başlıyor. Ama insanlar açısından doğal olan saymaya 1'den başlamaktır. O yüzden mesela kullanıcı 1 sayısını girdiğinde Python'ın bunu 0 olarak algılamasını sağlamamız gerekiyor. Bunun için x ve y değerlerinden 1 çıkarıyoruz.

Kullanıcıdan gerekli konum bilgilerini aldığımıza göre, bu bilgilere dayanarak X ve O işaretlerini oyun tahtası üzerine yerleştirebiliriz. Şimdi şu kodları dikkatlice inceleyin:

```
print("\n"*15)

if tahta[x][y] == "___":
    tahta[x][y] = işaret
    if işaret == "X".center(3):
        x_durumu += [[x, y]]
    elif işaret == "O".center(3):
        o_durumu += [[x, y]]
    sıra += 1
else:
    print("\nORASI DOLU! TEKRAR DENEYİN\n")
```

Burada öncelikle *15* adet satır başı karakteri basıyoruz. Böylece oyun tahtası için ekranda boş bir alan oluşturmuş oluyoruz. Bu satır tamamen güzel bir görüntü elde etmeye yönelik bir uygulamadır. Yani bu satırı yazmasanız da programınız çalışır. Veya siz kendi zevkinize göre daha farklı bir görünüm elde etmeye çalışabilirsiniz.

İkinci satırda gördüğümüz `if tahta[x][y] == "___":` kodu, oyun tahtası üzerindeki bir konumun halihazırda boş mu yoksa dolu mu olduğunu tespit etmemizi sağlıyor. Amacımız oyuncunun aynı konuma iki kez giriş yapmasını engellemek. Bunun için tahta üzerinde x ve y konumlarına denk gelen yerde "___" işaretinin olup olmadığına bakmamız yeterli olacaktır. Eğer bakılan konumda "___" işareti varsa orası boş demektir. O konuma işaret koyulabilir. Ama eğer o konumda "___" işareti yoksa X veya O işaretlerinden biri var demektir. Dolayısıyla o konuma işaret koyulamaz. Böyle bir durumda kullanıcıya "ORASI DOLU! TEKRAR DENEYİN" uyarısını gösteriyoruz.

Oyun tahtası üzerinde değişiklik yapabilmek için nasıl bir yol izlediğimize dikkat edin:

```
tahta[x][y] = işaret
```

Mesela oyuncu yukarıdan aşağıya 1; soldan sağa 2 sayısını girmişse, kullanıcıdan gelen sayılardan 1 çıkardığımız için, Python yukarıdaki kodu şöyle değerlendirecektir:

```
tahta[0][1] = işaret
```

Yani *tahta* adlı liste içindeki ilk listenin ikinci sırasına ilgili işaret yerleştirilecektir.

Ayrıca yukarıdaki kodlarda şu satırları da görüyoruz:

```
if işaret == "X".center(3):
    x_durumu += [[x, y]]
elif işaret == "O".center(3):
    o_durumu += [[x, y]]
```

Eğer işaret sırası X'te ise oyuncunun girdiği konum bilgilerini *x_durumu* adlı değişkene, eğer işaret sırası O'da ise konum bilgilerini *o_durumu* adlı değişkene yolluyoruz. Oyunu hangi işaretin kazandığını tespit edebilmemiz açısından bu kodlar büyük önem taşıyor. *x_durumu* ve *o_durumu* değişkenlerini *kazanma_ölçütleri* adlı liste ile karşılaştırarak oyunu kimin kazandığına karar vereceğiz.

Bu arada, oyunun en başında tanımladığımız *sıra* adlı değişkeni `if` bloğu içinde artırdığımıza dikkat edin. Bu sayede, kullanıcının yanlışlıkla aynı konuma iki kez işaret yerleştirmeye çalışması halinde işaret sırası değişmeyecek. Yani mesela o anda sıra X'te ise ve oyuncu yanlış bir konum girdiyse sıra yine X'te olacak. Eğer *sıra* değişkenini `if` bloğu içine yazmazsak, yanlış konum girildiğinde işaret sırası O'a geçecektir.

İsterseniz şimdiye kadar yazdığımız kodları şöyle bir topluca görelim:

```
tahta = [["___", "___", "___"],
```

```
        ["___", "___", "___"],
        ["___", "___", "___"]]

print("\n"*15)

for i in tahta:
    print("\t".expandtabs(30), *i, end="\n"*2)

kazanma_ölçütleri = [[[0, 0], [1, 0], [2, 0]],
                     [[0, 1], [1, 1], [2, 1]],
                     [[0, 2], [1, 2], [2, 2]],
                     [[0, 0], [0, 1], [0, 2]],
                     [[1, 0], [1, 1], [1, 2]],
                     [[2, 0], [2, 1], [2, 2]],
                     [[0, 0], [1, 1], [2, 2]],
                     [[0, 2], [1, 1], [2, 0]]]

x_durumu = []
o_durumu = []

sıra = 1
while True:
    if sıra % 2 == 0:
        işaret = "X".center(3)
    else:
        işaret = "O".center(3)

    print()
    print("İŞARET: {}\n".format(işaret))

    x = input("yukarıdan aşağıya [1, 2, 3]: ".ljust(30))
    if x == "q":
        break

    y = input("soldan sağa [1, 2, 3]: ".ljust(30))
    if y == "q":
        break

    x = int(x)-1
    y = int(y)-1

    print("\n"*15)
```

```
    if tahta[x][y] == "___":
        tahta[x][y] = işaret
        if işaret == "X".center(3):
            x_durumu += [[x, y]]
        elif işaret == "O".center(3):
            o_durumu += [[x, y]]
        sıra += 1
    else:
        print("\nORASI DOLU! TEKRAR DENEYİN\n")
```

Gördüğünüz gibi epey kod yazmışız. Kodlarımızı topluca incelediğimize göre yazmaya devam edebiliriz:

```
for i in tahta:
    print("\t".expandtabs(30), *i, end="\n"*2)
```

Bu kodların ne işe yaradığınız biliyorsunuz. Oyun tahtasının son durumunu kullanıcıya göstermek için kullanıyoruz bu kodları.

Sıra geldi oyunun en önemli kısmına. Bu noktada oyunu kimin kazandığını belirlememiz gerekiyor. Dikkatlice inceleyin:

```
for i in kazanma_ölçütleri:
    o = [z for z in i if z in o_durumu]
    x = [z for z in i if z in x_durumu]
    if len(o) == len(i):
        print("O KAZANDI!")
        quit()
    if len(x) == len(i):
        print("X KAZANDI!")
        quit()
```

Bu kodları anlayabilmek için en iyi yol uygun yerlere print() fonksiyonları yerleştirerek çıktıları incelemektir. Mesela bu kodları şöyle yazarak *o* ve *x* değişkenlerinin değerlerini izleyebilirsiniz:

```
for i in kazanma_ölçütleri:
    o = [z for z in i if z in o_durumu]
    x = [z for z in i if z in x_durumu]
    print("o: ", o)
    print("x: ", x)
    if len(o) == len(i):
        print("O KAZANDI!")
        quit()
    if len(x) == len(i):
```

```
        print("X KAZANDI!")
        quit()
```

Bu kodlar içindeki en önemli öğeler *o* ve *x* adlı değişkenlerdir. Burada, *o_durumu* veya *x_durumu* adlı listelerdeki değerlerle *kazanma_ölçütleri* adlı listedeki değerleri karşılaştırarak, ortak değerleri *o* veya *x* değişkenlerine yolluyoruz. Eğer ortak öğe sayısı 3'e ulaşırsa (`if len(o) == len(i):` veya `if len(x) == len(i):`), bu sayıyı yakalayan ilk işaret hangisiyse oyunu o kazanmış demektir.

Kodlarımızın son hali şöyle oldu:

```
tahta = [["___", "___", "___"],
         ["___", "___", "___"],
         ["___", "___", "___"]]

print("\n"*15)

for i in tahta:
    print("\t".expandtabs(30), *i, end="\n"*2)

kazanma_ölçütleri = [[[0, 0], [1, 0], [2, 0]],
                     [[0, 1], [1, 1], [2, 1]],
                     [[0, 2], [1, 2], [2, 2]],
                     [[0, 0], [0, 1], [0, 2]],
                     [[1, 0], [1, 1], [1, 2]],
                     [[2, 0], [2, 1], [2, 2]],
                     [[0, 0], [1, 1], [2, 2]],
                     [[0, 2], [1, 1], [2, 0]]]

x_durumu = []
o_durumu = []

sıra = 1
while True:
    if sıra % 2 == 0:
        işaret = "X".center(3)
    else:
        işaret = "O".center(3)

    print()
    print("İŞARET: {}\n".format(işaret))

    x = input("yukarıdan aşağıya [1, 2, 3]: ".ljust(30))
    if x == "q":
```

```
    break

y = input("soldan sağa [1, 2, 3]: ".ljust(30))
if y == "q":
    break

x = int(x)-1
y = int(y)-1

print("\n"*15)

if tahta[x][y] == "___":
    tahta[x][y] = işaret
    if işaret == "X".center(3):
        x_durumu += [[x, y]]
    elif işaret == "O".center(3):
        o_durumu += [[x, y]]
    sıra += 1
else:
    print("\nORASI DOLU! TEKRAR DENEYİN\n")

for i in tahta:
     print("\t".expandtabs(30), *i, end="\n"*2)

for i in kazanma_ölçütleri:
    o = [z for z in i if z in o_durumu]
    x = [z for z in i if z in x_durumu]

    if len(o) == len(i):
        print("O KAZANDI!")
        quit()
    if len(x) == len(i):
        print("X KAZANDI!")
        quit()
```

Gördüğünüz gibi, sadece şu ana kadar öğrendiğimiz bilgileri kullanarak bir oyun yazabilecek duruma geldik. Burada küçük parçaları birleştirerek bir bütüne nasıl ulaştığımızı özellikle görmenizi isterim. Dikkat ederseniz, yukarıdaki programda sadece karakter dizileri, sayılar, listeler ve birkaç fonksiyon var. Nasıl sadece 7 nota ile müzik şaheserleri meydana getirilebiliyorsa, yalnızca 4-5 veri tipi ile de dünyayı ayağa kaldıracak programlar da yazılabilir.

Listeleri temel olarak incelediğimize göre biraz da demetlerden söz edebiliriz.

Demetler

Demetler, özellikle görünüş olarak listelere çok benzeyen bir veri tipidir. Bu veri tipi de, tıpkı listeler gibi, farklı veri tiplerini içinde barındıran kapsayıcı bir veri tipidir.

Demet Tanımlamak

Demet tanımlamanın birkaç farklı yolu vardır. Nasıl karakter dizilerinin ayırt edici özelliği tırnak işaretleri, listelerin ayırt edici özelliği ise köşeli parantez işaretleri ise, demetlerin ayırt edici özelliği de normal parantez işaretleridir. Dolayısıyla bir demet tanımlamak için normal parantez işaretlerinden yararlanacağız:

```
>>> demet = ("ahmet", "mehmet", 23, 45)

>>> type(demet)

<class 'tuple'>
```

Gördüğünüz gibi, karakter dizilerinin type() sorgusuna *str*, listelerin ise *list* cevabı vermesi gibi, demetler de type() sorgusuna *tuple* cevabı veriyor.

Yalnız, dediğimiz gibi Python'da demet tanımlamanın birden fazla yolu vardır. Mesela yukarıdaki demeti şöyle de tanımlayabiliriz:

```
>>> demet = "ahmet", "mehmet", 23, 45
```

Gördüğünüz gibi, parantez işaretlerini kullanmadan, öğeleri yalnızca virgül işareti ile ayırdığımızda da elde ettiğimiz şey bir demet oluyor.

Demet oluşturmak için tuple() adlı bir fonksiyondan da yararlanabilirsiniz. Bu fonksiyon, liste oluşturan list() fonksiyonuna çok benzer:

```
>>> tuple('abcdefg')

('a', 'b', 'c', 'd', 'e', 'f', 'g')
```

Bu fonksiyonu kullanarak başka veri tiplerini demete dönüştürebilirsiniz:

```
>>> tuple(["ahmet", "mehmet", 34, 45])

('ahmet', 'mehmet', 34, 45)
```

Burada, *["ahmet", "mehmet", 34, 45]* adlı bir listeyi tuple() fonksiyonu yardımıyla demete dönüştürdük.

Tek Öğeli bir Demet Tanımlamak

Tek öğeli bir karakter dizisi oluşturabilmek için şu yolu izliyorduk hatırlarsanız:

```
>>> kardiz = 'A'
```

Bu tek öğeli bir karakter dizisidir. Bir de tek öğeli bir liste tanımlayalım:

```
>>> liste = ['ahmet']
```

Bu da tek öğeli bir listedir. Gelin bir de tek öğeli bir demet oluşturmaya çalışalım:

```
>>> demet = ('ahmet')
```

Bu şekilde tek öğeli bir demet oluşturduğunuzu zannediyorsunuz, ama aslında oluşturduğunuz şey basit bir karakter dizisinden ibaret! Gelin kontrol edelim:

```
>>> type(demet)

<class 'str'>
```

Python programlama dilinde tek öğeli bir demet oluşturma işlemi biraz ‘tuhaf’tır. Eğer tek öğeye sahip bir demet oluşturacaksak şöyle bir şey yazmalıyız:

```
>>> demet = ('ahmet',)
```

veya:

```
>>> demet = 'ahmet',
```

Gördüğünüz gibi, tek öğeli bir demet tanımlarken, o tek öğenin yanına bir tane virgül işareti yerleştiriyoruz. Böylece demet tanımlamak isterken, yanlışlıkla alelade bir şekilde ‘ahmet’ adlı bir karakter dizisini ‘demet’ adlı bir değişkene atamamış oluyoruz...

Demetlerin Öğelerine Erişmek

Eğer bir demet içinde yer alan herhangi bir öğeye erişmek isterseniz, karakter dizileri ve listelerden hatırladığınız yöntemi kullanabilirsiniz:

```
>>> demet = ('elma', 'armut', 'kiraz')
>>> demet[0]

'elma'

>>> demet[-1]
```

```
'kiraz'

>>> demet[:2]

('elma', 'armut')
```

Gördüğünüz gibi, daha önce öğrendiğimiz indeksleme ve dilimleme kuralları aynen demetler için de geçerli.

Demetlerle Listelerin Birbirinden Farkı

En başta da söylediğimiz gibi, demetlerle listeler birbirine çok benzer. Ama demetlerle listelerin birbirinden çok önemli bazı farkları da vardır. Bu iki veri tipi arasındaki en önemli fark, listelerin değiştirilebilir (*mutable*) bir veri tipi iken, demetlerin değiştirilemez (*immutable*) bir veri tipi olmasıdır. Yani tıpkı karakter dizileri gibi, demetler de bir kez tanımlandıktan sonra bunların üzerinde değişiklik yapmak mümkün değildir:

```
>>> demet = ('elma', 'armut', 'kiraz')
>>> demet[0] = 'karpuz'

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Gördüğünüz gibi, demetin herhangi bir öğesini değiştirmeye çalıştığımızda Python bize bir hata mesajı gösteriyor.

Bu bakımdan, eğer programın akışı esnasında üzerinde değişiklik yapmayacağınız veya değişiklik yapılmasını istemediğiniz birtakım veriler varsa ve eğer siz bu verileri liste benzeri bir taşıyıcı içine yerleştirmek istiyorsanız, listeler yerine demetleri kullanabilirsiniz. Ayrıca demetler üzerinde işlem yapmak listelere kıyasla daha hızlıdır. Dolayısıyla, performans avantajı nedeniyle de listeler yerine demetleri kullanmak isteyebilirsiniz.

Tahmin edebileceğiniz gibi, tıpkı karakter dizilerinde olduğu gibi, önceden tanımlanmış bir demetin üzerinde değişiklik yapabilmek için, örneğin bir demetle başka bir demeti birleştirmek için o demeti yeniden tanımlamak da mümkündür:

```
>>> demet = ('ahmet', 'mehmet)
>>> demet = demet + ('selin',)
```

Eğer sadece `demet + ('selin',)` demiş olsaydık. özgün demet üzerinde herhangi bir değişiklik yapmış olmayacaktık. Siz bu olguya karakter dizilerinden de aşinasınız. O yüzden, özgün demet üzerinde herhangi bir değişiklik yapabilmek için, daha doğrusu özgün demet üzerinde bir değişiklik yapmış gibi görünebilmek için, özgün demeti sıfırdan tanımlamamız gerekiyor...

Burada ayrıca ‘ahmet’ ve ‘mehmet’ öğelerinden oluşan bir demete ‘selin’ öğesini nasıl eklediğimize de dikkat edin. Asla unutmamalısınız: Python programlama dilinde sadece aynı tür verileri birbiriyle birleştirebilirsiniz. Mesela yukarıdaki örnekte ‘selin’ adlı öğeyi *demet* adlı demete bir karakter dizisi olarak ekleyemezsiniz:

```
>>> demet = demet + 'selin'

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "str") to tuple
```

Bu arada, yukarıdaki kodu şöyle yazdığınızda da aslında bir demetle karakter dizisini birleştirmeye çalışıyor olduğunuza dikkat edin:

```
>>> demet = demet + ('selin')
```

Hatırlarsanız, tek öğeli bir demet tanımlayabilmek için parantez içindeki tek öğenin yanına bir virgül işareti yerleştirmemiz gerekiyordu. Aksi halde demet değil, karakter dizisi tanımlamış oluyorduk. Zaten bir Python programcısı olarak, demetler üzerinde çalışırken en sık yapacağınız hata da demet tanımlamaya çalışırken yanlışlıkla karakter dizisi tanımlamak olacaktır.

Dediğimiz ve yukarıda da örneklerle gösterdiğimiz gibi, bir demeti yeni baştan tanımlayarak da o demet üzerinde değişiklik yapmış etkisi elde edebilirsiniz. Ancak elbette bir araya topladığınız veriler üzerinde sık sık değişiklikler yapacaksanız demetler yerine listeleri tercih etmelisiniz.

Demetlerin Kullanım Alanı

Demetleri ilk öğrendiğinizde bu veri tipi size son derece gereksizmiş gibi gelebilir. Ama aslında oldukça yaygın kullanılan bir veri tipidir bu. Özellikle programların ayar (*conf*) dosyalarında bu veri tipi sıklıkla kullanılır. Örneğin Python tabanlı bir web çatısı (*framework*) olan Django’nun *settings.py* adlı ayar dosyasında pek çok değer bir demet olarak saklanır. Mesela bir Django projesinde web sayfalarının şablonlarını (*template*) hangi dizin altında saklayacağınızı belirlediğiniz ayar şöyle görünür:

```
TEMPLATE_DIRS = ('/home/projects/djprojects/blog/templates',)
```

Burada, şablon dosyalarının hangi dizinde yer alacağını bir demet içinde gösteriyoruz. Bu demet içine birden fazla dizin adı yazabilirdik. Ama biz bütün şablon dosyalarını tek bir dizin altında tutmayı tercih ettiğimiz için tek öğeli bir demet tanımlamışız. Bu arada, daha önce de söylediğimiz gibi, demetlerle ilgili en sık yapacağınız hata, tek öğeli demet tanımlamaya çalışırken aslında yanlışlıkla bir karakter dizisi tanımlamak olacaktır. Örneğin yukarıdaki *TEMPLATE_DIRS* değişkenini şöyle yazsaydık:

```
TEMPLATE_DIRS = ('/home/projects/djprojects/blog/templates')
```

Aslında bir demet değil, alelade bir karakter dizisi tanımlamış olurduk...

başa dön

Listelerin ve Demetlerin Metotları

Listelerin Metotları

Burada, geçen bölümde kaldığımız yerden devam edeceğiz listeleri anlatmaya. Ağırlıklı olarak bu bölümde listelerin metotlarından söz edeceğiz. 'Metot' kavramını karakter dizilerinden hatırlıyorsunuz. Karakter dizilerini anlatırken bol miktarda metot görmüştük.

Python'da bütün veri tipleri bize birtakım metotlar sunar. Bu metotlar yardımıyla, ilgili veri tipi üzerinde önemli değişiklikler veya sorgulamalar yapabiliyoruz.

Hatırlarsanız bir veri tipinin hangi metotlara sahip olduğunu görmek için dir() fonksiyonundan yararlanıyorduk. Listelerde de durum farklı değil. Dolayısıyla şu komut bize listelerin metotlarını sıralayacaktır:

```
>>> dir(list)

['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__doc__'
, '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__',
 '__hash__', '__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__'
, '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '
__str__', '__subclasshook__', 'append', 'count', 'extend', 'index', 'insert',
'pop', 'remove', 'reverse', 'sort']
```

Gördüğünüz gibi, tıpkı karakter dizilerinde olduğu gibi, listelerin metotlarını görmek için de dir() fonksiyonuna parametre olarak veri tipinin teknik adını veriyoruz. Python'da listelerin teknik adı *list* olduğu için bu komutu dir(list) şeklinde kullanıyoruz. Elbette, eğer istersek, listelerin metotlarını almak için herhangi bir listeyi de kullanabiliriz. Mesela boş bir liste kullanalım:

```
>>> dir([])
```

Bu komut da dir(list) ile aynı çıktıyı verecektir. Bu listede bizi ilgilendiren metotlar ise şunlardır:

```
>>> [i for i in dir(list) if not "_" in i]

['append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

Metotlar, bir programcının hayatını önemli ölçüde kolaylaştıran araçlardır. Bu yüzden, 'Listeler' konusunun ilk bölümünde öğrendiğimiz listeye öğe ekleme, öğe çıkarma, öğe değiştirme, öğe silme gibi işlemleri orada anlattığımız yöntemlerle değil, biraz sonra göreceğimiz metotlar aracılığıyla yapmayı tercih edeceğiz. Ama tabii ki, metotları tercih edecek olmamız, birinci bölümde anlattığımız

yöntemleri bir kenara atmanızı gerektirmez. Unutmayın, bir dildeki herhangi bir özelliği siz kullanmasınız bile, etrafta bu özelliği kullanan başka programcılar var. Dolayısıyla en azından başkalarının yazdığı kodları anlayabilmek için dahi olsa, kendinizin kullanmayacağınız yöntem ve yolları öğrenmeniz gerekir.

append() metoduyla başlayalım...

append()

append kelimesi İngilizcede 'eklemek, ilave etmek, iliştirmek' gibi anlamlara gelir. append() metodunun görevi de kelime anlamıyla uyumludur. Bu metodu, bir listeye öğe eklemek için kullanıyoruz. Mesela:

```
>>> liste = ["elma", "armut", "çilek"]
>>> liste.append("erik")
```

Bu metot, yeni öğeyi listenin en sonuna ekler. Mesela yukarıdaki örnekte *"erik"* adlı karakter dizisi listede *"çilek"* adlı karakter dizisinin sağına eklendi.

Hatırlarsanız bir önceki bölümde listeye öğe ekleme işini + işleci ile de yapabileceğimizi söylemiştik. Dolayısıyla, aslında yukarıdaki kodu şöyle de yazabiliriz:

```
>>> liste = ["elma", "armut", "çilek"]
>>> liste = liste + ["erik"]
>>> print(liste)

['elma', 'armut', 'çilek', 'erik']
```

Bu iki yöntem birbiriyle aynı sonucu verse de hem pratiklik hem de işleyiş bakımından bu iki yöntemin birbirinden farklı olduğunu görüyoruz.

Pratiklik açısından bakarsak, append() metodununu kullanmanın + işlecini kullanmaya kıyasla daha kolay olduğunu herhalde kimse reddetmeyecektir. Bu iki yöntem işleyiş bakımından da birbirinden ayrılıyor. Zira + işlecini kullandığımızda listeye yeni bir öğe eklerken aslında *liste* adlı başka bir liste daha oluşturmuş oluyoruz. Hatırlarsanız önceki bölümlerde listelerin değiştirilebilir (*mutable*) veri tipleri olduğunu söylemiştik. İşte append() metodu sayesinde listelerin bu özelliğinden sonuna kadar yararlanabiliyoruz. + işlecini kullandığımızda ise, orijinal listeyi değiştirmek yerine yeni bir liste oluşturduğumuz için, sanki listelere karakter dizisi muamelesi yapmış gibi oluyoruz. Gördüğünüz gibi, listeye append() metodunu uyguladıktan sonra bunu bir değişkene atamamıza gerek kalmıyor. append() metodu orijinal liste üzerinde doğrudan değişiklik yapmamıza izin verdiği için daha az kod yazmamızı ve programımımızın daha performanslı çalışmasını sağlıyor.

+ işleci ile append() metodu işlev olarak birbirine benzese de bu iki yöntem arasında önemli farklılıklar da vardır. Mesela şu örneğe bir göz atalım:

```
işletim_sistemleri = ["Windows", "GNU/Linux", "Mac OS X"]
platformlar = ["IPhone", "Android", "S60"]
hepsi = işletim_sistemleri + platformlar
print(hepsi)

['Windows', 'GNU/Linux', 'Mac OS X', 'IPhone', 'Android', 'S60']
```

Burada iki farklı listeyi, + işleci kullanarak birleştirdik. Aynı işi append() metoduyla şu şekilde yapabiliriz:

```
işletim_sistemleri = ["Windows", "GNU/Linux", "Mac OS X"]
platformlar = ["IPhone", "Android", "S60"]
for i in platformlar:
    işletim_sistemleri.append(i)

print(işletim_sistemleri)
```

Burada *platformlar* adlı liste üzerinde bir for döngüsü kurmamızın nedeni, append() metodunun yalnızca tek bir parametre alabilmesidir. Yani bu metodu kullanarak bir listeye birden fazla öğe ekleyemezsiniz:

```
>>> liste = [1, 2, 3]
>>> liste.append(4, 5, 6)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: append() takes exactly one argument (3 given)
```

Bu sebeple, ekleyeceğiniz listenin öğeleri üzerinde bir for döngüsü kurmanız gerekir:

```
>>> liste = [1, 2, 3]
>>> for i in [4, 5, 6]:
...     liste.append(i)
...
>>> print(liste)

[1, 2, 3, 4, 5, 6]
```

Bir listeye birden fazla öğe eklemek için aklınıza şöyle bir yöntem de gelmiş olabilir:

```
>>> liste = [1, 2, 3]
>>> liste.append([4, 5, 6])
```

Ancak bu komutun çıktısı pek beklediğiniz gibi olmayabilir:

```
>>> print(liste)

[1, 2, 3, [4, 5, 6]]
```

Gördüğünüz gibi, [4, 5, 6] öğesi listeye tek parça olarak eklendi. Eğer istediğiniz şey buysa ne âlâ! Ama değilse, for döngüsü ya da + işleci ile istediğiniz çıktıyı elde edebilirsiniz.

Şöyle bir örnek daha düşünün: Diyelim ki kullanıcının girdiği bütün sayıları birbiriyle çarpan bir uygulama yazmak istiyoruz. Bunun için şöyle bir kod yazabiliriz:

```
sonuç = 1

while True:
    sayı = input("sayı (hesaplamak için q): ")
    if sayı == "q":
        break

    sonuç *= int(sayı)

print(sonuç)
```

Burada kullanıcı her döngüde bir sayı girecek ve programımız girilen bu sayıyı *sonuç* değişkeninin o anki değeriyle çarparak yine *sonuç* değişkenine gönderecek. Böylece kullanıcı tarafından girilen bütün sayıların çarpımını elde etmiş olacağız. Kullanıcının 'q' harfine basmasıyla birlikte de *sonuç* değişkeninin değeri ekranda görünecek. Yalnız burada birkaç sorun var. Diyelim ki kullanıcı hiçbir sayı girmeden 'q' harfine basarsa, *sonuç* değişkeninin *1* olan değeri ekranda görünecek ve bu şekilde kullanıcı yanlış bir sonuç elde etmiş olacak. Ayrıca çarpma işlemi için en az *2* adet sayı gerekiyor. Dolayısıyla kullanıcı *2*'den az sayı girerse de programımız yanlış sonuç verecektir. Kullanıcının yeterli miktarda sayı girip girmediğini tespit edebilmek için yine listelerden ve listelerin append() metodundan yararlanabiliriz:

```
kontrol = []
sonuç = 1

while True:
    sayı = input("sayı (hesaplamak için q): ")
    if sayı == "q":
        break
    kontrol.append(sayı)
    sonuç *= int(sayı)

if len(kontrol) < 2:
    print("Yeterli sayı girilmedi!")
else:
```

```
print(sonuç)
```

Burada önceki koda ilave olarak, *kontrol* adlı boş bir liste tanımladık. Bu liste kullanıcının girdiği sayıları depolayacak. Bir önceki örnekte kullanıcının girdiği sayıları hiçbir yerde depolamadık. Orada yaptığımız şey her döngüde kullanıcı tarafından girilen sayıyı *sonuç* değişkeninin değeriyle çarpıp yine *sonuç* değişkenine göndermekti. Dolayısıyla kullanıcı tarafından girilen sayılar bir yerde tutulmadığı için kaybolup gidiyordu. Burada ise *kontrol* adlı liste, kullanıcı tarafından girilen sayıları tuttuğu için, bu sayıları daha sonra istediğimiz gibi kullanabilme imkanına kavuşuyoruz.

Ayrıca bu ikinci kodlarda *kontrol* değişkeninin boyutuna bakarak kullanıcının *2*'den az sayı girip girmediğini denetliyoruz. Eğer *kontrol* listesinin uzunluğu *2*'den azsa kullanıcı çarpma işlemi için yeterli sayı girmemiş demektir. Böyle bir durumda çarpma işlemini yapmak yerine, kullanıcıya 'Yeterli sayı girilmedi!' şeklinde bir uyarı mesajı gösteriyoruz.

append() metodu listelerin en önemli metotlarından biridir. Hem kendi yazdığınız, hem de başkalarının yazdığı programlarda append() metodunu sıkça göreceksiniz. Dolayısıyla listelerin hiçbir metodunu bilmeseniz bile append() metodunu öğrenmelisiniz.

extend()

extend kelimesi İngilizcede 'genişletmek, yaymak' gibi anlamlara gelir. İşte extend() adlı metot da kelime anlamına uygun olarak listeleri 'genişletir'.

Şöyle bir düşündüğünüzde extend() metodunun append() metoduyla aynı işi yaptığını zannedebilirsiniz. Ama aslında bu iki metot işleyiş olarak birbirinden çok farklıdır.

append() metodunu kullanarak yazdığımız şu koda dikkatlice bakın:

```
li1 = [1, 3, 4]
li2 = [10, 11, 12]
li1. append(li2)

print(li1)
```

append() metodunu anlatırken söylediğimiz gibi, bu metot bir listeye her defasında sadece tek bir öğe eklenmesine izin verir. Yukarıda olduğu gibi, eğer bu metodu kullanarak bir listeye yine bir liste eklemeye çalışırsanız, eklediğiniz liste tek bir öğe olarak eklenecektir. Yani yukarıdaki kodlar size şöyle bir çıktı verecektir:

```
[1, 3, 4, [10, 11, 12]]
```

Gördüğünüz gibi, *[10, 11, 12]* listesi öteki listeye tek bir liste halinde eklendi. İşte extend() metodu bu tür durumlarda işinize yarayabilir. Mesela yukarıdaki örneği bir de extend() metodunu kullanarak yazalım:

```
li1 = [1, 3, 4]
li2 = [10, 11, 12]
li1. extend(li2)

print(li1)
```

Bu defa şöyle bir çıktı alıyoruz:

```
[1, 3, 4, 10, 11, 12]
```

Gördüğünüz gibi, extend() metodu tam da kelime anlamına uygun olarak listeyi yeni öğelerle genişletti.

Hatırlarsanız append() metodunu anlatırken şöyle bir örnek vermiştik:

```
işletim_sistemleri = ["Windows", "GNU/Linux", "Mac OS X"]
platformlar = ["IPhone", "Android", "S60"]
hepsi = işletim_sistemleri + platformlar
print(hepsi)
```

Burada + işlecini kullanarak *işletim_sistemleri* ve *platformlar* adlı listeleri birleştirerek *hepsi* adlı tek bir liste elde ettik. Aynı etkiyi append() metodunu kullanarak şu şekilde elde edebileceğimizi de söylemiştik orada:

```
işletim_sistemleri = ["Windows", "GNU/Linux", "Mac OS X"]
platformlar = ["IPhone", "Android", "S60"]
for i in platformlar:
    işletim_sistemleri.append(i)

print(işletim_sistemleri)
```

Esasında, append() metodunu kullanmaya kıyasla, burada + işlecini kullanmak sanki daha pratikmiş gibi görünüyor. Bir de şuna bakın:

```
işletim_sistemleri = ["Windows", "GNU/Linux", "Mac OS X"]
platformlar = ["IPhone", "Android", "S60"]
işletim_sistemleri.extend(platformlar)
print(işletim_sistemleri)
```

Gördüğünüz gibi, bu örnekte extend() metodunu kullanmak append() metodunu kullanmaya göre daha pratik ve makul. Çünkü bir listeye tek tek öğe eklemek açısından append() metodu daha uygundur, ama eğer yukarıda olduğu gibi bir listeye başka bir liste ekleyeceksek extend() metodunu kullanmayı tercih edebiliriz.

insert()

Bildiğiniz gibi, + işleci, append() ve extend() metotları öğeleri listenin sonuna ekliyor. Peki biz bir öğeyi listenin sonuna değil de, liste içinde başka bir konuma eklemek istersek ne yapacağız? İşte bunun için insert() adlı başka bir metottan yararlanacağız.

insert kelimesi 'yerleştirmek, sokmak' gibi anlamlara gelir. insert() metodu da bu anlama uygun olarak, öğeleri listenin istediğimiz bir konumuna yerleştirir. Dikkatlice inceleyin:

```
>>> liste = ["elma", "armut", "çilek"]
>>> liste.insert(0, "erik")
>>> print(liste)

['erik', 'elma', 'armut', 'çilek']
```

Gördüğünüz gibi insert() metodu iki parametre alıyor. İlk parametre, öğenin hangi konuma yerleştirileceğini, ikinci parametre ise yerleştirilecek öğenin ne olduğunu gösteriyor. Yukarıdaki örnekte *"erik"* öğesini listenin *0.* konumuna, yani listenin en başına yerleştiriyoruz.

ìnsert() metodu özellikle dosya işlemlerinde işinize yarar. Diyelim ki elimizde içeriği şöyle olan *deneme.txt* adlı bir dosya var:

```
Ahmet Özkoparan
Mehmet Veli
Serdar Güzel
Zeynep Güz
```

Bizim amacımız, 'Ahmet Özkoparan' satırından sonra 'Ferhat Yaz' diye bir satır daha eklemek. Yani dosyamızı şu hale getirmek istiyoruz:

```
Ahmet Özkoparan
Ferhat Yaz
Mehmet Veli
Serdar Güzel
Zeynep Güz
```

Biz henüz Python'da dosya işlemlerinin nasıl yapılacağını öğrenmedik. Ama hatırlarsanız bundan önceki bölümlerde birkaç yerde open() adlı bir fonksiyondan bahsetmiş ve bu fonksiyonun dosya işlemlerinde kullanıldığını söylemiştik. Mesela yukarıda bahsettiğimiz *deneme.txt* adlı dosyayı açmak için open() fonksiyonunu şu şekilde kullanabiliriz:

```
f = open("deneme.txt", "r")
```

Burada *deneme.txt* adlı dosyayı okuma modunda açmış olduk. Şimdi dosya içeriğini okuyalım:

```
içerik = f.readlines()
```

Bu satır sayesinde dosya içeriğini bir liste halinde alabildik. Eğer yukarıdaki kodlara şu eklemeyi yaparsanız, dosya içeriğini görebilirsiniz:

```
print(içerik)

['Ahmet Özkoparan\n', 'Mehmet Veli\n', 'Serdar Güzel\n', 'Zeynep Güz\n', '\n']
```

Gördüğünüz gibi, dosya içeriği basit bir listeden ibaret. Dolayısıyla listelerle yapabildiğimiz her şeyi *içerik* adlı değişkenle de yapabiliriz. Yani bu listeye öğe ekleyebilir, listeden öğe çıkarabilir ya da bu listeyi başka bir liste ile birleştirebiliriz.

Dosya içeriğini bir liste olarak aldığımıza göre şimdi bu listeye *"Ahmet Özkoparan"* öğesinden sonra *"Ferhat Yaz"* öğesini ekleyelim. Dikkatlice bakın:

```
içerik.insert(1, "Ferhat Yaz\n")
```

Dediğimiz gibi, f.readlines() satırı bize dosya içeriğini bir liste olarak verdi. Amacımız *"Ahmet Özkoparan"* öğesinden sonra *"Ferhat Yaz"* öğesini eklemek. Bunun için, liste metotlarından biri olan insert() metodunu kullanarak listenin *1.* sırasına *"Ferhat Yaz"* öğesini ekledik. Burada *"Ferhat Yaz"* öğesine *n* adlı satır başı karakterini de ilave ettiğimize dikkat edin. Bu eklemeyi neden yaptığımızı anlamak için satır başı karakterini çıkarmayı deneyebilirsiniz.

içerik adlı değişkenin değerini istediğimiz biçime getirdiğimize göre bu listeyi tekrar *deneme.txt* adlı dosyaya yazabiliriz. Ama bunun için öncelikle *deneme.txt* adlı dosyayı yazma modunda açmamız gerekiyor. Python'da dosyalar ya okuma ya da yazma modunda açılabilir. Okuma modunda açılan bir dosyaya yazılamaz. O yüzden dosyamızı bir de yazma modunda açmamız gerekiyor:

```
g = open("deneme.txt", "w")
```

open() fonksiyonunun ilk parametresi dosya adını gösterirken, ikinci parametresi dosyanın hangi modda açılacağını gösteriyor. Biz burada *deneme.txt* adlı dosyayı yazma modunda açtık. Buradaki *"w"* parametresi İngilizcede 'yazmak' anlamına gelen *write* kelimesinin ilk harfidir. Biraz önce ise *deneme.txt* dosyasını *"r"*, yani okuma (*read*) modunda açmıştık.

Dosyamız artık üzerine yazmaya hazır. Dikkatlice bakın:

```
g.writelines(içerik)
```

Burada, biraz önce istediğimiz biçime getirdiğimiz *içerik* adlı listeyi doğruda dosyaya yazdık. Bu işlem için writelines() adlı özel bir metottan yararlandık. Bu metotları birkaç bölüm sonra ayrıntılı olarak inceleyeceğiz. Biz şimdilik sadece sonuca odaklanalım.

Yapmamız gereken son işlem dosyayı kapatmak olmalı:

```
g.close()
```

Şimdi kodlara topluca bir bakalım:

```
f = open("deneme.txt", "r")
içerik = f.readlines()
içerik.insert(1, "Ferhat Yaz\n")

g = open("deneme.txt", "w")
g.writelines(içerik)
g.close()
```

Gördüğünüz gibi yaptığımız işlem şu basamaklardan oluşuyor:

1. Öncelikle dosyamızı okuma modunda açıyoruz (`f = open("deneme.txt", "r")`)
2. Ardından dosya içeriğini bir liste olarak alıyoruz (`içerik = f.readlines()`)
3. Aldığımız bu listenin *2.* sırasına *"Ferhat Yaz"* öğesini ekliyoruz (`içerik.insert(1, "Ferhat Yaz\n")`)
4. Listeyi istediğimiz şekle getirdikten sonra bu defa dosyamızı yazma modunda açıyoruz (`g = open("deneme.txt", "w")`)
5. Biraz önce düzenlediğimiz listeyi dosyaya yazıyoruz (`g.writelines(içerik)`)
6. Son olarak da, yaptığımız değişikliklerin etkin hale gelebilmesi için dosyamızı kapatıyoruz (`g.close()`)

Burada `insert()` metodunun bize nasıl kolaylık sağladığına dikkat edin. `insert()` metodu da listelerin önemli metotlarından biridir ve dediğimiz gibi, özellikle dosyaları manipüle ederken epey işimize yarar.

remove()

Bu metot listeden öğe silmemizi sağlar. Örneğin:

```
>>> liste = ["elma", "armut", "çilek"]
>>> liste.remove("elma")
>>> liste

['armut', 'çilek']
```

reverse()

Daha önce verdiğimiz örneklerde, liste öğelerini ters çevirmek için dilimleme yöntemini kullanabileceğimizi öğrenmiştik:

```
>>> meyveler = ["elma", "armut", "çilek", "kiraz"]
```

```
>>> meyveler[::-1]

['kiraz', 'çilek', 'armut', 'elma']
```

Eğer istersek, bu iş için, karakter dizilerini incelerken öğrendiğimiz reversed() fonksiyonunu da kullanabiliriz:

```
>>> reversed(meyveler)
```

Bu komut bize şu çıktıyı verir:

```
<list_reverseiterator object at 0x00DC9810>
```

Demek ki reversed() fonksiyonunu bir liste üzerine uyguladığımızda 'list_reverseiterator' adı verilen bir nesne elde ediyoruz. Bu nesnenin içeriğini görmek için birkaç farklı yöntemden yararlanabiliriz. Örneğin:

```
>>> print(*reversed(meyveler))

kiraz çilek armut elma
```

... veya:

```
>>> print(list(reversed(meyveler)))

['kiraz', 'çilek', 'armut', 'elma']
```

... ya da:

```
>>> for i in reversed(meyveler):
...     print(i)
...
kiraz
çilek
armut
elma
```

Gördüğünüz gibi, Python'da bir listeyi ters çevirmenin pek çok yöntemi var. Dilerseniz şimdi bu yöntemlere bir tane daha ekleyelim.

Python'da listelerin öğelerini ters çevirmek için yukarıdaki yöntemlere ek olarak listelerin reverse() metodunu da kullanabilirsiniz:

```
>>> liste = ["elma", "armut", "çilek"]
>>> liste.reverse()
```

```
>>> liste

['çilek', 'armut', 'elma']
```

İhtiyacınız olan çıktının türüne ve şekline göre yukarıdaki yöntemlerden herhangi birini tercih edebilirsiniz.

pop()

Tıpkı remove() metodu gibi, bu metot da bir listeden öğe silmemizi sağlar:

```
>>> liste = ["elma", "armut", "çilek"]
>>> liste.pop()
```

Ancak bu metot, remove() metodundan biraz farklı davranır. pop() metodunu kullanarak bir liste öğesini sildiğimizde, silinen öğe ekrana basılacaktır. Bu metot parametresiz olarak kullanıldığında listenin son öğesini listeden atar. Alternatif olarak, bu metodu bir parametre ile birlikte de kullanabilirsiniz. Örneğin:

```
>>> liste.pop(0)
```

Bu komut listenin *0.* öğesini listeden atar ve atılan öğeyi ekrana basar.

sort()

Yine listelerin önemli bir metodu ile karşı karşıyayız. sort() adlı bu önemli metot bir listenin öğelerini belli bir ölçüte göre sıraya dizmemizi sağlar. Basit bir örnek verelim. Diyelim ki elimizde şöyle bir liste var:

```
üyeler = ['Ahmet', 'Mehmet', 'Ceylan', 'Seyhan', 'Mahmut', 'Zeynep',
         'Abdullah', 'Kadir', 'Kemal', 'Kamil', 'Selin', 'Senem',
         'Sinem', 'Tayfun', 'Tuna', 'Tolga']
```

Bu listedeki isimleri mesela alfabe sırasına dizmek için sort() metodunu kullanabiliriz:

```
>>> liste.sort()
>>> print(liste)

['Abdullah', 'Ahmet', 'Ceylan', 'Kadir', 'Kamil', 'Kemal', 'Mahmut',
 'Mehmet', 'Selin', 'Senem', 'Seyhan', 'Sinem', 'Tayfun', 'Tolga',
 'Tuna', 'Zeynep']
```

Bu metot elbette yalnızca harfleri alfabe sırasına dizmek için değil sayıları sıralamak için de kullanılabilir:

```
>>> sayılar = [1, 0, -1, 4, 10, 3, 6]
>>> sayılar.sort()
>>> print(sayılar)

[-1, 0, 1, 3, 4, 6, 10]
```

Gördüğünüz gibi, sort() metodu öğeleri artan sıralamaya tabi tutuyor. Yani öğeler 'a, b, c' veya 1, 2, 3 şeklinde sıralanıyor. Bunun tersini yapmak da mümkündür. Yani istersek Python'ın sıralama işlemini 'c, b, a' şeklinde yapmasını da sağlayabiliriz. Bunun için sort() metodunun *reverse* parametresini kullanacağız:

```
>>> üyeler = ['Ahmet', 'Mehmet', 'Ceylan', 'Seyhan', 'Mahmut', 'Zeynep',
              'Abdullah', 'Kadir', 'Kemal', 'Kamil', 'Selin', 'Senem',
             'Sinem', 'Tayfun', 'Tuna', 'Tolga']

>>> üyeler.sort(reverse=True)
```

Gördüğünüz gibi sort() metodunun *reverse* adlı bir parametresine verdiğimiz *True* değeri sayesinde liste öğelerini ters sıraladık. Bu parametrenin öntanımlı değeri *False*'tur. Yani sort() metodu öntanımlı olarak öğeleri artıra artıra sıralar. Öğeleri azalta azalta sıralamak için *reverse* parametresinin *False* olan öntanımlı değerini *True* yapmamız yeterli olacaktır.

Gelin isterseniz sort() metodunu kullanarak bir örnek daha verelim. Elimizde şöyle bir liste olsun:

```
>>> isimler = ["Ahmet", "Işık", "İsmail", "Çiğdem", "Can", "Şule"]
```

Bu listedeki isimleri alfabe sırasına dizelim:

```
>>> isimler.sort()
>>> isimler

['Ahmet', 'Can', 'Işık', 'Çiğdem', 'İsmail', 'Şule']
```

Gördüğünüz gibi, çıktı pek beklediğimiz gibi değil. Tıpkı karakter dizilerini anlatırken öğrendiğimiz sorted() fonksiyonunda olduğu gibi, listelerin sort() metodu da Türkçe karakterleri düzgün sıralayamaz. Eğer Türkçe karakterleri sıralamamız gereken bir program yazıyorsak bizim sort() metodunun işleyişine müdahale etmemiz gerekir. Temel olarak, sorted() fonksiyonunu anlatırken söylediklerimiz burada da geçerlidir. Orada bahsettiğimiz locale modülü burada da çoğu durumda işimizi halletmemizi sağlar. Ama sorted() fonksiyonunu anlatırken de söylediğimiz gibi, locale modülü burada da 'i' ve 'ı' harflerini düzgün sıralayamaz. Türkçe harflerin tamamını düzgün sıralayabilmek için şöyle bir kod yazabiliriz:

```
harfler = "abcçdefgğhıijklmnoöprsştuüvyz"
çevrim = {harf: harfler.index(harf) for harf in harfler}
```

```
isimler = ["ahmet", "ışık", "ismail", "çiğdem", "can", "şule"]

isimler.sort(key=lambda x: çevrim.get(x[0]))

print(isimler)
```

Bu kodların bir kısmını anlayabiliyor, bir kısmını ise anlayamıyor olabilirsiniz. Çünkü burada henüz işlemediğimiz konular var. Zamanı geldiğinde bu kodların tamamını anlayabilecek duruma geleceksiniz. Siz şimdilik sadece bu kodlardan ne çıkarabildiğinize bakın yeter. Zaten bizim buradaki amacımız, sort() metodunun Türkçe harfleri de düzgün bir şekilde sıralayabileceğini göstermekten ibarettir.

Bu arada ufak bir uyarı yapmadan geçmeyelim: Yukarıdaki kodlar da esasında Türkçe kelimeleri tam anlamıyla düzgün bir şekilde sıralamak için yeterli değil. 'Gömülü Fonksiyonlar' konusunu incelerken, yeri geldiğinde bu konuya tekrar değinip, Türkçe kelimelerin nasıl doğru, tam ve eksiksiz bir biçimde sıralanacağını da tüm ayrıntılarıyla inceleyeceğiz.

index()

Karakter dizileri konusunu anlatırken bu veri tipinin index() adlı bir metodu olduğundan söz etmiştik hatırlarsanız. İşte liste veri tipinin de index() adında ve karakter dizilerinin index() metoduyla aynı işi yapan bir metodu bulunur. Bu metot bir liste öğesinin liste içindeki konumunu söyler bize:

```
>>> liste = ["elma", "armut", "çilek"]
>>> liste.index("elma")

0
```

Karakter dizilerinin index() metoduyla ilgili söylediğimiz her şey listelerin index() metodu için de geçerlidir.

count()

Karakter dizileri ile listelerin ortak metotlarından biri de count() metodudur. Tıpkı karakter dizilerinde olduğu gibi, listelerin count() metodu da bir öğenin o veri tipi içinde kaç kez geçtiğini söyler:

```
>>> liste = ["elma", "armut", "elma", "çilek"]
>>> liste.count("elma")

2
```

Karakter dizilerinin count() metoduyla ilgili söylediğimiz her şey listelerin count() metodu için de geçerlidir.

Demetlerin Metotları

Listelerin metotlarını incelediğimize göre, artık demetlerin metotlarına bakabiliriz.

Geçen bölümde de söylediğimiz gibi, listeler ve demetler birbirine benzer. Aralarındaki en önemli fark, listelerin değiştirilebilir bir veri tipi iken, demetlerin değiştirilemez bir veri tipi olmasıdır. Elbette bu fark, iki veri tipinin metotlarında da kendini gösterir. Demetler üzerinde değişiklik yapamadığımız için, bu veri tipi değişiklik yapmaya yarayan metotlara sahip değildir.

Demetlerin hangi metotları olduğunu şu komutla görebilirsiniz:

```
>>> dir(tuple)
```

Gördüğünüz gibi, bu veri tipinin bizi ilgilendiren iki metodu var:

1. index()
2. count()

index()

Listeler ve karakter dizileri konusunu anlatırken bu veri tiplerinin index() adlı bir metodu olduğundan söz etmiştik hatırlarsanız. İşte demet veri tipinin de index() adında ve listelerle karakter dizilerinin index() metoduyla aynı işi yapan bir metodu bulunur. Bu metot bir demet öğesinin demet içindeki konumunu söyler bize:

```
>>> demet = ("elma", "armut", "çilek")
>>> demet.index("elma")

0
```

Listelerin ve karakter dizilerinin index() metoduyla ilgili söylediğimiz her şey demetlerin index() metodu için de geçerlidir.

count()

Karakter dizileri, listeler ve demetlerin ortak metotlarından biri de count() metodudur. Tıpkı karakter dizileri ve listelerde olduğu gibi, demetlerin count() metodu da bir öğenin o veri tipi içinde kaç kez geçtiğini söyler:

```
>>> demet = ("elma", "armut", "elma", "çilek")
>>> demet.count("elma")
```

```
2
```

Karakter dizilerinin ve listelerin count() metoduyla ilgili söylediğimiz her şeydemetlerin count() metodu için de geçerlidir.

başa dön

Sayma Sistemleri

Sayılar olmadan bilgisayar ve programlama düşünülemez. O yüzden, önceki derslerimizde karakter dizilerini anlatırken şöyle bir değinip geçtiğimiz sayılar konusunu, sayma sistemleri konusunu da ilave ederek, birer programcı adayı olan bizleri yakından ilgilendirdiği için mümkün olduğunca ayrıntılı bir şekilde ele almaya çalışacağız.

Sayılar ve Sayma Sistemleri konusunu iki farklı bölümde inceleyeceğiz.

Sayılar konusunun temelini oluşturduğu için, öncelikle sayma sistemlerinden söz edelim.

Öncelikle ‘sayma sistemi’ kavramını tanımlayarak işe başlayalım. Nedir bu ‘sayma sistemi’ denen şey?

Sayma işleminin hangi ölçütlere göre yapılacağını belirleyen kurallar bütününe sayma sistemi adı verilir.

Dünyada yaygın olarak kullanılan dört farklı sayma sistemi vardır. Bunlar, onlu, sekizli, on altılı ve ikili sayma sistemleridir. Bu dördü arasında en yaygın kullanılan sayma sistemi ise, tabii ki, onlu sistemdir. İnsanların elleri ve ayaklarında on parmak olduğunu düşünürsek, bu sistemin neden daha yaygın kullanıldığını anlamak aslında hiç de zor değil!

Onlu sistemin yaygınlığını düşünerek, sayma sistemleri konusunu anlatmaya onlu sayma sisteminden başlayalım.

Onlu Sayma Sistemi

Biz insanlar genellikle hesap işlemleri için onlu sayma sistemini kullanırız. Hepinizin bildiği gibi bu sistem; *0*, *1*, *2*, *3*, *4*, *5*, *6*, *7*, *8* ve *9* olmak üzere toplam on rakamdan oluşur. Yani sayıları gösteren, birbirinden farklı toplam on simge (rakam) vardır bu sistemde. Bu on simgeyi kullanarak, olası bütün sayıları gösterebiliriz.

Bu arada terminoloji ile ilgili ufak bir açıklama yapalım:

Rakamlar, sayıları göstermeye yarayan simgelerdir. Onlu sayma sisteminde toplam on farklı rakam vardır. Bütün rakamlar birer sayıdır, ama bütün sayılar birer rakam değildir. Örneğin *8* hem bir rakam hem de bir sayıdır. Ancak mesela *32* bir sayı olup bu sayı, *3* ve *2* adlı iki farklı rakamın bir araya getirilmesi ile gösterilir. Yani *32* sayısı tek başına bir rakam değildir.

Açıklamamızı da yaptığımıza göre yolumuza devam edebiliriz.

İnsanlar yukarıda bahsettiğimiz bu onlu sisteme ve bu sistemi oluşturan rakamlara/simgelere o kadar alışmıştır ki, çoğu zaman başka bir sistemin varlığından veya var olma olasılığından haberdar bile değildir.

Ama elbette dünya üzerindeki tek sayma sistemi onlu sistem olmadığı gibi, sayıları göstermek için kullanılabilecek rakamlar da yukarıdakilerle sınırlı değildir.

Nihayetinde rakam dediğimiz şeyler insan icadı birtakım simgelerden ibarettir. Elbette doğada '2' veya '7' diye bir şey bulunmaz. Bizim yaygın olarak yukarıdaki şekilde gösterdiğimiz rakamlar Arap rakamlarıdır. Mesela Romalılar yukarıdakiler yerine I, II, III, IV, V, VI, VII, VIII, IX ve X gibi farklı simgeler kullanıyordu... Neticede *2* ve *II* aynı kavrama işaret ediyor. Sadece kullanılan simgeler birbirinden farklı, o kadar.

Onlu sayma sisteminde bir sayıyı oluşturan rakamlar *10*'un kuvvetleri olarak hesaplanır. Örneğin *1980* sayısını ele alalım. Bu sayıyı *10*'un kuvvetlerini kullanarak şu şekilde hesaplayabiliriz:

```
>>> (0 * (10 ** 0)) + (8 * (10 ** 1)) + (9 * (10 ** 2)) + (1 * (10 ** 3))

1980
```

Gördüğünüz gibi, sayının en sağındaki basamak *10*'un *0.* kuvveti olacak şekilde, sola doğru kuvveti artırarak ilerliyoruz.

Gelelim öteki sayma sistemlerine...

Sekizli Sayma Sistemi

Onlu sayma sisteminin aksine sekizli sayma sisteminde toplam sekiz rakam bulunur. Bu rakamlar şunlardır:

0, 1, 2, 3, 4, 5, 6, 7

Gördüğünüz gibi, onlu sistemde toplam on farklı simge varken, sekizli sistemde toplam sekiz farklı simge var.

Bu bölümün en başında da söylediğimiz gibi, insanlar onlu sayma sistemine ve bu sistemi oluşturan simgelere o kadar alışmıştır ki, çoğu zaman başka bir sistemin varlığından veya var olma olasılığından haberdar bile değildir. Hatta başka sayma sistemlerinden bir vesileyle haberdar olup, bu sistemleri öğrenmeye çalışanlar onlu sayma sistemine olan alışkanlıkları nedeniyle yeni sayma sistemlerini anlamakta dahi zorluk çekebilirler. Bunun birincil nedeni, iyi tanıdıklarını zannettikleri onlu sistemi de aslında o kadar iyi tanımıyor olmalarıdır.

O halde başka sayma sistemlerini daha iyi anlayabilmek için öncelikle yaygın olarak kullandığımız sayma sisteminin nasıl işlediğini anlamaya çalışalım:

Onlu sistemde toplam on farklı simge bulunur:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

9'dan büyük bir sayıyı göstermek gerektiğinde simge listesinin en başına dönülür ve basamak sayısı

bir artırılarak, semboller birleştirilir:

10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ..., 99, 100, ..., 999, 1000

İşte bu kural öteki sayma sistemleri için de geçerlidir. Mesela sekizli sayma sistemini ele alalım.

Dediğimiz gibi, sekizli sistemde toplam sekiz farklı simge bulunur:

0, 1, 2, 3, 4, 5, 6, 7

Bu sistemde *7*'den büyük bir sayıyı göstermek gerektiğinde, tıpkı onlu sistemde olduğu gibi, simge listesinin en başına dönüyoruz ve basamak sayısını bir artırarak sembolleri birleştiriyoruz:

10, 11, 12, 13, 14, 15, 16, 17, 20, ..., 77, 100

Onlu sayma sistemi ile sekizli sayma sistemi arasındaki farkı daha belirgin bir şekilde görebilmek için şu kodları yazalım:

```
sayı_sistemleri = ["onlu", "sekizli"]

print(("{:^5} "*len(sayı_sistemleri)).format(*sayı_sistemleri))

for i in range(17):
    print("{0:^5} {0:^5o}".format(i))
```

Bu kodlarda öğrenmediğimiz ve anlayamayacağımız hiçbir şey yok. Bu kodları oluşturan bütün parçaları önceki derslerimizde ayrıntılı olarak incelemiştik.

Bu kodlardan şöyle bir çıktı alacağız:

```
onlu  sekizli
 0     0
 1     1
 2     2
 3     3
 4     4
 5     5
 6     6
 7     7
 8    10
 9    11
10    12
11    13
12    14
13    15
```

```
14   16
15   17
16   20
```

Gördüğünüz gibi, onlu sistemde elimizde toplam on farklı simge olduğu için, elimizdeki simgeleri kullanarak 10. sayıya kadar ilerleyebiliyoruz. Bu noktadan sonra simge stoğumuz tükendiği için en başa dönüp bir basamak artırıyoruz ve simgeleri birbiriyle birleştirerek yeni sayılar elde ediyoruz.

Sekizli sistemde ise elimizde yalnızca sekiz farklı simge olduğu için, elimizdeki simgeleri kullanarak ancak 8. sayıya kadar gelebiliyoruz. Öteki sayıları gösterebilmek için bu noktadan sonra başa dönüp bir artırmamız ve simgeleri birbiriyle birleştirerek yeni sayılar elde etmemiz gerekiyor.

Sekizli sayma sisteminde bir sayıyı oluşturan rakamlar *8*'in kuvvetleri olarak hesaplanır. Örneğin sekizli sayma sistemindeki *3674* sayısını ele alalım. Bu sayıyı *8*'in kuvvetlerini kullanarak şu şekilde hesaplayabiliriz:

```
>>> (4 * (8 ** 0)) + (7 * (8 ** 1)) + (6 * (8 ** 2)) + (3 * (8 ** 3))

1980
```

Bu hesaplama şeklini onlu sayma sisteminden hatırlıyor olmalısınız. Gördüğünüz gibi, sekizli sistemdeki bir sayının her bir basamağını *8*'in kuvvetleri olarak hesapladığımızda, bu sayının onlu sistemdeki karşılığını elde ediyoruz.

On Altılı Sayma Sistemi

Şu ana kadar onlu ve sekizli sayma sistemlerinden bahsettik. Önemli bir başka sayma sistemi de on altılı sayma sistemidir.

Onlu sayma sisteminde on farklı rakam, sekizli sayma sisteminde sekiz farklı rakam olduğunu öğrenmiştik. On altılı sayma sisteminde ise, tahmin edebileceğiniz gibi, on altı farklı rakam bulunur:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f

Şimdiye kadar öğrenmiş olduğumuz sayma sistemleri arasındaki farkı daha net görmek için biraz önce yazdığımız kodlara on altılı sayma sistemini de ekleyelim:

```
sayı_sistemleri = ["onlu", "sekizli", "on altılı"]

print(("{:^8} "*len(sayı_sistemleri)).format(*sayı_sistemleri))

for i in range(17):
    print("{0:^8} {0:^8o} {0:^8x}".format(i))
```

Buradan şöyle bir çıktı alacağız:

onlu	sekizli	on altılı
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	10	8
9	11	9
10	12	a
11	13	b
12	14	c
13	15	d
14	16	e
15	17	f
16	20	10

Gördüğünüz gibi, onlu sistemde birbirinden farklı toplam *10* adet rakam/simge varken, sekizli sistemde toplam *8* farklı simge, on altılı sistemde ise toplam *16* farklı simge var. Yani onlu sistemde olası bütün sayılar eldeki *10* farklı simge ve bunların kombinasyonunun kullanılması yoluyla; sekizli sistemde *8* farklı simge ve bunların kombinasyonunun kullanılması yoluyla; on altılı sistemde ise *16* farklı simge ve bunların kombinasyonunun kullanılması yoluyla gösteriliyor. Bu sebeple onlu sistemde *9* sayısından itibaren bir basamak artırılıp simge listesinin başına dönülürken, sekizli sistemde *7* sayısından itibaren; on altılı sistemde ise *f* sayısından itibaren başa dönülüyor.

On altılı sistemde *9* sayısından sonra gelen harfleri yadırgamış olabilirsiniz. Bu durumu şöyle düşünün: Sayı dediğimiz şeyler insan icadı birtakım simgelerden ibarettir. Daha önce de söylediğimiz gibi, doğada ‘2’ veya ‘7’ diye bir şey göremezsiniz...

İşte on altılık sistemdeki sayıları gösterebilmek için de birtakım simgelere ihtiyaç var. İlk on simge, onluk sayma sistemindekilerle aynı. Ancak *10*‘dan sonraki sayıları gösterebilmek için elimizde başka simge yok. On altılık sistemi tasarlayanlar, bir tercih sonucu olarak, eksik sembolleri alfabe harfleriyle tamamlamayı tercih etmişler. Alfabe harfleri yerine pekala Roma rakamlarını da tercih edebilirlerdi. Eğer bu sistemi tasarlayanlar böyle tercih etmiş olsaydı bugün on altılık sistemi şöyle gösteriyor olabilirdik:

```
0
1
2
3
4
5
```

```
6
7
8
9
I
II
III
IV
V
VI
```

Bugün bu sayıları bu şekilde kullanmıyor olmamızın tek sebebi, sistemi tasarlayanların bunu böyle tercih etmemiş olmasıdır...

On altılı sayma sisteminde bir sayıyı oluşturan rakamlar *16*'nın kuvvetleri olarak hesaplanır. Peki ama bu sayma sistemindeki *a*, *b*, *c*, *d*, *e* ve *f* harfleriyle nasıl aritmetik işlem yapacağız? Örneğin on altılı sayma sistemindeki *7bc* sayısını ele alalım. Bu sayının onlu sistemdeki karşılığını *16*'nın kuvvetlerini kullanarak hesaplayabiliriz hesaplamasına, ama peki yukarıda bahsettiğimiz harfler ne olacak? Yani şöyle bir işlem tabii ki mümkün değil:

```
>>> ((c * 16 ** 0)) + ((b * 16 ** 1)) + ((7 * 16 ** 2))
```

Elbette *c* ve *b* sayılarını herhangi bir aritmetik işlemde kullanamayız. Bunun yerine, bu harflerin onlu sistemdeki karşılıklarını kullanacağız:

a –> 10

b –> 11

c –> 12

d –> 13

e –> 14

f –> 15

Buna göre:

```
>>> ((12 * (16 ** 0)) + ((11 * (16 ** 1)) + ((7 * (16 ** 2))

1980
```

Demek ki on altılı sistemdeki '7bc' sayısının onlu sistemdeki karşılığı *1980*'miş.

İkili Sayma Sistemi

Bildiğiniz, veya orada burada duymuş olabileceğiniz gibi, bilgisayarların temelinde iki tane sayı vardır: *0* ve *1*. Bilgisayarlar bütün işlemleri sadece bu iki sayı ile yerine getirir.

Onlu, sekizli ve on altılı sayı sistemleri dışında, özellikle bilgisayarların altyapısında tercih edilen bir başka sayı sistemi daha bulunur. İşte bu sistemin adı ikili (*binary*) sayı sistemidir. Nasıl onlu sistemde *10*, sekizli sistemde *8*, on altılı sistemde ise sayıları gösteren *16* farklı simge varsa, bu sayı sisteminde de sayıları gösteren toplam iki farklı sembol vardır: *0* ve *1*.

İkili sayı sisteminde olası bütün sayılar işte bu iki simge ile gösterilir.

Gelin isterseniz durumu daha net bir şekilde görebilmek için yukarıda verdiğimiz sayı sistemi tablosuna ikili sayıları da ekleyelim:

```
sayı_sistemleri = ["onlu", "sekizli", "on altılı", "ikili"]

print(("{:^9} "*len(sayı_sistemleri)).format(*sayı_sistemleri))

for i in range(17):
    print("{0:^9} {0:^9o} {0:^9x} {0:^9b}".format(i))
```

Bu kodlar şu çıktıyı verecektir:

```
onlu     sekizli  on altılı   ikili
 0         0         0         0
 1         1         1         1
 2         2         2        10
 3         3         3        11
 4         4         4       100
 5         5         5       101
 6         6         6       110
 7         7         7       111
 8        10         8      1000
 9        11         9      1001
 10       12         a      1010
 11       13         b      1011
 12       14         c      1100
 13       15         d      1101
 14       16         e      1110
 15       17         f      1111
 16       20        10     10000
```

Burada, onlu, sekizli ve on altılı sayı sistemleri için geçerli olan durumun aynen ikili sayı sistemi için de geçerli olduğunu rahatlıkla görebiliyoruz. İkili sayı sistemindeki mevcut sayıları gösterebilmemiz için toplam iki farklı simge var. Bunlar: *0* ve *1*. İkili sayı sisteminde *0* ve *1* diye saymaya başlayıp üçüncü sayıyı söylememiz gerektiğinde, elimizde *0* ve *1*'den başka simge olmadığı için bir basamak

artırıp simge listesinin başına dönüyoruz ve böylece onluk düzendeki *2* sayısını ikili düzende gösterebilmek için *0* ve *1*'den sonra *10* simgesini kullanıyoruz.

Bu söylediklerimizden sonra İnternet üzerinde sıkça karşılaştığınız şu sözün anlamını herhalde artık daha iyi anlıyor olmalısınız:

> İnsanlar 10'a ayrılır: İkili sistemi bilenler ve bilmeyenler!

Bu arada, elbette ikili düzendeki *10* sayısı 'on' şeklinde telaffuz edilmiyor. Bu sayıyı "bir-sıfır" diye seslendiriyoruz...

İkili sayma sisteminde bir sayıyı oluşturan rakamlar *2*'nin kuvvetleri olarak hesaplanır. Örneğin ikili sayma sistemindeki *1100* sayısını ele alalım. Bu sayıyı *2*'nin kuvvetlerini kullanarak şu şekilde hesaplayabiliriz:

```
>>> (0 * (2 ** 0)) + (0 * (2 ** 1)) + (1 * (2 ** 2)) + (1 * (2 ** 3))

12
```

Demek ki '1100' sayısı onlu sistemde *12* sayısına karşılık geliyormuş.

Sayma Sistemlerini Birbirine Dönüştürme

Sıklıkla kullanılan dört farklı sayma sistemini öğrendik. Peki biz bir sayma sisteminden öbürüne dönüştürme işlemi yapmak istersek ne olacak? Örneğin onlu sistemdeki bir sayıyı ikili sisteme nasıl çevireceğiz?

Python programlama dilinde bu tür işlemleri kolaylıkla yapmamızı sağlayan birtakım fonksiyonlar bulunur. Ayrıca özel fonksiyonları kullanmanın yanısıra karakter dizisi biçimlendirme (*string formatting*) yöntemlerini kullanarak da sayma sistemlerini birbirine dönüştürebiliriz. Biz burada her iki yöntemi de tek tek inceleyeceğiz.

Gelin isterseniz bu dönüştürme işlemleri için hangi özel fonksiyonların olduğuna bakalım önce.

Fonksiyon Kullanarak

bin()

Bu fonksiyon bir sayının ikili (*binary*) sayı sistemindeki karşılığını verir:

```
>>> bin(2)

'0b10'
```

Bu fonksiyonun çıktı olarak bir karakter dizisi verdiğine dikkat edin. Bu karakter dizisinin ilk iki

karakteri (*'0b'*), o sayının ikili sisteme ait bir sayı olduğunu gösteren bir işarettir. Bu bilgilerden yola çıkarak, yukarıdaki karakter dizisinin gerçek ikili kısmını almak için şu yöntemi kullanabilirsiniz:

```
>>> bin(2)[2:]

'10'
```

hex()

Bu fonksiyon, herhangi bir sayıyı alıp, o sayının on altılı sistemdeki karşılığını verir:

```
>>> hex(10)

'Oxa'
```

Tıpkı bin() fonksiyonunda olduğu gibi, hex() fonksiyonunun da çıktı olarak bir karakter dizisi verdiğine dikkat edin. Hatırlarsanız bin() fonksiyonunun çıktısındaki ilk iki karakter (*0b*), o sayının ikili sisteme ait bir sayı olduğunu gösteren bir işaret olarak kullanılıyordu. hex() fonksiyonunun çıktısındaki ilk iki karakter de (*0x*), o sayının on altılı sisteme ait bir sayı olduğunu gösteriyor.

oct()

Bu fonksiyon, herhangi bir sayıyı alıp, o sayının sekizli sistemdeki karşılığını verir:

```
>>> oct(10)

'0o12'
```

Tıpkı bin() ve hex() fonksiyonlarında olduğu gibi, oct() fonksiyonunun da çıktı olarak bir karakter dizisi verdiğine dikkat edin. Hatırlarsanız bin() ve hex() fonksiyonlarının çıktısındaki ilk iki karakter (*0b* ve *0x*), o sayıların hangi sisteme ait sayılar olduğunu gösteriyordu. Aynı şekilde oct() fonksiyonunun çıktısındaki ilk iki karakter de (*0o*), o sayının sekizli sisteme ait bir sayı olduğunu gösteriyor.

int()

Aslında biz bu fonksiyonu yakından tanıyoruz. Bildiğiniz gibi bu fonksiyon herhangi bir sayı veya sayı değerli karakter dizisini tam sayıya (*integer*) dönüştürmek için kullanılıyor. int() fonksiyonunun şimdiye kadar gördüğümüz işlevi dışında bir işlevi daha bulunur: Biz bu fonksiyonu kullanarak herhangi bir sayıyı onlu sistemdeki karşılığına dönüştürebiliriz:

```
>>> int('7bc', 16)
```

```
1980
```

Gördüğünüz gibi, bu fonksiyonu kullanırken dikkat etmemiz gereken bazı noktalar var. İlkin, eğer int() fonksiyonunu yukarıdaki gibi bir dönüştürme işlemi için kullanacaksak, bu fonksiyona verdiğimiz ilk parametrenin bir karakter dizisi olması gerekiyor. Dikkat etmemiz gereken ikinci nokta, int() fonksiyonuna verdiğimiz ikinci parametrenin niteliği. Bu parametre, dönüştürmek istediğimiz sayının hangi tabanda olduğunu gösteriyor. Yukarıdaki örneğe göre biz, on altı tabanındaki *7bc* sayısını on tabanına dönüştürmek istiyoruz.

Bir de şu örneklere bakalım:

```
>>> int('1100', 2)

12

>>> int('1100', 16)

4352
```

İlk örnekte, ikili sistemdeki *1100* sayısını onlu sisteme çeviriyoruz ve *12* sayısını elde ediyoruz. İkinci örnekte ise on altılı sistemdeki *1100* sayısını onlu sisteme çeviriyoruz ve *4352* sayısını elde ediyoruz.

Biçimlendirme Yoluyla

Esasında biz karakter dizisi biçimlendirme yöntemlerini kullanarak dönüştürme işlemlerini nasıl gerçekleştireceğimizi biliyoruz. Biz burada zaten öğrendiğimiz bu bilgileri tekrar ederek öğrendiklerimizi pekiştirme amacı güdeceğiz.

b

Bu karakteri kullanarak bir sayıyı ikili düzendeki karşılığına dönüştürebiliriz:

```
>>> '{:b}'.format(12)

'1100'
```

Bu karakter, bin() fonksiyonuyla aynı işi yapar.

x

Bu karakteri kullanarak bir sayıyı on altılı düzendeki karşılığına dönüştürebiliriz:

```
>>> '{:x}'.format(1980)
```

```
'7bc'
```

Bu karakter, hex() fonksiyonuyla aynı işi yapar.

o

Bu karakteri kullanarak bir sayıyı sekizli düzendeki karşılığına dönüştürebiliriz:

```
>>> ':o'.format(1980)

'3674'
```

Bu karakter, oct() fonksiyonuyla aynı işi yapar.

Bütün bu anlattıklarımızdan sonra (eğer o zaman anlamakta zorluk çekmişseniz) aşağıdaki kodları daha iyi anlamış olmalısınız:

```
sayı_sistemleri = ["onlu", "sekizli", "on altılı", "ikili"]

print(("{:^9} "*len(sayı_sistemleri)).format(*sayı_sistemleri))

for i in range(17):
    print("{0:^9} {0:^9o} {0:^9x} {0:^9b}".format(i))
```

Bu arada, yukarıda bir sayının, karakter dizisi biçimlendirme yöntemleri kullanılarak ikili, sekizli ve on altılı düzene nasıl çevrileceğini gördük. Bir sayıyı onlu düzene çevirmek için ise sadece int() fonksiyonunu kullanabiliyoruz. Böyle bir çevirme işlemini karakter dizisi biçimlendirme yöntemlerini kullanarak yapamıyoruz. Ama elbette, eğer başka bir sayma sisteminden onlu sisteme çevirdiğiniz bir sayıyı herhangi bir karakter dizisi içinde biçimlendirmek isterseniz şöyle bir kod kullanabilirsiniz:

```
>>> n = '7bc'
>>> "{} sayısının onlu karşılığı {:d} sayısıdır.".format(n, int(n, 16))
```

...veya:

```
>>> n = '7bc'
>>> "{} sayısının onlu karşılığı {} sayısıdır.".format(n, int(n, 16))
```

Zira bildiğiniz gibi, Python'da onlu sayıları temsil eden harf *d* harfidir. Eğer *{}* yapısı içinde herhangi bir harf kullanmazsanız yukarıdaki durumda Python *{:d}* yazmışsınız gibi davranacaktır.

Sayma Sistemlerinin Birbirlerine Karşı Avantajları

Böylece dört farklı sayı sisteminin hangi mantık üzerine işlediğini anlamış olduk. Ayrıca sayı sistemleri arasında dönüştürme işlemlerini de öğrendik.

İşte bilgisayarlar bu sayı sistemleri arasında sadece ikili sayı sistemini 'anlayabilir'. Aslında bu da hiç mantıksız değil. Bilgisayar dediğimiz şey, üzerinden elektrik geçen devrelerden ibaret bir makinedir. Eğer bir devrede elektrik yoksa o devrenin değeri ~0 volt iken, o devreden elektrik geçtiğinde devrenin değeri ~5 volttur. Gördüğünüz gibi, ortada iki farklı değer var: ~0 volt ve ~5 volt. Yukarıda anlattığımız gibi, ikili (*binary*) sayma sisteminde de iki değer bulunur: *0* ve *1*. Dolayısıyla ikili sayma sistemi bilgisayarın iç işleyişine en uygun sistemdir. ikili sistemde ~0 volt'u 0 ile, ~5 volt'u ise *1* ile temsil edebiliyoruz. Yani devreden elektrik geçtiğinde o devrenin değeri *1*, elektrik geçmediğinde ise *0* olmuş oluyor. Tabii bilgisayar açısından bakıldığında devrede elektrik vardır veya yoktur. Biz insanlar bu ikili durumu daha kolay bir şekilde temsil edebilmek için her bir duruma *0* ve *1* gibi bir ad veriyoruz.

Bilgisayarın işlemcisi sadece bu iki farklı durumu kullanarak her türlü hesaplama işlemini gerçekleştirebilir. Bu sebeple ikili sayı sistemi bilgisayarın çalışma mantığı için gayet yeterli ve uygundur. İkili sayı sistemi yerine mesela onlu sayı sistemini kullanmak herhalde simge israfından başka bir şey olmazdı. Neticede, dediğimiz gibi, bilgisayarın işleyebilmesi için iki farklı simge yeterlidir.

Dediğimiz gibi, ikili sayma sistemi bilgisayarın yapısına gayet uygundur. Ama biz insanlar açısından sadece iki simge yardımıyla saymaya çalışmak epey zor olacaktır. Ayrıca sayı büyüdükçe, ikili sistemde sayının kapladığı alan hızla ve kolayca artacak, yığılan bu sayıları idare etmek hiç de kolay olmayacaktır. İşte bu noktada devreye on altılı (*hexadecimal*) sayılar girer. Bu sayma sisteminde toplam *16* farklı rakam/simge olduğu için, büyük sayılar çok daha az yer kaplayacak şekilde gösterilebilir.

Bildiğiniz gibi, ikili sayma sistemindeki herbir basamağa 'bit' adı verilir. İkili sayma sistemini kullanarak, *0*'dan *256*'ya kadar sayabilmek için toplam *7* bitlik (yani *7* hanelik) bir yer kullanmanız gerekir. On altılı sistemde ise bu işlemi sadece iki basamakla halledebilirsiniz. Yani on altılı sistemde 00 ile FF arasına toplam 255 tane sayı sığdırılabilir. Dolayısıyla on altılı sistemi kullanarak, çok büyük sayıları çok az yer kullanarak gösterebilirsiniz:

```
>>> for i in range(256):
...     print(i, bin(i)[2:], hex(i)[2:])
...
0 0 0
(...)
255 11111111 ff
>>>
```

Gördüğünüz gibi, onlu sistemde *255* şeklinde, ikili sistemde ise *11111111* şeklinde gösterilen sayı on altılı sistemde yalnızca *ff* şeklinde gösterilebiliyor. Dolayısıyla, kullanım açısından, biz insanlar için on altılık sayma sisteminin ikili sisteme kıyasla çok daha pratik bir yöntem olduğunu söyleyebiliriz.

Ayrıca on altılı sistem, az alana çok veri sığdırabilme özelliği nedeniyle HTML renk kodlarının

gösterilmesinde de tercih edilir. Örneğin beyaz rengi temsil etmek için on altılı sistemdeki *#FFFFFF* ifadesini kullanmak *rgb(255,255,255)* ifadesini kullanmaya kıyasla çok daha mantıklıdır. Hatta *#FFFFFF* ifadesini *#FFF* şeklinde kısaltma imkanı dahi vardır.

başa dön

Sayılar

Geçen bölümde sayma sistemlerini ayrıntılı bir şekilde inceledik. Bu bölümde ise yine bununla bağlantılı bir konu olan sayılar konusunu ele alacağız. Esasında biz sayıların ne olduğuna ve Python'da bunların nasıl kullanılacağına dair tamamen bilgisiz değiliz. Buraya gelene kadar, sayılar konusunda epey şey söyledik aslında. Mesela biz Python'da üç tür sayı olduğunu biliyoruz:

1. Tam Sayılar (*integers*)
2. Kayan Noktalı Sayılar (*floating point numbers* veya kısaca *floats*)
3. Karmaşık Sayılar (*complex numbers*)

Eğer bir veri `type(veri)` sorgulamasına *int* cevabı veriyorsa o veri bir tam sayıdır. Eğer bir veri `type(veri)` sorgulamasına *float* cevabı veriyorsa o veri bir kayan noktalı sayıdır. Eğer bir veri `type(veri)` sorgulamasına *complex* cevabını veriyorsa o veri bir karmaşık sayıdır.

Mesela şunlar birer tam sayıdır:

15, 4, 33

Şunlar birer kayan noktalı sayıdır:

3.5, 6.6, 2.3

Şunlarsa birer karmaşık sayıdır:

3+3j, 5+2j, 19+10j

Ayrıca şimdiye kadar öğrendiklerimiz sayesinde bu sayıların çeşitli fonksiyonlar yardımıyla birbirlerine dönüştürülebileceğini de biliyoruz:

Fonksiyon	Görevi	Örnek
`int()`	Bir veriyi tam sayıya dönüştürür	`int('2')`
`float()`	Bir veriyi kayan noktalı sayıya dönüştürür	`float(2)`
`complex()`	Bir veriyi karmaşık sayıya dönüştürür	`complex(2)`

Dediğimiz gibi, bunlar bizim zaten sayılara dair bildiğimiz şeyler. Elbette bir de henüz öğrenmediklerimiz var.

Gelin şimdi bunların neler olduğunu inceleyelim.

Sayıların Metotları

Tıpkı öteki veri tiplerinde olduğu gibi, sayıların da bazı metotları bulunur. Bu metotları kullanarak sayılar üzerinde çeşitli işlemler gerçekleştirebiliriz.

Tam Sayıların Metotları

Dediğimiz gibi, Python'da birkaç farklı sayı tipi bulunur. Biz ilk olarak tam sayı (*integer*) denen sayı tipinin metot ve niteliklerini inceleyeceğiz.

Öncelikle hangi metotlar ve niteliklerle karşı karşıya olduğumuza bakalım:

```
>>> [i for i in dir(int) if not i.startswith("_")]

['bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator',
'real', 'to_bytes']
```

Bu listede şimdilik bizi ilgilendiren tek bir metot var. Bu metodun adı bit_length().

bit_length()

Bilgisayarlar hakkında bilmemiz gereken en önemli bilgilerden biri şudur: Bilgisayarlar ancak ve ancak sayılarla işlem yapabilir. Bilgisayarların işlem yapabildiği sayılar da onlu sistemdeki sayılar değil, ikili sistemdeki sayılardır. Yani 0'lar ve 1'ler.

Bilgisayar terminolojisinde bu 0'lar ve 1'lerden oluşan herbir basamağa 'bit' adı verilir. Yani ikili sayma sisteminde '0' ve '1' sayılarından herbiri 1 bit'tir. Mesela onlu sistemde *2* sayısının ikili sistemdeki karşılığı olan *10* sayısı iki bit'lik bir sayıdır. Onlu sistemdeki *100* sayısının ikili sistemdeki karşılığı olan *1100100* sayısı ise yedi bitlik bir sayıdır.

Bu durumu daha net bir şekilde görebilmek için şu kodları yazalım:

```
>>> for i in range(11):
...     print(i, bin(i)[2:], len(bin(i)[2:]), sep="\t")
...
0       0       1
1       1       1
2       10      2
3       11      2
4       100     3
5       101     3
6       110     3
7       111     3
8       1000    4
9       1001    4
10      1010    4
```

Burada ikinci sıradaki sayılar ilk sıradaki sayıların ikili sistemdeki karşılıklarıdır. Üçüncü sıradaki sayılar ise herbir sayının kaç bit olduğunu, yani bir bakıma ikili sayma sisteminde kaç basamağa sahip olduğunu gösteriyor.

İşte herhangi bir tam sayının kaç bit'lik bir yer kapladığını öğrenmek için, tam sayıların metotlarından biri olan bit_length() metodundan yararlanacağız:

```
>>> sayı = 10
>>> sayı.bit_length()

4
```

Demek ki *10* sayısı bellekte dört bitlik bir yer kaplıyormuş. Yani bu sayının ikili sistemdeki karşılığı olan *1010* sayısı dört basamaktan oluşuyormuş.

Yukarıdaki örneklerden de rahatlıkla çıkarabileceğiniz gibi, bit_length() metodu, ikili sayma sistemindeki bir sayı üzerine len() fonksiyonunun uygulanması ile eşdeğerdir. Yani:

```
>>> len(bin(10)[2:]) == (10).bit_length()
True
```

Bu arada şu son örnekte bir şey dikkatinizi çekmiş olmalı: bit_length() metodunu doğrudan sayılar üzerine uygulayamıyoruz. Yani:

```
>>> 10.bit_length()
  File "<stdin>", line 1
    10.bit_length()
               ^
SyntaxError: invalid syntax
```

Bu metodu sayılarla birlikte kullanabilmek için iki seçeneğimiz var: bit_length() metodunu uygulamak istediğimiz sayıyı önce bir değişkene atayabiliriz:

```
>>> a = 10
>>> a.bit_length()

4
```

...veya ilgili sayıyı parantez içine alabiliriz:

```
>>> (10).bit_length()

4
```

Bu durum, yani sayıyı parantez içinde gösterme zorunluluğu, *10* sayısının sağına bir nokta işareti koyduğumuzda, Python'ın bu sayıyı bir kayan noktalı sayı olarak değerlendirmesinden kaynaklanıyor. Yani biz '10' yazıp, bit_length() metodunu bu sayıya bağlama amacıyla sayının sağına bir nokta koyduğumuz anda, Python bu sayının bir kayan noktalı sayı olduğunu zannediyor. Çünkü Python

açısından, *10.* sayısı bir kayan noktalı sayıdır. Bunu teyit edelim:

```
>>> type(10.)

<class 'float'>
```

Kayan noktalı sayıların bit_length() adlı bir metodu olmadığı için de Python'ın bize bir hata mesajı göstermekten başka yapabileceği bir şey kalmıyor.

Kayan Noktalı Sayıların Metotları

Python'da tam sayılar dışında kayan noktalı sayıların da olduğunu biliyoruz. Bu sayı tipinin şu metotları vardır:

```
>>> [i for i in dir(float) if not i.startswith("_")]

['as_integer_ratio', 'conjugate', 'fromhex', 'hex', 'imag', 'is_integer', 'real']
```

Biz bu metotlar arasından, as_integer_ratio() ve is_integer() adlı metotlarla ilgileneceğiz.

as_integer_ratio()

Bu metot, birbirine bölündüğünde ilgili kayan noktalı sayıyı veren iki adet tam sayı verir bize. Örnek üzerinden açıklayalım:

```
>>> sayı = 4.5
>>> sayı.as_integer_ratio()

(9, 2)
```

9 sayısını *2* sayısına böldüğümüzde *4.5* sayısını elde ederiz. İşte as_integer_ratio() metodu, bu *9* ve *2* sayılarını bize ayrı ayrı verir.

is_integer()

Bir kayan noktalı sayının ondalık kısmında *0* harici bir sayının olup olmadığını kontrol etmek için bu metodu kullanıyoruz. Örneğin:

```
>>> (12.0).is_integer()

True

>>> (12.5).is_integer()
```

```
False
```

Karmaşık Sayıların Metotları

Gelelim karmaşık sayıların metot ve niteliklerine...

```
>>> [i for i in dir(complex) if not i.startswith("_")]

['conjugate', 'imag', 'real']
```

Gördüğünüz gibi, karmaşık sayıların da birkaç tane metot ve niteliği var. Biz bunlar arasından imag ve real adlı nitelikleri inceleyeceğiz.

imag

Bir gerçek bir de sanal kısımdan oluşan sayılara karmaşık sayılar (*complex*) adı verildiğini biliyorsunuz. Örneğin şu bir karmaşık sayıdır:

```
12+4j
```

İşte imag adlı nitelik, bize bir karmaşık sayının sanal kısmını verir:

```
>>> c = 12+4j
>>> c.imag

4.0
```

real

real adlı nitelik bize bir karmaşık sayının gerçek kısmını verir:

```
>>> c = 12+4j
>>> c.real

12.0
```

Aritmetik Fonksiyonlar

Python programlama dili, bize sayılarla rahat çalışabilmemiz için bazı fonksiyonlar sunar. Bu fonksiyonları kullanarak, karmaşık aritmetik işlemleri kolayca yapabiliriz.

Biz bu bölümde Python'ın bize sunduğu bu gömülü fonksiyonları tek tek inceleyeceğiz.

Gömülü fonksiyonlar, Python programlama dilinde, herhangi bir özel işlem yapmamıza gerek

olmadan, kodlarımız içinde doğrudan kullanabileceğimiz fonksiyonlardır. Biz şimdiye kadar pek çok gömülü fonksiyonla zaten tanışmıştık. O yüzden gömülü fonksiyonlar bizim yabancısı olduğumuz bir konu değil. Mesela buraya gelene kadar gördüğümüz, len(), range(), type(), open(), print() ve id() gibi fonksiyonların tamamı birer gömülü fonksiyondur. Biz bu fonksiyonları ilerleyen derslerde çok daha ayrıntılı bir şekilde inceleyeceğiz. Ama şu anda bile fonksiyonlar konusunda epey bilgiye sahibiz.

Şimdiye kadar öğrendiğimiz gömülü fonksiyonlardan şu listede yer alanlar, matematik işlemlerinde kullanılmaya uygun olanlardır:

1. complex()
2. float()
3. int()
4. pow()
5. round()
6. hex()
7. oct()
8. bin()

Biz bu fonksiyonların ne işe yaradığını önceki derslerimizde zaten ayrıntılı olarak incelemiştik. O yüzden burada bunlardan söz etmeyeceğiz. Onun yerine, henüz öğrenmediğimiz, ama mutlaka bilmemiz gereken gömülü fonksiyonları ele alacağız.

O halde hiç vakit kaybetmeden yola koyulalım...

abs()

Bu fonksiyon bize bir sayının mutlak değerini verir:

```
>>> abs(-2)

2

>>> abs(2)

2
```

divmod()

Bu fonksiyon, bir sayının bir sayıya bölünmesi işleminde **bölümü** ve **kalanı** verir:

```
>>> divmod(10, 2)

(5, 0)
```

10 sayısı *2* sayısına bölündüğünde 'bölüm' *5*, 'kalan' ise *0*'dır.

Bir örnek daha verelim:

```
>>> divmod(14, 3)

(4, 2)
```

Bu sonuçtan gördüğünüz gibi, aslında divmod() fonksiyonu şu kodlarla aynı işi yapıyor:

```
>>> 14 // 3, 14 % 3
```

Bu fonksiyonun gerçekleştirdiği bölme işleminin bir 'taban bölme' işlemi olduğuna özellikle dikkatinizi çekmek istiyorum.

max()

Size şöyle bir soru sorduğumu düşünün: Acaba aşağıdaki listede yer alan sayıların en büyüğü kaçtır?

```
[882388, 260409, 72923, 692476, 131925, 259114, 47630, 84513, 25413, 614654,
239479, 299159, 175488, 345972, 458112, 791030, 243610, 413702, 565285, 773607,
131583, 979177, 247202, 615485, 647512, 556823, 242460, 852928, 893126, 792435,
273 904, 544434, 627222, 601984, 966446, 384143, 308858, 915106, 914423, 826315,
258 342, 188056, 934954, 253918, 468223, 262875, 462902, 370061, 336521, 367829,
147 846, 838385, 605377, 175140, 957437, 105779, 153499, 435097, 9934, 435761,
98906 6, 357279, 341319, 420455, 220075, 28839, 910043, 891209, 975758, 140968,
837021 , 526798, 235190, 634295, 521918, 400634, 385922, 842289, 106889, 742531,
359913 , 842431, 666182, 516933, 22222, 445705, 589281, 709098, 48521, 513501,
277645, 860937, 655966, 923944, 7895, 77482, 929007, 562981, 904166, 619260,
616293, 203 512, 67534, 615578, 74381, 484273, 941872, 110617, 53517, 402324,
156156, 839504 , 625325, 694080, 904277, 163914, 756250, 809689, 354050, 523654,
26723, 167882, 103404, 689579, 121439, 158946, 485258, 850804, 650603, 717388,
981770, 573882, 358726, 957285, 418479, 851590, 960182, 11955, 894146, 856069,
369866, 740623, 867622, 616830, 894801, 827179, 580024, 987174, 638930, 129200,
214789, 45268, 4 55924, 655940, 335481, 845907, 942437, 759380, 790660, 432715,
858959, 289617, 7 57317, 982063, 237940, 141714, 939369, 198282, 975017, 785968,
49954, 854914, 99 6780, 121633, 436419, 471, 776271, 91626, 209175, 894281,
417963, 624464, 736535 , 418888, 506194, 591087, 64075, 50252, 952943, 25878,
217085, 223996, 416042, 4 84123, 810460, 423284, 956886, 237772, 960241, 601551,
830147, 449088, 364567, 3 37281, 524358, 980387, 393760, 619710, 100181, 96738,
275199, 553783, 975654, 66 2536, 979103, 869504, 702350, 174361, 970250, 267625,
661580, 444662, 871532, 88 1977, 981660, 446047, 508758, 530694, 608789, 339540,
242774, 637473, 874011, 73 2999, 511638, 744144, 710805, 641326, 88085, 128487,
59732, 739340, 443638, 8303 33, 832136, 882277, 403538, 441349, 721048, 32859]
```

İşte böyle bir soruyu çözmek için max() fonksiyonundan yararlanabilirsiniz. Yukarıdaki listeyi *sayılar* adlı bir değişkene atadığımızı varsayarsak, aşağıdaki kod bize listedeki en büyük sayıyı verecektir:

```
>>> max(sayılar)
```

Yukarıdaki örneklerde max() fonksiyonunu kullanarak bir dizi içindeki en **büyük** sayıyı bulduk. Peki bu fonksiyonu kullanarak bir dizi içindeki en **uzun** karakter dizisini bulabilir miyiz? Evet, bulabiliriz.

Diyelim ki elimizde şöyle bir liste var:

```
isimler = ["ahmet", "mehmet", "necla", "sedat", "abdullah",
        "gıyaseddin", "sibel", "can", "necmettin", "savaş", "özgür"]
```

Amacımız bu liste içindeki en uzun kelimeyi bulmak. İşte bunu max() fonksiyonu ile yapabiliriz. Dikkatlice bakın:

```
print(max(isimler, key=len))
```

Bu kodları çalıştırdığımızda, listedeki en uzun isim olan 'gıyaseddin'i elde edeceğiz.

Gördüğünüz gibi, max() fonksiyonu *key* adlı özel bir parametre daha alıyor. Bu parametreye biz 'len' değerini verdik. Böylece max() fonksiyonu liste içindeki öğeleri uzunluklarına göre sıralayıp en uzun öğeyi bize sundu.

Hatırlarsanız geçen bölümde şöyle bir kod yazmıştık:

```
sayı_sistemleri = ["onlu", "sekizli", "on altılı", "ikili"]

print(("{:^9} "*len(sayı_sistemleri)).format(*sayı_sistemleri))

for i in range(17):
    print("{0:^9} {0:^9o} {0:^9x} {0:^9b}".format(i))
```

Bu kodlar, farklı sayma sistemleri arasındaki farkları daha net görmemizi sağlamıştı. Yalnız burada dikkat ettiyseniz, *sayı_sistemleri* adlı listeye her öğe ekleyişimizde, listedeki en uzun değeri dikkate alarak karakter dizisi biçimlendiricileri içindeki, öğeler arasında ne kadar boşluk bırakılacağını belirleyen sayıları güncelliyorduk. Mesela yukarıdaki örnekte, öğeler arasında yeterince boşluk bırakabilmek için bu sayıyı *9* olarak belirlemiştik. İşte şimdi öğrendiğimiz max() fonksiyonunu kullanarak bu sayının otomatik olarak belirlenmesini sağlayabiliriz. Dikkatlice inceleyin:

```
sayı_sistemleri = ["onlu", "sekizli", "on altılı", "ikili"]

en_uzun = len(max(sayı_sistemleri, key=len))
```

```
print(("{:^{aralık}} "*len(sayı_sistemleri)).format(*sayı_sistemleri, aralık=en_uzun))

for i in range(17):
    print("{0:^{1}} {0:^{1}o} {0:^{1}x} {0:^{1}b}".format(i, en_uzun))
```

Gördüğünüz gibi, max() fonksiyonunu ve bu fonksiyonun *key* parametresini kullanarak, oluşturduğumuz tablodaki öğelerin arasına uygun boşluğu otomatik olarak eklemiş olduk. Bunun için, *sayı_sistemleri* adlı listedeki en uzun öğenin uzunluk miktarını temel aldık.

min()

Bu fonksiyon, max() fonksiyonun yaptığı işin tam tersini yapar. Yani bu fonksiyonu kullanarak bir dizi içindeki en küçük sayıyı bulabilirsiniz:

```
>>> min(sayılar)
```

Tıpkı max() fonksiyonunda olduğu gibi, min() fonksiyonunda da *key* parametresini kullanabilirsiniz. Mesela max() fonksiyonunu anlatırken verdiğimiz isim listedindeki en kısa ismi bulabilmek için şu kodu yazabilirsiniz:

```
print(min(isimler, key=len))
```

sum()

Bu fonksiyon bir dizi içinde yer alan bütün sayıları birbiriyle toplar. Örneğin:

```
>>> a = [10, 20, 43, 45 , 77, 2, 0, 1]
>>> sum(a)

198
```

Eğer bu fonksiyonun, toplama işlemini belli bir sayının üzerine gerçekleştirmesini istiyorsanız şu kodu yazabilirsiniz:

```
>>> sum(a, 10)

208
```

sum() fonksiyonuna bu şekilde ikinci bir parametre verdiğinizde, bu ikinci parametre toplam değere eklenecektir.

başa dön

Temel Dosya İşlemleri

Hatırlarsanız print() fonksiyonunu anlatırken, bu fonksiyonun *file* adlı bir parametresi olduğundan söz etmiştik. Bu parametre yardımıyla print() fonksiyonunun çıktılarını bir dosyaya gönderebiliyorduk. Böylece print() fonksiyonunun bu özelliği sayesinde, Python'daki 'Dosya Girdi/Çıktısı' (*File I/O*) konusuyla da ilk kez tanışmış olmuştuk.

Ayrıca print() fonksiyonu dışında, open() adlı başka bir fonksiyon yardımıyla da dosyaları açabileceğimizi ve bu dosyaların üzerinde çeşitli işlemleri gerçekleştirebileceğimizi öğrenmiştik. Ancak gerek print() fonksiyonunun *file* parametresi, gerekse open() fonksiyonuyla şimdiye kadar yaptığımız örnekler aracılığıyla öğrendiklerimiz dosyalara ilişkin çok sınırlı işlemleri yerine getirmemizi sağlıyordu.

İşte biz bu bölümde, dosya girdi/çıktısı konusuna ilişkin bildiklerimizi bir adım öteye götüreceğiz ve gerçek anlamda dosyaları nasıl manipüle edeceğimizi öğreneceğiz.

Programcılık maceramız boyunca dosyalarla bol bol muhatap olacaksınız. O yüzden bu konuyu olabildiğince ayrıntılı ve anlaşılır bir şekilde anlatmaya çalışacağız.

Dediğimiz gibi, biz esasında bu noktaya gelinceye kadar çeşitli fonksiyonlar ve bunların birtakım parametreleri aracılığıyla dosya işlemlerinden az da olsa zaten söz etmiştik. Dolayısıyla aslında tamamen yabancısı olduğunuz bir konuyla karşı karşıya olmanız gibi bir durum söz konusu değil. Biz bu bölümde, zaten aşina olduğumuz bir konuyu çok daha derinlemesine ele alacağız.

Python programlama dilinde dosyalarla uğraşırken bütün dosya işlemleri için temel olarak tek bir fonksiyondan yararlanacağız. Bu fonksiyonu siz zaten tanıyorsunuz. Fonksiyonumuzun adı open().

Dosya Oluşturmak

Dediğimiz gibi, Python programlama dilinde dosya işlemleri için open() adlı bir fonksiyondan yararlanacağız. İşte dosya oluşturmak için de bu fonksiyonu kullanacağız.

Önceki derslerimizde verdiğimiz örneklerden de bildiğiniz gibi, open() fonksiyonunu temel olarak şöyle kullanıyoruz:

```
f = open(dosya_adı, kip)
```

Not

open() fonksiyonu *dosya_adı* ve *kip* dışında başka parametreler de alır. İlerleyen sayfalarda bu parametrelerden de söz edeceğiz.

Mesela “tahsilat.txt” adlı bir dosyayı **yazma** kipinde açmak için şöyle bir komut kullanıyoruz:

```
tahsilat_dosyası = open("tahsilat_dosyası.txt", "w")
```

Burada ‘tahsilat_dosyası.txt’ ifadesi dosyamızın adını belirtiyor. *“w”* harfi ise bu dosyanın yazma kipinde açıldığını söylüyor.

Yukarıdaki komutu çalıştırdığınızda, o anda hangi dizin altında bulunuyorsanız o dizin içinde *tahsilat_dosyası.txt* adlı boş bir dosyanın oluştuğunu göreceksiniz.

Bu arada, dosya adını yazarken, dosya adı ile birlikte o dosyanın hangi dizin altında oluşturulacağını da belirleyebilirsiniz. Örneğin:

```
dosya = open("/dosyayı/oluşturmak/istediğimiz/dizin/dosya_adı", "w")
```

Eğer dosya adını dizin belirtmeden yazarsanız, oluşturduğunuz dosya, o anda hangi dizin altında bulunuyorsanız orada oluşacaktır.

Ayrıca dosyayı barındıran dizin adlarını yazarken, dizinleri ayırmak için ters taksim (\) yerine düz taksim (/) kullanmaya dikkat edin. Aksi halde, dizin adı oluşturmaya çalışırken yanlışlıkla kaçış dizileri oluşturabilirsiniz. Esasında siz bu olguya hiç yabancı değilsiniz. Zira kaçış dizilerini anlatırken şöyle bir örnek verdiğimizi hatırlıyor olmalısınız:

```
print("C:\aylar\nisan\toplam masraf")
```

İşte eğer bu örnekte olduğu gibi ters taksim işaretleri ile oluşturulmuş dizin adları kullanırsanız programınız hata verecektir:

```
>>> open("C:\aylar\nisan\toplam masraf\masraf.txt", "w")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 22] Invalid argument: 'C:\x07ylar\nisan\toplam masraf\masraf.txt'
```

Bunun sebebi, bildiğiniz gibi, Python’ın *\a*, *\n* ve *\t* ifadelerini birer kaçış dizisi olarak algılamasıdır. Bu durumdan kaçabilmek için, dizin adlarını ters taksim işareti ile ayırmanın dışında, *r* adlı kaçış dizisinden de yararlanabilirsiniz:

```
>>> open(r"C:\aylar\nisan\toplam masraf\masraf.txt", "w")
```

...veya ters taksim işaretlerini çiftleyebilirsiniz:

```
>>> open("C:\\aylar\\nisan\\toplam masraf\\masraf.txt", "w")
```

Bu şekilde, eğer bilgisayarınızda *C:\aylar\nisan\toplam masraf* adlı bir dizin varsa, o dizin içinde *masraf.txt* adlı bir dosya oluşturulacaktır.

Böylece Python programlama dilinde boş bir dosyanın nasıl oluşturulacağını öğrenmiş olduk. O halde gelin isterseniz şimdi bu dosyanın içini nasıl dolduracağımızı öğrenelim.

Dosyaya Yazmak

Bir dosyayı, yukarıda gösterdiğimiz şekilde yazma kipinde açtığımız zaman, Python bizim için içi boş bir dosya oluşturacaktır. Peki biz bu dosyanın içini nasıl dolduracağız?

Python programlama dilinde, open() fonksiyonu ile yazma kipinde açtığımız bir dosyaya bir veri yazabilmek için dosyaların write() adlı metodundan yararlanacağız.

Siz aslında bu metodun da nasıl kullanılacağını çok iyi biliyorsunuz:

```
dosya.write(yazılacak_şeyler)
```

Gelin bu formülü somutlaştıracak bir örnek verelim. Mesela yukarıda oluşturduğumuz tahsilat dosyasının içine bazı veriler girelim.

Önce dosyamızı nasıl oluşturacağımızı hatırlayalım:

```
ths = open("tahsilat_dosyası.txt", "w")
```

Şimdi de bu dosyaya şu bilgileri girelim:

```
ths.write("Halil Pazarlama: 120.000 TL")
```

Yani programımız şöyle görünsün:

```
ths = open("tahsilat_dosyası.txt", "w")
ths.write("Halil Pazarlama: 120.000 TL")
```

Bu komutları verdiğinizde, *tahsilat_dosyası.txt* adlı dosyanın içine şu bilgilerin işlendiğini göreceksiniz:

```
Halil Pazarlama: 120.000 TL
```

Eğer dosyayı açtığınızda bu bilgi yerine hâlâ boş bir dosya görüyorsanız, sebebi tamponda tutulan verilerin henüz dosyaya işlenmemiş olmasıdır.

Not

Bu konuyu print() fonksiyonunun *flush* adlı parametresini incelerken öğrendiğimizi hatırlıyor olmalısınız.

Eğer durum böyleyse, dosyanızı kapatmanız gerekiyor. Bunu close() adlı başka bir metotla yapabildiğimizi biliyorsunuz:

```
ths.close()
```

Bu arada, bu söylediklerimizden, eğer yazdığınız bilgiler zaten dosyaya işlenmişse dosyayı kapatmanıza gerek olmadığı anlamını çıkarmayın. Herhangi bir şekilde açtığınız dosyaları kapatmanız, özellikle dosyanın açılmasıyla birlikte kullanılmaya başlayan ve arka planda çalışan kaynakların serbest bırakılması açısından büyük önem taşıyor. O yüzden açtığımız dosyaların tamamını programın işleyişi sona erdiğinde kapatmayı unutmuyoruz. Yani yukarıdaki programı tam olarak şöyle yazıyoruz:

```
ths = open("tahsilat_dosyası.txt", "w")
ths.write("Halil Pazarlama: 120.000 TL"),
ths.close()
```

Bu kodlarda sırasıyla şu işlemleri gerçekleştirdik:

1. *tahsilat_dosyası* adlı bir dosyayı yazma kipinde açarak, bu adda bir dosya oluşturulmasını sağladık,
2. write() metodunu kullanarak bu dosyaya bazı bilgiler girdik,
3. Dosyamıza yazdığımız bilgilerin dosyaya işlendiğinden emin olmak ve işletim sisteminin dosyanın açılması ve dosyaya veri işlenmesi için devreye soktuğu bütün kaynakları serbest bırakmak için close() metoduyla programımızı kapattık.

Bu arada, bu başlığı kapatmadan önce önemli bir bilgi daha verelim. Python'da bir dosyayı *"w"* kipinde açtığımızda, eğer o adda bir dosya ilgili dizin içinde zaten varsa, Python bu dosyayı sorgusuz sualsiz silip, yerine aynı adda başka bir boş dosya oluşturacaktır. Yani mesela yukarıda *tahsilat_dosyası.txt* adlı dosyayı oluşturup içine bir şeyler yazdıktan sonra bu dosyayı yine *"w"* kipinde açmaya çalışırsanız, Python bu dosyanın bütün içeriğini silip, yine *tahsilat_dosyası.txt* adını taşıyan başka bir dosya oluşturacaktır. O yüzden dosya işlemleri sırasında bu *"w"* kipini kullanırken dikkat ediyoruz ve disk üzerinde var olan dosyalarımızı yanlışlıkla silmiyoruz.

Böylece bir dosyanın nasıl oluşturulacağını, nasıl açılacağını ve içine birtakım bilgilerin nasıl girileceğini kabataslak da olsa öğrenmiş olduk. Şimdi de dosyaları nasıl **okuyacağımızı** öğrenelim.

Dosya Okumak

Bir önceki başlıkta dosyaların içine bilgi girme işleminin Python programlama dilinde nasıl yapıldığını inceledik. Elbette bir dosyaya yazabilmenin yanısıra, bilgisayarınızda halihazırda var olan bir dosyayı okumak da isteyeceksiniz. Peki bunu nasıl yapacaksınız?

Python'da bir dosyayı okumak için yukarıda anlattığımız yazma yöntemine benzer bir yöntem kullanacağız. Bildiğiniz gibi, bir dosyayı yazma kipinde açmak için *"w"* harfini kullanıyoruz. Bir dosyayı okuma kipinde açmak için ise *"r"* harfini kullanacağız.

Mesela, bilgisayarımızda var olan *fihrist.txt* adlı dosyayı okumak üzere açalım:

```
fihrist = open("fihrist.txt", "r")
```

Bir dosyayı open() fonksiyonu yardımıyla açarken kip parametresi için *"r"* harfini kullanırsak, Python o dosyayı okuma yetkisiyle açacaktır. Yalnız burada şöyle bir özellik var: Eğer bir dosyayı okuma kipinde açacaksanız, bu *"r"* harfini hiç belirtmeseniz de olur. Yani şu komut bilgisayarımızdaki *fihrist.txt* adlı dosyayı okuma kipinde açacaktır:

```
fihrist = open("fihrist.txt")
```

Dolayısıyla bir dosyayı açarken kip belirtmediğimizde Python bizim o dosyayı okuma kipinde açmak istediğimizi varsayacaktır.

Hatırlarsanız, *"w"* kipiyle açtığımız bir dosyaya yazmak için write() adlı bir metottan yararlanıyorduk. *"r"* kipiyle açtığımız bir dosyayı okumak için ise read(), readline() ve readlines() adlı üç farklı metottan yararlanacağız.

Yukarıdaki üç metot da Python'da dosya okuma işlemlerini gerçekleştirmemizi sağlar. Peki bu metotların üçü de aynı işi yapıyorsa neden tek bir metot değil de üç farklı metot var?

Bu metotların üçü de dosya okumaya yarasa da, verdikleri çıktılar birbirinden farklıdır. O yüzden farklı amaçlar için farklı metodu kullanmanız gereken durumlarla karşılaşabilirsiniz.

Bu metotlar arasındaki farkı anlamanın en kolay yolu bu üç metodu sırayla kullanıp, çıktıları incelemektir.

Öncelikle içeriği şu olan, *fihrist.txt* adlı bir dosyamızın olduğunu varsayalım:

```
Ahmet Özbudak : 0533 123 23 34
Mehmet Sülün  : 0532 212 22 22
Sami Sam      : 0542 333 34 34
```

Şimdi bir dosya açıp şu kodları yazalım:

```
fihrist = open("fihrist.txt")
print(fihrist.read())
```

Bu kodları çalıştırdığımızda, eğer kullandığınız işletim sistemi GNU/Linux ise muhtemelen şu çıktıyı elde edeceksiniz:

```
Ahmet Özbudak : 0533 123 23 34
Mehmet Sülün  : 0532 212 22 22
Sami Sam      : 0542 333 34 34
```

Ama eğer bu kodları Windows'ta çalıştırdıysanız Türkçe karakterler bozuk çıkmış olabilir. Bu

durumu şimdilik görmezden gelin. Birazdan bu durumun nedenini açıklayacağız.

Yukarıda elde ettiğimiz şey bir karakter dizisidir bunu şu şekilde teyit edebileceğinizi biliyorsunuz:

```
fihrist = open("fihrist.txt")
print(type(fihrist.read()))
```

Gördüğünüz gibi, read() metodu bize, dosyanın bütün içeriğini bir karakter dizisi olarak veriyor. Bir de şuna bakalım:

```
fihrist = open("fihrist.txt")
print(fihrist.readline())
```

Burada da readline() metodunu kullandık. Bu kodlar bize şöyle bir çıktı veriyor:

```
Ahmet Özbudak : 0533 123 23 34
```

read() metodu bize dosya içeriğinin tamamını veriyordu. Gördüğünüz gibi readline() metodu tek bir satır veriyor. Yani bu metot yardımıyla dosyaları satır satır okuyabiliyoruz.

Bu metodun işleyiş tarzını daha iyi görebilmek için bu kodları dosyaya yazıp çalıştırmak yerine etkileşimli kabuk üzerinden de çalıştırabilirsiniz:

```
>>> fihrist = open("fihrist.txt", "r")
>>> print(fihrist.readline())

Ahmet Özbudak : 0533 123 23 34

>>> print(fihrist.readline())

Mehmet Sülün  : 0532 212 22 22

>>> print(fihrist.readline())

Sami Sam      : 0542 333 34 34
```

Gördüğünüz gibi, readline() metodu gerçekten de dosyayı satır satır okuyor.

Son satırı da okuduktan sonra, readline() metodunu tekrar çalıştırırsak ne olur peki? Bakalım:

```
>>> print(fihrist.readline())
```

Gördüğünüz gibi, bu defa hiçbir çıktı almadık. Çünkü dosyada okunacak satır kalmadı. Bu yüzden de Python bize boş bir çıktı verdi. Bu durumu daha net görmek için kodu etkileşimli kabukta print() olmadan yazabilirsiniz:

```
>>> fihrist.readline()

''
```

Gerçekten de elimizdeki şey boş bir karakter dizisi... Demek ki bir dosya tamamen okunduktan sonra, Python otomatik olarak tekrar dosyanın başına dönmüyor. Böyle bir durumda dosyanın başına nasıl geri döneceğimizi inceleyeceğiz, ama isterseniz biz başka bir konuyla devam edelim.

Not

Bir dosyanın tamamı okunduktan sonra otomatik olarak başa sarılmaması özelliği sadece `readline()` metodu için değil, öteki bütün dosya okuma metotları için de geçerlidir. Yani bir dosyayı `read()`, `readline()` veya `readlines()` metotlarından herhangi biri ile okuduğunuzda imleç başa dönmez.

Dediğimiz ve gösterdiğimiz gibi, `read()` ve `readline()` metotları bize bir karakter dizisi döndürüyor. Bu iki metot arasındaki fark ise, `read()` metodunun dosyanın tamamını önümüze sererken, `readline()` metodunun dosyayı satır satır okuyup, her defasında tek bir satırı önümüze sürmesidir. Bir de `readlines()` metodunun ne yaptığına bakalım...

Şu kodları yazalım:

```
fihrist = open("fihrist.txt")
print(fihrist.readlines())
```

Bu kodları yazdığımızda şuna benzer bir çıktı alacağız:

```
['Ahmet Özbudak : 0533 123 23 34\n', 'Mehmet Sülün  : 0532 212 22 22\n',
 'Sami Sam      : 0542 333 34 34']
```

Gördüğünüz gibi, bu defa karakter dizisi yerine bir liste ile karşılaşıyoruz. Demek ki `read()` ve `readline()` metotları çıktı olarak bize bir karakter dizisi verirken, `readlines()` metodu liste veriyormuş. Bunun neden önemli bir bilgi olduğunu artık gayet iyi biliyor olmanız lazım. Zira bir verinin tipi, o veriyle neler yapıp neler yapamayacağımızı doğrudan etkiler...

Dosyaları Otomatik Kapatma

Daha önce de söylediğimiz gibi, bir dosyayı açıp bu dosya üzerinde gerekli işlemleri yaptıktan sonra bu dosyayı açık bırakmamak büyük önem taşır. Dolayısıyla üzerinde işlem yaptığımız bütün dosyaları, işimiz bittikten sonra, mutlaka kapatmalıyız. Çünkü bir dosya açıldığında işletim sistemi, sistem kaynaklarının bir kısmını bu dosyaya ayırır. Eğer dosyayı açık bırakırsak, sistem kaynaklarını gereksiz yere meşgul etmiş oluruz. Ancak farklı sebeplerden, dosyalar açıldıktan sonra kapanmayabilir. Örneğin açtığınız dosyayı kapatmayı unutmuş olabilirsiniz. Yani programınızın hiçbir

yerinde close() metodunu kullanmamışsınızdır. Bunun dışında, programınızdaki bir hata da dosyaların kapanmasını engelleyebilir. Örneğin bir dosya açıldıktan sonra programda beklenmeyen bir hata gerçekleşirse, programınız asla close() satırına ulaşamayabilir. Bu durumda da açılan dosya kapanmadan öylece bekler.

Bu tür durumlara karşı iki seçeneğiniz var:

1. try... except... finally... bloklarından yararlanmak
2. with adlı bir deyimi kullanmak

Birinci yöntemden daha önce de bahsettiğimizi hatırlıyorsunuz. Hata yakalama bölümünü anlatırken bununla ilgili şöyle bir örnek vermiştik:

```
try:
    dosya = open("dosyaadı", "r")
    ...burada dosyayla bazı işlemler yapıyoruz...
    ...ve ansızın bir hata oluşuyor...
except IOError:
    print("bir hata oluştu!")
finally:
    dosya.close()
```

Bu yöntem gayet uygun ve iyi bir yöntemdir. Ancak Python bize bu tür durumlar için çok daha pratik bir yöntem sunar. Dikkatlice bakın:

```
with open("dosyaadı", "r") as dosya:
    print(dosya.read())
```

Dosyalarımızı bu şekilde açıp üzerlerinde işlemlerimizi yaptığımızda Python dosyayı bizim için kendisi kapatacaktır. Bu şekilde bizim ayrıca bir close() satırı yazmamıza gerek yok. with deyimini kullanmamız sayesinde, dosya açıldıktan sonra arada bir hata oluşsa bile Python dosyayı sağsalim kapatıp sistem kaynaklarının israf edilmesini önleyecektir.

Dosyayı İleri-Geri Sarmak

Dosya okumak için kullanılan metotları anlatırken, dosya bir kez okunduktan sonra imlecin otomatik olarak dosyanın başına dönmediğini görmüştük. Yani mesela read() metoduyla dosyayı bir kez okuduktan sonra, dosyayı tekrar okumak istersek elde edeceğimiz şey boş bir karakter dizisi olacaktır. Çünkü dosya okunduktan sonra okunacak başka bir satır kalmamış, imleç dosya sonuna ulaşmış ve otomatik olarak da başa dönmemiştir. Bu olguyu etkileşimli kabuk üzerinde daha net bir şekilde görebileceğinizi biliyorsunuz.

Peki dosyayı tamamen okuduktan sonra tekrar başa dönmek istersek ne yapacağız? Bir dosya tamamen okunduktan sonra tekrar başa dönmek için dosyaların seek() adlı bir metodundan yararlanacağız.

Mesela şu örneklere bakalım. Bu örnekleri daha iyi anlamak için bunları Python'ın etkileşimli kabuğunda çalıştırmanızı tavsiye ederim:

```
>>> f = open("python.txt")
>>> f.read()

'Bu programlama dili Guido Van Rossum adlı Hollandalı bir
programcı\ntarafından 90'lı yılların başında geliştirilmeye başlanmıştır.
Çoğu insan,\nisminin Python olmasına aldanarak, bu programlama dilinin,
adını piton\nyılanından aldığını düşünür. Ancak zannedildiğinin aksine bu
programlama dilinin\nadı piton yılanından gelmez. Guido Van Rossum bu
programlama dilini, The Monty\nPython adlı bir İngiliz komedi grubunun,
Monty Python's Flying Circus adlı\ngösterisinden esinlenerek adlandırmıştır.
Ancak her ne kadar gerçek böyle olsa\nda, Python programlama dilinin pek çok
yerde bir yılan figürü ile temsil\nedilmesi neredeyse bir gelenek halini
almıştır.\n'
```

Burada open() fonksiyonunu kullanarak *python.txt* adlı bir dosyayı açıp, read() metodu yardımıyla da bu dosyanın içeriğini okuduk. Bu noktada dosyayı tekrar okumaya çalışırsak elde edeceğimiz şey boş bir karakter dizisi olacaktır:

```
>>> f.read()

''
```

Çünkü dosya bir kez tamamen okunduktan sonra imleç otomatik olarak başa dönmüyor. Dosyayı tekrar okumak istiyorsak, bunu başa bizim sarmamız lazım. İşte bunun için seek() metodunu kullanacağız:

```
>>> f.seek(0)
```

Gördüğünüz gibi seek() metodunu bir parametre ile birlikte kullandık. Bu metoda verdiğimiz parametre, dosya içinde kaçıncı bayt konumuna gideceğimizi gösteriyor. Biz burada *0* sayısını kullanarak dosyanın ilk baytına, yani en başına dönmüş olduk. Artık dosyayı tekrar okuyabiliriz:

```
>>> f.read()

'Bu programlama dili Guido Van Rossum adlı Hollandalı bir
programcı\ntarafından 90'lı yılların başında geliştirilmeye başlanmıştır.
Çoğu insan,\nisminin Python olmasına aldanarak, bu programlama dilinin,
adını piton\nyılanından aldığını düşünür. Ancak zannedildiğinin aksine bu
programlama dilinin\nadı piton yılanından gelmez. Guido Van Rossum bu
programlama dilini, The Monty\nPython adlı bir İngiliz komedi grubunun,
Monty Python's Flying Circus adlı\ngösterisinden esinlenerek adlandırmıştır.
```

```
Ancak her ne kadar gerçek böyle olsa\nda, Python programlama dilinin pek çok
yerde bir yılan figürü ile temsil\nedilmesi neredeyse bir gelenek halini
almıştır.\n'
```

Elbette seek() metodunu kullanarak istediğiniz bayt konumuna dönebilirsiniz. Mesela eğer dosyanın *10.* baytının bulunduğu konuma dönmek isterseniz bu metodu şöyle kullanabilirsiniz:

```
>>> f.seek(10)
```

Eğer o anda dosyanın hangi bayt konumunda bulunduğunuzu öğrenmek isterseniz de tell() adlı başka bir metottan yararlanabilirsiniz. Bu metodu parametresiz olarak kullanıyoruz:

```
>>> f.tell()

20
```

Bu çıktıya göre o anda dosyanın *20.* baytının üzerindeyiz...

Bu arada, dosya içinde bulunduğumuz konumu baytlar üzerinden tarif etmemizi biraz yadırgamış olabilirsiniz. Acaba neden karakter değil de bayt? Biraz sonra bu konuya geleceğiz. Biz şimdilik önemli başka bir konuya değinelim.

Dosyalarda Değişiklik Yapmak

Buraya kadar, Python'da bir dosyanın nasıl oluşturulacağını, boş bir dosyaya nasıl veri girileceğini ve varolan bir dosyadan nasıl veri okunacağını öğrendik. Ama varolan ve içi halihazırda dolu bir dosyaya nasıl veri ekleneceğini bilmiyoruz. İşte şimdi bu işlemin nasıl yapılacağını tartışacağız.

Ancak burada önemli bir ayrıntıya dikkatinizi çekmek istiyorum. Dosyaların neresinde değişiklik yapmak istediğiniz büyük önem taşır. Unutmayın, dosyaların başında, ortasında ve sonunda değişiklik yapmak birbirlerinden farklı kavramlar olup, birbirinden farklı işlemlerin uygulanmasını gerektirir.

Biz bu bölümde dosyaların baş tarafına, ortasına ve sonuna nasıl veri eklenip çıkarılacağını ayrı ayrı tartışacağız.

Dosyaların Sonunda Değişiklik Yapmak

Daha önce de söylediğimiz gibi, Python'da bir dosyayı açarken, o dosyayı hangi kipte açacağımızı belirtmemiz gerekiyor. Yani eğer bir dosyayı okumak istiyorsak dosyayı *"r"* kipinde, yazmak istiyorsak da *"w"* kipinde açmamız gerekiyor. Bildiğiniz gibi *"w"* kipi dosya içeriğini tamamen siliyor.

Eğer bir dosyayı **tamamen silmeden**, o dosyaya ekleme yapmak veya o dosyada herhangi bir değişiklik yapmak istiyorsak, dosyamızı buraya kadar öğrendiğimiz iki kipten daha farklı bir kiple açmamız gerekiyor. Şimdi öğreneceğimiz bu yeni kipin adı *"a"*. Yani Python'da içi boş olmayan bir

dosyada değişiklik yapabilmek için *"a"* adlı bir kipten yararlanacağız:

```
f = open(dosya_adı, "a")
```

Örneğin yukarıda verdiğimiz *fihrist.txt* adlı dosyayı bu kipte açalım ve dosyaya yeni bir girdi ekleyelim:

```
with open("fihrist.txt", "a") as f:
    f.write("Selin Özden\t: 0212 222 22 22")
```

Gördüğünüz gibi, dosyaya yeni eklediğimiz girdiler otomatik olarak dosyanın sonuna ilave ediliyor. Burada şu noktaya dikkat etmeniz lazım. Dosyanın sonunda bir satır başı karakterinin (*\n*) bulunup bulunmamasına bağlı olarak, dosyaya eklediğiniz yeni satırlar düzgün bir şekilde bir alt satıra geçebileceği gibi, dosyanın son satırının yanına da eklenebilir. Dolayısıyla duruma göre yukarıdaki satırı şu şekilde yazmanız gerekebilir:

```
with open("fihrist.txt", "a") as f:
    f.write("\nSelin Özden\t: 0212 222 22 22")
```

Burada bir alt satıra geçebilmek için 'Selin' ifadesinden önce bir satır başı karakteri eklediğimize dikkat edin. Ayrıca eğer bu satırdan sonra bir başka satır daha ekleyecekseniz, ilgili satırın sonuna da bir satır başı karakteri koymanız gerekebilir:

```
with open("fihrist.txt", "a") as f:
    f.write("Selin Özden\t: 0212 222 22 22\n")
```

Karşı karşıya olduğunuz duruma göre, satır başı karakterlerine ihtiyacınız olup olmadığını ve ihtiyacınız varsa bunları nereye yerleştireceğinizi kendiniz değerlendirmelisiniz.

Dosyaların Başında Değişiklik Yapmak

Bir önceki bölümde dosya sonuna nasıl yeni satır ekleyeceğimizi öğrendik. Ama siz programcılık maceranız sırasında muhtemelen dosyaların sonuna değil de, en başına ekleme yapmanız gereken durumlarla da karşılaşacaksınız. Python'da bu işi yapmak da çok kolaydır.

Örnek olması açısından, *fihrist.txt* adlı dosyanın içeriğini ele alalım:

```
Ahmet Özbudak : 0533 123 23 34
Mehmet Sülün  : 0532 212 22 22
Sami Sam      : 0542 333 34 34
Selin Özden   : 0212 222 22 22
```

Dosya içeriği bu. Eğer bu dosyayı *"a"* kipi ile açtıktan sonra doğrudan write() metodunu kullanarak bir ekleme yaparsak, yeni değer dosyanın sonuna eklenecektir. Ama biz mesela şu veriyi:

```
Sedat Köz     : 0322 234 45 45
```

‘Ahmet Özbudak : 0533 123 23 34’ girdisinin hemen üstüne, yani dosyanın sonuna değil de en başına eklemek istersek ne yapacağız?

Öncelikle şu kodları deneyelim:

```
with open("fihrist.txt", "r") as f:
    veri = f.read()
    f.seek(0) #Dosyayı başa sarıyoruz
    f.write("Sedat Köz\t: 0322 234 45 45\n"+veri)
```

Bu kodları bir dosyaya kaydedip çalıştırdığımızda Python bize şu hatayı verecektir:

```
istihza@netbook:~/Desktop$ python3 deneme.py
Traceback (most recent call last):
  File "deneme.py", line 4, in <module>
    f.write("Sedat Köz\t: 0322 234 45 45\n"+veri)
io.UnsupportedOperation: not writable
```

Bu hatayı almamızın sebebi dosyayı ‘okuma’ kipinde açmış olmamız. Çünkü bir dosyayı okuma kipinde açtığımızda o dosya üzerinde yalnızca okuma işlemleri yapabiliriz. Dosyaya yeni veri ekleme kısmına gelindiğinde, dosya yalnızca okuma yetkisine sahip olduğu için, Python bize yukarıdaki hata mesajını verecek, dosyanın ‘yazılamaz’ olduğundan şikayet edecektir.

Peki dosyayı *“w”* karakteri yardımıyla yazma kipinde açarsak ne olur? O zaman da şu meş’um hatayı alırız:

```
istihza@netbook:~/Desktop$ python3 deneme.py
Traceback (most recent call last):
  File "deneme.py", line 2, in <module>
    veri = f.read()
io.UnsupportedOperation: not readable
```

Gördüğünüz gibi, bu kez de dosyanın okunamadığına ilişkin bir hata alıyoruz. Çünkü biz bu kez de dosyayı ‘yazma’ kipinde açtık. Ancak burada şöyle bir durum var. Bildiğiniz gibi, bir dosyayı *“w”* kipi ile açtığımızda, Python bize hiçbir şey sormadan varolan içeriği silecektir. Burada da yukarıda yazdığımız kodlar yüzünden dosya içeriğini kaybettik. Unutmayın, dosya okuma-yazma işlemleri belli bir takım riskleri içinde barındırır. O yüzden bu tür işlemleri yaparken fazladan dikkat göstermeliyiz.

Yukarıda da gördüğümüz gibi, dosyamızı *“r”* veya *“w”* kiplerinde açmak işe yaramadı. Peki ne yapacağız? Bunun cevabı çok basit: Dosyamızı hem okuma hem de yazma kipinde açacağız. Bunun için de farklı bir kip kullanacağız. Dikkatlice bakın:

```
with open("fihrist.txt", "r+") as f:
    veri = f.read()
    f.seek(0) #Dosyayı başa sarıyoruz
```

```
    f.write("Sedat Köz\t: 0322 234 45 45\n"+veri)
```

Burada *"r+"* adlı yeni bir kip kullandığımıza dikkat edin. *"+"* işareti bir dosyayı hem okuma hem de yazma kipinde açmamıza yardımcı olur. İşte bu işareti *"r"* kipiyle birlikte *"r+"* şeklinde kullanarak dosyamızı hem okuma hem de yazma kipinde açmayı başardık. Artık ilgili dosya üzerinde hem okuma hem de yazma işlemlerini aynı anda gerçekleştirebiliriz.

Yukarıdaki kodlarda ilk satırın ardından şöyle bir kod yazdık:

```
veri = f.read()
```

Böylece dosyanın bütün içeriğini *veri* adlı bir değişkene atamış olduk. Peki bu işlemi yapmazsak ne olur? Yani mesela şöyle bir kod yazarsak:

```
with open("fihrist.txt", "r+") as f:
    f.seek(0)
    f.write("Sedat Köz\t: 0322 234 45 45\n")
```

Bu şekilde 'Sedat Köz\t: 0322 234 45 45\n' satırı, dosyadaki ilk satırı silip onun yerine geçecektir. Çünkü *f.seek(0)* ile dosyanın başına dönüp o noktaya, yani dosyanın ilk satırına bir veri ekledikten sonra Python öbür satırları otomatik olarak bir alt satıra kaydırmaz. Bunun yerine ilk satırdaki verileri silip onun yerine, yeni eklenen satırı getirir. Eğer yapmak istediğiniz şey buysa ne âlâ. Bu kodları kullanabilirsiniz. Ama bizim istediğimiz şey bu değil. O yüzden *veri = f.read()* satırını kullanarak dosya içeriğini bir değişken içinde depoluyoruz ve böylece bu verileri kaybetmemiş oluyoruz.

Bu satırın ardından gelen *f.seek(0)* satırının ne işe yaradığını biliyorsunuz. Biz yeni veriyi dosyanın en başına eklemek istediğimiz için, doğal olarak bu kod yardımıyla dosyanın en başına sarıyoruz. Böylece şu kod:

```
f.write("Sedat Köz\t: 0322 234 45 45\n"+veri)
```

Sedat Köz\t: 0322 234 45 45\n' satırını dosyanın en başına ekliyor. Ayrıca burada, biraz önce *veri* değişkenine atadığımız dosya içeriğini de yeni eklediğimiz satırın hemen arkasına ilave ettiğimize dikkat edin. Eğer bunu yapmazsanız, elinizde sadece Sedat Köz'ün iletişim bilgilerini barındıran bir dosya olacaktır...

Dosyaların Ortasında Değişiklik Yapmak

Gördüğünüz gibi, Python'da bir dosyanın en sonuna ve en başına veri eklemek çok zor değil. Birkaç satır yardımıyla bu işlemleri rahatlıkla yapabiliyoruz. Peki ya bir dosyanın en başına veya en sonuna değil de rastgele bir yerine ekleme yapmak istersek ne olacak?

Hatırlarsanız, Python'da her veri tipinin farklı özellikleri olduğundan, her veri tipinin farklı açılardan birbirlerine karşı üstünlükleri ya da zayıflıkları olduğundan söz etmiştik. Dediğimiz gibi, Python'da bazı işler için bazı veri tiplerini kullanmak daha pratik ve avantajlı olabilir. Örneğin karakter dizileri

değiştirilemeyen veri tipleri olduğu için, mesela bir metinde değişiklik yapmamız gereken durumlarda, eğer mümkünse listeleri kullanmak daha mantıklı olabilir. Zira bildiğiniz gibi, karakter dizilerinin aksine listeler değiştirilebilir veri tipleridir.

Önceki sayfalarda bir dosyayı okurken üç farklı metottan yararlanabileceğimizi öğrenmiştik. Bu metotların read(), readline() ve readlines() adlı metotlar olduğunu biliyorsunuz. Bu üç metottan read() adlı olanı bize çıktı olarak bir karakter dizisi veriyor. readline() metodu ise dosyaları satır satır okuyor ve bize yine bir karakter dizisi veriyor. Sonuncu metot olan readlines() ise bize bir liste veriyor. readline() metodundan farklı olarak readlines() metodu dosyanın tamamını bir çırpıda okuyor.

Bu üç metot arasından, adı readlines() olanının, dosyaların herhangi bir yerinde değişiklik yapmak konusunda bize yardımcı olabileceğini tahmin etmiş olabilirsiniz. Çünkü dediğimiz gibi readlines() metodu bize bir dosyanın içeriğini liste halinde veriyor. Bildiğiniz gibi listeler, üzerinde değişiklik yapılabilen veri tipleridir. Listelerin bu özelliğinden yararlanarak, dosyaların herhangi bir yerinde yapmak istediğimiz değişiklikleri rahatlıkla yapabiliriz. Şimdi dikkatlice bakın şu kodlara:

```
with open("fihrist.txt", "r+") as f:
    veri = f.readlines()
    veri.insert(2, "Sedat Köz\t: 0322 234 45 45\n")
    f.seek(0)
    f.writelines(veri)
```

Bu kodları bir dosyaya kaydedip çalıştırdıysanız, istediğimiz işlemi başarıyla yerine getirdiğini görmüşsünüzdür. Peki ama bu kodlar nasıl çalışıyor?

Yukarıdaki kodlarda dikkatimizi çeken pek çok özellik var. İlk olarak gözümüze çarpan şey, dosyayı *"r+"* kipinde açmış olmamız. Bu şekilde dosyayı hem okuma hem de yazma kipinde açmış oluyoruz. Çünkü dosyada aynı anda hem okuma hem de yazma işlemleri gerçekleştireceğiz.

Daha sonra şöyle bir satır yazdık:

```
veri = f.readlines()
```

Bu sayede dosyadaki bütün verileri bir liste olarak almış olduk. Liste adlı veri tipi ile ne yapabiliyorsak, bu şekilde aldığımız dosya içeriği üzerinde de aynı şeyleri yapabiliriz. Bizim amacımız bu listenin *2.* sırasına yeni bir satır eklemek. Bu işlemi listelerin insert() adlı metodu yardımıyla rahatlıkla yapabiliriz:

```
veri.insert(2, "Sedat Köz\t: 0322 234 45 45\n")
```

Bu şekilde liste üzerinde istediğimiz değişiklikleri yaptıktan sonra tekrar dosyanın başına dönmemiz lazım. Çünkü readlines() metoduyla dosyayı bir kez tam olarak okuduktan sonra imleç o anda dosyanın en sonunda bulunuyor. Eğer dosyanın en başına dönmeden herhangi bir yazma işlemi gerçekleştirirsek, yazılan veriler dosyanın sonuna eklenecektir. Bizim yapmamız gereken şey

dosyanın en başına sarıp, değiştirilmiş verilerin dosyaya yazılmasını sağlamak olmalı. Bunu da şu satır yardımıyla yapıyoruz:

```
f.seek(0)
```

Son olarak da bütün veirleri dosyaya yazıyoruz:

```
f.writelines(veri)
```

Şimdiye kadar dosyaya yazma işlemleri için write() adlı bir metottan yararlanmıştık. Burada ise writelines() adlı başka bir metot görüyoruz. Peki bu iki metot arasındaki fark nedir?

write() metodu bir dosyaya yalnızca karakter dizilerini yazabilir. Bu metot yardımıyla dosyaya liste tipinde herhangi bir veri yazamazsınız. Eğer mutlaka write() metodunu kullanmak isterseniz, liste üzerinde bir for döngüsü kurmanız gerekir. O zaman yukarıdaki kodları şöyle yazmanız gerekir:

```
with open("fihrist.txt", "r+") as f:
    veri = f.readlines()
    veri.insert(2, "Sedat Köz\t: 0322 234 45 45\n")
    f.seek(0)
    for öğe in veri:
        f.write(öğe)
```

writelines() adlı metot ise bize dosyaya liste tipinde verileri yazma imkanı verir. Dolayısıyla herhangi bir döngü kurmak zorunda kalmadan listeleri dosyalarımıza yazabiliriz.

Böylece Python'da dosyaların herhangi bir yerine nasıl yazabileceğimizi öğrenmiş olduk. Bu arada eğer isteseydik yukarıdaki kodları şöyle de yazabilirdik:

```
with open("fihrist.txt", "r") as f:
    veri = f.readlines()

with open("fihrist.txt", "w") as f:
    veri.insert(2, "Sedat Köz\t: 0322 234 45 45\n")
    f.writelines(veri)
```

Bir önceki kodlardan farklı olarak bu kodlarda dosyamızı önce okuma kipinde açıp verileri *veri* adlı bir değişken içinde sakladık. Ardından aynı dosyayı bir kez de yazma kipinde açarak, gerekli değişiklikleri liste üzerinde gerçekleştirdikten sonra bütün verileri dosyaya yazdık.

Unutmayın, Python'da herhangi bir işlemi pek çok farklı şekilde gerçekleştirebilirsiniz. Biz yukarıda olası yöntemlerden bazılarını ele aldık. Zaten bütün yöntemleri tek tek göstermemiz pek mümkün olmazdı. Siz dosyalara ilişkin bilgilerinizi ve farklı araçları kullanarak aynı işlemleri çok daha farklı şekillerde de yapabilirsiniz. Yani karşı karşıya olduğunuz duruma değerlendirip, yukarıdaki kodlardan uygun olanını veya kendi bulduğunuz bambaşka bir yöntemi kullanabilirsiniz.

Bu arada, aslında yukarıdaki kodlarda uyguladığımız yöntem biraz güvensiz. Çünkü aynı dosyayı hem okuyup hem de bu dosyaya yeni veri ekliyoruz. Eğer bu işlemlerin herhangi bir aşamasında bir hata oluşursa, bütün değişiklikleri dosyaya işleyemeden dosya içeriğini tümden kaybedebiliriz. Bu tür risklere karşı en uygun çözüm, okuma ve yazma işlemlerini ayrı dosyalar üzerinde gerçekleştirmektir. Bunun nasıl yapılacağından biraz sonra söz edeceğiz. Biz şimdi başka bir konuya değinelim.

Dosyaya Erişme Kipleri

Dosyalar konusunu anlatırken yukarıda verdiğimiz örneklerden de gördüğünüz gibi, Python'da dosyalara erişimin türünü ve niteliğini belirleyen bazı kipler var. Bu kipler dosyaların açılırken hangi yetkilere sahip olacağını veya olmayacağını belirliyor. Gelin isterseniz bu kipleri tek tek ele alalım.

Kip	Açıklaması
"r"	Bu öntanımlı kiptir. Bu kip dosyayı okuma yetkisiyle açar. Ancak bu kipi kullanabilmemiz için, ilgili dosyanın disk üzerinde halihazırda var olması gerekir. Eğer bu kipte açılmak istenen dosya mevcut değilse Python bize bir hata mesajı gösterecektir. Dediğimiz gibi, bu öntanımlı kiptir. Dolayısıyla dosyayı açarken herhangi bir kip belirtmezsek Python dosyayı bu kipte açmak istediğimizi varsayacaktır.
"w"	Bu kip dosyayı yazma yetkisiyle açar. Eğer belirttiğiniz adda bir dosya zaten disk üzerinde varsa, Python hiçbir şey sormadan dosya içeriğini silecektir. Eğer belirttiğiniz adda bir dosya diskte yoksa, Python o adda bir dosyayı otomatik olarak oluşturur.
"a"	Bu kip dosyayı yazma yetkisiyle açar. Eğer dosya zaten disk üzerinde mevcutsa içeriğinde herhangi bir değişiklik yapılmaz. Bu kipte açtığınız bir dosyaya eklediğiniz veriler varolan verilere ilave edilir. Eğer belirttiğiniz adda bir dosya yoksa Python otomatik olarak o adda bir dosyayı sizin için oluşturacaktır.
"x"	Bu kip dosyayı yazma yetkisiyle açar. Eğer belirttiğiniz adda bir dosya zaten disk üzerinde varsa, Python varolan dosyayı silmek yerine size bir hata mesajı gösterir. Zaten bu kipin *"w"* kipinden farkı, varolan dosyaları silmemesidir. Eğer belirttiğiniz adda bir dosya diskte yoksa, bu kip yardımıyla o ada sahip bir dosya oluşturabilirsiniz.
"r+"	Bu kip, bir dosyayı hem yazma hem de okuma yetkisiyle açar. Bu kipi kullanabilmeniz için, belirttiğiniz dosyanın disk üzerinde mevcut olması gerekir.
"w+"	Bu kip bir dosyayı hem yazma hem de okuma yetkisiyle açar. Eğer dosya mevcutsa içerik silinir, eğer dosya mevcut değilse oluşturulur.
"a+"	Bu kip bir dosyayı hem yazma hem de okuma yetkisiyle açar. Eğer dosya zaten disk üzerinde mevcutsa içeriğinde herhangi bir değişiklik yapılmaz. Bu kipte açtığınız bir dosyaya eklediğiniz veriler varolan verilere ilave edilir. Eğer belirttiğiniz adda bir dosya yoksa Python otomatik olarak o adda bir dosyayı sizin için oluşturacaktır.
"x+"	Bu kip dosyayı hem okuma hem de yazma yetkisiyle açar. Eğer belirttiğiniz adda bir dosya zaten disk üzerinde varsa, Python varolan dosyayı silmek yerine size bir hata mesajı gösterir. Zaten bu kipin *"w+"* kipinden farkı, varolan dosyaları silmemesidir. Eğer belirttiğiniz adda bir dosya diskte yoksa, bu kip yardımıyla o ada sahip bir dosya oluşturup bu dosyayı hem okuma hem de yazma yetkisiyle açabilirsiniz.
"rb"	Bu kip, metin dosyaları ile ikili (*binary*) dosyaları ayırt eden sistemlerde ikili dosyaları

	okuma yetkisiyle açmak için kullanılır. *"r"* kipi için söylenenler bu kip için de geçerlidir.
"wb"	Bu kip, metin dosyaları ile ikili dosyaları ayırt eden sistemlerde ikili dosyaları yazma yetkisiyle açmak için kullanılır. *"w"* kipi için söylenenler bu kip için de geçerlidir.
"ab"	Bu kip, metin dosyaları ile ikili dosyaları ayırt eden sistemlerde ikili dosyaları yazma yetkisiyle açmak için kullanılır. *"a"* kipi için söylenenler bu kip için de geçerlidir.
"xb"	Bu kip, metin dosyaları ile ikili dosyaları ayırt eden sistemlerde ikili dosyaları yazma yetkisiyle açmak için kullanılır. *"x"* kipi için söylenenler bu kip için de geçerlidir.
"rb+"	Bu kip, metin dosyaları ile ikili dosyaları ayırt eden sistemlerde ikili dosyaları hem okuma hem de yazma yetkisiyle açmak için kullanılır. *"r+"* kipi için söylenenler bu kip için de geçerlidir.
"wb+"	Bu kip, metin dosyaları ile ikili dosyaları ayırt eden sistemlerde ikili dosyaları hem okuma hem de yazma yetkisiyle açmak için kullanılır. *"w+"* kipi için söylenenler bu kip için de geçerlidir.
"ab+"	Bu kip, metin dosyaları ile ikili dosyaları ayırt eden sistemlerde ikili dosyaları hem okuma hem de yazma yetkisiyle açmak için kullanılır. *"a+"* kipi için söylenenler bu kip için de geçerlidir.
"xb+"	Bu kip, metin dosyaları ile ikili dosyaları ayırt eden sistemlerde ikili dosyaları hem okuma hem de yazma yetkisiyle açmak için kullanılır. *"x+"* kipi için söylenenler bu kip için de geçerlidir.

Bütün bu tabloya baktığınızda ilk bakışta sanki bir sürü farklı erişim kipi olduğunu düşünmüş olabilirsiniz. Ama aslında tabloyu biraz daha incelerseniz, temel olarak *"r"*, *"w"*, *"a"*, "*x*" ve *"b"* kiplerinin olduğunu, geri kalan kiplerin ise bunların kombinasyonlarından oluştuğunu göreceksiniz.

Daha önce de söylediğimiz gibi, dosya işlemlerini pek çok farklı yöntemle gerçekleştirebilirsiniz. Yukarıdaki tabloyu dikkatlice inceleyerek, yapmak istediğiniz işleme uygun kipi rahatlıkla seçebilirsiniz.

Bu arada, yukarıdaki tabloda değindiğimiz ikili (*binary*) dosyalardan henüz söz etmedik. Bir sonraki bölümde bu dosya türünü de ele alacağız.

başa dön

Dosyaların Metot ve Nitelikleri

Dosyalara ilişkin olarak bir önceki bölümde anlattığımız şeylerin kafanıza yatması açısından size şu bilgiyi de verelim: Dosyalar da, tıpkı karakter dizileri ve listeler gibi, Python programlama dilindeki veri tiplerinden biridir. Dolayısıyla tıpkı karakter dizileri ve listeler gibi, dosya (*file*) adlı bu veri tipinin de bazı metotları ve nitelikleri vardır. Gelin isterseniz bu metot ve niteliklerin neler olduğunu şöyle bir listeleyelim:

```
dosya = open("falanca_dosya.txt", "w")
print(*[metot for metot in dir(dosya) if not metot.startswith("_")], sep="\n")
```

Bu kodlar, dosya adlı veri tipinin bizi ilgilendiren bütün metotlarını alt alta ekrana basacaktır. Eğer yukarıdaki kodları anlamakta zorluk çektiyseniz, bunları şöyle de yazabilirsiniz:

```
dosya = open("falanca_dosya.txt", "w")

for metot in dir(dosya):
    if not metot.startswith("_"):
        print(metot, sep="\n")
```

Bildiğiniz gibi bu kodlar bir öncekiyle tamamen aynı anlama geliyor.

Bu kodları çalıştırdığınızda karşınıza pek çok metot çıkacak. Biz buraya gelene kadar bu metotların en önemlilerini zaten inceledik. İncelemediğimiz yalnızca birkaç önemli metot (ve nitelik) kaldı. Gelin isterseniz henüz incelemediğimiz bu önemli metot ve nitelikleri gözden geçirelim.

closed() Metodu

Bu metot, bir dosyanın kapalı olup olmadığını sorgulamamızı sağlar. Dosya adının *f* olduğunu varsayarsak, bu metodu şöyle kullanıyoruz:

```
f.closed()
```

Eğer *f* adlı bu dosya kapalıysa *True* çıktısı, açıksa *False* çıktısı verilecektir.

readable() Metodu

Bu metot bir dosyanın okuma yetkisine sahip olup olmadığını sorgulamamızı sağlar. Eğer bir dosya *"r"* gibi bir kiple açılmışsa, yani o dosya 'okunabilir' özellikle ise bu metot bize *True* çıktısı verir. Ama eğer dosya yazma kipinde açılmışsa bu metot bize *False* çıktısı verecektir.

writable() Metodu

Bu metot bir dosyanın yazma yetkisine sahip olup olmadığını sorgulamamızı sağlar. Eğer bir dosya *"w"* gibi bir kiple açılmışsa, yani o dosya 'yazılabilir' özellikle ise bu metot bize *True* çıktısı verir. Ama eğer dosya okuma kipinde açılmışsa bu metot bize *False* çıktısı verecektir.

truncate() Metodu

Bu metot, henüz işlemediğimiz metotlar arasında en önemlilerinden biridir. Bu metot yardımıyla dosyalarımızı istediğimiz boyuta getirebiliyoruz.

İngilizcede *truncate* kelimesi 'budamak, kırpmak' gibi anlamlara gelir. Bu metodun yaptığı iş de bu anlamıyla uyumludur. Bu metodu temel olarak şöyle kullanıyoruz:

```
>>> with open("falanca.txt", "r+") as f:
...     f.truncate()
```

Bu komutu bu şekilde kullandığımızda dosyanın bütün içeriği silinecektir. Yani bu kodlar, sanki dosyayı *"w"* kipiyle açmışsınız gibi bir etki ortaya çıkaracaktır.

truncate() metodu yukarıda gördüğünüz şekilde parametresiz olarak kullanılabileceği gibi, parametreli olarak da kullanılabilir. Bu metodun parantezleri arasına, dosyanın kaç baytlık bir boyuta sahip olmasını istediğinizi yazabilirsiniz. Örneğin:

```
>>> with open("falanca.txt", "r+") as f:
...     f.truncate(10)
```

Bu kodlar, *falanca.txt* adlı dosyanın ilk *10* baytı dışındaki bütün verileri siler. Yani dosyayı yalnızca *10* baytlık bir boyuta sahip olacak şekilde kırpar.

Gelin isterseniz bu metotla ilgili bir örnek verelim. Elimizdeki dosyanın şu içeriğe sahip olduğunu varsayalım:

```
Ahmet Özbudak : 0533 123 23 34
Mehmet Sülün  : 0532 212 22 22
Sami Sam      : 0542 333 34 34
```

Amacımız dosyadaki şu iki satırı tamamen silmek:

```
Mehmet Sülün  : 0532 212 22 22
Sami Sam      : 0542 333 34 34
```

Yani dosyanın yeni içeriğinin tam olarak şöyle olmasını istiyoruz:

```
Ahmet Özbudak : 0533 123 23 34
```

Bunun için truncate() metodundan yararlanarak şu kodları yazabiliriz:

```
with open("fihrist.txt", "r+") as f:
    f.readline()
    f.seek(f.tell())
    f.truncate()
```

Bu kodları bir dosyaya kaydedip çalıştırdığınızda, istediğiniz sonucu elde ettiğinizi göreceksiniz.

Burada sırasıyla şu işlemleri gerçekleştirdik:

1. Önce dosyamızı hem okuma hem de yazma kipinde açtık. Çünkü aynı dosya üzerinde hem okuma hem de yazma işlemleri gerçekleştireceğiz:

   ```
   with open("fihrist.txt", "r+") as f:
   ```

2. Ardından dosyadan tek bir satır okuduk:

   ```
   f.readline()
   ```

3. Daha sonra, truncate() metodunun imleç konumundan itibaren kırpma işlemi gerçekleştirebilmesi için imleci dosya içinde o anda bulunduğumuz konuma, yani ikinci satırın başına getirdik. Bildiğiniz gibi dosyaların tell() metodu, o anda dosya içinde hangi konumda bulunduğumuzu bildiriyor. Biz biraz önce yazdığımız readline() komutu yardımıyla dosyadan bir satır okuduğumuz için, o anda ikinci satırın başında bulunuyoruz. İşte seek() metodunu ve tell() metodundan elde ettiğimiz bu konum bilgisini kullanarak imleci istediğimiz konuma getirdik:

   ```
   f.seek(f.tell())
   ```

4. İmleci istediğimiz konuma getirdiğimize göre artık kırpma işlemini gerçekleştirebiliriz:

   ```
   f.truncate()
   ```

Artık elimizde tek satırlık bir dosya var...

truncate() metodunun, yukarıda anlattığımızdan farklı bir özelliği daha var. Her ne kadar *truncate* kelimesi 'kırpmak' anlamına gelse ve bu metotla dosya boyutlarını küçültebilsek bile, bu metodu kullanarak aynı zamanda dosya boyutlarını artırabiliriz de. Örneğin boyutu *1* kilobayt olan bir dosyayı *3* kilobayta çıkarmak için bu metodu şöyle kullanabiliriz:

```
>>> f = open("fihrist.txt", "r+")
>>> f.truncate(1024*3)
>>> f.close()
```

Dosyanın boyutunu kontrol edecek olursanız, dosyanın gerçekten de 3 kilobayt'a çıktığını göreceksiniz. Peki bu metot bu işi nasıl yapıyor? Aslında bunun cevabı çok basit: Dosyanın sonuna gereken miktarda *0* ekleyerek... Zaten eğer *fihrist.txt* adlı bu dosyayı tekrar açıp okursanız bu durumu

kendiniz de görebilirsiniz:

```
>>> f = open("fihrist.txt")
>>> f.read()
```

Gördüğünüz gibi, dosya sıfırlarla dolu.

mode Niteliği

Bu nitelik, bize bir dosyanın hangi kipte açıldığına dair bilgi verir:

```
>>> f = open("falanca.txt")
>>> f.mode

'r'
```

Demek ki bu dosya *"r"* kipinde açılmış...

name Niteliği

Bu nitelik, bize bir dosyanın adını verir:

```
>>> f.name

'falanca.txt'
```

encoding Niteliği

Bu nitelik, bize bir dosyanın hangi dil kodlaması ile kodlandığını söyler:

```
>>> f.encoding

'utf-8'
```

veya:

```
>>> f.encoding

'cp1254' #Windows
```

Not

Bu 'dil kodlaması' konusunu ilerleyen sayfalarda ayrıntılı olarak inceleyeceğiz.

Böylece dosyaların en önemli metot ve niteliklerini incelemiş olduk. Bu arada, gerek bu derste, gerekse önceki derslerde verdiğimiz örneklerden, ‘metot’ ile ‘nitelik’ kavramları arasındaki farkı anladığınızı zannediyorum. Metotlar bir iş yaparken, nitelikler bir değer gösterir. Nitelikler basit birer değişkenden ibarettir. Metotlar ise bir işin nasıl yapılacağı ile ilgili süreci tanımlar. Esasında bu ikisi arasındaki farkları çok fazla kafaya takmanıza gerek yok. Zamanla (özellikle de başka programların kaynak kodlarını incelemeye başladığınızda) bu ikisi arasındaki farkı bariz bir biçimde göreceksiniz. O noktaya geldiğinizde, zaten kavramlar arasındaki farkları görmeniz konusunda biz de size yardımcı olmaya çalışacağız.

başa dön

İkili (*Binary*) Dosyalar

Dosyalar çoğunlukla iki farklı sınıfa ayrılır: Metin dosyaları ve ikili dosyalar. Metin dosyaları derken neyi kastettiğimiz az çok anlaşılıyor. Eğer bir dosyayı bir metin düzenleyici ile açtığınızda herhangi bir dilde yazılmış 'okunabilir' bir metin görüyorsanız, o dosya bir metin dosyasıdır. Mesela Notepad, Gedit, Kwrite veya benzeri metin düzenleyicileri kullanarak oluşturduğunuz dosyalar birer metin dosyasıdır. Şimdiye kadar verdiğimiz bütün örnekler metin dosyalarını içeriyordu. Peki 'ikili' (*binary*) dosya ne demek?

İkili dosyalar ise, metin dosyalarının aksine, metin düzenleyicilerle açılamayan, açılmaya çalışıldığında ise çoğunlukla anlamsız karakterler içeren bir dosya türüdür. Resim dosyaları, müzik dosyaları, video dosyaları, MS Office dosyaları, LibreOffice dosyaları, OpenOffice dosyaları, vb. ikili dosyalara örnektir.

Önceki bölümlerde de ifade ettiğimiz gibi, bilgisayarlar yalnızca sayılarla işlem yapabilir. Bilgisayarların üzerinde işlem yapabildiği bu sayıların *0* ve *1* adlı iki sayı olduğunu biliyoruz.

Peki bu iki farklı sayıyı kullanarak neler yapabiliriz? Aslında, bu iki farklı sayıyı kullanarak her türlü işlemi yapabiliriz: Basit veya karmaşık aritmetik hesaplamalar, metin düzenleme, resim veya video düzenleme, web siteleri hazırlama, uzaya mekik gönderme... Bütün bu işlemleri sadece iki farklı sayı kullanarak yapabiliriz. Daha doğrusu bilgisayarlar yapabilir.

Durum böyle olmasına rağmen, ilk bilgisayarlar yalnızca hesaplama işlemleri için kullanılıyordu. Yani metin içeren işlemleri yapmak bilgisayarların değil, mesela daktiloların görevi olarak görülüyordu. Bu durumu telefon teknolojisi ile kıyaslayabilirsiniz. Bildiğiniz gibi, ilk telefonlar yalnızca iki kişi arasındaki sesli iletişimi sağlamak için kullanılıyordu. Ama yeni nesil telefonlar artık ikiden fazla kişi arasındaki sesli ve görüntülü iletişimi sağlayabilmenin yanısıra, önceleri birbirinden farklı cihazlarla gerçekleştirilen işlemleri artık tek başına yerine getirebiliyor.

İlk bilgisayarlarda ise metinlerin, daha doğrusu karakterlerin görevi bir hayli sınırlıydı.

Başta da söylediğimiz gibi, çoğunlukla dosyalar iki farklı sınıfa ayrılır: Metin dosyaları ve ikili dosyalar. Ama işin aslı sadece tek bir dosya türü vardır: İkili dosyalar (*binary files*). Yani bilgisayarlar açısından bütün dosyalar, içlerinde ne olursa olsun, birer ikili dosyadır ve içlerinde sadece 0'ları ve 1'leri barındırır. İşte bu 0 ve 1'lerin ne anlama geleceğini, işletim sistemleri ve bu sistemler üzerine kurulu yazılımlar belirler. Eğer bir dosya metin dosyasıysa bu dosyadaki 0 ve 1'ler birer karakter/harf olarak yorumlanır. Ama eğer dosya bir ikili dosyaysa dosya içindeki 0 ve 1'ler özel birtakım veriler olarak ele alınır ve bu verileri okuyan yazılıma göre değer kazanır. Örneğin eğer ilgili dosya bir resim dosyasıyla, bu dosya herhangi bir resim görüntüleyici yazılım ile açıldığında karşımıza bir resim çıkar. Eğer ilgili dosya bir video dosyasıyla, bu dosya bir video görüntüleyici yazılım ile açıldığında karşımıza bir video çıkar. Bu olgudan bir sonraki bölümde daha ayrıntılı olarak söz edeceğiz. Biz şimdilik işin sadece pratiğine yoğunlaşalım ve temel olarak iki farklı dosya çeşidi olduğunu varsayalım: Metin dosyaları ve ikili dosyalar.

Buraya gelene kadar hep metin dosyalarından söz etmiştik. Şimdi ise ikili dosyalardan söz edeceğiz.

Hatırlarsanız metin dosyalarını açmak için temel olarak şöyle bir komut kullanıyorduk:

```
f = open(dosya_adı, 'r')
```

Bu şekilde bir metin dosyasını okuma kipinde açmış oluyoruz. Bir metin dosyasını değil de, ikili bir dosyayı açmak için ise şu komutu kullanacağız:

```
f = open(dosya_adı, 'rb')
```

Dosyaya erişme kiplerini gösterdiğimiz tabloda ikili erişim türlerini de verdiğimizi hatırlıyorsunuz.

Peki neden metin dosyaları ve ikili dosyalar için farklı erişim kipleri kullanıyoruz?

İşletim sistemleri satır sonları için birbirinden farklı karakterler kullanırlar. Örneğin GNU/Linux dağıtımlarında satır sonları *\n* karakteri ile gösterilir. Windows işletim sistemi ise satır sonlarını *\r\n* karakterleriyle gösterir. İşte Python herhangi bir dosyayı açarken, eğer o dosya bir metin dosyası ise, satır sonlarını gösteren karakterleri, dosyanın açıldığı işletim sistemine göre ayarlar. Yani satır sonlarını standart bir hale getirerek *\n* karakterine dönüştürür.

Metin dosyaları ile ikili dosyalar arasında önemli bir fark bulunur: Bir metin dosyasındaki ufak değişiklikler dosyanın okunamaz hale gelmesine yol açmaz. Olabilecek en kötü şey, değiştirilen karakterin okunamaz hale gelmesidir. Ancak ikili dosyalarda ufak değişiklikler dosyanın tümden bozulmasına yol açabilir. Dolayısıyla Python'ın yukarıda bahsedilen satır sonu değişiklikleri ikili dosyaların bozulmasına yol açabilir. Yani eğer siz ikili bir dosyayı *'rb'* yerine sadece *'r'* gibi bir kiple açarsanız dosyanın bozulmasına yol açabilirsiniz. İkili bir dosyayı *'rb'* (veya *'wb'*, *'ab'*, *'xb'*, vb.) gibi bir kipte açtığınızda Python satır sonlarına herhangi bir değiştirme-dönüştürme işlemi uygulamaz. Böylece dosya bozulma riskiyle karşı karşıya kalmaz. O yüzden, metin dosyalarını ve ikili dosyaları açarken farklı kipler kullanmamız gerektiğine dikkat ediyoruz.

İkili Dosyalarla Örnekler

Gelin isterseniz bu noktada birkaç örnek verelim.

PDF Dosyalarından Bilgi Alma

Tıpkı resim, müzik ve video dosyaları gibi, *PDF* dosyaları da birer ikili dosyadır. O halde hemen önümüze bir *PDF* dosyası alalım ve bu dosyayı okuma kipinde açalım:

```
>>> f = open("falanca.pdf", "rb")
```

Şimdi de bu dosyadan *10* baytlık bir veri okuyalım:

```
>>> f.read(10)

b'%PDF-1.3\n4'
```

Bu çıktıda gördüğünüz *'b'* işaretine şimdilik takılmayın. Birazdan bunun ne olduğunu bütün ayrıntılarıyla anlatacağız. Biz bu harfin, elimizdeki verinin bayt türünde bir veri olduğunu gösteren bir işaret olduğunu bilelim yeter.

Gördüğünüz gibi, bir *PDF* dosyasının ilk birkaç baytını okuyarak hem dosyanın bir *PDF* belgesi olduğunu teyit edebiliyoruz, hem de bu *PDF* belgesinin, hangi *PDF* sürümü ile oluşturulduğunu anlayabiliyoruz. Buna göre bu belge PDF talimatnamesinin 1.3 numaralı sürümü ile oluşturulmuş.

Eğer biz bu belgeyi bir ikili dosya olarak değil de bir metin dosyası olarak açmaya çalışsaydık şöyle bir hata alacaktık:

```
>>> f = open("falanca.pdf")
>>> okunan = f.read()

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Python33\lib\encodings\cp1254.py", line 23, in decode
    return codecs.charmap_decode(input,self.errors,decoding_table)[0]
UnicodeDecodeError: 'charmap' codec can't decode byte 0x9d in position 527: char
acter maps to <undefined>
```

Python'ın bu dosyanın bir ikili dosya olduğu konusunda bilgilendirerek, dosyanın düzgün bir şekilde açılıp okunabilmesini sağlıyoruz.

Gelin bu *PDF* belgesi üzerinde biraz daha çalışalım.

PDF belgelerinde, o belge hakkında bazı önemli bilgiler veren birtakım özel etiketler bulunur. Bu etiketler şunlardır:

Etiket	Anlamı
/Creator	Belgeyi oluşturan yazılım
/Producer	Belgeyi PDF'e çeviren yazılım
/Title	Belgenin başlığı
/Author	Belgenin yazarı
/Subject	Belgenin konusu
/Keywords	Belgenin anahtar kelimeleri
/CreationDate	Belgenin oluşturulma zamanı
/ModDate	Belgenin değiştirilme zamanı

Bu etiketlerin tamamı bütün *PDF* dosyalarında tanımlı değildir. Ama özellikle */Producer* etiketi her *PDF* dosyasında bulunur.

Şimdi örnek olması bakımından elimize bir *PDF* dosyası alalım ve bunu güzelce okuyalım:

```
>>> f = open("falanca.pdf", "rb")
```

```
>>> okunan = f.read()
```

Şimdi de */Producer* ifadesinin dosya içinde geçtiği noktanın sıra numarasını bulalım. Bildiğiniz gibi, dosyaların read() metodu bize bir karakter dizisi verir. Yine bildiğiniz gibi, karakter dizilerinin index() metodu yardımıyla bir öğenin karakter dizisi içinde geçtiği noktayı bulabiliyoruz. Yani:

```
>>> producer_index = okunan.index(b"/Producer")
```

Burada */Producer* ifadesinin başına *'b'* harfini yerleştirmeyi unutmuyoruz. Çünkü şu anda yaptığımız işlem ikili bir dosya içinde geçen birtakım baytları arama işlemidir.

producer_index değişkeni, '/Producer' ifadesinin ilk baytının dosya içindeki konumunu tutuyor. Kontrol edelim:

```
>>> producer_index

4077883
```

Bu değerin gerçekten de '/Producer' ifadesinin ilk baytını depoladığını teyit edelim:

```
>>> okunan[producer_index]

47
```

Daha önce de dediğimiz gibi, bilgisayarlar yalnızca sayıları görür. Bu sayının hangi karaktere karşılık geldiğini bulmak için chr() fonksiyonundan yararlanabilirsiniz:

```
>>> chr(okunan[producer_index])

'/'
```

Gördüğünüz gibi, gerçekten de *producer_index* değişkeni '/Producer' ifadesinin ilk baytının dosya içindeki konumunu gösteriyor. Biz bu konumu ve bu konumun 50-60 bayt ötesini sorgularsak, *PDF* belgesini üreten yazılımın adına ulaşabiliriz. Dikkatlice bakın:

```
>>> okunan[producer_index:producer_index+50]

b'/Producer (Acrobat Distiller 2.0 for Macintosh)\r/T'
```

Hatta eğer bu çıktı üzerine split() metodunu uygularsak, çıktıyı daha kullanışlı bir hale getirebiliriz:

```
>>> producer = okunan[producer_index:producer_index+50].split()
>>> producer

[b'/Producer', b'(Acrobat', b'Distiller', b'2.0', b'for', b'Macintosh)', b'/T']
```

Bu şekilde, ihtiyacımız olan bilginin istediğimiz parçasına kolayca ulaşabiliriz:

```
>>> producer[0]

b'/Producer'

>>> producer[1]

b'(Acrobat'

>>> producer[1:3]

[b'(Acrobat', b'Distiller']
```

Elbette bu yöntem, bir *PDF* dosyasından gerekli etiketleri almanın en iyi yöntemi değildir. Ama henüz Python bilgimiz bu kadarını yapmamıza müsaade ediyor. Ancak yine de, yukarıda örnek, bir ikili dosyadan nasıl veri alınacağı konusunda size iyi bir fikir verecektir.

Resim Dosyalarının Türünü Tespit Etme

Dediğimiz gibi, resim dosyaları, müzik dosyaları, video dosyaları ve benzeri dosyalar birer ikili dosyadır. Mesela resim dosyalarını ele alalım. Diyelim ki, resimlerin hangi türde olduğunu tespit eden bir program yazmak istiyorsunuz. Yani yazdığınız bu programla bir resim dosyasının *PNG* mi, *JPEG* mi, *TIFF* mi, yoksa *BMP* mi olduğunu anlamak istiyorsunuz.

Peki bir resim dosyasının hangi türde olduğunu bulmak için uzantısına baksanız olmaz mı? Asla unutmayın dosya uzantıları ile dosya biçimleri arasında doğrudan bir bağlantı yoktur. O yüzden dosya uzantıları, dosya biçimini anlamak açısından güvenilir bir yöntem değildir. Bir resim dosyasının sonuna hangi uzantıyı getirirseniz getirin, o dosya bir resim dosyasıdır. Yani mesela bir resim dosyasının uzantısı yanlışlıkla veya bilerek *.doc* olarak değiştirilmişse, o dosya bir WORD dosyası haline gelmez. İşte yazacağınız program, bir resim dosyasının uzantısı ne olursa olsun, hatta dosyanın bir uzantısı olmasa bile, o dosyanın hangi türde olduğunu söyleyebilecek.

Bir resim dosyasının hangi türde olduğunu anlayabilmek için ilgili dosyanın ilk birkaç baytını okumanız yeterlidir. Bu birkaç bayt içinde o resim dosyasının türüne dair bilgileri bulabilirsiniz.

Resim dosyalarının türlerini birbirinden ayırt etmenizi sağlayacak verilerin ne olduğunu, ilgili resim türünün teknik şartnamesine bakarak öğrenebilirsiniz. Ancak teknik şartnameler genellikle okuması zor metinlerdir. Bu yüzden, doğrudan şartnameyi okumak yerine, Internet üzerinde kısa bir araştırma yaparak konuyu daha kolay anlamanızı sağlayacak yardımcı belgelerden de yardım alabilirsiniz.

JPEG

JPEG şartnamesini http://www.jpeg.org/public/jfif.pdf adresinde bulabilirsiniz. *JPEG* dosya biçimini daha iyi anlamanızı sağlayacak yardımcı kaynaklar ise şunlardır:

1. http://www.faqs.org/faqs/jpeg-faq/part1/section-15.html
2. http://www.mikekunz.com/image_file_header.html

Yukarıda verdiğimiz adreslerdeki bilgilere göre bir *JPEG* dosyasının en başında şu veriler bulunur:

```
FF  D8     FF     E0     ?  ?  4A     46     49     46     00
```

Ancak eğer ilgili *JPEG* dosyası bir CANON fotograf makinesi ile oluşturulmuşsa bu veri dizisi şöyle de olabilir:

```
FF  D8     FF     E0     ?  ?  45  78  69  66  00
```

Burada soru işareti ile gösterdiğimiz kısım, yani dosyanın *5.* ve *6.* baytları farklı *JPEG* dosyalarında birbirinden farklı olabilir. Dolayısıyla bir *JPEG* dosyasını başka resim dosyalarından ayırabilmek için ilginin ilk dört baytına bakmamız, sonraki iki baytı atlamamız ve bunlardan sonra gelen beş baytı kontrol etmemiz yeterli olacaktır.

Yukarıda gördükleriniz birer on altılı (*hex*) sayıdır. Bunlar onlu düzende sırasıyla şu sayılara karşılık gelir:

```
255 216 255 224 ? ? 74 70 73 70 0
255 216 255 224 ? ? 45 78 69 66 0 #canon
```

Bu diziler içinde özellikle şu dört sayı bizi yakından ilgilendiriyor:

```
74 70 73 70
45 78 69 66 #canon
```

Bu sayılar sırasıyla ‘J’, ‘F’, ‘I’, ‘F’ ve ‘E’, ‘x’, ‘i’, ‘f’ harflerine karşılık gelir. Yani bir *JPEG* dosyasını ayırt edebilmek için ilgili dosyanın 7-10 arası baytlarının ne olduğuna bakmamız yeterli olacaktır. Eğer bu aralıkta ‘JFIF’ veya ‘Exif’ ifadeleri varsa, o dosya bir *JPEG* dosyasıdır. Buna göre şöyle bir kod yazabiliriz:

```
f = open(dosya_adı, 'rb')
data = f.read(10)
if data[6:11] in [b"JFIF", b"Exif"]:
    print("Bu dosya JPEG!")
else:
    print("Bu dosya JPEG değil!")
```

Burada herhangi bir resim dosyasının ilk on baytını okuduk öncelikle:

```
data = f.read(10)
```

Çünkü aradığımız bilgiler ilk on bayt içinde yer alıyor.

Daha sonra okuduğumuz kısmın 7 ila 10. baytları arasında kalan verinin ne olduğuna bakıyoruz:

```
if data[6:11] in [b"JFIF", b"Exif"]:
    ...
```

Eğer ilgili aralıkta 'JFIF' veya 'Exif' baytları yer alıyorsa bu dosyanın bir *JPEG* dosyası olduğuna karar veriyoruz.

Yukarıdaki kodları elinizdeki bir *JPEG* dosyasına uygulayarak kendi kendinize pratik yapabilirsiniz.

Mesela benim elimde *d1.jpg*, *d2.jpg* ve *d3.jpeg* adlı üç farklı *JPEG* dosyası var:

```
dosyalar = ["d1.jpg", "d2.jpg", "d3.jpeg"]
```

Bu dosyaların ilk onar baytını okuyorum:

```
for f in dosyalar:
    okunan = open(f, 'rb').read(10)
    print(okunan)
```

Buradan şu çıktıyı alıyorum:

```
d1.jpg      b'\xff\xd8\xff\xe0\x00\x10JFIF'
d2.jpg      b'\xff\xd8\xff\xe1T\xaaExif'
d3.jpeg      b'\xff\xd8\xff\xe0\x00\x10JFIF'
```

Gördüğünüz gibi bu çıktılar yukarıda *JPEG* dosyalarına ilişkin olarak verdiğimiz bayt dizilimi ile uyuşuyor. Mesela ilk dosyayı ele alalım:

```
d1.jpg      b'\xff\xd8\xff\xe0\x00\x10JFIF'
```

Burada şu baytlar var:

```
\xff \xd8 \xff \xe0 \x00 \x10 J F I F
```

Sayıların başındaki *\x* işaretleri bunların birer on altılı sayı olduğunu gösteren bir işarettir. Dolayısıyla yukarıdakileri daha net inceleyebilmek için şöyle de yazabiliriz:

```
ff d8 ff e0 00 10 J F I F
```

Şimdi de ikinci dosyanın çıktısını ele alalım:

```
d2.jpg      b'\xff\xd8\xff\xe1T\xaaExif'
```

Burada da şu baytlar var:

```
ff d8 ff e1T aa E x i f
```

İşte dosyaların türünü ayırt etmek için bu çıktılardaki son dört baytı kontrol etmemiz yeterli olacaktır:

```
for f in dosyalar:
    okunan = open(f, 'rb').read(10)
    if okunan[6:11] in [b'JFIF', b'Exif']:
        print("Evet {} adlı dosya bir JPEG!".format(f))
    else:
        print("{} JPEG değil!".format(f))
```

Bu kodları elinizde bulunan farklı türdeki dosyalara uygulayarak, aldığınız çıktıları inceleyebilirsiniz.

PNG

PNG dosya biçiminin teknik şartnamesine http://www.libpng.org/pub/png/spec/ adresinden ulaşabilirsiniz.

Ayrıca yardımcı kaynak olarak da http://www.fileformat.info/format/png/egff.htm adresindeki belgeyi kullanabilirsiniz.

Şartnamade, http://www.libpng.org/pub/png/spec/1.2/PNG-Rationale.html#R.PNG-file-signature sayfasındaki bilgiye göre bir *PNG* dosyasının ilk *8* baytı mutlaka aşağıdaki değerleri içeriyor:

onlu değer	137 80 78 71 13 10 26 10
on altılı değer	89 50 4e 47 0d 0a 1a 0a
karakter değeri	\211 P N G \r \n \032 \n

Şimdi elimize herhangi bir *PNG* dosyası alarak bu durumu teyit edelim:

```
>>> f = open(falanca.png", "rb")

>>> okunan = f.read(8)
```

Şartnamede de söylendiği gibi, bir *PNG* dosyasını öteki türlerden ayırt edebilmek için dosyanın ilk *8* baytına bakmamız yeterli olacaktır. O yüzden biz de yukarıdaki kodlarda sadece ilk *8* baytı okumakla yetindik.

Bakalım ilk *8* baytta neler varmış:

```
>>> okunan

b'\x89PNG\r\n\x1a\n'
```

Bu değerin, şartnamedeki karakter değeri ile aynı olup olmadığını sorgulayarak herhangi bir dosyanın *PNG* olup olmadığına karar verebilirsiniz:

```
>>> okunan == b"\211PNG\r\n\032\n"

True
```

Dolayısıyla şuna benzer bir kod yazarak, farklı resim dosyalarının türünü tespit edebilirsiniz:

```
for f in dosyalar:
    okunan = open(f, 'rb').read(10)
    if okunan[6:11] in [b'JFIF', b'Exif']:
        print("{} adlı dosya bir JPEG!".format(f))
    elif okunan[:8] == b"\211PNG\r\n\032\n":
        print("{} adlı dosya bir PNG!".format(f))
    else:
        print("Türü bilinmeyen dosya: {}".format(f))
```

Bu kodlarda bir resim dosyasının ilk *10* baytını okuduk. 7-11 arası baytların içinde 'JFIF' veya 'Exif' baytları varsa o dosyanın bir *JPEG* olduğuna; ilk *8* bayt *b"211PNGrn032n"* adlı bayt dizisine eşitse de o dosyanın bir *PNG* olduğuna karar veriyoruz.

GIF

GIF şartnamesine http://www.w3.org/Graphics/GIF/spec-gif89a.txt adresinden ulaşabilirsiniz.

Bir dosyanın *GIF* olup olmadığına karar verebilmek için ilk *3* baytını okumanız yeterli olacaktır. Standart bir *GIF* dosyasının ilk üç baytı 'G', 'I' ve 'F' karakterlerinden oluşur. Dosyanın sonraki *3* baytı ise *GIF*'in sürüm numarasını verir. 30/10/2013 itibariyle *GIF* standardının şu sürümleri bulunmaktadır:

1. 87a - Mayıs 1987
2. 89a - Temmuz 1989

Dolayısıyla standart bir *GIF* dosyasının ilk *6* baytı şöyledir:

'GIF87a' veya 'GIF89a'

Eğer bir dosyanın *GIF* olup olmadığını anlamak isterseniz dosyanın ilk *3* veya *6* baytını denetlemeniz yeterli olacaktır:

```
for f in dosyalar:
    okunan = open(f, 'rb').read(10)
    if okunan[6:11] in [b'JFIF', b'Exif']:
        print("{} adlı dosya bir JPEG!".format(f))
    elif okunan[:8] == b"\211PNG\r\n\032\n":
        print("{} adlı dosya bir PNG!".format(f))
    elif okunan[:3] == b'GIF':
```

```
        print("{} adlı dosya bir GIF!".format(f))
    else:
        print("Türü bilinmeyen dosya: {}".format(f))
```

TIFF

TIFF şartnamesine http://partners.adobe.com/public/developer/en/tiff/TIFF6.pdf adresinden ulaşabilirsiniz. Bu şartnameye göre bir *TIFF* dosyası şunlardan herhangi biri ile başlar:

1. 'II'
2. 'MM'

Dolayısıyla, bir *TIFF* dosyasını tespit edebilmek için dosyanın ilk *2* baytına bakmanız yeterli olacaktır:

```
for f in dosyalar:
    okunan = open(f, 'rb').read(10)
    if okunan[6:11] in [b'JFIF', b'Exif']:
        print("{} adlı dosya bir JPEG!".format(f))
    elif okunan[:8] == b"\211PNG\r\n\032\n":
        print("{} adlı dosya bir PNG!".format(f))
    elif okunan[:3] == b'GIF':
        print("{} adlı dosya bir GIF!".format(f))
    elif okunan[:2] in [b'II', b'MM']:
        print("{} adlı dosya bir TIFF!".format(f))
    else:
        print("Türü bilinmeyen dosya: {}".format(f))
```

BMP

BMP türündeki resim dosyalarına ilişkin bilgi için http://www.digitalpreservation.gov/formats/fdd/fdd000189.shtml adresine başvurabilirsiniz.

Buna göre, *BMP* dosyaları 'BM' ile başlar. Yani:

```
for f in dosyalar:
    okunan = open(f, 'rb').read(10)
    if okunan[6:11] in [b'JFIF', b'Exif']:
        print("{} adlı dosya bir JPEG!".format(f))
    elif okunan[:8] == b"\211PNG\r\n\032\n":
        print("{} adlı dosya bir PNG!".format(f))
    elif okunan[:3] == b'GIF':
        print("{} adlı dosya bir GIF!".format(f))
    elif okunan[:2] in [b'II', b'MM']:
```

```
        print("{} adlı dosya bir TIFF!".format(f))
    elif okunan[:2] in [b'BM']:
        print("{} adlı dosya bir BMP!".format(f))
    else:
        print("Türü bilinmeyen dosya: {}".format(f))
```

Gördüğünüz gibi ikili dosyalar, baytların özel bir şekilde dizildiği ve özel bir şekilde yorumlandığı bir dosya türüdür. Dolayısıyla ikili dosyalarla çalışabilmek için, ikili dosyanın bayt dizilimini yakından tanımak gerekiyor.

başa dön

Basit bir İletişim Modeli

Bu bölümde, bilgisayarların çalışma mantığını, verileri nasıl işlediğini, sayılarla karakter dizilerini nasıl temsil ettiğini daha iyi ve daha net bir şekilde anlayabilmek için basit bir iletişim modeli kuracağız.

Şimdi şöyle bir durum hayal edin: Diyelim ki, hatlar üzerinden iletilen elektrik akımı yoluyla bir arkadaşınızla haberleşmenizi sağlayacak bir sistem tasarlıyorsunuz. Bu sistem, verici tarafında elektrik akımının gönderilmesini sağlayan bir anahtardan, alıcı tarafında ise, gelen akımın şiddetine göre loş veya parlak ışık veren bir ampulden oluşuyor. Eğer vericiden gönderilen elektrik akımı düşükse alıcı loş bir ışık, eğer gelen akım yüksekse alıcı parlak bir ışık görecek. Elbette eğer isterseniz düşük akım-yüksek akım karşıtlığı yerine akım varlığı-akım yokluğu karşıtlığını da kullanabilirsiniz. Böylece vericiden akım gönderildiğinde ampul yanar, gönderilmediğinde ise söner. Bana düşük akım-yüksek akım karşıtlığı daha kullanışlı geldiği için böyle tercih ettim. Siz tabii ki öbür türlüsünü de tercih edebilirsiniz.

Yukarıda bahsedildiği gibi sistemimizi kurduk diyelim. Peki ama bu sistem verici ile alıcı arasında basit de olsa bir iletişim kurmamızı nasıl olacak da sağlayacak?

Aslında bunun cevabı ve mantığı çok basit. Gördüğünüz gibi, bu sistemde iki farklı durum söz konusu: Loş ışık ve parlak ışık (veya yanan ampul ve sönmüş ampul).

Bu ikili yapıyı, tahmin edebileceğiniz gibi, ikili (*binary*) sayma sistemi aracılığıyla rahatlıkla temsil edebiliriz. Mesela loş ışık durumuna *0*, parlak ışık durumuna ise *1* diyebiliriz. Dolayısıyla verici, ampulün loş ışık vermesini sağlayacak düşük bir akım gönderdiğinde bunun değerini *0*, ampulün yüksek ışık vermesini sağlayacak yüksek bir akım gönderdiğinde ise bunun değerini *1* olarak değerlendirebiliriz.

Burada yaptığımız dönüştürme işlemine teknik olarak ‘kodlama’ (*encoding*) adı verilir. Bu kodlama sistemine göre biz, iki farklı elektrik akımı değerini, yani loş ışık ve parlak ışık değerlerini sırasıyla ikili sistemdeki *0* ve *1* sayıları ile eşleştirip, loş ışığa *0*, parlak ışığa ise *1* dedik.

Hemen anlayacağınız gibi, bahsettiğimiz bu hayali sistem, telgraf iletişimine çok benziyor. İşte gerçekte de kullanılan telgraf sistemine çok benzeyen bu basitleştirilmiş model bizim bilgisayarların çalışma mantığını da daha net bir şekilde anlamamızı sağlayacak.

8 Bitlik bir Sistem

Hatırlarsanız ikili sayma sisteminde *0*‘lar ve *1*‘lerin oluşturduğu her bir basamağa ‘bit’ adını veriyorduk.

Not

Bit kelimesi İngilizcede ‘**bi**nary’ (ikili) ve ‘digi**t**‘ (rakam) kelimelerinin birleştirilmesi ile

üretilmiştir.

Bu bilgiye göre mesela *0* sayısı bir bitlik bir sayı iken, *1001* sayısı dört bitlik bir sayıdır. İletişimimizi eksiksiz bir biçimde sağlayabilmemiz, yani gereken bütün karakterleri temsil edebilmemiz için, sistemimizin 8 hanelik bir sayı kapasitesine sahip olması, yani teknik bir dille ifade etmek gerekirse sistemimizin 8 bitlik olması herhalde yeterli olacaktır.

8 bitlik bir iletişim sisteminde *10*'a kadar şu şekilde sayabiliriz:

```
>>> for i in range(10):
...     print(bin(i)[2:].zfill(8))
...
00000000
00000001
00000010
00000011
00000100
00000101
00000110
00000111
00001000
00001001
```

Verici tarafındaki kişi elindeki anahtar yardımıyla farklı kuvvetlere sahip sinyalleri art arda göndererek yukarıda gösterildiği gibi on farklı sayıyı alıcıya iletebilir. Sistemimizin 8 bitlik olduğunu düşünürsek karşı tarafa *0* sayısı ile birlikte toplam 2 ** 8 = 256 farklı sinyal gönderebiliriz:

```
>>> for i in range(256):
...     print(bin(i)[2:].zfill(8))

00000000
00000001
00000010
00000011
00000100
...
...
...
11111001
11111010
11111011
11111100
11111101
```

```
11111110
11111111
```

Gördüğünüz gibi, bizim 8 bitlik bu sistemle gönderebileceğimiz son sinyal, yani sayı *255*'tir. Bu sistemle bundan büyük bir sayıyı gönderemeyiz. Bu durumu kendi gözlerinizle görmek için şu kodları çalıştırın:

```
>>> for i in range(256):
...     print(bin(i)[2:], i.bit_length(), sep="\t")
```

Burada ilk sütun *256*'ya kadar olan sayıların ikili sistemdeki karşılıklarını, ikinci sütun ise bu sayıların bit uzunluğunu gösteriyor. Bu çıktıyı incelediğinizde de göreceğiniz gibi, 8 bit uzunluğa sahip son sayı *255*'tir. *256* sayısı ise 9 bit uzunluğa sahiptir. Yani *256* sayısı mecburen bizim sistemimizin dışındadır:

```
>>> bin(255)[2:]

'11111111'


>>> (255).bit_length()

8

>>> bin(256)[2:]

'100000000'

>>> (256).bit_length()

9
```

Dediğimiz gibi, bu sistemde elimizde toplam 8 bit var. Yani bu sistemi kullanarak *0*'dan *256*'ya kadar sayıp, bu sayıları alıcıya iletebiliriz.

Peki verici ile alıcı arasında birtakım sayıları gönderip alabilmek ne işimize yarar? Yani bu iş neden bu kadar önemli?

Bu soruların cevabını birazdan vereceğiz, ama ondan önce daha önemli bir konuya değinelim.

Hata Kontrolü

Buraya kadar her şey yolunda. Alıcı ve verici arasındaki iletişimi elektrik akımı vasıtasıyla, 8 bitlik bir sistem üzerinden sağlayabiliyoruz. Ancak sistemimizin çok önemli bir eksiği var. Biz bu sistemde hiçbir hata kontrolü yapmıyoruz. Yani vericiden gelen mesajın doğruluğunu test eden hiçbir

ölçütümüz yok. Zira alıcı ile verici arasında gidip gelen veriler pek çok farklı şekilde ve sebeple bozulmaya uğrayabilir. Örneğin, gönderilen veri alıcı tarafından doğru anlaşılamayabilir veya elektrik sinyallerini ileten kablolardaki arızalar sinyallerin doğru iletilmesini engelleyebilir.

İşte bütün bunları hesaba katarak, iletişimin doğru bir şekilde gerçekleşebilmesini sağlamak amacıyla sistemimiz için basit bir hata kontrol süreci tasarlayalım.

Dediğimiz gibi, elimizdeki sistem toplam *256*'ya kadar saymamıza olanak tanıyor. Çünkü bizim sistemimiz 8 bitlik bir sistem. Bu sisteme bir hata kontrol mekanizması ekleyebilmek için veri iletimini 8 bitten 7 bite çekeceğiz. Yani iletişimimizi toplam 2 ** 7 = 127 sayı ile sınırlayacağız. Boşta kalan 8. biti ise bahsettiğimiz bu hata kontrol mekanizmasına ayıracağız.

Peki hata kontrol mekanizmamız nasıl işleyecek?

Çok basit: Vericiden alıcıya ulaşan verilerin tek mi yoksa çift mi olduğuna bakacağız.

Buna göre sistemimiz şöyle çalışacak:

Diyelim ki verici alıcıya sinyaller aracılığıyla şu sayıyı göndermek istiyor:

```
0110111
```

Bu arada, bunun 7 bitlik bir sayı olduğuna dikkat edin. Dediğimiz gibi, biz kontrol mekanizmamızı kurabilmek için elimizdeki 8 bitlik kapasitenin 7 bitini kullanacağız. Boşta kalan 8. biti ise kontrol mekanizmasına tahsis edeceğiz.

Ne diyorduk? Evet, biz karşı tarafa 7 bitlik bir sayı olan *0110111* sayısını göndermek istiyoruz. Bu sayıyı göndermeden önce, içindeki *1*'lerin miktarına bakarak bu sayının tek mi yoksa çift mi olduğuna karar verelim. Burada toplam beş adet *1* sayısı var. Yani bu sayı bir tek sayıdır. Eğer göndermek istediğimiz sayı bir tek sayı ise, karşı tarafa ulaştığında da bir tek sayı olmalıdır.

Biz bu sistem için şöyle bir protokol tasarlayabiliriz:

> Bu sistemde bütün sayılar karşı tarafa bir 'tek sayı' olarak iletilmelidir. Eğer iletilen sayılar arasında bir çift sayı varsa, o sayı hatalı iletilmiş veya iletim esnasında bozulmuş demektir.

Peki biz iletilen bütün sayıların bir tek sayı olmasını nasıl sağlayacağız? İşte bu işlemi, boşa ayırdığımız o 8. bit ile gerçekleştireceğiz:

> Eğer karşı tarafa iletilen bir sayı zaten tekse, o sayının başına *0* ekleyeceğiz. Böylece sayının teklik-çiftlik durumu değişmemiş olacak. Ama eğer iletilecek sayı çiftse, o sayının başına *1* ekleyeceğiz. Böylece çift sayıyı, sistemimizin gerektirdiği şekilde, tek sayıya çevirmiş olacağız.

Örnek olarak *0110111* sayısını verelim. Bu sayıda toplam beş adet *1* var. Yani bu sayı bir tek sayı. Dolayısıyla bu sayının başına bir adet *0* ekliyoruz:

```
0 0110111
```

Böylece sayımızın teklik-çiftlik durumu değişmemiş oluyor. Karşı taraf bu sayıyı aldığında *1*'lerin miktarına bakarak bu verinin doğru iletildiğinden emin oluyor.

Bir de şu sayıya bakalım:

```
1111011
```

Bu sayıda toplam altı adet *1* sayısı var. Yani bu sayı bir çift sayı. Bir sayının sistemimiz tarafından 'hatasız' olarak kabul edilebilmesi için bu sayının bir tek sayı olması gerekiyor. Bu yüzden biz bu sayıyı tek sayıya çevirmek için başına bir adet *1* sayı ekliyoruz:

```
1 1111011
```

Böylece sayımızın içinde toplam yedi adet *1* sayısı olmuş ve böylece sayımız tek sayıya dönüşmüş oluyor.

Teknik olarak ifade etmemiz gerekirse, yukarıda yaptığımız kontrol türüne 'eşlik denetimi' (*parity check*) adı verilir. Bu işlemi yapmamızı sağlayan bit'e ise 'eşlik biti' (*parity bit*) denir. İki tür eşlik denetimi bulunur:

1. Tek eşlik denetimi (*odd parity check*)
2. Çift eşlik denetimi (*even parity check*)

Biz kendi sistemimizde hata kontrol mekanizmasını bütün verilerin bir 'tek sayı' olması gerekliliği üzerine kurduk. Yani burada bir 'tek eşlik denetimi' gerçekleştirmiş olduk. Elbette bütün verilerin bir çift sayı olması gerekliliği üzerine de kurabilirdik bu sistemi. Yani isteseydik 'çift eşlik denetimi' de yapabilirdik. Bu tamamen bir tercih meselesidir. Bu tür sistemlerde yaygın olarak 'tek eşlik denetimi' kullanıldığı için biz de bunu tercih ettik.

Bu örneklerden de gördüğünüz gibi, toplam 8 bitlik kapasitemizin 7 bitini veri aktarımı için, kalan 1 bitini ise alınıp verilen bu verilerin doğruluğunu denetlemek için kullanıyoruz. Elbette kullandığımız hata kontrol mekanizması epey zayıf bir sistemdir. Ama, iletişim sistemleri arasında verilerin hatasız bir şekilde aktarılıp aktarılamadığını kontrol etmeye yarayan bir sistem olan eşlik denetiminin, bugün bilgisayarın belleklerinde (RAM) dahi kullanılmaya devam ettiğini söylemeden geçmeyelim...

Karakterlerin Temsili

Yukarıda anlattıklarımızdan da gördüğünüz gibi, sistemimizi kullanarak 7 bit üzerinden toplam 127 sayı gönderebiliyoruz. Tabii ki sistemimiz 8 bit olduğu için 1 bit de boşta kalıyor. İşte boşta duran bu 1 biti ise eşlik denetimi için kullanıyoruz. Ama elbette alıcı ile verici arasında sayı alışverişi yapmak pek de heyecan uyandırıcı bir faaliyet değil. Karşı tarafa sayısal mesajlar yerine birtakım sözel mesajlar iletebilsek herhalde çok daha keyifli olurdu...

Şunu asla unutmayın. Eğer bir noktadan başka bir noktaya en az iki farklı sinyal yolu ile birtakım sayısal verileri gönderebiliyorsanız aynı şekilde sözel verileri de rahatlıkla gönderebilirsiniz. Tıpkı düşük voltaj ve yüksek voltaj değerlerini sırasıyla *0* ve *1* sayıları ile temsil ettiğiniz gibi, karakterleri

de bu iki sayı ile temsil edebilirsiniz. Yapmanız gereken tek şey hangi sayıların hangi karakterlere karşılık geleceğini belirlemekten ibarettir. Mesela elimizde sayılarla karakterleri eşleştiren şöyle bir tablo olduğunu varsayalım:

sayı	karakter	sayı	karakter	sayı	karakter	sayı	karakter
0	‘a’	1	‘b’	10	‘c’	11	‘d’
100	‘e’	101	‘f’	110	‘g’	111	‘h’
1000	‘i’	1001	‘j’	1010	‘k’	1011	‘l’
1100	‘m’	1101	‘n’	1110	‘o’	1111	‘p’
10000	‘q’	10001	‘r’	10010	‘s’	10011	‘t’
10100	‘u’	10101	‘v’	10110	‘w’	10111	‘x’
11000	‘y’	11001	‘z’	11010	‘A’	11011	‘B’
11100	‘C’	11101	‘D’	11110	‘E’	11111	‘F’
100000	‘G’	100001	‘H’	100010	‘I’	100011	‘J’
100100	‘K’	100101	‘L’	100110	‘M’	100111	‘N’
101000	‘O’	101001	‘P’	101010	‘Q’	101011	‘R’
101100	‘S’	101101	‘T’	101110	‘U’	101111	‘V’
110000	‘W’	110001	‘X’	110010	‘Y’	110011	‘Z’

Bu tabloda toplam *52* karakter ile *52* sayı birbiriyle eşleştirilmiş durumda. Mesela vericiden *0* sinyali geldiğinde bu tabloya göre biz bunu ‘a’ harfi olarak yorumlayacağız. Örneğin karşı tarafa ‘python’ mesajını iletmek için sırasıyla şu sinyalleri göndereceğiz:

```
1111, 11000, 10011, 111, 1110, 1101
```

Gördüğünüz gibi, elimizdeki 127 sayının 52’sini harflere ayırdık ve elimizde 75 tane daha sayı kaldı. Eğer isterseniz geri kalan bu sayıları da birtakım başka karakterlere veya işaretlere ayırarak, alıcı ve verici arasındaki bütün iletişimin eksiksiz bir şekilde gerçekleşmesini sağlayabilirsiniz. Örneğin şöyle bir tablo oluşturabilirsiniz:

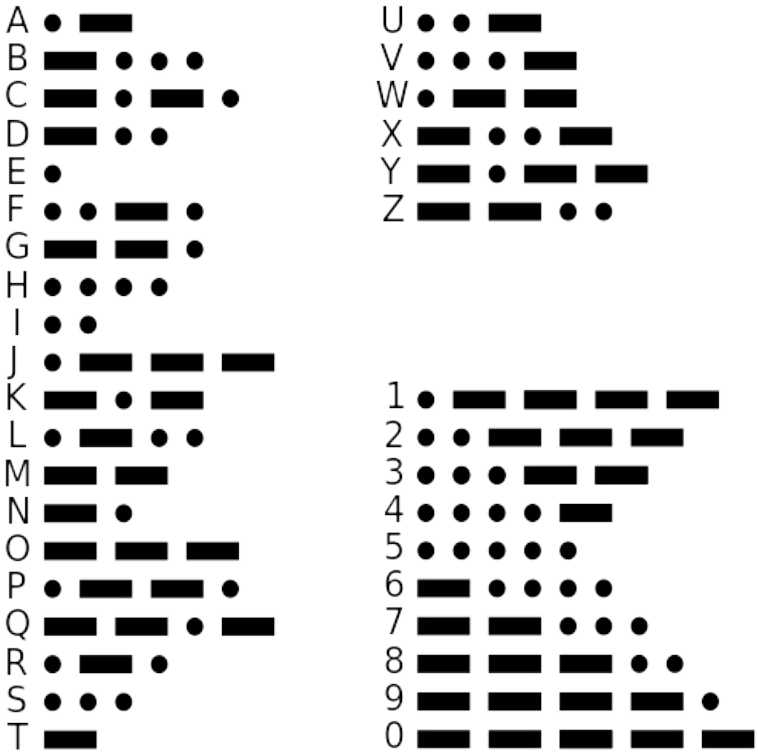
sayı	karakter	sayı	karakter	sayı	karakter	sayı	karakter
0	‘0’	1	‘1’	10	‘2’	11	‘3’
100	‘4’	101	‘5’	110	‘6’	111	‘7’
1000	‘8’	1001	‘9’	1010	‘a’	1011	‘b’
1100	‘c’	1101	‘d’	1110	‘e’	1111	‘f’
10000	‘g’	10001	‘h’	10010	‘i’	10011	‘j’
10100	‘k’	10101	‘l’	10110	‘m’	10111	‘n’
11000	‘o’	11001	‘p’	11010	‘q’	11011	‘r’
11100	‘s’	11101	‘t’	11110	‘u’	11111	‘v’
100000	‘w’	100001	‘x’	100010	‘y’	100011	‘z’
100100	‘A’	100101	‘B’	100110	‘C’	100111	‘D’
101000	‘E’	101001	‘F’	101010	‘G’	101011	‘H’

101100	‘I’	101101	‘J’	101110	‘K’	101111	‘L’
110000	‘M’	110001	‘N’	110010	‘O’	110011	‘P’
110100	‘Q’	110101	‘R’	110110	‘S’	110111	‘T’
111000	‘U’	111001	‘V’	111010	‘W’	111011	‘X’
111100	‘Y’	111101	‘Z’	111110	‘!’	111111	‘”’
1000000	‘#’	1000001	‘$’	1000010	‘%’	1000011	‘&’
1000100	“’””	1000101	“’(‘	1000110	‘)’	1000111	‘*’
1001000	‘+’	1001001	‘,’	1001010	‘-‘	1001011	‘.’
1001100	‘/’	1001101	‘:’	1001110	‘;’	1001111	‘<’
1010000	‘=’	1010001	‘>’	1010010	‘?’	1010011	‘@’
1010100	‘[‘	1010101	‘\’	1010110	‘]’	1010111	‘^’
1011000	‘_’	1011001	‘’	1011010	‘{‘	1011011	‘’
1011100	‘}’	1011101	‘~’	1011110	‘ ‘	1011111	‘t’
1100000	‘n’	1100001	‘r’	1100010	‘x0b’	1100011	‘x0c’

Aslında yukarıda anlattığımız sayı-karakter eşleştirme işleminin, ta en başta yaptığımız sinyal-sayı eşleştirme işlemiyle mantık olarak aynı olduğuna dikkatinizi çekmek isterim.

Sistemimizi tasarlarken, iletilen iki farklı sinyali *0* ve *1* sayıları ile temsil etmiştik. Yani bu sinyalleri *0* ve *1*‘ler halinde kodlamıştık. Şimdi ise bu sayıları karakterlere dönüştürüyoruz. Yani yine bir kodlama (*encoding*) işlemi gerçekleştiriyoruz.

Baştan beri anlattığımız bu küçük iletişim modeli, sayıların ve karakterlerin nasıl temsil edilebileceği konusunda bize epey bilgi verdi. Bu arada, yukarıda anlattığımız sistem her ne kadar hayali de olsa, bu sisteme benzeyen sistemlerin tarih boyunca kullanıldığını ve hatta bugün kullandığımız bütün iletişim sistemlerinin de yukarıda anlattığımız temel üzerinde şekillendiğini belirtmeden geçmeyelim. Örneğin telgraf iletişiminde kullanılan Mors alfabesi yukarıda tarif ettiğimiz sisteme çok benzer. Mors alfabesi, kısa ve uzun sinyallerle karakterlerin eşleştirilmesi yoluyla oluşturulmuştur. Mors sisteminde farklı sinyaller (tıpkı bizim sistemimizde olduğu gibi) farklı harflere karşılık gelir:



Mors alfabesinin bizim oluşturduğumuz sisteme mantık olarak ne kadar benzediğine dikkat edin. Bu sistemin benzeri biraz sonra göstereceğimiz gibi, modern bilgisayarlarda da kullanılmaktadır.

başa dön

Karakter Kodlama (*Character Encoding*)

Bu bölüme gelinceye kadar Python programlama dilindeki karakter dizisi, liste ve dosya adlı veri tiplerine ilişkin epey söz söyledik. Artık bu veri tiplerine dair hemen hemen bütün ayrıntıları biliyoruz. Ancak henüz öğrenmediğimiz, ama programcılık maceramız açısından mutlaka öğrenmemiz gereken çok önemli bir konu daha var. Bu önemli konunun adı, karakter kodlama.

Bu bölümde 'karakter kodlama' adlı hayati konuyu işlemenin yanısıra, son birkaç bölümde üstünkörü bir şekilde üzerinden geçtiğimiz, ama derinlemesine incelemeye pek fırsat bulamadığımız bütün konuları da ele almaya çalışacağız. Bu konuyu bitirdikten sonra, önceki konuları çalışırken zihninizde oluşmuş olabilecek boşlukların pek çoğunun dolduğunu farkedeceksiniz. Sözün özü, bu bölümde hem yeni şeyler söyleyeceğiz, hem de halihazırda öğrendiğimiz şeylerin bir kez daha üzerinden geçerek bunların zihnimizde iyiden iyine pekişmesini sağlayacağız.

Hatırlarsanız önceki derslerimizde karakter dizilerinin encode() adlı bir metodu olduğundan söz etmiştik. Aynı şekilde, dosyaların da *encoding* adlı bir parametresi olduğunu söylemiştik. Ayrıca bu *encoding* konusu, ilk derslerimizde metin düzenleyici ayarlarını anlatırken de karşımıza çıkmıştı. Orada, yazdığımız programlarda özellikle Türkçe karakterlerin düzgün görünebilmesi için, kullandığımız metin düzenleyicinin dil kodlaması (*encoding*) ayarlarını düzgün yapmamız gerektiğini üstüne basa basa söylemiştik. Biz şu ana kadar bu konuyu ayrıntılı olarak ele almamış da olsak, siz şimdiye kadar yazdığınız programlarda Türkçe karakterleri kullanırken halihazırda pek çok problemle karşılaşmış ve bu sorunların neden kaynaklandığını anlamakta zorlanmış olabilirsiniz.

İşte bu bölümde, o zaman henüz bilgimiz yetersiz olduğu için ertelediğimiz bu *encoding* konusunu bütün ayrıntılarıyla ele alacağız ve yazdığımız programlarda Türkçe karakterleri kullanırken neden sorunlarla karşılaştığımızı, bu sorunun temelinde neyin yattığını anlamaya çalışacağız.

O halde hiç vakit kaybetmeden bu önemli konuyu incelemeye başlayalım.

Giriş

Önceki bölümlerde sık sık tekrar ettiğimiz gibi, bilgisayar dediğimiz şey, üzerinden elektrik geçen devrelerden oluşmuş bir sistemdir. Eğer bir devrede elektrik yoksa o devrenin değeri *0* volt iken, o devreden elektrik geçtiğinde devrenin değeri yaklaşık *+5* volttur.

Gördüğünüz gibi, ortada iki farklı değer var: *0* volt ve *+5* volt. İkili (*binary*) sayma sisteminde de iki değer bulunur: *0* ve *1*. İşte biz bu *0* volt'u ikili sistemde *0* ile, *+5* volt'u ise *1* ile temsil ediyoruz. Yani devreden elektrik geçtiğinde o devrenin değeri *1*, elektrik geçmediğinde ise *0* olmuş oluyor. Tabii bilgisayar açısından bakıldığında devrede elektrik vardır veya yoktur. Biz insanlar bu ikili durumu daha kolay bir şekilde manipüle edebilmek için farklı voltaj durumlarından her birine sırasıyla *0* ve *1* gibi bir ad veriyoruz. Yani iki farklı voltaj değerini iki farklı sayı halinde 'kodlamış' oluyoruz...

Hatırlarsanız bir önceki bölümde tasarladığımız basit iletişim modelinde de ampulün loş ışık vermesini sağlayan düşük elektrik sinyallerini *0* ile, parlak ışık vermesini sağlayan yüksek elektrik sinyallerini ise *1* ile temsil etmiştik. Bu temsil işine de teknik olarak 'kodlama' (*encoding*) adı verildiğini söylemiştik. İşte bilgisayarlar açısından da benzer bir durum söz konusudur. Bilgisayarlarda da *0* volt ve *+5* volt değerleri sırasıyla ikili sayma sistemindeki *0* ve *1* sayıları halinde kodlanabilir.

Sözün özü ilk başta yalnızca iki farklı elektrik sinyali vardır. Elbette bu elektrik sinyalleri ile doğrudan herhangi bir işlem yapamayız. Mesela elektrik sinyallerini birbiriyle toplayıp, birbirinden çıkaramayız. Ama bu sinyalleri bir sayma sistemi ile temsil edersek (yani bu sinyalleri o sayma sisteminde kodlarsak), bunları kullanarak, örneğin, aritmetik işlemleri rahatlıkla gerçekleştirebiliriz. Mesela *0* volt ile *+5* voltu birbiriyle toplayamayız, ama *0* voltu ikili sistemdeki *0* sayısıyla, *+5* voltu ise ikili sistemdeki *1* sayısıyla kodladıktan sonra bu ikili sayılar arasında her türlü aritmetik işlemi gerçekleştirebiliriz.

Bilgisayarlar yalnızca iki farklı voltaj durumundan anladığı ve bu iki farklı voltaj durumu da ikili sayma sistemindeki iki farklı sayı ile kolayca temsil edilebildiği için, ilk bilgisayarlar çoğunlukla sadece hesap işlemlerinde kullanılıyordu. Karakterlerin/harflerin bilgisayar dünyasındaki işlevi bir hayli kısıtlıydı. Metin oluşturma işi o zamanlarda daktilo ve benzeri araçların görevi olarak görülüyordu. Bu durumu, telefon teknolojisi ile kıyaslayabilirsiniz. İlk telefonlar da yalnızca iki kişi arasındaki sesli iletişimi sağlamak gibi kısıtlı bir amaca hizmet ediyordu. Bugün ise, geçmişte pek çok farklı cihaza paylaştırılmış görevleri akıllı telefonlar aracılığıyla tek elden halledebiliyoruz.

Peki bir bilgisayar yalnızca elektrik sinyallerinden anlıyorsa, biz mesela bilgisayarları nasıl oluyor da metin girişi için kullanabiliyoruz?

Bu sorunun cevabı aslında çok açık: Birtakım elektrik sinyallerini, birtakım aritmetik işlemleri gerçekleştirebilmek amacıyla nasıl birtakım sayılar halinde kodlayabiliyorsak; birtakım sayıları da, birtakım metin işlemlerini gerçekleştirebilmek amacıyla birtakım karakterler halinde kodlayabiliriz.

Peki ama nasıl?

Bir önceki bölümde bahsettiğimiz basit iletişim modeli aracılığıyla bunun nasıl yapılacağını anlatmıştık. Tıpkı bizim basit iletişim sistemimizde olduğu gibi, bilgisayarlar da yalnızca elektrik sinyallerini görür. Tıpkı orada yaptığımız gibi, bilgisayarlarda da hangi elektrik sinyalinin hangi sayıya; hangi sayının da hangi karaktere karşılık geleceğini belirleyebiliriz. Daha doğrusu, bilgisayarların gördüğü bu elektrik sinyallerini sayılara ve karakterlere dönüştürebiliriz. Dışarıdan girilen karakterleri de, bilgisayarların anlayabilmesi için tam aksi istikamette sayıya, oradan da elektrik sinyallerine çevirebiliriz. İşte bu dönüştürme işlemine karakter kodlama (*character encoding*) adı verilir.

Bu noktada şöyle bir soru akla geliyor: Tamam, sayıları karakterlere, karakterleri de sayılara dönüştüreceğiz. Ama peki hangi sayıları hangi karakterlere, hangi karakterleri de hangi sayılara dönüştüreceğiz? Yani mesela ikili sistemdeki *0* sayısı hangi karaktere, *1* sayısı hangi karaktere, *10* sayısı hangi karaktere karşılık gelecek?

Siz aslında bu sorunun cevabını da biliyorsunuz. Yine bir önceki bölümde anlattığımız gibi, hangi sayıların hangi karakterlere karşılık geleceğini, sayılarla karakterlerin eşleştirildiği birtakım tablolalar yardımıyla rahatlıkla belirleyebiliriz.

Bu iş ilk başta kulağa çok kolaymış gibi geliyor. Esasında iş kolaydır, ama şöyle bir problem var: Herkes aynı sayıları aynı karakterlerle eşleştirmiyor olabilir. Mesela durumu bir önceki bölümde tasarladığımız basit iletişim modeli üzerinden düşünelim. Diyelim ki, başta yalnızca bir arkadaşınızla ikinizin arasındaki iletişimi sağlamak için tasarladığınız bu sistem başkalarının da dikkatini çekmiş olsun... Tıpkı sizin gibi, başkaları da loş ışık-parlak ışık karşıtlığı üzerinden birbiriyle iletişim kurmaya karar vermiş olsun. Ancak sistemin temeli herkesçe aynı şekilde kullanılıyor olsa da, karakter eşleştirme tablolarını herkes aynı şekilde kullanmıyor olabilir. Örneğin başkaları, kendi ihtiyaçları çerçevesinde, farklı sayıların farklı karakterlerle eşleştirildiği farklı tablolar tasarlamış olabilir. Bu durumun dezavantajı, farklı sistemlerle üretilen mesajların, başka sistemlerde aslı gibi görüntülenemeyecek olmasıdır. Örneğin 'a' harfinin *1010* gibi bir sayıyla temsil edildiği sistemle üretilen bir mesaj, aynı harfin mesela *1101* gibi bir sayıyla temsil edildiği sistemde düzgün görüntülenemeyecektir. İşte aynı şey bilgisayarlar için de geçerlidir.

1960'lı yılların ilk yarısına kadar her bilgisayar üreticisi, sayılarla karakterlerin eşleştirildiği, birbirinden çok farklı tablolar kullanıyordu. Yani her bilgisayar üreticisi farklı karakterleri farklı sayılarla eşleştiriyordu. Örneğin bir bilgisayarda *10* sayısı 'a' harfine karşılık geliyorsa, başka bir bilgisayarda *10* sayısı 'b' harfine karşılık gelebiliyordu. Bu durumun doğal sonucu olarak, iki bilgisayar arasında güvenilir bir veri aktarımı gerçekleştirmek mümkün olmuyordu. Hatta daha da vahimi, aynı firma içinde bile birden fazla karakter eşleştirme tablosunun kullanıldığı olabiliyordu...

Peki bu sorunun çözümü ne olabilir?

Cevap elbette standartlaşma.

Standartlaşma ilerleme ve uygarlık açısından çok önemli bir kavramdır. Standartlaşma olmadan ilerleme ve uygarlık düşünülemez. Eğer standartlaşma diye bir şey olmasaydı, mesela A4 piller boy ve en olarak standart bir ölçüye sahip olmasaydı, evde kullandığınız küçük aletlerin pili bittiğinde uygun pili satın almakta büyük zorluk çekerdiniz. Banyo-mutfak musluklarındaki plastik contanın belli bir standardı olmasaydı, conta eskidiğinde yenisini alabilmek için eskisinin ölçülerini inceden inceye hesaplayıp bu ölçülere göre yeni bir conta arayışına çıkmanız gerekirdi. Herhangi bir yerden bulduğunuz contayı herhangi bir muslukta kullanamazdınız. İşte bu durumun aynısı bilgisayarlar için de geçerlidir. Eğer bugün karakterlerle sayıları eşleştirme işlemi belli bir standart üzerinden yürütülüyor olmasaydı, kendi bilgisayarınızda oluşturduğunuz bir metni başka bir bilgisayarda açtığınızda aynı metni göremezdiniz. İşte 1960'lı yıllara kadar bilgisayar dünyasında da aynen buna benzer bir sorun vardı. Yani o dönemde, hangi sayıların hangi karakterlerle eşleşeceği konusunda uzlaşma olmadığı için, farklı bilgisayarlar arasında metin değiş tokuşu pek mümkün değildi.

1960'lı yılların başında IBM şirketinde çalışan Bob Bemer adlı bir bilim adamı bu kargaşanın sona ermesi gerektiğine karar verip, herkes tarafından benimsenecek ortak bir karakter kodlama sistemi üzerinde ilk çalışmaları başlattı. İşte ASCII ('aski' okunur) böylece hayatımıza girmiş oldu.

Peki bu 'ASCII' denen şey tam olarak ne anlama geliyor? Gelin bu sorunun cevabını, en baştan

başlayarak ve olabildiğince ayrıntılı bir şekilde vermeye çalışalım.

ASCII

Bilgisayarların iki farklı elektrik sinyali ile çalıştığını, bu iki farklı sinyalin de *0* ve *1* sayıları ile temsil edildiğini, bilgisayarla metin işlemleri yapabilmek için ise bu sayıların belli karakterlerle eşleştirilmesi gerektiğini söylemiştik.

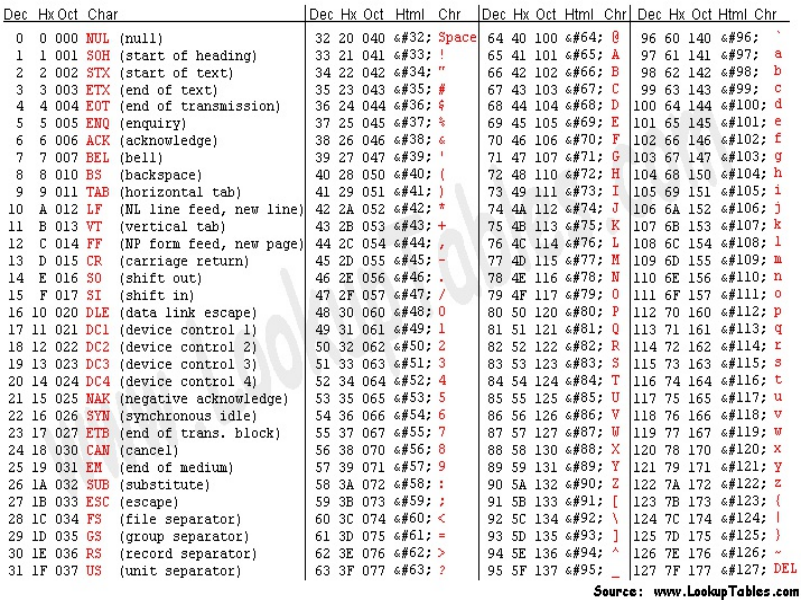
Yukarıda da bahsettiğimiz gibi, uygarlık ve ilerleme açısından standartlaşma önemli bir basamaktır. Şöyle düşünün: Biz bilgisayarların çalışma prensibinde iki farklı elektrik sinyali olduğunu biliyoruz. Biz insanlar olarak, işlerimizi daha kolay yapabilmek için, bu sinyalleri daha somut birer araç olan *0* ve *1* sayılarına atamışız. Eğer devrede elektrik yoksa bu durumu *0* ile, eğer devrede elektrik varsa bu durumu *1* ile temsil ediyoruz. Esasında bu da bir uzlaşma gerektirir. Devrede elektrik yoksa bu durumu pekala *0* yerine *1* ile de temsil edebilirdik... Eğer elektrik sinyallerinin temsili üzerinde böyle bir uzlaşmazlık olsaydı, her şeyden önce hangi sinyalin hangi sayıya karşılık geleceği konusunda da ortak bir karara varmamız gerekirdi.

Elektriğin var olmadığı durumu *0* yerine *1* ile temsil etmek akla pek yatkın olmadığı için uzlaşmada bir problem çıkmıyor. Ama karakterler böyle değildir. Onlarca (hatta yüzlerce ve binlerce) karakterin sayılarla eşleştirilmesi gereken bir durumda, ortak bir eşleştirme düzeni üzerinde uzlaşma sağlamak hiç de kolay bir iş değildir. Zaten 1960'lı yılların başına kadar da böyle bir uzlaşma sağlanabilmiş değildi. Dediğimiz gibi, her bilgisayar üreticisi sayıları farklı karakterlerle eşleştiriyor, yani birbirlerinden tamamen farklı karakter kodlama sistemleri kullanıyordu.

İşte bu kargaşayı ortadan kaldırmak gayesiyle, Bob Bemer ve ekibi hangi sayıların hangi karakterlere karşılık geleceğini belli bir standarda bağlayan bir tablo oluşturdu. Bu standarda ise *American Standard Code for Information Interchange*, yani 'Bilgi Alışverişi için Standart Amerikan Kodu' veya kısaca 'ASCII' adı verildi.

7 Bitlik bir Sistem

ASCII adı verilen sistem, birtakım sayıların birtakım karakterlerle eşleştirildiği basit bir tablodan ibarettir. Bu tabloyu http://www.asciitable.com/ adresinde görebilirsiniz:



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

İsterseniz bu tabloyu Python yardımıyla kendiniz de oluşturabilirsiniz:

```
for i in range(128):
    if i % 4 == 0:
        print("\n")

    print("{:<3}{:>8}\t".format(i, repr(chr(i))), sep="", end="")
```

Not

Bu kodlarda repr() fonksiyonu dışında bilmediğiniz ve anlayamayacağınız hiçbir şey yok. Biraz sonra repr() fonksiyonundan da bahsedeceğiz. Ama dilerseniz, bu fonksiyonun ne işe yaradığı konusunda en azından bir fikir sahibi olmak için, yukarıdaki kodları bir de repr() olmadan yazmayı ve aldığınız çıktıyı incelemeyi deneyebilirsiniz.

ASCII tablosunda toplam 128 karakterin sayılarla eşleştirilmiş durumda olduğunu görüyorsunuz. Bir önceki bölümde bahsettiğimiz basit iletişim modelinde anlattıklarımızdan da aşina olduğunuz gibi, 128 adet sayı 7 bite karşılık gelir (2**7=128). Yani 7 bit ile gösterilebilecek son sayı *127*'dir. Dolayısıyla ASCII 7 bitlik bir sistemdir.

ASCII tablosunu şöyle bir incelediğimizde ilk 32 öğenin göze ilk başta anlamsız görünen birtakım karakterlerden oluştuğunu görüyoruz:

sayı	karakter	sayı	karakter	sayı	karakter	sayı	karakter
0	'\x00'	1	'\x01'	2	'\x02'	3	'\x03'
4	'\x04'	5	'\x05'	6	'\x06'	7	'\x07'
8	'\x08'	9	'\t'	10	'\n'	11	'\x0b'
12	'\x0c'	13	'\r'	14	'\x0e'	15	'\x0f'
16	'\x10'	17	'\x11'	18	'\x12'	19	'\x13'

20	‘\x14’	21	‘\x15’	22	‘\x16’	23	‘\x17’
24	‘\x18’	25	‘\x19’	26	‘\x1a’	27	‘\x1b’
28	‘\x1c’	29	‘\x1d’	30	‘\x1e’	31	‘\x1f’

Not

Bu arada, asciitable.com adresinden baktığınız tablo ile yukarıdaki tablonun birbirinden farklı olduğunu zannedebilirsiniz ilk bakışta. Ama aslında arada herhangi bir fark yok. Yalnızca iki tablonun karakterleri gösterim şekli birbirinden farklı. Örneğin asciitable.com’daki tabloda 9 sayısının ‘TAB (horizontal tab)’ adlı bir karaktere atandığını görüyoruz. Yukarıdaki tabloda ise 9 sayısının yanında *\t* adlı kaçış dizisi var. Gördüğünüz gibi, ‘TAB (horizontal tab)’ ifadesi ile *\t* ifadesi aynı karaktere atıfta bulunuyor. Yalnızca bunların gösterimleri birbirinden farklı, o kadar.

Aslında bu karakter salatası arasında bizim tanıdığımız birkaç karakter de yok değil. Mesela 9. sıradaki *\t* öğesinin sekme oluşturan kaçış dizisi olduğunu söyledik. Aynı şekilde, 10. sıradaki *\n* öğesinin satır başına geçiren kaçış dizisi olduğunu, 13. sıradaki *\r* öğesinin ise satırı başa alan kaçış dizisi olduğunu da biliyoruz. Bu tür karakterler ‘basılamayan’ (*non-printing*) karakterlerdir. Yani mesela ekranda görüntülenebilen ‘a’, ‘b’, ‘c’, ‘!’, ‘?’, ‘=’ gibi karakterlerden farklı olarak bu ilk 32 karakter ekranda görünmez. Bunlara aynı zamanda ‘kontrol karakterleri’ (*control characters*) adı da verilir. Çünkü bu karakterler ekranda görüntülenmek yerine, metnin akışını kontrol eder. Bu karakterlerin ne işe yaradığını şu tabloyla tek tek gösterebiliriz (tablo http://tr.wikipedia.org/wiki/ASCII adresinden alıntıdır):

Sayı	**Karakter**	**Sayı**	**Karakter**
0	boş	16	veri bağlantısından çık
1	başlık başlangıcı	17	aygıt denetimi 1
2	metin başlangıcı	18	aygıt denetimi 2
3	metin sonu	19	aygıt denetimi 3
4	aktarım sonu	20	aygıt denetimi 4
5	sorgu	21	olumsuz bildirim
6	bildirim	22	zaman uyumlu boşta kalma
7	zil	23	aktarım bloğu sonu
8	geri al	24	iptal
9	yatay sekme	25	ortam sonu
10	satır besleme/yeni satır	26	değiştir
11	dikey sekme	27	çık
12	form besleme/yeni sayfa	28	dosya ayırıcısı
13	satır başı	29	grup ayırıcısı
14	dışarı kaydır	30	kayıt ayırıcısı
15	içeri kaydır	31	birim ayırıcısı

Gördüğünüz gibi, bunlar birer harf, sayı veya noktalama işareti değil. O yüzden bu karakterler

ekranda görünmez. Ama bir metindeki veri, satır ve paragraf düzeninin nasıl olacağını, metnin nerede başlayıp nerede biteceğini ve nasıl görüneceğini kontrol ettikleri için önemlidirler.

Geri kalan sayılar ise doğrudan karakterlere, sayılara ve noktalama işaretlerine tahsis edilmiştir:

sayı	**karakter**	**sayı**	**karakter**	**sayı**	**karakter**	**sayı**	**karakter**
32	‘ ‘	33	‘!’	34	‘”’	35	‘#’
36	‘$’	37	‘%’	38	‘&’	39	“’”
40	‘(‘	41	‘)’	42	‘*’	43	‘+’
44	‘,’	45	‘-‘	46	‘.’	47	‘/’
48	‘0’	49	‘1’	50	‘2’	51	‘3’
52	‘4’	53	‘5’	54	‘6’	55	‘7’
56	‘8’	57	‘9’	58	‘:’	59	‘;’
60	‘<’	61	‘=’	62	‘>’	63	‘?’
64	‘@’	65	‘A’	66	‘B’	67	‘C’
68	‘D’	69	‘E’	70	‘F’	71	‘G’
72	‘H’	73	‘I’	74	‘J’	75	‘K’
76	‘L’	77	‘M’	78	‘N’	79	‘O’
80	‘P’	81	‘Q’	82	‘R’	83	‘S’
84	‘T’	85	‘U’	86	‘V’	87	‘W’
88	‘X’	89	‘Y’	90	‘Z’	91	‘[‘
92	‘\’	93	‘]’	94	‘^’	95	‘_’
96	‘’‘	97	‘a’	98	‘b’	99	‘c’
100	‘d’	101	‘e’	102	‘f’	103	‘g’
104	‘h’	105	‘i’	106	‘j’	107	‘k’
108	‘l’	109	‘m’	110	‘n’	111	‘o’
112	‘p’	113	‘q’	114	‘r’	115	‘s’
116	‘t’	117	‘u’	118	‘v’	119	‘w’
120	‘x’	121	‘y’	122	‘z’	123	‘{‘
124	‘|’	125	‘}’	126	‘~’	127	‘x7f’

İşte 32 ile 127 arası sayılarla eşleştirilen yukarıdaki karakterler yardımıyla metin ihtiyaçlarımızın büyük bölümünü karşılayabiliriz. Yani ASCII adı verilen bu eşleştirme tablosu sayesinde bilgisayarların sayılarla birlikte karakterleri de işleyebilmesini sağlayabiliriz.

1960’lı yıllara gelindiğinde, bilgisayarlar 8 bit uzunluğundaki verileri işleyebiliyordu. Yani, ASCII sisteminin gerçeklendiği (yani hayata geçirildiği) bilgisayarlar 8 bitlik bir kapasiteye sahipti. Bu 8 bitin 7 biti karakterle ayrılmıştı. Dolayısıyla mevcut bütün karakterler 7 bitlik bir alana sığdırılmıştı. Boşta kalan *8.* bit ise, veri aktarımının düzgün gerçekleştirilip gerçekleştirilmediğini denetlemek amacıyla ‘doğruluk kontrolü’ için kullanılıyordu. Bu kontrole teknik olarak ‘eşlik denetimi’ (*parity check*), bu eşlik denetimini yapmamızı sağlayan bit’e ise ‘eşlik biti’ (*parity bit*) adı verildiğini biliyorsunuz. Geçen bölümde bu teknik teknik terimlerin ne anlama geldiğini açıklamış, hatta bunlarla

ilgili basit bir örnek de vermiştik.

Adından da anlaşılacağı gibi, ASCII bir Amerikan standardıdır. Dolayısıyla hazırlanışında İngilizce temel alınmıştır. Zaten ASCII tablosunu incelediğinizde, bu tabloda Türkçeye özgü harflerin bulunmadığını göreceksiniz. Bu sebepten, bu standart ile mesela Türkçeye özgü karakterleri gösteremeyiz. Çünkü ASCII standardında 'ş', 'ç', 'ğ' gibi harfler kodlanmamıştır. Özellikle Python'ın 2.x serisini kullanmış olanlar, ASCII'nin bu yetersizliğinin nelere sebep olduğunu gayet iyi bilir. Python'ın 2.x serisinde mesela doğrudan şöyle bir kod yazamayız:

```
print("Merhaba Şirin Baba!")
```

"Merhaba Şirin Baba! adlı karakter dizisinde geçen 'Ş' harfi ASCII dışı bir karakterdir. Yani bu harf ASCII ile temsil edilemez. O yüzden böyle bir kod yazıp bu kodu çalıştırdığımızda Python bize şöyle bir hata mesajı gösterecektir:

```
  File "deneme.py", line 1
SyntaxError: Non-ASCII character '\xde' in file deneme.py on line 1, but no
encoding declared; see http://www.python.org/peps/pep-0263.html for details
```

Aynen anlattığımız gibi, yukarıdaki hata mesajı da kodlar arasında ASCII olmayan bir karakter yer aldığından yakınıyor...

ASCII'nin her ne kadar yukarıda bahsettiğimiz eksiklikleri olsa da bu standart son derece yaygındır ve piyasada bulunan pek çok sistemde kullanılmaya devam etmektedir. Örneğin size kullanıcı adı ve parola soran hemen hemen bütün sistemler bu ASCII tablosunu temel alır veya bu tablodan etkilenmiştir. O yüzden çoğu yerde kullanıcı adı ve/veya parola belirlerken Türkçe karakterleri kullanamazsınız. Hatta pek çok yazı tipinde yalnızca ASCII tablosunda yer alan karakterlerin karşılığı bulunur. Bu yüzden, mesela blogunuzda kullanmak üzere seçip beğendiğiniz çoğu yazı tipi 'ş', 'ç', 'ğ', 'ö' gibi harfleri göstermeyebilir. Yukarıda 'Merhaba Şirin Baba!' örneğinde de gösterdiğimiz gibi, Python'ın 2.x serisinde de öntanımlı olarak ASCII kodlama biçimi kullanılıyordu. O yüzden Python'ın 2.x sürümlerinde Türkçe karakterleri gösterebilmek için daha fazla ilave işlem yapmak zorunda kalıyorduk.

Sözün özü, eğer yazdığınız veya kendiniz yazmamış da olsanız herhangi bir sebeple kullanmakta olduğunuz bir programda Türkçe karakterlere ilişkin bir hata alıyorsanız, bu durumun en muhtemel sebebi, kullandığınız programın veya sistemin, doğrudan ASCII'yi veya ASCII'ye benzer başka bir sistemi temel alarak çalışıyor olmasıdır. ASCII tablosunda görünen 128 karakter dışında kalan hiçbir karakter ASCII ile kodlanamayacağı için, özellikle farklı dillerin kullanıldığı bilgisayarlarda çalışan programlar kaçınılmaz olarak karakterlere ilişkin pek çok hata verecektir. Örneğin, karakter kodlamalarına ilişkin olarak yukarıda bahsettiğimiz ayrıntılardan habersiz bir Amerikalı programcının yazdığı bir programa Türkçe veri girdiğinizde bu program bir anda tuhaf görünen hatalar verip çökecektir...

Genişletilmiş ASCII

Dediğimiz gibi, ASCII 7 bitlik bir karakter kümesidir. Bu standardın ilk çıktığı dönemde 8. bitin hata

kontrolü için kullanıldığını söylemiştik. Sonraki yıllarda 8. bitin hata kontrolü için kullanılmasından vazgeçildi. Böylece 8. bit yine boşa düşmüş oldu. Bu bitin boşa düşmesi ile elimizde yine toplam *128* karakterlik bir boşluk olmuş oldu. Dediğimiz gibi 7 bit ile toplam *128* sayı-karakter eşleştirilebilirken, 8 bit ile toplam *256* sayı-karakter eşleştirilebilir. Ne de olsa:

```
>>> 2**7

128

>>> 2**8

256
```

İşte bu fazla bit, farklı kişi, kurum ve organizasyonlar tarafından, İngilizcede bulunmayan ama başka dillerde bulunan karakterleri temsil etmek için kullanıldı. Ancak elbette bu fazladan bitin sağladığı *128* karakter de dünyadaki bütün karakterlerin temsil edilmesine yetmez. Bu yüzden 8. bitin sunduğu boşluk, birbirinden farklı karakterleri gösteren çeşitli tabloların ortaya çıkmasına sebep oldu. Bu birbirinden farklı tablolara genel olarak 'kod sayfası' adı verilir. Örneğin Microsoft şirketinin Türkiye'ye gönderdiği bilgisayarlarda tanımlı 'cp857' adlı kod sayfasında 128 ile 256 aralığında Türkçe karakterlere de yer verilmişti (bkz. http://msdn.microsoft.com/en-us/library/cc195068.aspx)

Bu tabloya baktığınızda baştan 128'e kadar olan karakterlerin standart ASCII tablosu ile aynı olduğunu göreceksiniz. 128. karakterden itibaren ise Türkçeye özgü harfler tanımlanır. Mesela bu tabloda 128. karakter Türkçedeki büyük 'ç' harfi iken, 159. karakter küçük 'ş' harfidir. Bu durumu şu Python kodları ile de teyit edebilirsiniz:

```
>>> "Ç".encode("cp857")

b'\x80'

>>> "ş".encode("cp857")

b'\x9f'
```

Bu arada bu sayıların onaltılı sayma düzenine göre olduğunu biliyorsunuz. Onlu düzende bunların karşılığı sırasıyla şudur:

```
>>> int("80", 16)

128

>>> int("9f", 16)

159
```

Burada karakter dizilerinin encode() adlı metodunu kullandığımıza dikkat edin. Bu metot yardımıyla herhangi bir karakteri herhangi bir karakter kodlama sistemine göre kodlayabiliriz. Mesela yukarıdaki iki örnekte 'Ç' ve 'ş' harflerini 'cp857' adlı kod sayfasına göre kodladık ve bunların bu kod sayfasında hangi sayılara karşılık geldiğini bulduk.

cp857 numaralı kod sayfasında 'Ç' ve 'ş' harfleri yer aldığı için, biz bu harfleri o kod sayfasına göre kodlayabiliyoruz. Ama mesela ASCII kodlama sisteminde bu harfler bulunmaz. O yüzden bu harfleri ASCII sistemine göre kodlayamayız:

```
>>> "Ç".encode("ascii")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character '\xc7' in position
0: ordinal not in range(128)
```

Tıpkı hata mesajında da söylendiği gibi:

```
Unicode Kodlama Hatası: 'ascii' kod çözücüsü, 0 konumundaki '\xc7' adlı
karakteri kodlayamıyor. Sayı 0-128 aralığında değil.
```

Gerçekten de onlu sistemde *199* sayısına karşılık gelen bu onaltılı '\xc7' sayısı ASCII'nin kapsadığı sayı aralığının dışında kalmakta, bu yüzden de ASCII kod çözücüsü ile kodlanamamaktadır.

Dediğimiz gibi, Microsoft Türkiye'ye gönderdiği bilgisayarlarda 857 numaralı kod sayfasını tanımlıyordu. Ama mesela Arapça konuşulan ülkelere gönderdiği bilgisayarlarda ise, http://msdn.microsoft.com/en-us/library/cc195061.aspx adresinden görebileceğiniz 708 numaralı kod sayfasını tanımlıyordu. Bu kod sayfasını incelediğinizde, 128 altı karakterlerin standart ASCII ile aynı olduğunu ancak 128 üstü karakterlerin Türkçe kod sayfasındaki karakterlerden farklı olduğunu göreceksiniz. İşte 128 üstü karakterler bütün dillerde birbirinden farklıdır. Bu farklılığın ne sonuç doğurabileceğini tahmin edebildiğinizi zannediyorum. Elbette, mesela kendi bilgisayarınızda yazdığınız bir metni Arapça konuşulan bir ülkedeki bilgisayara gönderdiğinizde, doğal olarak metin içindeki Türkçeye özgü karakterlerin yerinde başka karakterler belirecektir.

Bu bölümün başında da söylediğimiz gibi, Genişletilmiş ASCII sisteminde 128 ile 256 aralığı için pek çok farklı karakter eşleştirme tabloları kullanılıyordu. Mesela Microsoft şirketi bu aralık için kendine özgü birtakım kod sayfaları tasarlamıştı. Bu kod sayfalarına örnek olarak yukarıda cp857 ve cp708 numaralı kod sayfalarını örnek vermiştik.

Elbette 128 ile 256 aralığını dolduran, yalnızca Microsoft'a ait kod sayfaları yoktu piyasada. Aynı aralığı farklı karakterlerle dolduran pek çok başka eşleştirme tablosu da dolaşıyordu etrafta. Örneğin özellikle Batı Avrupa dillerindeki karakterleri temsil edebilmek için oluşturulmuş 'latin1' (öbür adıyla ISO-8859-1) adlı karakter kümesi bugün de yaygın olarak kullanılan sistemlerinden biridir. Almancada olup da ASCII sistemi ile temsil edilemeyen 'ö', 'ß', 'ü' gibi harfler ve Fransızcada olup da yine ASCII sistemi ile temsil edilemeyen 'ç' ve 'é' gibi harfler bu karakter kümesinde temsil edilebiliyordu. Eğer dilerseniz bu karakter kümesini de http://www.fileformat.info/info/charset/ISO-

8859-1/list.htm adresinden inceleyebilirsiniz.

Yalnız burada önemli bir ayrıntıyı not düşelim. 'Genişletilmiş ASCII', standart ASCII gibi genel kabul görmüş tek bir sistem değildir.Genişletilmiş ASCII dediğimizde zaten tek bir karakter kümesi akla gelmiyor. Dolayısıyla ASCII dendiğinde anlamamız gereken şey 128 karakterlik bir sayı-karakter eşleştirme tablosudur. ASCII hiçbir zaman bu 128 karakterin ötesine geçip de 256 karakterlik bir aralığı temsil etmiş değildir. Dolayısıyla 127. sayının ötesindeki karakterleri kapsayan sistem ASCII değildir. 'Genişletilmiş ASCII' kavramı, temel ASCII sisteminde temsil edilen sayı-karakter çiftlerinin pek çok farklı kurum ve kuruluş tarafından birbirinden farklı biçimlerde 'genişletilmesiyle' oluşturulmuş, ancak ASCII'nin kendisi kadar standartlaşamamış bir sistemler bütünüdür. Bu sistem içinde pek çok farklı kod sayfası (veya karakter kümesi) yer alır. Tek başına 'Genişletilmiş ASCII' ifadesi açıklayıcı olmayıp; ASCII'nin hangi karakter kümesine göre genişletildiğinin de belirtilmesi gerekir.

Bütün bu anlattıklarımızdan şu sonucu çıkarıyoruz: ASCII bilgisayarlar arasında güvenli bir şekilde veri aktarımını sağlamak için atılmış en önemli ve en başarılı adımlardan bir tanesidir. Bu güçlü standart sayesinde uzun yıllar bilgisayarlar arası temel iletişim başarıyla sağlandı. Ancak bu standardın zayıf kaldığı nokta 7 bitlik olması ve boşta kalan 8. bitin tek başına dünyadaki bütün dilleri temsil etmeye yeterli olmamasıdır.

1 Karakter == 1 Bayt

ASCII standardı, her karakterin 1 bayt ile temsil edilebileceği varsayımı üzerine kurulmuştur. Bildiğiniz gibi, 1 bayt (geleneksel olarak) 8 bit'e karşılık gelir. Peki 1 bayt'ın 8 bit'e karşılık gelmesinin nedeni nedir? Aslında bunun özel bir nedeni yok. 1 destede neden 10 öğe, 1 düzinede de 12 öğe varsa, 1 bayt'ta da 8 bit vardır... Yani biz insanlar öyle olmasına karar verdiğimiz için 1 destede 10 öğe, 1 düzinede 12 öğe, 1 bayt'ta ise 8 bit vardır.

Dediğimiz gibi ASCII standardı 7 bitlik bir sistemdir. Yani bu standartta en büyük sayı olan 127 yalnızca 7 bit ile gösterilebilir:

```
>>> bin(127)[2:]

'1111111'
```

127 sayısı 7 bit ile gösterilebilecek son sayıdır:

```
>>> (127).bit_length()

7

>>> (128).bit_length()

8
```

8 bitlik bir sistem olan Genişletilmiş ASCII ise 0 ile 255 arası sayıları temsil edebilir:

```
>>> bin(255)[2:]

'11111111'
```

255 sayısı 8 bit ile gösterilebilecek son sayıdır:

```
>>> (255).bit_length()

8

>>> (256).bit_length()

9
```

Dolayısıyla ASCII'de ve Genişletilmiş ASCII'de 1 baytlık alana toplam 256 karakter sığdırılabilir. Eğer daha fazla karakteri temsil etmek isterseniz 1 bayttan fazla bir alana ihtiyaç duyarsınız.

Bu arada, olası bir yanlış anlamayı önleyelim:

1 bayt olma durumu mesela doğrudan 'a' harfinin kendisi ile ilgili bir şey değildir. Yani 'a' harfi 1 bayt ile gösterilebiliyorken, mesela 'ş' harfi 1 bayt ile gösterilemiyorsa, bunun nedeni 'ş' harfininin 'tuhaf bir harf' olması değildir! Eğer ASCII gibi bir sistem Türkiye'de tasarlanmış olsaydı, herhalde 'ş' harfi ilk 128 sayı arasında kendine bir yer bulurdu. Mesela böyle bir sistemde muhtemelen 'x', 'w' ve 'q' harfleri, Türk alfabesinde yer almadıkları için, dışarıda kalırdı. O zaman da 'ş', 'ç', 'ğ' gibi harflerin 1 bayt olduğunu, 'x', 'w' ve 'q' gibi harflerin ise 1 bayt olmadığını söylerdik.

UNICODE

İlk bilgisayarların ABD çıkışlı olması nedeniyle, bilgisayarlar çoğunlukla ABD'de üretilip ABD pazarına satılıyordu. Bu nedenle İngilizce alfabeyi temel alan ASCII gibi bir sistem bu pazarın karakter temsil ihtiyaçlarını %99 oranında karşılıyordu. Ancak bilgisayarların ABD dışına çıkması ve ABD dışında da da yayılmaya başlamasının ardından, ASCII'nin yetersizlikleri de iyice görünür olmaya başladı. Çünkü ASCII tablosunda, İngilizce dışındaki dillerde bulunan aksanlı ve noktalı harflerin (é, ä, ö, ç gibi) hiçbiri bulunmuyordu.

İlk zamanlarda insanlar aksanlı ve noktalı harfleri ASCII tablosundaki benzerleriyle değiştirerek kullanmaya razı olmuşlardı (é yerine e; ä yerine a; ö yerine o; ç yerine c gibi). Ancak bu çözüm Avrupa dillerini kullananların sorununu kısmen çözüyor da olsa, Asya dillerindeki problemi çözemez. Çünkü ASCII tablosunu kullanarak Çince ve Japonca gibi dillerdeki karakterleri herhangi bir şekilde temsil etmeniz mümkün değildir.

Bu sıkıntıyı kısmen de olsa giderebilmek için, yukarıda da bahsetmiş olduğumuz, 128-256 arasındaki boşluktan yararlanılmaya başlandı. Dediğimiz gibi, ASCII 7 bitlik bir sistem olduğu için, 8 bitlik

bilgisayarlarda fazladan 1 bitin boşta kalmasına izin verir. İşte bu 1 bitlik boşluk dünyanın çeşitli ülkeleri tarafından kendi karakter ihtiyaçlarını karşılamak için kullanıldı. Dolayısıyla Almanlar 128-256 arasını farklı karakterlerle, Fransızlar başka karakterlerle, Yunanlar ise bambaşka karakterlerle doldurdurlar.

Hatırlarsanız ASCII'nin ortaya çıkış sebebi bilgisayarlar arasında veri alışverişini mümkün kılmaktı. ASCII Amerika'daki bilgisayarlar arasında sağlıklı bir veri alışverişi gerçekleştirilmesini rahatlıkla mümkün kılıyordu. Ama bilgisayarların dünyaya yayılması ile birlikte ilk baştaki veri aktarımı problemi tekrar ortaya çıktı. Bu defa da, mesela Türkiye'den gönderilen bir metin (örneğin bir e.posta) Almanya'daki bilgisayarlarda düzgün görüntülenemeyebiliyordu. Örneğin Windows-1254 (cp1254) numaralı kod sayfası ile kodlanmış Türkçe bir metin, Almanya'da Windows-1250 numaralı kod sayfasının tanımlı olduğu bir bilgisayarda, aynı sayıların her iki kod sayfasında farklı karakterlere karşılık gelmesi nedeniyle düzgün görüntülenemez.

Not

Windows-1254 adlı kod sayfası için http://en.wikipedia.org/wiki/Windows-1254 adresine; Windows-1250 adlı kod sayfası için ise http://en.wikipedia.org/wiki/Windows-1250 adresine bakabilirsiniz.

İşte nasıl 1960'lı yılların başında Bob Bemer ve arkadaşları bilgisayarlar arasında sağlıklı bir veri iletişimi sağlamak için kolları sıvayıp ASCII gibi bir çözüm ürettiyse, ASCII ve Genişletilmiş ASCII ile kodlanamayan karakterleri de kodlayıp, uluslar arasında çok geniş çaplı veri alışverişine izin verebilmek amacıyla Xerox şirketinden Joe Becker, Apple şirketinden ise Lee Collins ve Mark Davis UNICODE adlı bir çözüm üzerinde ilk çalışmaları başlattı.

Peki tam olarak nedir bu UNICODE denen şey?

Aslında Unicode da tıpkı ASCII gibi bir standarttır. Unicode'un bir proje olarak ortaya çıkışı 1987 yılına dayanır. Projenin amacı, dünyadaki bütün dillerde yer alan karakterlerin tek, benzersiz ve doğru bir biçimde temsil edilebilmesidir. Yani bu projenin ortaya çıkış gayesi, ASCII'nin yetersiz kaldığı noktaları telafi etmektir.

Sınırsız Bitlik bir Sistem

Unicode standardı ile ilgili olarak bilmemiz gereken ilk şey bu standardın ASCII'yi tamamen görmezden gelmiyor olmasıdır. Daha önce de söylediğimiz gibi, ASCII son derece yaygın ve güçlü bir standarttır. Üstelik ASCII standardı yaygın olarak kullanılmaya da devam etmektedir. Bu sebeple ASCII ile halihazırda kodlanmış karakterler UNICODE standardında da aynı şekilde kodlanmıştır. Dolayısıyla ASCII UNICODE sisteminin bir alt kümesi olduğu için, ASCII ile uyumlu olan bütün sistemler otomatik olarak UNICODE ile de uyumludur. Ancak tabii bunun tersi geçerli değildir.

UNICODE'un ASCII'den en önemli farkı, UNICODE'un ASCII'ye kıyasla çok daha büyük miktarda karakterin kodlanmasına izin vermesidir. ASCII yalnızca 128 karakterin kodlanmasına izin verirken

UNICODE 1.000.000'dan fazla karakterin kodlanmasına izin verir.

UNICODE sistemini devasa bir karakter tablosu olarak hayal edebilirsiniz. Bildiğiniz gibi ASCII 7 bitlik bir sistemdir. Bu sebeple de sadece 128 karakteri kodlayabilir. UNICODE ilk ortaya çıktığında 16 bitlik bir sistem olarak tasarlanmıştı. Dolayısıyla UNICODE daha ilk çıkışında 2**16=65536 karakterin kodlanmasına izin veriyordu. Bugün ise UNICODE sisteminin böyle kesin bir sınırı yoktur. Çünkü 'bilmem kaç bitlik bir sistem' kavramı UNICODE için geçerli değildir. Dediğimiz gibi, UNICODE'u, ucu bucağı olmayan dev bir karakter tablosu olarak düşünebilirsiniz. Bu tabloya istediğimiz kadar karakteri ekleyebiliriz. Bizi engelleyen sınırlı bir bit kavramı mevcut değildir. Çünkü UNICODE sisteminin kendisi, ASCII sisteminin aksine, doğrudan doğruya karakterleri kodlamaz. UNICODE'un yaptığı şey karakterleri tanımlamaktan ibarettir.

Unicode sisteminde her karakter tek ve benzersiz bir 'kod konumuna' (*code point*) karşılık gelir. Kod konumları şu formüle göre gösterilir:

```
U+sayının_onaltılı_değeri
```

Örneğin 'a' harfinin kod konumu şudur:

```
u+0061
```

Buradaki *0061* sayısı onaltılı bir sayıdır. Bunu onlu sayı sistemine çevirebilirsiniz:

```
>>> int("61", 16)

97
```

Hatırlarsanız 'a' harfinin ASCII tablosundaki karşılığı da *97* idi.

Esasında ASCII ile UNICODE birbirleri ile karşılaştırılamayacak iki farklı kavramdır. Neticede ASCII bir kodlama biçimidir. UNICODE ise pek çok farklı kodlama biçimini içinde barındıran devasa bir sistemdir.

Not

Unicode standardına http://www.unicode.org/versions/Unicode6.2.0/UnicodeStandard-6.2.pdf adresinden ulaşabilirsiniz.

UTF-8 Kod Çözücüsü

Dediğimiz gibi UNICODE devasa bir tablodan ibarettir. Bu tabloda karakterlere ilişkin birtakım bilgiler bulunur ve bu sistemde her karakter, kod konumları ile ifade edilir. UNICODE kendi başına karakterleri kodlamaz. Bu sistemde tanımlanan karakterleri kodlama işi kod çözücülerin görevidir.

UNICODE sistemi içinde UTF-1, UTF-7, UTF-8, UTF-16 ve UTF-32 adlı kod çözücüler bulunur.

UTF-8, UNICODE sistemi içindeki en yaygın, en bilinen ve en kullanışlı kod çözücüdür.

UTF-8 adlı kod çözücünün kodlayabildiği karakterlerin listesine http://www.fileformat.info/info/charset/UTF-8/list.htm adresinden ulaşabilirsiniz. Bu listenin sayfalar dolusu olduğuna ve her sayfaya, sayfanın en altındaki 'More...' bağlantısı ile ulaşabileceğinize dikkat edin.

1 Karakter != 1 Bayt

ASCII sisteminde her karakterin 1 bayt'a karşılık geldiğini söylemiştik. Ancak 1 bayt dünyadaki bütün karakterleri kodlamaya yetmez. Geri kalan karakterleri de kodlayabilmek için 1 bayttan fazlasına ihtiyacımız var. Mesela karakter kodlama için:

```
1 bayt kullanırsak toplam 2**8  = 256
2 bayt kullanırsak toplam 2**16 = 65,536
3 bayt kullanırsak toplam 2**24 = 16,777,216
4 bayt kullanırsak toplam 2**32 = 4,294,967,296
```

karakter kodlayabiliriz. Bu durumu şu Python kodları ile de gösterebiliriz:

```
>>> for i in range(1, 5):
...     print("{} bayt kullanırsak toplam 2**{:<2} = {:,}".format(i, i*8, (2**(i*8))))
```

Görünüşe göre biz 4 baytlık bir sistem kullanırsak gelmiş geçmiş bütün karakterleri rahatlıkla temsil etmeye yetecek kadar alana sahip oluyoruz. Ancak burada şöyle bir durum var. Bildiğiniz gibi, 0 ile 256 aralığındaki karakterler yalnızca 1 bayt ile temsil edilebiliyor. 256 ile 65,536 arasındaki karakterler için ise 2 bayt yeter. Aynı şekilde 65,536 ile 16,777,216 aralığındaki sayılar için de 3 bayt yeterli. Bu durumda eğer biz bütün karakterleri 4 bayt ile temsil edecek olursak, korkunç derece bir israfa düşmüş oluruz. Çünkü ASCII gibi bir kodlama sisteminde yalnızca 1 bayt ile temsil edilebilecek bir karakterin kapladığı alan bu sistemle boşu boşuna 4 kat artmış olacaktır.

Bu sorunun çözümü elbette sabit boyutlu karakter kodlama biçimleri yerine değişken boyutlu karakter kodlama biçimleri kullanmaktır. İşte UNICODE sistemi içindeki UTF-8 adlı kod çözücü, karakterleri değişken sayıda baytlar halinde kodlayabilir. UTF-8, UNICODE sistemi içinde tanımlanmış karakterleri kodlayabilmek için 1 ile 4 bayt arası değerleri kullanır. Böylece de bu kod çözücü UNICODE sistemi içinde tanımlanmış bütün karakterleri temsil edebilir.

Bu durumu bir örnek üzerinden göstermeye çalışalım:

```
>>> for s in l:
...     print("{:<5}{:<15}{:<15}".format(s, s.encode("utf-8"), len(s.encode("utf-8"))))
...
a    b'a'           1
b    b'b'           1
c    b'c'           1
```

```
ç   b'\xc3\xa7'   2
d   b'd'          1
e   b'e'          1
f   b'f'          1
g   b'g'          1
ğ   b'\xc4\x9f'   2
h   b'h'          1
ı   b'\xc4\xb1'   2
i   b'i'          1
j   b'j'          1
k   b'k'          1
l   b'l'          1
m   b'm'          1
n   b'n'          1
o   b'o'          1
ö   b'\xc3\xb6'   2
p   b'p'          1
r   b'r'          1
s   b's'          1
ş   b'\xc5\x9f'   2
t   b't'          1
u   b'u'          1
ü   b'\xc3\xbc'   2
v   b'v'          1
y   b'y'          1
z   b'z'          1
```

Burada ilk sütun Türk alfabesindeki tek tek harfleri gösteriyor. İkinci sütun ise bu harflerin UTF-8 ile kodlandığında nasıl göründüğünü. Son sütunda ise UTF-8 ile kodlanan Türk harflerinin kaç baytlık yer kapladığını görüyoruz.

Bu tabloyu daha iyi anlayabilmek için mesela buradaki ‘ç’ harfini ele alalım:

```
>>> 'ç'.encode('utf-8')

b'\xc3\xa7'
```

Burada Python’ın kendi yerleştirdiği karakterleri çıkarırsak (‘b’ ve ‘\x’ gibi) elimizde şu onaltılı sayı kalır:

```
c3a7
```

Bu onaltılı sayının onlu sistemdeki karşılığı şudur:

```
>>> int('c3a7', 16)

50087
```

50087 sayısının ikili sayma sistemindeki karşılığı ise şudur:

```
>>> bin(50087)

'0b1100001110100111'
```

Gördüğünüz gibi, bu sayı 16 bitlik, yani 2 baytlık bir sayıdır. Bunu nasıl teyit edeceğinizi biliyorsunuz:

```
>>> (50087).bit_length()

2
```

http://www.fileformat.info/info/charset/UTF-8/list.htm adresine gittiğinizde de UTF-8 tablosunda 'ç' harfinin 'c3a7' sayısıyla eşleştirildiğini göreceksiniz.

Bir de UTF-8'in 'a' harfini nasıl temsil ettiğine bakalım:

```
>>> "a".encode("utf-8")

b'a'
```

'a' harfi standart ASCII harflerinden biri olduğu için Python doğrudan bu harfin kendisini gösteriyor. Eğer bu harfin hangi sayıya karşılık geldiğini görmek isterseniz şu kodu kullanabilirsiniz:

```
>>> ord("a")

97
```

Daha önce de söylediğimiz gibi, UNICODE sistemi ASCII ile uyumludur. Yani ASCII sisteminde tanımlanmış bir harf hangi sayı değerine sahipse, UNICODE içindeki bütün kod çözücüleri de o harf için aynı sayıyı kullanır. Yani mesela 'a' harfi hem ASCII'de, hem UTF-8'de *97* sayısı ile temsil edilir. Bu sayı 256'dan küçük olduğu için yalnızca 1 bayt ile temsil edilir. Ancak standart ASCII dışında kalan karakterler, farklı kod çözücüler tarafından farklı sayılarla eşleştirilecektir. Bununla ilgili şöyle bir çalışma yapabiliriz:

```
kod_çözücüler = ['UTF-8', 'cp1254', 'latin-1', 'ASCII']

harf = 'İ'

for kç in kod_çözücüler:
```

```
    try:
        print("'{}' karakteri {} ile {} olarak "
              "ve {} sayısıyla temsil edilir.".format(harf, kç,
                                                      harf.encode(kç),
                                                      ord(harf)))
    except UnicodeEncodeError:
        print("'{}' karakteri {} ile temsil edilemez!".format(harf, kç))
```

Bu programı çalıştırdığımızda şuna benzer bir çıktı alırız:

```
'İ' karakteri UTF-8 ile b'\xc4\xb0' olarak ve 304 sayısıyla temsil edilir
'İ' karakteri cp1254 ile b'\xdd' olarak ve 304 sayısıyla temsil edilir.
'İ' karakteri latin-1 ile temsil edilemez!
'İ' karakteri ASCII ile temsil edilemez!
```

Bu ufak programı kullanarak hangi karakterin hangi kod çözücü ile nasıl temsil edildiğini (veya temsil edilip edilemediğini) görebilirsiniz.

Eksik Karakterler ve encode Metodu

Dediğimiz ve örneklerden de gördüğümüz gibi, her karakter her kod çözücü ile çözülemeyebilir. Mesela Windows-1254 adlı kod sayfasında bulunan bir karakter Windows-1250 adlı kod sayfasında bulunamadığında, bulunmayan karakterin yerine bir soru işareti (veya başka bir simge) yerleştirilecektir.

Aslında siz bu olguya hiç yabancı değilsiniz. İnternette dolaşırken mutlaka anlamsız karakterlerle dolu web sayfalarıyla karşılaşmışsınızdır. Bu durumun sebebi, ilgili sayfanın dil kodlamasının (*encoding*) düzgün belirtilmemiş olmasıdır. Yani sayfanın HTML kodları arasında *meta charset* etiketi ya hiç yazılmamış ya da yanlış yazılmıştır. Eğer bu etiket hiç yazılmamışsa, Internet tarayıcınız dil kodlamasının ne olduğunu kendince tahmin etmeyece çalışacak, çoğunlukla da yanlış bir karar verecektir. Tarayıcınız metnin dilini düzgün tespit edemediği için de bu metni yanlış bir karakter tablosu ile eşleştirecek, o karakter tablosunda tanımlanmamış karakterler yerine bir soru işareti veya başka anlamsız bir simge yerleştirecektir. Metni düzgün görüntüleyebilmek için tarayıcınızın dil kodlamasının yapıldığı menü öğesini bulup, doğru dil kodlamasını kendiniz seçeceksiniz. Böyle bir şeyi hayatınız boyunca en az bir kez yapmak zorunda kaldığınıza eminim...

Bir karakter kümesinde herhangi bir karakter bulunamadığında, bulunamayan bu karakterin yerine neyin geleceği, tamamen aradaki yazılıma bağlıdır. Örneğin söz konusu olan bir Python programıysa, ilgili karakter bulunamadığında öntanımlı olarak bu karakterin yerine hiçbir şey koyulmaz. Onun yerine program çökmeye bırakılır... Ancak böyle bir durumda ne yapılacağını isterseniz kendiniz de belirleyebilirsiniz.

Bunun için karakter dizilerinin encode() metodunun *errors* adlı parametresinden yararlanacağız. Bu parametre dört farklı değer alabilir:

Parametre	Anlamı
‘strict’	Karakter temsil edilemiyorsa hata verilir
‘ignore’	Temsil edilemeyen karakter görmezden gelinir
‘replace’	Temsil edilemeyen karakterin yerine bir ‘?’ işareti koyulur
‘xmlcharrefreplace’	Temsil edilemeyen karakter yerine XML karşılığı koyulur

Bu parametreleri şöyle kullanıyoruz:

```
>>> "bu Türkçe bir cümledir.".encode("ascii", errors="strict")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character '\xfc' in
position 4: ordinal not in range(128)
```

‘strict’ zaten öntanımlı değerdir. Dolayısıyla eğer *errors* parametresine herhangi bir değer vermezsek Python sanki ‘strict’ değerini vermişiz gibi davranacak ve ilgili karakter kodlaması ile temsil edilemeyen bir karakter ile karşılaşıldığında hata verecektir:

```
>>> "bu Türkçe bir cümledir.".encode("ascii")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character '\xfc' in
position 4: ordinal not in range(128)
```

Gelelim öteki değerlerin ne yaptığına:

```
>>> "bu Türkçe bir cümledir.".encode("ascii", errors="ignore")

b'bu Trke bir cmledir.'
```

Gördüğünüz gibi, *errors* parametresine ‘ignore’ değerini verdiğimizde, temsil edilemeyen karakterler görmezden geliniyor:

```
>>> "bu Türkçe bir cümledir.".encode("ascii", errors="replace")

b'bu T?rk?e bir c?mledir.'
```

Burada ise ‘replace’ değerini kullandık. Böylece temsil edilemeyen karakterlerin yerine birer *?* işareti koyuldu:

```
>>> "bu Türkçe bir cümledir.".encode("ascii", errors="xmlcharrefreplace")

b'bu T&#252;rk&#231;e bir c&#252;mledir.'
```

Son olarak ise ‘xmlcharrefreplace’ değerinin ne yaptığını görüyoruz. Eğer *errors* parametresine ‘xmlcharrefreplace’ değerini verecek olursak, temsil edilemeyen her bir harf yerine o harfin XML karşılığı yerleştirilir. Bu değer, programınızdan alacağınız çıktıyı bir XML dosyasında kullanacağınız durumlarda işinize yarayabilir.

Dosyalar ve Karakter Kodlama

Dosyalar konusunu anlatırken, Python’da bir dosyanın open() fonksiyonu ile açılacağını söylemiştik. Bildiğiniz gibi open() fonksiyonunu şu şekilde kullanıyoruz:

```
>>> f = open(dosya_adı, dosya_açma_kipi)
```

Burada biz open() fonksiyonunu iki farklı parametre ile birlikte kullandık. Ancak aslında belirtmemiz gereken önemli bir parametresi daha var bu fonksiyonun. İşte bu parametrenin adı *encoding*‘dir.

Gelin şimdi bu parametrenin ne olduğuna ve nasıl kullanıldığına bakalım:

encoding

Tahmin edebileceğiniz gibi, *encoding* parametresi bir dosyanın hangi kod çözücü ile açılacağını belirtmemizi sağlar. Python’da dosyalar öntanımlı olarak locale adlı bir modülün getpreferredencoding() adlı fonksiyonunun gösterdiği kod çözücü ile açılır. Siz de dosyalarınızın varsayılan olarak hangi kod çözücü ile açılacağını öğrenmek için şu komutları yazabilirsiniz:

```
>>> import locale
>>> locale.getpreferredencoding()
```

İşte eğer siz *encoding* parametresini belirtmezseniz, dosyalarınız yukarıdaki çıktıda görünen kod çözücü ile açılacaktır.

GNU/Linux dağıtımlarında bu çıktı çoğunlukla UTF-8 olacaktır. O yüzden GNU/Linux’ta dosyalarınız muhtemelen *encoding* belirtmeseniz bile düzgün görünecektir. Ancak Windows’ta locale.getpreferredencoding() değeri cp1254 olacağı için, mesela UTF-8 ile kodlanmış dosyalarınızda özellikle Türkçe karakterler düzgün görüntülenemeyecektir. O yüzden, dosyalarınızın hangi kod çözücü ile kodlanmış olduğunu open() fonksiyonuna vereceğiniz *encoding* parametresi aracılığıyla her zaman belirtmelisiniz:

```
>>> f = open(dosya, encoding='utf-8')
```

Diyelim ki açmak istediğiniz dosya cp1254 adlı kod çözücü ile kodlanmış olsun. Eğer siz bu dosyayı açarken cp1254 adlı kod çözücüyü değil de başka bir kod çözücüyü yazarsanız elbette dosyadaki karakterler düzgün görüntülenemeyecektir.

Örneğin cp1254 ile kodlanmış bir belgeyi UTF-8 ile açmaya kalkışırsanız veya siz hiçbir kod çözücü

belirtmediğiniz halde kullandığınız işletim sistemi öntanımlı olarak dosyaları açmak için cp1254 harici bir kod çözücüyü kullanıyorsa, dosyayı okuma esnasında şuna benzer bir hata alırsınız:

```
>>> f = open("belge.txt", encoding="utf-8")
>>> f.read(50)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Python33\lib\codecs.py", line 300, in decode
    (result, consumed) = self._buffer_decode(data, self.errors, final)
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xde in position 79: invalid
 continuation byte
```

Gördüğünüz gibi, dosyamız bizim kullanmaya çalıştığımız kod çözücüden (UTF-8) farklı bir kod çözücü ile (cp1254) kodlanmış olduğu için, doğal olarak karakterler doğru sayılarla eşleştirilemiyor. Bu da kaçınılmaz olarak yukarıdaki hatanın verilmesine sebep oluyor.

Aslında siz bu hatayı tanıyorsunuz. encode() metodunu anlatırken bunun ne olduğundan ve bu hataya karşı neler yapabileceğinizden söz etmiştik.

Hatırlarsanız bu tür hatalara karşı ne tepki verileceğini belirleyebilmek için encode() metodunda *errors* adlı bir parametreyi kullanabiliyorduk. İşte open() fonksiyonunda da aynı *errors* parametresi bulunur.

errors

Dediğimiz gibi, bir dosyanın doğru görüntülenebilmesi ve okunabilmesi için, sahip olduğu kodlama biçiminin doğru olarak belirtilmesi gerekir. Ama okuyacağınız dosyaların hangi kodlama sistemine sahip olduğunu doğru tahmin etmeniz her zaman mümkün olmayabilir. Böyle durumlarda, programınızın çökmesini önlemek için çeşitli stratejiler belirlemeniz gerekir.

Bir önceki bölümde verdiğimiz örnekten de gördüğünüz gibi, eğer Python, açılmaya çalışılan dosyadaki karakterleri *encoding* parametresinde gösterilen kod çözücü ile çözemezse öntanımlı olarak bir hata mesajı üretip programdan çıkacaktır. Ancak sizin istediğiniz şey her zaman bu olmayabilir. Mesela dosyadaki karakterler doğru kodlanamasa bile programınızın çökmemesini tercih edebilirsiniz. İşte bunun için *errors* parametresinden yararlanacaksınız.

Bu parametreyi encoding() metodundan hatırlıyorsunuz. Bu parametre orada nasıl kullanılıyorsa, open() fonksiyonunda da aynı şekilde kullanılır. Dikkatlice bakın:

```
>>> f = open(dosya_adı, encoding='utf-8', errors='strict')
```

Bu zaten *errors* parametresinin öntanımlı değeridir. Dolayısıyla 'strict' değerini belirtmeseniz de öntanımlı olarak bu değeri belirtmişsiniz gibi davranılacaktır.

```
>>> f = open(dosya_adı, encoding='utf-8', errors='ignore')
```

Burada ise ‘ignore’ değerini kullanarak, Python’ın kodlanamayan karakterleri görmezden gelmesini sağlıyoruz.

```
>>> f = open(dosya_adı, encoding='utf-8', errors='replace')
```

‘replace’ değeri ise kodlanamayan karakterlerin yerine *\ufffd* karakterini yerleştirecektir. Bu karakter işlev bakımından, encode() metodunu anlatırken gördüğümüz ‘?’ işaretine benzer. Bu karaktere teknik olarak ‘UNICODE Değiştirme Karakteri’ (*UNICODE Replacement Character*) adı verilir. Bazı yerlerde bu karakteri elmas şeklinde siyah bir küp içine yerleştirilmiş soru işareti şeklinde görebilirsiniz.

Peki encode() metodunu anlatırken *errors* parametresi ile birlikte kullanabildiğimiz ‘xmlcharrefreplace’ değerini open() fonksiyonu ile birlikte kullanabilir miyiz?

Hayır, open() fonksiyonu, *errors* parametresinde bu değerin kullanılmasına izin vermez.

Konu ile ilgili Fonksiyonlar

Bu bölümde, karakter kodlama işlemleri esnasında işimize yarayacak bazı fonksiyonları ele alacağız.

repr()

İnceleyeceğimiz ilk fonksiyonun adı repr(). Esasında biz bu fonksiyonu önceki derslerimizde de birkaç örnekte kullanmıştık. Belki o zaman bu fonksiyonun ne işe yaradığını deneme-yanılma yoluyla anlamış olabilirsiniz. Eğer henüz bu fonksiyonun görevini anlamadıysanız da mesele değil. Bu bölümde bu fonksiyonu ve işlevini ayrıntılı bir şekilde anlatmaya çalışacağız.

Dilerseniz repr() fonksiyonunu anlatmaya bir örnek ile başlayalım.

Şimdi Python’ın etkileşimli kabuğunu açarak şu kodu yazın:

```
>>> "Python programlama dili"
```

Bu kodu yazıp *ENTER* düğmesine bastığınızda şöyle bir çıktı alacağınızı biliyorsunuz:

```
>>> 'Python programlama dili'
```

Dikkat ettiyseniz, yukarıdaki kodların çıktısında karakter dizisi tırnak işaretleri içinde gösteriliyor. Eğer bu karakter dizisini print() fonksiyonu içine yazarsanız o tırnak işaretleri kaybolacaktır:

```
>>> print("Python programlama dili")

Python programlama dili
```

Peki bu iki farklı çıktının sebebi ne?

Python programlama dilinde nesneler iki farklı şekilde temsil edilir:

1. Python'ın göreceği şekilde
2. Kullanıcının göreceği şekilde

Yukarıdaki ilk kullanım, yazdığımız kodu Python programlama dilinin nasıl gördüğünü gösteriyor. İkinci kullanım ise aynı kodu bizim nasıl gördüğümüzü gösteriyor. Zaten bu yüzden, etkileşimli kabukta print() fonksiyonu içinde yazmadığımız karakter dizilerinin çıktılarını ekranda görebildiğimiz halde, aynı karakter dizilerini bir dosyaya yazıp kaydettiğimizde ekranda çıktı olarak görebilmek için bunları print() fonksiyonu içine yazmamız gerekiyor.

Bu söylediklerimiz biraz karmaşık gelmiş olabilir. İsterseniz ne anlatmaya çalıştığımızı daha açık bir örnek üzerinde gösterelim. Şimdi tekrar etkileşimli kabuğu açıp şu kodu çalıştıralım:

```
>>> "birinci satır\n"
```

Bu komut bize şu çıktıyı verdi:

```
'birinci satır\n'
```

Şimdi aynı kodu bir de şöyle yazalım:

```
>>> print("birinci satır\n")

birinci satır
```

Gördüğünüz gibi, ilk kodun çıktısında satır başı karakteri (*\n*) görünürken, ikinci kodun çıktısında bu karakter görünmüyor (ama işlevini yerine getiriyor. Yani satır başına geçilmesini sağlıyor).

İşte bunun sebebi, ilk kodun Python'ın bakış açısını yansıtırken, ikinci kodun bizim bakış açımızı yansıtmasıdır.

Peki bu bilgi bizim ne işimize yarar?

Şimdi şöyle bir örnek düşünün:

Diyelim ki elimizde şöyle bir değişken var:

```
>>> a = "elma "
```

Şimdi bu değişkeni ekrana çıktı olarak verelim:

```
>>> print(a)

```

```
elma
```

Gördüğünüz gibi, bu çıktıya bakarak, *a* değişkeninin tuttuğu karakter dizisinin son tarafında bir adet boşluk karakteri olduğunu anlayamıyoruz. Bu yüzden bu değişkeni şöyle bir program içinde kullanmaya çalıştığımızda neden bozuk bir çıktı elde ettiğimizi anlamak zor olabilir:

```
>>> print("{} kilo {} kaldı!".format(23, a))

23 kilo elma  kaldı!
```

Gördüğünüz gibi, *"elma"* karakter dizisinin son tarafında bir boşluk olduğu için 'elma' ile 'kaldı' kelimeleri arasında gereksiz bir açıklık meydana geldi.

Bu boşluğu print() ile göremiyoruz, ama bu değişkeni print() olmadan yazdırdığımızda o boşluk da görünür:

```
>>> a

'elma '
```

Bu sayede programınızdaki aksaklıkları giderme imkanı kazanmış olur, şu kodu yazarak gereksiz boşlukları atabilirsiniz:

```
>>> print("{} kilo {} kaldı!".format(23, a.strip()))

23 kilo elma kaldı!
```

Daha önce de dediğimiz gibi, başında print() olmayan ifadeler, bir dosyaya yazılıp çalıştırıldığında çıktıda görünmez. O halde biz yukarıdaki özellikten yazdığımız programlarda nasıl yararlanacağız. İşte burada yardımımıza repr() adlı bir fonksiyon yetişecek. Bu fonksiyonu şöyle kullanıyoruz:

```
print(repr("karakter dizisi"))
```

Bu kodu bir dosyaya yazıp kaydettiğimizde şöyle bir çıktı alıyoruz:

```
'karakter dizisi\n'
```

Gördüğünüz gibi hem tırnak işaretleri, hem de satır başı karakteri çıktıda görünüyor. Eğer repr() fonksiyonunu kullanmasaydık şöyle bir çıktı alacaktık:

```
karakter dizisi
```

repr() fonksiyonu özellikle yazdığımız programlardaki hataları çözmeye çalışırken çok işimize yarar. Çünkü print() fonksiyonu, kullanıcının gözüne daha cazip görünecek bir çıktı üretebilmek için arkaplanda neler olup bittiğini kullanıcıdan gizler. İşte arkaplanda neler döndüğünü, print()

fonksiyonunun bizden neleri gizlediğini görebilmek için bu repr() fonksiyonundan yararlanabiliriz.

Not

repr() fonksiyonu ile ilgili gerçek hayattan bir örnek için istihza.com/blog/windows-python-3-2de-bir-hata.html adresindeki yazımızı okuyabilirsiniz.

Bütün bu açıklamalar bize şunu söylüyor: repr() fonksiyonu, bir karakter dizisinin Python tarafından nasıl temsil edildiğini gösterir. Yukarıda biz bu fonksiyonun nasıl kullanıldığına dair ayrıntıları verdik. Ancak bu fonksiyonun, yine yukarıdaki işleviyle bağlantılı olmakla birlikte biraz daha farklı görünen bir işlevi daha bulunur.

Hatırlarsanız, ilk derslerimizde *r* adlı bir kaçış dizisinden söz etmiştik. Bu kaçış dizisini şöyle kullanıyorduk:

```
print(r"\n")
```

Bildiğiniz gibi, *\n* kaçış dizisi bir alt satıra geçmemizi sağlıyor. İşte *r* adlı kaçış dizisi *\n* kaçış dizisinin bu işlevini baskılayarak, bizim *\n* kaçış dizisinin kendisini çıktı olarak verebilmemizi sağlıyor.

O halde bu noktada size şöyle bir soru sormama izin verin:

Acaba bir değişkene atanmış kaçış dizilerinin işlevini nasıl baskılayabiliriz? Yani mesela elimizde şöyle bir değişken bulunuyor olsun:

```
yeni_satır = "\n"
```

Biz bu değişkenin değerini nasıl ekrana yazdıracağız?

Eğer bunu doğrudan print() fonksiyonuna gönderirsek ne olacağını biliyorsunuz: Yeni satır karakteri işlevini yerine getirecek ve biz de yeni satır karakterinin kendisini değil, yaptığı işin sonucunu (yani satır başına geçildiğini) göreceğiz.

İşte bu tür durumlar için de repr() fonksiyonundan yararlanabilirsiniz:

```
print(repr('\n'))
```

Böylece satır başı karakterinin işlevi baskılanacak ve biz çıktıda bu karakterin kendisini göreceğiz.

Hatırlarsanız ASCII konusunu anlatırken şöyle bir örnek vermiştik:

```
for i in range(128):
    if i % 4 == 0:
        print("\n")
```

```
    print("{:<3}{:>8}\t".format(i, repr(chr(i))), sep="", end="")
```

İşte burada, repr() fonksiyonunun yukarıda sözünü ettiğimiz işlevinden yararlanıyoruz. Eğer bu kodlarda repr() fonksiyonunu kullanmazsak, ASCII tablosunu oluşturan karakterler arasındaki *\n*, *\a*, *\t* gibi kaçış dizileri ekranda görünmeyecek, bunun yerine bu kaçış dizileri doğrudan işlevlerini yerine getirecek, bu da bizim istediğimiz ASCII tablosunu üretmemize engel olacaktır.

ascii()

ascii() fonksiyonu biraz önce öğrendiğimiz repr() fonksiyonuna çok benzer. Örneğin:

```
>>> repr("asds")

'"asds"'

>>> ascii("asds")

'"asds"'
```

Bu iki fonksiyon, *ASCII* tablosunda yer almayan karakterlere karşı tutumları yönünden birbirlerinden ayrılır. Örneğin:

```
>>> repr("İ")

'"İ"'

>>> ascii("İ")

'"\\u0130"'
```

Gördüğünüz gibi, repr() fonksiyonu *ASCII* tablosunda yer almayan karakterleri de göründükleri gibi temsil ediyor. ascii() fonksiyonu ise bu karakterlerin UNICODE kod konumlarını (*code points*) gösteriyor.

Bir örnek daha verelim:

```
>>> repr("€")

'"€"'

>>> ascii("€")

'"\\u20ac"'
```

ascii() fonksiyonunun UNICODE kod konumlarını gösterme özelliğinin bir benzerini daha önce öğrendiğimiz encode() metodu yardımıyla da elde edebilirsiniz:

```
>>> "€".encode("unicode_escape")

b'\\u20ac'
```

Ancak ascii() fonksiyonunun *str* tipinde, encode() metodunun ise *bytes* tipinde bir çıktı verdiğine dikkat edin.

ord()

Bu fonksiyon, bir karakterin sayı karşılığını verir:

```
>>> ord("\n")

10

>>> ord("€")

8364
```

chr()

Bu fonksiyon, bir sayının karakter karşılığını verir:

```
>>> chr(10)

'\n'

>>> chr(8364)

'€'
```

başa dön

Baytlar (Bytes) ve Bayt Dizileri (Bytearrays)

Bu bölüme gelinceye kadar veri tipi olarak karakter dizilerinden, listelerden ve dosyalardan söz etmiştik. Bu bölümde ise Python programlama dilindeki iki veri tipinden daha söz edeceğiz. Birbirleriyle doğrudan bağlantılı oldukları için bu bölümde birlikte ele alacağımız bu veri tiplerinin adı 'baytlar'(*bytes*) ve 'bayt dizileri' (*bytearrays*).

Bu bölümde yalnızca 'baytlar' ve 'bayt dizileri' adlı veri tiplerinden söz etmeyeceğiz. Bu iki yeni veri tipini bilgi dağarcığımıza eklemenin yanısıra, önceki bölümlerde öğrendiğimiz konuları zihnimizde pekiştirmeye ve sağlamlaştırmaya da devam edeceğiz.

Giriş

Bilgisayar teknolojisi ve bilimi açısından 'karakter' tamamen soyut bir kavramdır. Son birkaç bölümdür üstüne basa basa tekrar ettiğimiz gibi, karakter dediğimiz şey, bilgisayarların anlayabildiği tek kavram olan sayılara biz insanların atadığı birtakım işaretlerden ibarettir. Dolayısıyla bilgisayarlar açısından karakterler değil, ikili sayma düzenindeki birtakım sayılar, yani bitler ve baytlar vardır.

Teknik olarak 1 bit, ikili sayma sistemindeki her bir basamağa verilen isimdir. Zaten 'bit' kelimesinin de İngilizcede 'ikili basamak' anlamına gelen '*binary digit* ifadesinin kısaltması olduğunu geçen bölümde öğrenmiştiniz.

Örneğin ikili sayma sistemindeki *0*, bir bitlik bir sayı iken, *100* üç bitlik bir sayıdır. Bu bit'lerin *8* tanesi bir araya gelince 'bayt' denen birimi oluşturur. Yani bayt, 8 adet bit'ten oluşan bir birimdir. Nasıl bir düzinede 10, bir destede de 12 öğe olmasını biz insanlar tercih etmiş ve belirlemişsek, bir bayt'ta da 8 bit olmasını yine biz insanlar tercih etmiş ve belirlemişizdir.

Önceki derslerimizde de öğrendiğimiz gibi, 8 adet bit, yani 1 bayt, Genişletilmiş ASCII sisteminde bir adet karakteri temsil etmek için kullanılabilecek en büyük birim olarak tasarlanmıştır. Yani Genişletilmiş ASCII tablolarının en sonundaki 255 numaralı karakteri temsil edebilmek için 8 adet bit, yani toplam 1 bayt kullanmamız gerekir. Standart ASCII sistemi ise 7 bitlik bir sistem olduğu için, bir adet karakteri temsil etmek için kullanılabilecek en büyük birimin 7 bit olduğunu biliyorsunuz. Dolayısıyla ASCII sistemindeki son karaktere karşılık gelen 127. sayıyı temsil edebilmek için toplam 7 bit yeterlidir.

Farklı bir sistem olan UTF-8 ise birden fazla bayt kullanarak çok sayıda karakteri temsil etmeye imkan tanır. UTF-8 ile, duruma göre 1, 2, 3 veya 4 bayt kullanarak, UNICODE sistemi içinde tanımlanmış bütün karakterleri temsil edebilirsiniz. UTF-8, değişken boyutlu bir kodlama sistemi olması sayesinde, bir karakteri temsil edebilmek için kaç bayt gerekiyorsa, o karakteri temsil etmek için o kadar bayt kullanır. Ama mesela UTF-32 adlı kod çözücü hangi karakter olursa olsun hepsini 4 bayt (32 bit) ile temsil eder. Bu durumda aslında tek baytla temsil edilebilecek 'a', 'b', 'c' gibi

karakterler de boşu boşuna 4 bayt yer kaplamış olur. Zaten UTF-8'in bu kadar yaygın ve gözde olmasının nedeni de hem çok sayıda karakteri kodlayabilmesi, hem de bu işi yaparken tasarruflu olmayı başarabilmesidir.

Python programlama dilinde karakter dizileri UNICODE kod konumları şeklinde temsil edilir. Dolayısıyla *str* adı verilen veri tipi esasında karakter dizilerini birtakım UNICODE kod konumları şeklinde gösteren soyut bir yapıdır. Yani biz Python'da karakter dizileri üzerinde işlem yaparken aslında baytlarla değil, UNICODE kod konumları ile muhatap oluyoruz. Ancak UNICODE kod konumları da tamamen soyut kavramlardır. Bunları bilgisayarın belleğinde bu şekilde temsil edemezsiniz ya da bu kod konumlarını herhangi bir ağ üzerinden başka bilgisayarlara iletemezsiniz. Bu kod konumlarını anlamlı bir şekilde kullanabilmek için öncelikle bunları bilgisayarların anlayabileceği bir biçim olan baytlara çevirmeniz gerekir. Çünkü dediğimiz gibi bilgisayarlar yalnızca bitler ve baytlardan anlar. İşte kod çözücülerin görevi de zaten bu kod konumlarını baytlara çevirmektir.

Esasında programcılık maceranız boyunca genellikle metin ihtiyaçlarınızı UNICODE kod konumları üzerinden halledeceksiniz. Python sistemdeki öntanımlı kod çözücüyü kullanarak bu kod konumlarını alttan alta bayta çevirip bellekte saklayacaktır. Ama eğer yazdığınız programlarda herhangi bir şekilde doğrudan baytlarla muhatap olmanız gerekirse *str* veri tipini değil, *bytes* adlı başka bir veri tipini kullanacaksınız. Örneğin ikili (*binary*) dosyalar üzerinde çeşitli çalışmalar yapacaksanız ve bu ikili dosyalara birtakım veriler girecekseniz, gireceğiniz bu veriler *bytes* tipinde olacaktır.

Bütün bu sebeplerden ötürü, *str* ve *bytes* veri tipleri arasındaki farkı anlamak, yazdığınız programların kararlılığı ve sağlamlığı açısından büyük önem taşır. O anda elinizde olan verinin hangi tipte olduğunu bilmezseniz, bu verinin, programınızın çalışması esnasında size ne tür tuzaklar kurabileceğini de kestiremezsiniz. Örneğin bütün karakterlerin 1 bayt olduğunu ve bunların da yalnızca 0 ile 127 arası sayılarla temsil edilebileceğini zanneden yazılımcıların tasarladığı programlara Türkçe karakterler girdiğinizde nasıl bu programlar patır patır dökülüyorsa, eğer siz de baytlar ve karakterler arasındaki farkı anlamazsanız sizin yazdığınız programlar da hiç beklemediğiniz bir anda tökezleyebilir.

Örneğin yazdığınız bir programın bir aşamasında programa yalnızca tek karakterlik verilerin girilmesi temeli üzerinden bir işlem yaptığınızı düşünün. Yani programınız içinde yapacağınız bir işlem, birden fazla karakter girişinin engellenmesini gerektiriyor olsun.

Bunun için şöyle bir şey yazmış olun:

```
a = "k"

if len(a) > 1:
    print("Lütfen yalnızca tek bir karakter giriniz!")
else:
    print("Teşekkürler!")
```

Ben burada temsili olarak *a* adlı bir değişken oluşturdum ve örnek olması açısından da bunun değerini 'k' olarak belirledim. Bu değerlerle programımız düzgün bir şekilde çalışır. Çünkü *a*

değişkeninin değeri tek bir karakter olan ‘k’ harfi. Ama eğer *a* değişkeninin değeri mesela ‘kz’ gibi bir şey olsaydı programımız ‘Lütfen yalnızca tek bir karakter giriniz!’ uyarısı verecekti...

Şimdi bu *a* değişkeninin sizin tarafınızdan belirlenmediğini, bu değerin başka bir kaynaktan geldiğini düşünün. Eğer size bu değeri gönderen kaynak, bu değeri UNICODE kod konumu olarak gönderiyorsa programınız düzgün çalışır. Ama peki ya gelen bu veri bayt olarak geliyorsa ne olacak?

Yukarıda verdiğimiz örneğin neden önemli olduğunu, daha doğrusu bu örnekle ne demek istediğimiz ve nereye varmaya çalıştığımızı anlamamış olabilirsiniz. Ama endişe etmenize hiç gerek yok. Zira bu bölümde yukarıda sorduğumuz sorunun cevabını derinlemesine ele alacağız. Bu bölümün sonuna vardığımızda neler olup bittiğini ve baytların neden bu kadar önemli olduğunu gayet iyi anlıyor olacaksınız.

Eskisi ve Yenisi

Gelin isterseniz tam olarak ne ile karşı karşıya olduğumuzu daha iyi anlayabilmek için Python3 öncesi durumun nasıl olduğuna bakalım. Eğer geçmişte Python programlama dilinin karakter dizileri ve baytları nasıl ele aldığını bilirsek bugünkü durumu ve dolayısıyla genel olarak karakter dizisi ve bayt kavramını çok daha net bir şekilde kavrayabiliriz.

Python’ın 2.x sürümlerinde, bir karakter dizisi tanımladığınızda Python bu karakter dizisini bir bayt dizisi olarak temsil ediyordu. Örneğin:

```
>>> kardiz = "e"
```

Burada *kardiz* adlı değişkenin değeri, bir baytlık bir karakter dizisidir. Bunu `len()` fonksiyonu ile teyit edelim:

```
>>> len(kardiz)

1
```

Bir de şuna bakalım:

```
>>> kardiz = "ş"
```

Burada ise *kardiz* adlı değişkenin değerinin kaç baytlık bir karakter dizisi olduğu, yani bir bakıma `len()` fonksiyonunun ne çıktı vereceği işletim sisteminden işletim sistemine farklılık gösterir. Eğer kullandığınız işletim sistemi Windows ise muhtelemen `len(kardiz)` komutu *1* çıktısı verecektir. Ama eğer bu komutu GNU/Linux dağıtımlarından birinde veriyorsanız alacağınız çıktı büyük ihtimalle *2* olacaktır.

Dediğimiz gibi, Python2’de *str* veri tipi bize bir dizi bayt verir. Dolayısıyla bu veri tipinin içinde tuttuğu karakter dizisinin kaç bayt ile gösterileceği, sistemdeki öntanımlı kod çözücünün hangisi olduğuna bağlıdır. Kullandığınız işletim sisteminde öntanımlı kod çözücünün hangisi olduğunu şu

komutla bulabilirsiniz:

```
>>> import locale
>>> locale.getpreferredencoding()
```

Eğer Windows kullanıyorsanız buradan alacağınız çıktı muhtemelen cp1254 olacaktır. cp1254, Microsoft'un Türkçe için özel olarak kullandığı bir kod sayfası olduğu için, 128 ile 256 sayıları arasında Türkçe karakterleri içerir. O yüzden bu kodlama sisteminde Türkçe karakterler 1 bayt ile gösterilebilir. Bu kod sayfasının içeriğinde hangi karakterlerin hangi sayılara karşılık geldiğini görmek için en.wikipedia.org/wiki/Windows-1254 adresindeki tabloyu inceleyebilirsiniz.

Ama eğer yukarıdaki komutların çıktısı UTF-8 veya başka bir kod çözücü ise, Türkçe karakterler 1 bayt ile gösterilemeyeceği için `len(kardiz)` komutu *1* değil, *2* çıktısı verecektir.

Bir de şuna bakalım:

```
>>> len("€")
```

Bu komutu hangi işletim sisteminde verdiğinize bağlı olarak yukarıdaki komuttan alacağınız çıktı farklı olacaktır. *str* tipi Python2'de karakter dizilerini bayt olarak temsil eder. Bu temsilin de hangi kurallara göre yapılacağı kullanılan kod çözücüye bağlıdır. Eğer karakter dizileri baytlara çevrilirken cp1254 adlı kod çözücü kullanılırsa, bu kod çözücü '€' simgesini tek bayt ile gösterilebildiği için yukarıdaki komut *1* çıktısı verir. Ama UTF-8 adlı kod çözücü '€' simgesini *3* baytla gösterebildiği için yukarıdaki komutun çıktısı da buna paralel olarak *3* olacaktır.

str veri tipi ile gösterilen bu karakter dizilerinin içindeki baytlara ulaşmak için şu yöntemi kullanabilirsiniz:

```
>>> "ş"[0]

'\xc5'

>>> "ş"[1]

'\x9f'
```

Gördüğünüz gibi, *str* veri tipi gerçekten de bize bir dizi bayt veriyor. Eğer karakter dizilerini baytlarına göre değil de sahip oldukları karakter sayısına göre saymak isterseniz bunları UNICODE olarak tanımlanız gerekiyor:

```
>>> len(u'ş')

1
```

Python3 ile birlikte yukarıda bahsettiğimiz durumda bazı değişiklikler oldu. Artık *str* veri tipi UNICODE kod konumlarını döndürüyor. Dolayısıyla artık her karakter dizisi, sahip oldukları

karakter sayısına göre sayılabiliyor:

```
>>> len("ş")

1

>>> len("€")

1
```

İşte eğer Python2'deki *str* veri tipini elde etmek istiyorsanız, Python3'te *bytes* adlı yeni veri tipini kullanmanız gerekiyor.

Bayt Tanımlamak

Bildiğiniz gibi Python programlama dilinde her veri tipinin kendine özgü bir tanımlanma biçimi var. Örneğin bir liste tanımlamak için şöyle bir şey yazıyoruz:

```
>>> liste = []
```

Böylece boş bir liste tanımlamış olduk. Aynı şekilde karakter dizilerini de şöyle tanımlıyorduk:

```
>>> kardiz = ''
```

Bu şekilde de boş bir karakter dizisi tanımlamış olduk. İşte boş bir bayt tanımlamak için de şu yapıyı kullanıyoruz:

```
>>> bayt = b''
```

Gelin tanımladığımız bu veri tipinin bayt olduğunu teyit edelim:

```
>>> type(bayt)

<class 'bytes'
```

Gördüğünüz gibi, gerçekten de bayt tipinde bir veri tanımlamışız. Nasıl karakter dizileri 'str', listeler 'list' ifadesiyle gösteriliyorsa, baytlar da 'bytes' ifadesi ile gösterilir.

Peki bu şekilde bir bayt veri tipi tanımlamak ne işimize yarar?

Hatırlarsanız bayt veri tipini ikili (*binary*) dosyaları anlatırken de görmüştük. Orada da söylediğimiz gibi, ikili dosyaları okuduğunuzda elde edeceğiniz şey karakter dizisi değil bayttır. Aynı şekilde, ikili dosyalara da ancak baytları yazabilirsiniz. Dolayısıyla eğer ikili dosyalarla birtakım işlemler yapacaksanız bu bayt veri tipini yoğun olarak kullanacağınızdan hiç şüpheniz olmasın. Yani bayt veri tipi kolayca görmezden gelebileceğiniz gereksiz bir veri tipi değildir.

bytes() Fonksiyonu

Bayt veri tipi temel olarak ASCII karakterleri kabul eder. Dolayısıyla ASCII tablosu dışında kalan karakterleri doğrudan bayt olarak temsil edemezsiniz:

```
>>> b'ş'

  File "<stdin>", line 1
SyntaxError: bytes can only contain ASCII literal characters.
```

Ama ASCII dışında kalan karakterleri de bayt'a dönüştürmenin bir yolu var. Bunun için bytes() adlı bir fonksiyondan yararlanacağız:

```
>>> b = bytes("ş", "utf-8")
```

Gördüğünüz gibi, ilgili karakterin hangi kod çözücü ile kodlanacağını belirterek, bayt tipinde bir veri oluşturabiliyoruz.

Tahmin edebileceğiniz gibi, bytes() fonksiyonu, belirttiğimiz kod çözücü ile kodlanamayan karakterlerle karşılaşılması durumunda ne yapılacağını belirlememizi sağlayan *errors* adlı bir parametreye de sahiptir:

```
>>> b = bytes("Fırat", "ascii", errors="xmlcharrefreplace")
>>> b

b'F&#305;rat'
```

Önceki derslerimizde *errors* parametresinin hangi değerleri alabileceğini tartışmıştık. Orada anlattığımız şeyler burada da geçerlidir.

Baytların Metotları

Bütün veri tiplerinde olduğu gibi, *bytes* adlı veri tipinin de birtakım metotları bulunur. Bu metotların listesini almak için şu komutu kullanabileceğinizi biliyorsunuz:

```
>>> dir(bytes)
```

Listeye baktığınızda bu metotları karakter dizilerinin metotları ile hemen hemen aynı olduğunu göreceksiniz. Baytların metotları arasında olup da karakter dizilerinin metotları arasında olmayan metotları şu şekilde elde edebilirsiniz:

```
>>> for i in dir(bytes):
...     if i not in dir(str):
...         print(i)
```

```
decode
fromhex
```

Gördüğünüz gibi, decode() ve fromhex() adlı metotlar baytlarda var, ama karakter dizilerinde yok. O yüzden biz de bu bölümde yalnızca bu iki metodu incelemekle yetineceğiz. Çünkü öteki metotları zaten karakter dizilerinden tanıyorsunuz.

decode

Hatırlarsanız karakter dizilerinin encode() adlı bir metodu vardı. Bu metot yardımıyla karakter dizilerini belli bir kodlama biçimine göre kodlayabiliyor, yani bunları baytlara çevirebiliyorduk. Mesela 'İ' harfini UTF-8 ile kodlayalım:

```
>>> "İ".encode("utf-8")

b'\xc4\xb0'
```

Aynı harfi cp1254 ile kodlarsak şu çıktıyı elde ederiz:

```
>>> "İ".encode("cp1254")

b'\xdd'
```

Tahmin edebileceğiniz gibi, bu harfi ASCII ile kodlayamayız:

```
>>> "İ".encode("ascii")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character '\u0130' in position 0:
 ordinal not in range(128)
```

İşte bu kodlama işlemini tersine çevirebilmek, yani baytları belli bir kodlama biçimine göre karakter dizilerine dönüştürebilmek için decode() metodundan yararlanacağız:

```
>>> b"\xc4\xb0".decode("utf-8")

'İ'
```

Bu baytları bir de başka kodlama sistemleri ile kodlamayı deneyelim:

```
>>> b"\xc4\xb0".decode("cp1254")

'Ä°'
```

Gördüğünüz gibi, cp1254 adlı kod çözücü bu baytı çözebiliyor, ama yanlış çözüyor! Çünkü bu baytın gösterdiği sayı cp1254 adlı kod sayfasında 'İ'ye değil, başka bir karaktere karşılık geliyor. Aslında başka iki karaktere, yani C4 ve B0 ile gösterilen *Ä* ve ° karakterlerine karşılık geliyor... Bu durumu http://en.wikipedia.org/wiki/Windows-1254 adresine gidip kendiniz de görebilirsiniz.

Bu baytları bir de ASCII ile çözmeye çalışalım:

```
>>> b"\xc4\xb0".decode("ascii")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc4 in position 0: ordinal
not in range(128)
```

Elbette, bu karakter 128'den büyük bir sayıya karşılık geldiği için ASCII tarafından çözülemeyecektir.

fromhex

Bu metot, onaltılı sayma sistemindeki bir sayıdan oluşan bir karakter dizisini alıp, bayta dönüştürür. Bu metodu şöyle kullanıyoruz:

```
>>> bytes.fromhex("c4b0")

b'\xc4\xb0'
```

Gördüğünüz gibi, bu metot bir onaltılı sayı olan *c4b0*'ı alıp, bize bir bayt nesnesi veriyor.

Bayt Dizileri

bytes adlı veri tipi ile elde ettiğimiz veri tıpkı karakter dizileri gibi, üzerinde değişiklik yapılamayan bir veridir. Dolayısıyla bir *bytes* nesnesi üzerinde değişiklik yapabilmek için o nesneyi tekrar tanımlamamız gerekir:

```
>>> b = b'PDF'
>>> v = b'-1.7'
>>> b = b + v
>>> b

b'PDF-1.7'
```

Ama Python programlama dilinde *bytes* veri tipi dışında, baytlara ilişkin ikinci veri tipi daha bulunur. *bytearray* adlı bu veri tipi, *bytes* veri tipinin aksine, üzerinde değişiklik yapılabilen bir veri tipidir.

Python'da *bytearray* veri tipini şu şekilde tanımlıyoruz:

```
>>> pdf = bytearray(b'PDF-1.7')
```

Gördüğünüz gibi, bir bayt dizisi tanımlayabilmek için bytearray() adlı bir fonksiyondan faydalanıyoruz.

Bayt Dizilerinin Metotları

Bayt dizileri bir bakıma listelerle baytların karışımı gibidir. dir(bytearray) gibi bir komutla bu veri tipinin metotlarını inceleyecek olursanız, bu veri tipinin hem baytlardan hem de listelerden birtakım metotlar aldığını görürsünüz.

Bu veri tipi listelerin şu metotlarına sahiptir:

1. append
2. clear
3. copy
4. count
5. extend
6. index
7. insert
8. pop
9. remove
10. reverse

Bu veri tipi baytların ise şu metotlarına sahiptir:

1. capitalize
2. center
3. count
4. decode
5. endswith
6. expandtabs
7. find
8. fromhex
9. index
10. isalnum
11. isalpha
12. isdigit
13. islower
14. isspace
15. istitle
16. isupper
17. join
18. ljust
19. lower

20. lstrip
21. maketrans
22. partition
23. replace
24. rfind
25. rindex
26. rjust
27. rpartition
28. rsplit
29. rstrip
30. split
31. splitlines
32. startswith
33. strip
34. swapcase
35. title
36. translate
37. upper
38. zfill

başa dön

Sözlükler

Şu ana kadar Python programlama dilinde veri tipi olarak karakter dizilerini, sayıları, listeleri, demetleri ve dosyaları öğrendik. Yeni veri tipleri öğrendikçe Python'daki hareket alanımızın da genişlediğini siz de farketmişsinizdir. Bu bölümde yine Python'daki önemli veri tiplerinden birini inceleyeceğiz. Bu defa inceleyeceğimiz veri tipinin adı sözlük. İngilizcede buna *dictionary* diyorlar.

Sözlükler de, tıpkı daha önceki derslerimizde öğrendiğimiz karakter dizileri, sayılar, listeler, demetler ve dosyalar gibi programlama maceramız boyunca işlerimizi bir hayli kolaylaştıracak ve hareket imkanımızı genişletecek veri tiplerinden biridir.

Öteki veri tiplerinde olduğu gibi, sözlüklerin de birtakım metotları vardır. İşte bu bölümde hem genel olarak sözlüklerden söz edeceğiz, hem de bu veri tipinin metotlarını en ince ayrıntısına kadar inceleyeceğiz.

Sözlük denen veri tipi Python programlama dilinin son derece kullanışlı ve işe yarar araçlarından bir tanesidir. Programlama alanında ilerledikçe, bu veri tipinin neler yapabileceğini görüp şaşıracağınızı rahatlıkla söyleyebilirim.

Esasında biz daha önceki derslerimizin birinde sözlük adlı bu veri tipinden üstünkörü de olsa söz etmiştik. Yani aslında bu veri tipiyle tanışıklığımız eskiye dayanıyor.

Hatırlayacaksınız, karakter dizilerinin str.maketrans() ve translate() adlı metotlarını anlatırken, Türkçeye özgü karakterleri ve bunların noktasız karşılıklarını içeren *çeviri_tablosu* adını verdiğimiz şöyle bir değişken tanımlamıştık:

```
çeviri_tablosu = {"Ö": "O",
                  "Ç": "C",
                  "Ü": "U",
                  "Ç": "C",
                  "İ": "I",
                  "ı": "i",
                  "Ğ": "G",
                  "ö": "o",
                  "ş": "s",
                  "ü": "u",
                  "Ş": "S",
                  "ğ": "g"}
```

Burada *çeviri_tablosu* değişkeni içinde gösterdiğimiz biçimin Python'daki adının 'sözlük' olduğunu da ifade etmiştik. İşte bu bölümde, orada şöyle bir değinip geçtiğimiz bu veri tipini çok daha ayrıntılı bir şekilde ele alma imkanımız olacak.

Hem eski bilgilerimize dayanarak, hem de yukarıda anlattıklarımızdan yola çıkarak sözlük veri tipinin

ne olduğuna dair halihazırda kafamızda bir fikir oluşmuş olduğunu söyleyebiliriz.

Sözlükler öteki veri tiplerine kıyasla biraz farklı bir görünüşe sahip bir veri tipidir. Biz birazdan sözlüklerin yapısını derinlemesine inceleyeceğiz.

Ancak sözlüklerin yapısını incelemeye geçmeden önce öğrenmemiz gereken bir şey var. Tıpkı öteki veri tiplerinde olduğu gibi, sözlüklerle de çalışabilmek için öncelikle bu veri tipini tanımlamış olmamız gerekiyor. O yüzden isterseniz sözlüklerin yapısından söz etmeden önce bir sözlüğü nasıl tanımlayacağımızdan bahsedelim.

Sözlük Tanımlamak

Dediğimiz gibi, karakter dizilerini anlatırken verdiğimiz sözlük örneği sayesinde sözlüklerin neye benzediğini az çok biliyoruz. Gelin isterseniz sözlüklerin nasıl tanımlandığını inceleyerek bu veri tipinin derinliklerine doğru ilk kulaçlarımızı atalım.

Python programlama dilindeki sözlük veri tipi, gerçek hayatta 'sözlük' denince aklınıza gelen şeye çok benzer. Mesela gerçek hayatta 'kitap' kelimesinin İngilizce bir sözlükteki karşılığı *book* kelimesidir. Dolayısıyla 'kitap' ve '*book*' kelimeleri arasındaki ilişkiyi herhalde şu şekilde temsil edebiliriz:

kitap: book

Bu manzara bize 'kitap' kelimesinin karşılığının '*book*' olduğunu açık bir şekilde gösteriyor. Eğer bu durumu Python'daki sözlük veri tipiyle göstermek isteseydik şöyle bir şey yazacaktık:

```
>>> kelimeler = {"kitap": "book"}
```

Burada, içeriği sözlük veri tipi olan *kelimeler* adlı bir değişken tanımladık. Gördüğünüz gibi, listelere benzer bir şekilde sözlük veri tipi de içinde farklı veri tiplerini barındıran, 'kapsayıcı' bir veri tipidir. Burada sözlüğümüz iki adet karakter dizisinden oluşuyor.

Yukarıdaki sözlüğü nasıl tanımladığımıza çok dikkat edin. Nasıl ki listelerin ayırt edici özelliği köşeli parantezlerdi, sözlüklerin ayırt edici özelliği de küme parantezleridir.

Esasında sözlük dediğimiz şey en basit haliyle şöyle görünür:

```
>>> sözlük = {}
```

Bu örnek boş bir sözlüktür. İsterseniz yukarıdaki veri tipinin gerçekten de bir sözlük olduğunu kanıtlayalım:

```
>>> type(sözlük)

<class 'dict'>
```

Sözlüklerin Python programlama dilindeki teknik karşılığı dict ifadesidir. type(sözlük) sorgusu <class 'dict'> çıktısı verdiğine göre, *sözlük* adlı değişkenin gerçekten de bir sözlük olduğunu söyleyebiliyoruz.

Yukarıda şöyle bir sözlük örneği verdiğimizi hatırlıyorsunuz:

```
>>> kelimeler = {"kitap": "book"}
```

Python programlama diline özellikle yeni başlayanlar, sözlüklerin görünüşü nedeniyle bir sözlükteki öğe sayısı konusunda tereddüte kapılabilir, örneğin yukarıdaki sözlüğün *2* öğeden oluştuğu yanılgısına düşebilir. O halde bu noktada size şöyle bir soru sormama izin verin: Acaba bu sözlükte kaç öğe var? Hemen bakalım:

```
>>> len(kelimeler)

1
```

Demek ki elimizdeki veri tipi bir adet öğeye sahip bir sözlükmüş. Gördüğünüz gibi, "kitap": "book" ifadesi tek başına bir öğe durumundadır. Yani burada *"kitap"* karakter dizisini ayrı, *"book"* karakter dizisini ayrı bir öğe olarak almıyoruz. Bu ikisi tek bir sözlük öğesi oluşturuyor. Hatırlarsanız, listelerde öğeleri birbirinden ayırmak için virgül işaretlerinden yararlanıyorduk. Sözlüklerde de birden fazla öğeyi birbirinden ayırmak için virgül işaretlerinden yararlanacağız:

```
>>> kelimeler = {"kitap": "book", "bilgisayar": "computer"}
```

Bir önceki örnek tek öğeliydi. Bu sözlük ise *2* öğeye sahiptir:

```
>>> len(kelimeler)

2
```

İlk derslerimizden bu yana sürekli olarak vurguladığımız gibi, Python programlama dilinde doğru kod yazmak kadar okunaklı kod yazmak da çok önemlidir. Mesela bir sözlüğü şöyle tanımladığımızda kodlarımızın pek okunaklı olmayacağını söyleyebiliriz:

```
sözlük = {"kitap": "book", "bilgisayar": "computer", "programlama": "programming",
"dil": "language", "defter": "notebook"}
```

Teknik olarak baktığımızda bu kodlarda hiçbir problem yok. Ancak sözlükleri böyle sağa doğru uzayacak şekilde tanımladığımızda okunaklılığı azaltmış oluyoruz. Bu yüzden yukarıdaki sözlüğü şöyle yazmayı tercih edebiliriz:

```
sözlük = {"kitap"      : "book",
          "bilgisayar" : "computer",
          "programlama": "programming",
```

```
        "dil"       : "language",
        "defter"    : "notebook"}
```

Bu şekilde sözlükteki öğeler arasındaki ilişki daha belirgin, yazdığınız kodlar da daha okunaklı bir hale gelecektir.

Python'da bir sözlük oluşturmanın başka yolları da olmakla birlikte, en temel sözlük oluşturma yöntemi yukarıdaki örneklerde gösterdiğimiz gibidir. Biz ilerleyen sayfalarda sözlük oluşturmanın farklı yöntemlerini de ele alacağız. Ancak şimdilik 'sözlük tanımlama' konusunu burada noktalayıp sözlüklerle ilgili önemli bir konuya daha değinelim.

Sözlük Öğelerine Erişmek

Yukarıdaki örneklerden bir sözlüğün en basit şekilde nasıl tanımlanacağını öğrendik. Peki tanımladığımız bir sözlüğün öğelerine nasıl erişeceğiz?

Hemen basit bir örnek verelim. Daha önce tanımladığımız şu sözlüğe bir bakalım mesela:

```
sözlük = {"kitap"     : "book",
        "bilgisayar" : "computer",
        "programlama": "programming",
        "dil"       : "language",
        "defter"    : "notebook"}
```

Bu sözlükte birtakım Türkçe kelimeler ve bunların İngilizce karşılıkları var. Şimdi mesela bu sözlükteki 'kitap' adlı öğeye erişelim:

```
print(sözlük["kitap"])
```

Bu kodları çalıştırdığımızda şöyle bir çıktı alırız:

```
book
```

Yukarıdaki örnekten anladığımız gibi, sözlük öğelerine erişmek için şöyle bir formül kullanıyoruz:

```
sözlük[sözlük_öğesinin_adı]
```

Aynı şekilde *sözlük* değişkeni içindeki 'bilgisayar' öğesinin karşılığını almak istersek şöyle bir kod yazıyoruz:

```
print(sözlük["bilgisayar"])
```

Bu da bize *"computer"* çıktısını veriyor.

Karakter dizilerini anlatırken verdiğimiz *çeviri_tablosu* adlı sözlüğe ve orada anlattıklarımıza geri dönelim şimdi. Artık sözlük adlı veri tipiyle iyiden iyiye tanıştığımıza göre, orada anlattıklarımız

zihninizde daha net bir hale gelmiş olmalı.

Oradaki tablomuz şöyleydi:

```
çeviri_tablosu = {"Ö": "O",
                  "ç": "c",
                  "Ü": "U",
                  "Ç": "C",
                  "İ": "I",
                  "ı": "i",
                  "Ğ": "G",
                  "ö": "o",
                  "ş": "s",
                  "ü": "u",
                  "Ş": "S",
                  "ğ": "g"}
```

Mesela bu sözlükteki *"Ö"* öğesinin karşılığını elde etmek için şöyle bir kod yazdığımızı gayet iyi hatırlıyorsunuz:

```
print(çeviri_tablosu["Ö"])
```

Bu kodları bir dosyaya kaydedip çalıştırdığımızda şöyle bir çıktı alıyorduk:

```
O
```

Gördüğünüz gibi sözlükteki *"Ö"* adlı öğeyi parantez içinde belirttiğimiz zaman, Python bize bu öğenin karşısındaki değeri veriyor. Dolayısıyla sözlük içinde *"Ö"* öğesinin karşılığı *"O"* harfi olduğu için de çıktımız *"O"* oldu.

Sözlüğün öteki öğelerini ise şu şekilde alabiliyoruz:

```
print(çeviri_tablosu["Ö"])
print(çeviri_tablosu["ç"])
print(çeviri_tablosu["Ü"])
print(çeviri_tablosu["Ç"])
print(çeviri_tablosu["İ"])
print(çeviri_tablosu["ı"])
print(çeviri_tablosu["Ğ"])
print(çeviri_tablosu["ö"])
print(çeviri_tablosu["Ş"])
print(çeviri_tablosu["ğ"])
```

Ancak kod tekrarından kaçınmak için yukarıdaki kodları şu şekilde sadeleştirme imkanımızın da olduğunu biliyorsunuz:

```
for i in çeviri_tablosu:
    print(i)
```

Gördüğünüz gibi, sözlük içinde iki nokta üst üste işaretinin sol tarafında görünen öğeleri parantez içinde yazarak, iki nokta üst üste işaretinin sağ tarafındaki değerleri elde edebiliyoruz.

Eğer bir sözlük içinde bulunmayan bir öğeye erişmeye çalışırsak Python bize KeyError tipinde bir hata mesajı verecektir. Mesela yukarıdaki sözlüğü temel alacak olursak şöyle bir sorgu hata verecektir:

```
>>> print(çeviri_tablosu["Z"])

Traceback (most recent call last):
  File "deneme.py", line 14, in <module>
    print(çeviri_tablosu["Z"])
KeyError: 'Z'
```

Sözlükte *"Z"* kaydı bulunmadığı için doğal olarak Python'ın bize bir hata mesajı göstermekten başka çaresi kalmıyor.

Sözlükler ile ilgili epey bilgi edindik. Dilerseniz bu öğrendiklerimizi örnek bir uygulama üzerinde somutlaştırmaya çalışalım. Mesela Python'daki sözlükleri kullanarak basit bir telefon defteri uygulaması yazalım:

```
telefon_defteri = {"ahmet öz" : "0532 532 32 32",
                   "mehmet su": "0543 543 42 42",
                   "seda naz" : "0533 533 33 33",
                   "eda ala"  : "0212 212 12 12"}

kişi = input("Telefon numarasını öğrenmek için bir kişi adı girin: ")

cevap = "{} adlı kişinin telefon numarası: {}"

print(cevap.format(kişi, telefon_defteri[kişi]))
```

Burada öncelikle isimler ve telefon numaralarından oluşan, sözlük veri tipinde bir telefon defteri oluşturduk:

```
telefon_defteri = {"ahmet öz" : "0532 532 32 32",
                   "mehmet su": "0543 543 42 42",
                   "seda naz" : "0533 533 33 33",
                   "eda ala"  : "0212 212 12 12"}
```

Bu kodlarda bilmediğimiz hiçbir şey yok. Sözlüklere dair öğrendiklerimizi kullanarak oluşturduğumuz oldukça basit bir sözlüktür bu.

Daha sonra kullanıcıdan, telefon numarasını öğrenmek için bir kişi adı girmesini istiyoruz. Bunu da şu kodlar yardımıyla yapıyoruz:

```
kişi = input("Telefon numarasını öğrenmek için bir kişi adı girin: ")
```

Ardından da telefon defterinde sorgulama yapacak olan kullanıcıya göstereceğimiz cevap için bir şablon oluşturuyoruz:

```
cevap = "{} adlı kişinin telefon numarası: {}"
```

Mesela kullanıcı *"ahmet öz"* ismini sorgulamışsa ona şöyle bir cevap vereceğiz:

```
"ahmet öz adlı kişinin telefon numarası 0532 532 32 32"
```

Eğer aranan isim telefon defterinde varsa, bir önceki adımda tanımladığımız cevap şablonuna göre kullanıcıyı bilgilendiriyoruz. Ama eğer eğer isim defterde yoksa, programımız hata veriyor. Bunu önlemek için şöyle bir kod yazabilirsiniz:

```
telefon_defteri = {"ahmet öz" : "0532 532 32 32",
                   "mehmet su": "0543 543 42 42",
                   "seda naz" : "0533 533 33 33",
                   "eda ala"  : "0212 212 12 12"}

kişi = input("Telefon numarasını öğrenmek için bir kişi adı girin: ")

if kişi in telefon_defteri:
    cevap = "{} adlı kişinin telefon numarası: {}"
    print(cevap.format(kişi, telefon_defteri[kişi]))
else:
    print("Aradığınız kişi telefon rehberinde yok!")
```

Gördüğünüz gibi, `if kişi in telefon_defteri` satırı yardımıyla öncelikle aranan ismin sözlükte olup olmadığını denetledik. Eğer aranan isim sözlükte yer alıyorsa bu telefon numarasını kullanıcılarımıza gösteriyoruz. Aksi durumda aranan kişinin telefon rehberinde olmadığı konusunda kullanıcılarımızı bilgilendiriyoruz.

Gördüğünüz gibi, sözlükler gerçekten de bize Python programlama maceramızda yepyeni olanakların kapısını açabilecek kadar güçlü bir veri tipi. Bu veri tipini programlarınızda bolca kullanacaksınız.

Yukarıda verdiğimiz telefon defteri uygulamasına şöyle bir baktığınızda bu uygulamanın aslında geliştirilmeye bir hayli açık olduğu dikkatinizi çekmiştir. Mesela biz bu uygulamada sadece kendi tanımladığımız bir telefon defteri üzerinden sorgulama yapmaya izin verdik. Örneğin kullanıcı bu telefon defterine kendi isim-telefon çiftlerini giremiyor. Bu veri tipini etkili bir şekilde kullanmamızı sağlayacak araçlardan henüz yoksun olduğumuz için yukarıda tanımladığımız uygulama çok basit kaldı. O halde, sözlük veri tipini daha verimli ve etkili bir biçimde kullanabilmek için hiç vakit

kaybetmeden bu veri tipinin derinliklerine doğru yol almaya devam edelim.

Sözlüklerin Yapısı

Yukarıdaki örneklerden, Python'da bir sözlüğün nasıl tanımlanacağını ve bir sözlüğün öğelerine nasıl erişileceğini öğrendik. Gelin isterseniz şimdi sözlük veri tipinin yapısına ilişkin bazı ayrıntıları inceleyelim.

Mesela şu örneği tekrar önümüze alalım:

```
sözlük = {"kitap": "book"}
```

Burada iki nokta üst üste işaretinden önce ve sonra birer tane karakter dizisi görüyoruz. Bu karakter dizileri *"kitap"* ve *"book"*. Dediğimiz gibi, sözlükler de tıpkı listeler gibi, farklı veri tiplerinin bir araya gelmesi ile oluşan birleşik/kapsayıcı bir veri tipidir. Dolayısıyla bir sözlük içinde sadece karakter dizilerini değil, başka veri tiplerini de görebilirsiniz. İlerleyen sayfalarda sözlüklere ilişkin daha karmaşık örnekler verdiğimizde sözlüklerin hangi veri tiplerini içerebileceğini de göreceğiz.

Ne dedik? Sözlük içinde iki nokta üst üste işaretinin solunda ve sağında *"kitap"* ve *"book"* adlı karakter dizileri var. Teknik olarak, iki nokta üst üste işaretinin solundaki karakter dizisine 'anahtar' (*key*), sağındaki karakter dizisine ise 'değer' (*value*) adı verilir. Bu bilgilere bakarak sözlük için şöyle bir tanım verebiliriz:

> Sözlükler; anahtar ve değer çiftlerinin birbirleriyle eşleştirildiği bir veri tipidir. Dolayısıyla sözlükler bu anahtar ve değer çiftleri arasında birebir ilişki kurar.

Mesela yukarıdaki örnekte *"kitap"* öğesi anahtar, *"book"* öğesi ise değerdir. İşte sözlük dediğimiz şey, bu anahtar ve değer çifti arasında birebir ilişki kuran bir veri tipidir. Yani sözlük adlı veri tipi, bir anahtarı bir değerle eşleştirme görevi görür.

Sözlüklerin bu özelliğini, sözlük öğelerine erişirken gayet net bir şekilde görebiliyoruz.

Yukarıdaki örneklerde tanımladığımız sözlüklerde sadece karakter dizilerini kullandık. Ama aslında sözlükler farklı veri tiplerinden oluşabilir. Mesela:

```
sözlük = {"sıfır": 0,
          "bir"  : 1,
          "iki"  : 2,
          "üç"   : 3,
          "dört" : 4,
          "beş"  : 5}
```

Burada sözlük içinde hem sayıları hem de karakter dizilerini kullandık. Aynı şekilde sözlük içinde listelere de yer verebiliriz:

```
sözlük = {"Ahmet Özkoparan": ["İstanbul", "Öğretmen", 34],
```

```
        "Mehmet Yağız"   : ["Adana", "Mühendis", 40],
        "Seda Bayrak"    : ["İskenderun", "Doktor", 30]}
```

Mesela bu sözlükte *"Seda Bayrak"* adlı kişinin bilgilerine ulaşmak istersek şöyle bir kod yazabiliriz:

```
print(sözlük["Seda Bayrak"])
```

Bu kod bize şöyle bir çıktı verecektir:

```
['İskenderun', 'Doktor', 30]
```

Gördüğünüz gibi, sözlük içinde *"Seda Bayrak"* adlı öğenin karşısındaki bilgi listesine ulaşabildik.

İstersek sözlükleri, içlerinde başka sözlükleri barındıracak şekilde de tanımlayabiliriz:

```
kişiler = {"Ahmet Özkoparan": {"Memleket": "İstanbul",
                               "Meslek"  : "Öğretmen",
                               "Yaş"     : 34},

           "Mehmet Yağız"   : {"Memleket": "Adana",
                               "Meslek"  : "Mühendis",
                               "Yaş"     : 40},

           "Seda Bayrak"    : {"Memleket": "İskenderun",
                               "Meslek"  : "Doktor",
                               "Yaş"     : 30}}
```

Böylece şöyle kodlar yazabiliriz:

```
print(kişiler["Mehmet Yağız"]["Memleket"])
print(kişiler["Seda Bayrak"]["Yaş"])
print(kişiler["Ahmet Özkoparan"]["Meslek"])
```

Yukarıdaki yapının benzerini listeler konusundan hatırlıyor olmalısınız. İç içe geçmiş listelerin öğelerine ulaşırken de buna benzer bir sözdiziminden yararlanıyorduk. Örneğin:

```
liste = [["Ahmet", "Mehmet", "Ayşe"],
         ["Sedat", "Serkan", "Selin"],
         ["Zeynep", "Nur", "Eda"]]
```

Burada bir liste içinde iç içe geçmiş üç farklı liste ile karşı karşıyayız. Mesela ilk listenin ilk öğesi olan *"Ahmet"* adlı öğeye erişmek istersek şöyle bir kod yazmamız gerekiyor:

```
print(liste[0][0])
```

İşte iç içe geçmiş sözlüklerin öğelerine ulaşmak için de buna benzer bir kod yazmamız gerekiyor.

Örneğin *kişiler* adlı sözlükteki *"Mehmet Yağız"* adlı kişinin yaşına ulaşmak istersek şöyle bir şey yazacağız:

```
print(kişiler["Mehmet Yağız"]["Yaş"])
```

Gelin isterseniz *kişiler* adlı sözlüğü kullanarak basit bir irtibat listesi uygulaması yazalım. Böylece sözlüklere elimizi alıştırmış oluruz:

```
kişiler = {"Ahmet Özkoparan": {"Memleket": "İstanbul",
                               "Meslek"  : "Öğretmen",
                               "Yaş"     : 34},

           "Mehmet Yağız"   : {"Memleket": "Adana",
                               "Meslek"  : "Mühendis",
                               "Yaş"     : 40},

           "Seda Bayrak"    : {"Memleket": "İskenderun",
                               "Meslek"  : "Doktor",
                               "Yaş"     : 30}}

isim = "Hakkında ayrıntılı bilgi edinmek \
istediğiniz kişinin adını girin: "

arama = input(isim)

ayrıntı = input("Memleket/Meslek/Yaş? ")

print(kişiler[arama][ayrıntı])
```

Tıpkı bir önceki telefon defteri uygulamamız gibi, bu irtibat listesi uygulaması da geliştirilmeye açıktır. Ancak henüz bu iki uygulamayı geliştirmemizi sağlayacak bilgilerden yoksunuz. Bu uygulamaları istediğimiz kıvama sokabilmek için sözlüklere dair öğrenmemiz gereken başka şeyler de var.

Sözlüklerin öteki veri tiplerinden önemli bir farkı, sözlük içinde yer alan öğelerin herhangi bir sıralama mantığına sahip olmamasıdır. Yani sözlükteki öğeler açısından 'sıra' diye bir kavram yoktur.

Örneğin bir liste, demet veya karakter dizisi içindeki öğelere; bu öğelerin o liste, demet veya karakter dizisi içindeki sıralarına göre erişebilirsiniz:

```
>>> liste = ["Ahmet", "Mehmet", "Zeynep"]
>>> liste[0]

'Ahmet'
```

```
>>> liste[-1]

'Zeynep'
```

Ancak sözlükler açısından böyle bir şey söz konusu değildir:

```
>>> sözlük = {'elma': 'apple',
...           'armut': 'pear',
...           'çilek': 'strawberry'}
>>> sözlük[0]

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 0
```

Gördüğünüz gibi, sözlükler üzerinde sıralamaya dayalı bir sorgulama yapmaya çalıştığımızda Python bize bir hata mesajı gösteriyor.

Bu durumun etkilerini şurada da görebilirsiniz:

Dikkatlice bakın:

```
>>> sözlük = {'a': '0', 'b': '1', 'c': '2'}
>>> sözlük

{'a': '0', 'c': '2', 'b': '1'}
```

Bu çıktıyı iyi inceleyin. Göreceğiniz gibi, çıktıda görünen öğeler bizim sözlüğü tanımladığımız sıradaki gibi değil. Biz sözlüğü ‘a’, ‘b’ ve ‘c’ şeklinde sıralayarak tanımladık, ama çıktı ‘a’, ‘c’ ve ‘b’ şeklinde oldu. O yüzden sözlükler üzerinde çalışırken öğelerin sırasına dayalı herhangi bir işlem yapmak hiç mantıklı değildir. Çünkü sözlükteki öğeleri tanımlarken kullandığınız sıralama düzeninin çıktıda da aynen korunacağının herhangi bir garantisi bulunmaz.

Sözlüklere Öğe Eklemek

Tıpkı listeler gibi, sözlükler de büyüyüp küçülebilen bir veri tipidir. Yani bir sözlüğü ilk kez tanımladıktan sonra istediğimiz zaman bu sözlüğe yeni öğeler ekleyebilir veya varolan öğeleri çıkarabiliriz. Biz şimdi bir sözlüğe nasıl öğe ekleyeceğimizi inceleyeceğiz.

Diyelim ki elimizde şöyle boş bir sözlük var:

```
>>> sözlük = {}
```

Bu listeye öğe eklemek için şöyle bir formül kullanacağız:

```
>>> sözlük[anahtar] = değer
```

Bu formülü bir örnek üzerinden somutlaştıralım:

```
>>> sözlük["Ahmet"] = "Adana"
```

Böylece sözlüğe, anahtarı *"Ahmet"*, değeri ise *"Adana"* olan bir öğe eklemiş olduk. Sözlüğümüzün son durumunu kontrol edelim:

```
>>> print(sözlük)

{'Ahmet': 'Adana'}
```

Gördüğünüz gibi, "Ahmet" öğesi sözlüğe eklendi. Artık bu öğeye normal yollardan ulaşabiliriz:

```
>>> print(sözlük["Ahmet"])

Adana
```

Elimiz alışsın diye sözlüğe öğe eklemeye devam edelim:

```
>>> sözlük["Mehmet"] = "İstanbul"
>>> sözlük

{'Ahmet': 'Adana', 'Mehmet': 'İstanbul'}

>>> sözlük["Seda"] = "Mersin"
>>> sözlük

{'Ahmet': 'Adana', 'Mehmet': 'İstanbul', 'Seda': 'Mersin'}

>>> sözlük["Eda"] = "Tarsus"
>>> sözlük

{'Ahmet': 'Adana', 'Eda': 'Tarsus', 'Mehmet': 'İstanbul', 'Seda': 'Mersin'}
```

Özellikle son çıktıya dikkatlice bakın. Sözlüğe en son *"Eda"* öğesini eklemiştik. Ama sözlüğü ekrana bastığımızda bu öğenin sözlüğün sonuna değil ortasına bir yere yerleştiğini görüyoruz. Bu durumun, sözlüklerin sırasız bir veri tipi olmasından kaynaklandığını biliyorsunuz.

Gelin pratik olması açısından birkaç örnek daha verelim.

Elimizde şöyle bir sözlük olsun:

```
>>> personel = {"Mehmet Öz": "AR-GE Müdürü",
```

```
...         "Samet Söz": "Genel Direktör",
...         "Sedat Gün": "Proje Müdürü"}
```

Şimdi bu sözlüğe "Turgut Özben": "Mühendis" anahtar-değer çiftini ekleyelim:

```
>>> personel["Turgut Özben"] = "Mühendis"
```

Sözlüğümüzün son halini görelim:

```
>>> print(personel)

{'Samet Söz': 'Genel Direktör', 'Mehmet Öz': 'AR-GE Müdürü', 'Turgut Özben':
'Mühendis', 'Sedat Gün': 'Proje Müdürü'}
```

Gördüğünüz gibi eklemek istediğimiz öğe sözlüğe eklenmiş. Ancak bu öğenin sözlüğün en sonuna değil, sözlük içine rastgele bir şekilde yerleştirildiğine dikkatinizi çekmek isterim. Çünkü, dediğimiz gibi, sözlükler sırasız bir veri tipidir.

Gelin bu konuyu daha iyi anlamak için bir örnek daha verelim.

Önce *notlar* adında boş bir sözlük tanımlayalım:

```
>>> notlar = {}
```

Bu sözlüğe öğrencilerin sınavdan aldıkları notları ekleyeceğiz:

```
>>> notlar["Ahmet"] = 45
>>> notlar["Mehmet"] = 77
>>> notlar["Seda"] = 98
>>> notlar["Deniz"] = 95
>>> notlar["Ege"] = 95
>>> notlar["Zeynep"] = 100
```

Sözlüğümüzün son halini görelim:

```
>>> print(notlar)

{'Seda': 98, 'Ege': 95, 'Mehmet': 77, 'Zeynep': 100, 'Deniz': 95, 'Ahmet': 45}
```

Bu noktada sözlüklerin önemli bir özelliğinden bahsetmemiz uygun olacak. Bir sözlüğe değer olarak bütün veri tiplerini verebiliriz. Yani:

```
>>> sözlük = {}
>>> sözlük = {'a': 1}
>>> sözlük = {'a': (1,2,3)}
>>> sözlük = {'a': 'kardiz'}
```

```
>>> sözlük = {'a': [1,2,3]}
```

Gördüğünüz gibi, sözlükler değer olarak her türlü veri tipini kabul ediyor. Ama durum sözlük anahtarları açısından böyle değildir. Yani sözlüklere anahtar olarak her veri tipini atayamayız. Bir değerin 'anahtar' olabilmesi için, o öğenin değiştirilemeyen (*immutable*) bir veri tipi olması gerekir. Python'da şimdiye kadar öğrendiğimiz şu veri tipleri değiştirilemeyen veri tipleridir:

1. Demetler
2. Sayılar
3. Karakter Dizileri

Şu veri tipleri ise değiştirilebilen veri tipleridir:

1. Listeler
2. Sözlükler

Dolayısıyla bir sözlüğe ancak şu veri tiplerini ekleyebiliriz:

1. Demetler
2. Sayılar
3. Karakter Dizileri

Şu kodları dikkatlice inceleyin:

Önce boş bir sözlük oluşturalım:

```
>>> sözlük = {}
```

Bu sözlüğe anahtar olarak bir demet ekleyelim:

```
>>> l = (1,2,3)
>>> sözlük[l] = 'falanca'
>>> sözlük

{(1, 2, 3): 'falanca'}
```

Bir sayı ekleyelim:

```
>>> l = 45
>>> sözlük[l] = 'falanca'
>>> sözlük

{45: 'falanca', (1, 2, 3): 'falanca'}
```

Bir karakter dizisi ekleyelim:

```
>>> l = 'kardiz'
>>> sözlük[l] = 'falanca'
>>> sözlük

{'kardiz': 'falanca', 45: 'falanca', (1, 2, 3): 'falanca'}
```

Yukarıdakiler, değiştirilemeyen veri tipleri olduğu için sözlüklere anahtar olarak eklenebildi.

Bir de şunlara bakalım:

Sözlüğümüze anahtar olarak bir liste eklemeye çalışalım:

```
>>> l = [1,2,3]
>>> sözlük[l] = 'falanca'

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Kümemize bir sözlük eklemeye çalışalım:

```
>>> l = {"a": 1, "b": 2, "c": 3}
>>> sözlük[l] = 'falanca'

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'
```

Sözlüklerle çalışırken sözlüklerin bu özelliğine karşı uyanık olmalıyız.

Sözlük Öğeleri Üzerinde Değişiklik Yapmak

Sözlükler değiştirilebilir veri tipleridir. Dolayısıyla sözlükler üzerinde rahatlıkla istediğimiz değişikliği yapabiliriz.

Sözlükler üzerinde değişiklik yapma işlemi, biraz önce öğrendiğimiz, sözlüklere yeni öğe ekleme işlemiyle aynıdır. Dikkatlice bakın:

```
>>> notlar = {'Seda': 98, 'Ege': 95, 'Mehmet': 77,
... 'Zeynep': 100, 'Deniz': 95, 'Ahmet': 45}
```

Sözlüğümüz bu. Şimdi bu sözlükteki ‘Ahmet’ adlı kişinin *45* olan notunu *65* olarak değiştirelim:

```
>>> notlar["Ahmet"] = 65
>>> print(notlar)
```

```
{'Seda': 98, 'Ege': 95, 'Mehmet': 77, 'Zeynep': 100, 'Deniz': 95, 'Ahmet': 65}
```

Gördüğünüz gibi Ahmet'in notu *65* olarak değişmiş...

Sözlük Üreteçleri (*Dictionary Comprehensions*)

Hatırlarsanız listeleri anlatırken liste üreteçleri adı bir kavramdan söz etmiştik. Liste üreteçlerini kullanarak tek satırda ve hızlı bir şekilde listeler oluşturabiliyorduk. Aynı şey sözlükler için de geçerlidir. Tıpkı liste üreteçlerinde olduğu gibi, sözlük üreteçleri sayesinde tek satırda ve hızlı bir şekilde sözlükler üretebiliriz.

Örneğin elimizde, Türkçe alfabedeki harfleri içeren *harfler* adlı şöyle bir liste olduğunu düşünün:

```
>>> harfler = 'abcçdefgğhıijklmnoöprsştuüvyz'
```

Amacımız bu harflerin her birine bir numara vermek. Yani nihai olarak şöyle bir sözlük elde etmek istiyoruz:

```
{'ğ': 8,
 'v': 26,
 'ş': 22,
 'u': 24,
 't': 23,
 'ö': 18,
 'ı': 10,
 'p': 19,
 's': 21,
 'r': 20,
 'ü': 25,
 'y': 27,
 'ç': 3,
 'z': 28,
 'e': 5,
 'd': 4,
 'g': 7,
 'f': 6,
 'a': 0,
 'c': 2,
 'b': 1,
 'm': 15,
 'l': 14,
 'o': 17,
 'n': 16,
```

```
'i': 11,
'h': 9,
'k': 13,
'j': 12}
```

Bunun için birkaç farklı yöntemden yararlanabiliriz. Örneğin:

```
>>> sözlük = {}
>>> for i in harfler:
...     sözlük = harfler.index(i)
```

veya:

```
>>> sözlük = {}
>>> for i in range(len(harfler)):
...     sözlük[harfler[i]] = i
```

İşte bu işlemleri sözlük üreteçlerini kullanarak çok daha hızlı ve pratik bir şekilde halledebiliriz. Dikkatlice bakın:

```
>>> sözlük = {i: harfler.index(i) for i in harfler}
```

Bir örnek daha verelim. Diyelim ki elinizde şöyle bir isim listesi var:

```
isimler = ["ahmet", "mehmet", "fırat", "zeynep", "selma", "abdullah", "cem"]
```

Amacınız, bu isimleri ve her bir ismin kaç harften oluştuğunu gösteren bir sözlük elde etmek. Yani nihai olarak şöyle bir şey olsun istiyorsunuz elinizde:

```
{'zeynep': 6,
 'cem': 3,
 'abdullah': 8,
 'ahmet': 5,
 'mehmet': 6,
 'fırat': 5,
 'selma': 5}
```

İşte bu görev için de sözlük üreteçlerinden yararlanabilirsiniz:

```
>>> isimler = ["ahmet", "mehmet", "fırat", "zeynep", "selma", "abdullah", "cem"]
>>> sözlük = {i: len(i) for i in isimler}
>>> sözlük

{'zeynep': 6, 'cem': 3, 'abdullah': 8, 'ahmet': 5, 'mehmet': 6, 'fırat': 5, 'selma': 5}
```

Bildiğiniz gibi sözlükler, her biri birbirinden *:* işareti ile ayrılan birtakım anahtar-değer çiftlerinden

oluşuyor. İşte yukarıdaki sözlük üreteci yapısında biz *:* işaretinin sol tarafına *isimler* adlı listedeki her bir öğeyi; sağ tarafına da bu öğelerin uzunluklarını bir çırpıda ekliyoruz.

başa dön

Sözlüklerin Metotları

Tıpkı öteki veri tiplerinde olduğu gibi, sözlüklerin de birtakım metotları bulunur. Bu bölümde sözlüklerin şu metotlarını inceleyeceğiz:

1. clear()
2. copy()
3. fromkeys()
4. get()
5. items()
6. keys()
7. pop()
8. popitem()
9. setdefault()
10. update()
11. values()

İlk olarak keys() metoduyla başlayalım.

keys()

Sözlükleri tarif ederken, sözlüklerin anahtar-değer çiftlerinden oluşan bir veri tipi olduğunu söylemiştik. Bir sözlüğü normal yollardan ekrana yazdırırsanız size hem anahtarları hem de bunlara karşılık gelen değerleri verecektir. Ama eğer bir sözlüğün sadece anahtarlarını almak isterseniz keys() metodundan yararlanabilirsiniz:

```
>>> sözlük = {"a": 0,
...           "b": 1,
...           "c": 2,
...           "d": 3}
>>> print(sözlük.keys())

dict_keys(['b', 'c', 'a', 'd'])
```

Gördüğünüz gibi, sözlük.keys() komutu bize bir *dict_keys* nesnesi veriyor. Bu nesneyi programınızda kullanabilmek için isterseniz, bunu listeye, demete veya karakter dizisine dönüştürebilirsiniz:

```
>>> liste = list(sözlük.keys())
>>> liste

['b', 'c', 'a', 'd']
```

```
>>> demet = tuple(sözlük.keys())
>>> demet

('b', 'c', 'a', 'd')

>>> kardiz = "".join(sözlük.keys())
>>> kardiz

'bcad'
```

Son örnekte sözlük anahtarlarını karakter dizisine dönüştürmek için str() fonksiyonunu değil, karakter dizilerinin join() adlı metodunu kullandığımıza dikkat edin. Çünkü tuple() ve list() fonksiyonlarının aksine str() fonksiyonu, sözlükteki anahtarların nasıl bir ölçüte göre karakter dizisine çevrileceğine dair bir kural içermez. Zira siz bu sözlük anahtarlarını pek çok farklı şekilde karakter dizisine çevirebilirsiniz. Örneğin öğeleri karakter dizisi içine yerleştirirken öğelerin arasına virgül koymak isteyebilirsiniz:

```
>>> kardiz = ', '.join(sözlük.keys())
>>> kardiz

'b, c, a, d'
```

Eğer sözlük anahtarlarını str() fonksiyonu yardımıyla karakter dizisine dönüştürmeye kalkışırsanız beklemediğiniz bir çıktı alırsınız.

values()

keys() metodu bir sözlüğün anahtarlarını veriyor. Bir sözlüğün değerlerini ise values() metodu verir:

```
>>> sözlük
{'b': 1, 'c': 2, 'a': 0, 'd': 3}

>>> print(sözlük.values())

dict_values([1, 2, 0, 3])
```

Gördüğünüz gibi, bu metottan bir *dict_values* nesnesi alıyoruz. Tıpkı keys() metodunda olduğu gibi, values() metodunda da bu çıktıyı başka veri tiplerine dönüştürme imkanına sahibiz:

```
>>> liste = list(sözlük.values())
>>> liste
```

```
[1, 2, 0, 3]

>>> demet = tuple(sözlük.values())
>>> demet

(1, 2, 0, 3)
```

Yalnız bu verileri karakter dizisine dönüştürmeye çalıştığınızda ufak bir problemle karşılacaksınız:

```
>>> kardiz = "".join(sözlük.values())

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 0: expected str instance, int found
```

Bunun sebebi, sözlükteki değerlerin *int* tipinde olmasıdır. Bildiğiniz gibi, sadece aynı tip verileri birbiriyle birleştirebiliriz. Eğer birleştirmek istediğimiz veriler birbirinden farklı tipte ise, bunları birleştirmeden önce bir dönüştürme işlemi yapmamız gerekir:

```
>>> kardiz = "".join([str(i) for i in sözlük.values()])
>>> kardiz

'1203'
```

Gördüğünüz gibi, sözlükteki değerlerin her birini, tek bir liste üreteci içinde karakter dizisine dönüştürdük ve ortaya çıkan listeyi karakter dizilerinin join() metodu yardımıyla, öğelerin arasında hiçbir boşluk bırakmadan *kardiz* adlı bir karakter dizisi içine yerleştirdik. Elbette eğer isteseydik bu öğelerin her birinin arasına bir virgül de koyabilirdik:

```
>>> kardiz = ", ".join([str(i) for i in sözlük.values()])
>>> kardiz

'1, 2, 0, 3'
```

items()

Bu metot, bir sözlüğün hem anahtarlarını hem de değerlerini aynı anda almamızı sağlar:

```
>>> sözlük.items()

dict_items([('a', 0), ('c', 2), ('b', 1)])
```

Gördüğünüz gibi, tek bir liste içinde iki öğeli demetler halinde hem anahtarları hem de değerleri

görebiliyoruz. Bu metot sıklıkla for döngüleri ile birlikte kullanılarak bir sözlüğün anahtar ve değerlerinin manipüle edilebilmesini sağlar:

```
>>> for anahtar, değer in sözlük.items():
...     print("{} = {}".format(anahtar, değer))
...
a = 0
c = 2
b = 1
```

get()

Bu metot sözlüklerin en kullanışlı metotlarından biridir. Bu metot pek çok durumda işinizi bir hayli kolaylaştırır.

Diyelim ki şöyle bir program yazdık:

```
#!/usr/bin/env python3.0

ing_sözlük = {"dil": "language", "bilgisayar": "computer", "masa": "table"}

sorgu = input("Lütfen anlamını öğrenmek istediğiniz kelimeyi yazınız:")

print(ing_sözlük[sorgu])
```

Bu programı çalıştırdığımızda eğer kullanıcı "ing_sözlük" adıyla belirttiğimiz sözlük içinde bulunan kelimelerden birini yazarsa, o kelimenin karşılığını alacaktır. Diyelim ki kullanıcımız soruya "dil" diye cevap verdi. Bu durumda ekrana "dil" kelimesinin sözlükteki karşılığı olan "language" yazdırılacaktır. Peki ya kullanıcı sözlükte tanımlı olmayan bir kelime yazarsa ne olacak? Öyle bir durumda programımız hata verecektir. Programımız için doğru yol, hata vermektense, kullanıcıyı kelimenin sözlükte olmadığı konusunda bilgilendirmektir. Bunu klasik bir yaklaşımla şu şekilde yapabiliriz:

```
ing_sözlük = {"dil": "language", "bilgisayar": "computer", "masa": "table"}

sorgu = input("Lütfen anlamını öğrenmek istediğiniz kelimeyi yazınız:")

if sorgu not in ing_sözlük:
    print("Bu kelime veritabanımızda yoktur!")

else:
    print(ing_sözlük[sorgu])
```

Ama açıkçası bu pek verimli bir yaklaşım sayılmaz. Yukarıdaki yöntem yerine sözlüklerin get()

metodundan faydalanabiliriz. Bakalım bunu nasıl yapıyoruz:

```
ing_sözlük = {"dil": "language", "bilgisayar": "computer", "masa": "table"}

sorgu = input("Lütfen anlamını öğrenmek istediğiniz kelimeyi yazınız:")

print(ing_sözlük.get(sorgu, "Bu kelime veritabanımızda yoktur!"))
```

Gördüğünüz gibi, burada çok basit bir metot yardımıyla bütün dertlerimizi hallettik. Sözlüklerin get() adlı metodu, parantez içinde iki adet argüman alır. Birinci argüman sorgulamak istediğimiz sözlük öğesidir. İkinci argüman ise bu öğenin sözlükte bulunmadığı durumda kullanıcıya hangi mesajın gösterileceğini belirtir. Buna göre, yukarıda yaptığımız şey, önce "sorgu" değişkenini sözlükte aramak, eğer bu öğe sözlükte bulunamıyorsa da kullanıcıya, "Bu kelime veritabanımızda yoktur!" cümlesini göstermekten ibarettir...

Gelin isterseniz bununla ilgili bir örnek daha yapalım.

Diyelim ki bir havadurumu programı yazmak istiyoruz. Bu programda kullanıcı bir şehir adı girecek. Program da girilen şehre ait havadurumu bilgilerini ekrana yazdıracak. Bu programı klasik yöntemle şu şekilde yazabiliriz:

```
#!/usr/bin/env python3

soru = input("Şehrinizin adını tamamı küçük harf olacak şekilde yazın:")

if soru == "istanbul":
    print("gök gürültülü ve sağanak yağışlı")

elif soru == "ankara":
    print("açık ve güneşli")

elif soru == "izmir":
    print("bulutlu")

else:
    print("Bu şehre ilişkin havadurumu bilgisi bulunmamaktadır.")
```

Yukarıdaki, gayet geçerli bir yöntemdir. Ama biz istersek bu kodları "get" metodu yardımıyla çok daha verimli ve sade bir hale getirebiliriz:

```
#!/usr/bin/env python3

soru = input("Şehrinizin adını tamamı küçük harf olacak şekilde yazın:")

cevap = {"istanbul": "gök gürültülü ve sağanak yağışlı",
```

```
        "ankara": "açık ve güneşli", "izmir": "bulutlu"}

print(cevap.get(soru, "Bu şehre ilişkin havadurumu bilgisi bulunmamaktadır."))
```

clear()

Sözlüklerin, inceleyeceğimiz ilk metodu clear(). Bu kelime İngilizce'de "temizlemek" anlamına gelir. Görevi sözlükteki öğeleri temizlemektir. Yani içi dolu bir sözlüğü bu metot yardımıyla tamamen boşaltabiliriz:

```
   >>> lig = {"şampiyon": "Adana Demirspor", "ikinci": "Mersin İdman Yurdu",
... "üçüncü": "Adana Gençlerbirliği"}
```

İsterseniz sözlüğümüzü boşaltmadan önce bu sözlükle biraz çalışalım:

Sözlüğümüzün öğelerine şöyle ulaşıyoruz:

```
  >>> lig

  {'şampiyon': 'Adana Demirspor', 'ikinci': 'Mersin İdman Yurdu',
'üçüncü': 'Adana Gençlerbirliği'}
```

Eğer bu sözlüğün öğelerine tek tek erişmek istersek şöyle yapıyoruz:

```
>>> lig["şampiyon"]

'Adana Demirspor'

>>> lig["üçüncü"]

'Adana Gençlerbirliği'
```

Şimdi geldi bu sözlüğün bütün öğelerini silmeye:

```
>>> lig.clear()
```

Şimdi sözlüğümüzün durumunu tekrar kontrol edelim:

```
>>> lig

{}
```

Gördüğünüz gibi artık "lig" adlı sözlüğümüz bomboş. clear() metodunu kullanarak bu sözlüğün bütün öğelerini sildik. Ama tabii ki bu şekilde sözlüğü silmiş olmadık. Boş da olsa bellekte hâlâ "lig" adlı bir sözlük duruyor. Eğer siz "lig"i ortadan kaldırmak isterseniz "del" adlı bir parçacıktan

yararlanmanız gerekir:

```
>>> del lig
```

Kontrol edelim:

```
>>> lig

NameError: name 'lig' is not defined
```

Gördüğünüz gibi artık "lig" diye bir şey yok... Bu sözlüğü bellekten tamamen kaldırdık.

clear() adlı metodun ne olduğunu ve ne işe yaradığını gördüğümüze göre başka bir metoda geçebiliriz.

copy()

Diyelim ki elimizde şöyle bir sözlük var:

```
>>> hava_durumu = {"İstanbul": "yağmurlu", "Adana": "güneşli", ... "İzmir": "bulutlu"}
```

Biz bu sözlüğü kopyalamak istiyoruz. Hemen şöyle bir şey deneyelim:

```
>>> yedek_hava_durumu = hava_durumu
```

Artık elimizde aynı sözlükten iki tane var:

```
>>> hava_durumu

{'İstanbul': 'yağmurlu', 'Adana': 'güneşli', 'İzmir': 'bulutlu'}

>>> yedek_hava_durumu

{'İstanbul': 'yağmurlu', 'Adana': 'güneşli', 'İzmir': 'bulutlu'}
```

Şimdi hava_durumu adlı sözlüğe bir öğe ekleyelim:

```
>>> hava_durumu["Mersin"] = "sisli"

>>> hava_durumu

{'İstanbul': 'yağmurlu', 'Adana': 'güneşli', 'Mersin': 'sisli', 'İzmir': 'bulutlu'}
```

Şimdi bir de yedek_hava_durumu adlı sözlüğün durumuna bakalım:

```
>>> yedek_hava_durumu

{'İstanbul': 'yağmurlu', 'Adana': 'güneşli', 'Mersin': 'sisli', 'İzmir': 'bulutlu'}
```

Gördüğünüz gibi, hava_durumu adlı sözlüğe yaptığımız ekleme yedek_hava_durumu adlı sözlüğü de etkiledi. Hatırlarsanız buna benzer bir durumla daha önce listeleri anlatırken de karşılaşmıştık. Çünkü varolan bir sözlüğü veya listeyi başka bir değişkene atadığımızda aslında yaptığımız şey bir kopyalama işleminden ziyade bellekteki aynı nesneye gönderme yapan iki farklı isim belirlemekten ibaret. Yani sözlüğümüzü bellekteki bir nesne olarak düşünürsek, bu nesneye atıfta bulunan, "hava_durumu" ve "yedek_hava_durumu" adlı iki farklı isim belirlemiş oluyoruz. Eğer istediğimiz şey bellekteki nesneden iki adet oluşturmak ve bu iki farklı nesneyi iki farklı isimle adlandırmak ise yukarıdaki yöntemi kullanmak istemediğiniz sonuçlar doğurabilir. Yani amacınız bir sözlüğü yedekleyip orijinal sözlüğü korumaksa ve yukarıdaki yöntemi kullandıysanız, hiç farkında olmadan orijinal sözlüğü de değiştirebilirsiniz. İşte böyle durumlarda imdadımıza sözlüklerin "copy" metodu yetişecek. Bu metodu kullanarak varolan bir sözlüğü gerçek anlamda kopyalayabilir, yani yedekleyebiliriz...

```
>>> hava_durumu = {"İstanbul": "yağmurlu", "Adana": "güneşli", ... "İzmir": "bulutlu"}
```

Şimdi bu sözlüğü yedekliyoruz. Yani kopyalıyoruz:

```
>>> yedek_hava_durumu = hava_durumu.copy()
```

Bakalım hava_durumu adlı sözlüğe ekleme yapınca yedek_hava_durumu adlı sözlüğün durumu ne oluyor?

```
>>> hava_durumu["Mersin"] = "sisli"

>>> hava_durumu

{'İstanbul': 'yağmurlu', 'Adana': 'güneşli', 'Mersin': 'sisli', 'İzmir':
'bulutlu'}
```

yedek_hava_durumu adlı sözlüğe bakalım:

```
>>> yedek_hava_durumu

{'İstanbul': 'yağmurlu', 'Adana': 'güneşli', 'İzmir': 'bulutlu'}
```

Gördüğünüz gibi bu defa sözlüklerin birinde yapılan değişiklik öbürünü etkilemedi... copy metodu sağolsun!...

fromkeys()

fromkeys() metodu öteki metotlardan biraz farklıdır. Bu metot mevcut sözlük üzerinde işlem

yapmaz. fromkeys()'in görevi yeni bir sözlük oluşturmaktır. Bu metot yeni bir sözlük oluştururken listeler veya demetlerden yararlanır. Şöyle ki:

```
>>> elemanlar = "Ahmet", "Mehmet", "Can"

>>> adresler = dict.fromkeys(elemanlar, "Kadıköy")

>>> adresler

{'Ahmet': 'Kadıköy', 'Mehmet': 'Kadıköy', 'Can': 'Kadıköy'}
```

Gördüğünüz gibi öncelikle "elemanlar" adlı bir demet tanımladık. Daha sonra da "adresler" adlı bir sözlük tanımlayarak, fromkeys() metodu yardımıyla anahtar olarak "elemanlar" demetindeki öğelerden oluşan, değer olarak ise "Kadıköy"ü içeren bir sözlük meydana getirdik.

En başta tanımladığımız "elemanlar" demeti liste de olabilirdi. Hatta tek başına bir karakter dizisi dahi yazabilirdik oraya...

pop()

Bu metodu listelerden hatırlıyoruz. Bu metot listelerle birlikte kullanıldığında, listenin en son öğesini silip, silinen öğeyi de ekrana basıyordu. Eğer bu metodu bir sıra numarası ile birlikte kullanırsak, listede o sıra numarasına karşılık gelen öğe siliniyor ve silinen bu öğe ekrana basılıyordu. Bu metodun sözlüklerdeki kullanımı da az çok buna benzer. Ama burada farkı olarak, pop metodunu argümansız bir şekilde kullanamıyoruz. Yani pop metodunun parantezi içinde mutlaka bir sözlük öğesi belirtmeliyiz:

```
>>> sepet = {"meyveler": ("elma", "armut"), "sebzeler": ("pırasa", "fasulye"),
... "içecekler": ("su", "kola", "ayran")}

>>> sepet.pop("meyveler")
```

Bu komut, sözlükteki "meyveler" anahtarını silecek ve sildiği bu öğenin değerini ekrana basacaktır. Eğer silmeye çalıştığımız anahtar sözlükte yoksa Python bize bir hata mesajı gösterecektir:

```
>>> sepet.pop("tatlılar")

KeyError: 'tatlilar'
```

Bir program yazarken böyle bir durumla karşılaşmak istemeyiz çoğu zaman. Yani bir sözlük içinde arama yaparken, aranan öğenin sözlükte bulunmadığı bir durumda kullanıcıya mekanik ve anlamsız bir hata göstermek yerine, daha anlaşılır bir mesaj iletmeyi tercih edebiliriz. Hatırlarsanız sözlüklerin get() metodunu kullanarak benzer bir şey yapabiliyorduk. Şu anda incelemekte olduğumuz pop() metodu da bize böyle bir imkan verir. Bakalım:

```
>>> sepet.pop("tatlılar", "Silinecek öğe yok!")
```

Böylelikle sözlükte bulunmayan bir öğeyi silmeye çalıştığımızda Python bize bir hata mesajı göstermek yerine, "Silinecek öğe yok!" şeklinde daha anlamlı bir mesaj verecektir...

popitem()

popitem() metodu da bir önceki bölümde öğrendiğimiz pop() metoduna benzer. Bu iki metodun görevleri hemen hemen aynıdır. Ancak pop() metodu parantez içinde bir parametre alırken, popitem() metodunun parantezi boş, yani parametresiz olarak kullanılır. Bu metot bir sözlükten rastgele öğeler silmek için kullanılır. Daha önce de pek çok kez söylediğimiz gibi, sözlükler sırasız veri tipleridir. Dolayısıyla popitem() metodunun öğeleri silerken kullanabileceği bir sıra kavramı yoktur. Bu yüzden bu metot öğeleri rastgele silmeyi tercih eder...

```
>>> sepet = {"meyveler": ("elma", "armut"), "sebzeler": ("pırasa", "fasulye")}

>>> sepet.popitem()
```

Bu komut sözlükten rastgele bir anahtarı, değerleriyle birlikte sözlükten silecektir. Eğer sözlük boşsa bu metot bize bir hata mesajı gösterir.

setdefault()

Bu metot epey enteresan, ama bir o kadar da yararlı bir araçtır... Bu metodun ne işe yaradığını doğrudan bir örnek üzerinde görelim:

```
>>> sepet = {"meyveler": ("elma", "armut"), "sebzeler": ("pırasa", "fasulye")}

>>> sepet.setdefault("içecekler", ("su", "kola"))
```

Bu komut yardımıyla sözlüğümüz içinde "içecekler" adlı bir anahtar oluşturduk. Bu anahtarın değeri ise *("su", "kola")* oldu... Bir de şuna bakalım:

```
>>> sepet.setdefault("meyveler", ("erik", "çilek"))

('elma', 'armut')
```

Gördüğünüz gibi, sözlükte zaten "meyveler" adlı bir anahtar bulunduğu için, Python aynı adı taşıyan ama değerleri farklı olan yeni bir "meyveler" anahtarı oluşturmadı. Demek ki bu metot yardımıyla bir sözlük içinde arama yapabiliyor, eğer aradığımız anahtar sözlükte yoksa, setdefault() metodu içinde belirttiğimiz özellikleri taşıyan yeni bir anahtar-değer çifti oluşturabiliyoruz.

update()

İnceleyeceğimiz son metot update() metodu... Bu metot yardımıyla oluşturduğumuz sözlükleri yeni verilerle güncelleyeceğiz. Diyelim ki elimizde şöyle bir sözlük var:

```
>>> stok = {"elma": 5, "armut": 10, "peynir": 6, "sosis": 15}
```

Stoğumuzda 5 adet elma, 10 adet armut, 6 kutu peynir, 15 adet de sosis var. Diyelim ki daha sonraki zamanlarda bu stoğa mal giriş-çıkışı oldu ve stoğun son hali şöyle:

```
>>> yeni_stok = {"elma": 3, "armut": 20, "peynir": 8, "sosis": 4, "sucuk": 6}
```

Yapmamız gereken şey, stoğumuzu yeni bilgilere göre güncellemek olacaktır. İşte bu işlemi update() metodu ile yapabiliriz:

```
>>> stok.update(yeni_stok)

>>> print(stok)

{'peynir': 8, 'elma': 3, 'sucuk': 6, 'sosis': 4, 'armut': 20}
```

Böylelikle malların son miktarlarına göre stok bilgilerimizi güncellemiş olduk...

başa dön

Kümeler ve Dondurulmuş Kümeler

Bu bölümde Python'daki iki veri tipini daha inceleyeceğiz. İnceleyeceğimiz veri tiplerinin adı küme ve dondurulmuş küme.

Özellikle kümeleri öğrendiğimizde, bu veri tipinin kendine has birtakım özellikleri sayesinde bunların kimi zaman hiç tahmin bile edemeyeceğimiz yerlerde işimize yaradığını göreceğiz. Normalde uzun uzun kod yazmayı gerektiren durumlarda kümeleri kullanmak, bir-iki satırla işlerimizi halletmemizi sağlayabilir.

Bu bölümde kümeler dışında, bir de dondurulmuş kümelerden söz edeceğiz. Bu iki veri tipi birbiriyle ilişkilidir. O yüzden bu iki veri tipini tek bölümde ele alacağız.

İsterseniz anlatmaya önce kümelerle başlayalım.

Kümeler

Tıpkı listeler, demetler, karakter dizileri, sayılar ve dosyalar gibi kümeler de Python'daki veri tiplerinden biridir. Adından da az çok tahmin edebileceğiniz gibi kümeler, matematikten bildiğimiz "küme" kavramıyla sıkı sıkıya bağlantılıdır. Bu veri tipi, matematikteki kümelerin sahip olduğu bütün özellikleri taşır. Yani matematikteki kümelerden bildiğimiz kesişim, birleşim ve fark gibi özellikler Python'daki kümeler için de geçerlidir.

Küme Oluşturmak

Kümelerin bize sunduklarından faydalanabilmek için elbette öncelikle bir küme oluşturmamız gerekiyor. Küme oluşturmak çok kolay bir işlemdir. Örneğin boş bir kümeyi şöyle oluşturuyoruz:

```
>>> boş_küme = set()
```

Listeler, demetler ve sözlüklerin aksine kümelerin ayırt edici bir işareti yoktur. Küme oluşturmak için *set()* adlı özel bir fonksiyondan yararlanıyoruz.

Yukarıdaki boş veri tipinin bir küme olduğunu nasıl teyit edeceğinizi biliyorsunuz:

```
>>> type(boş_küme)

<class 'set'>
```

Gördüğünüz gibi, Python programlama dilinde kümeler *set* ifadesiyle gösteriliyor.

Yukarıda boş bir küme oluşturduk. İçinde öğe de barındıran kümeleri ise şu şekilde oluşturuyoruz:

```
>>> küme = set(["elma", "armut", "kebap"])
```

Böylelikle, içinde öğe barındıran ilk kümemizi başarıyla oluşturduk. Dikkat ederseniz, küme oluştururken listelerden faydalandık. Gördüğünüz gibi *set()* fonksiyonu içindeki öğeler bir liste içinde yer alıyor. Dolayısıyla yukarıdaki tanımlamayı şöyle de yapabiliriz:

```
>>> liste = ["elma", "armut", "kebap"]

>>> küme = set(liste)
```

Bu daha temiz bir görüntü oldu. Elbette küme tanımlamak için mutlaka liste kullanmak zorunda değiliz. İstersek demetleri de küme haline getirebiliriz:

```
>>> demet = ("elma", "armut", "kebap")

>>> küme = set(demet)
```

Hatta ve hatta karakter dizilerinden dahi küme yapabiliriz:

```
>>> kardiz = "Python Programlama Dili için Türkçe Kaynak"

>>> küme = set(kardiz)
```

Kullandığımız karakter dizisinin böyle uzun olmasına da gerek yok. Tek karakterlik dizilerden bile küme oluşturabiliriz:

```
>>> kardiz = "a"

>>> küme = set(kardiz)
```

Ama sayılardan küme oluşturamayız:

```
>>> n = 10

>>> küme = set(n)

TypeError: 'int' object is not iterable
```

Peki sözlükleri kullanarak küme oluşturabilir miyiz? Elbette, neden olmasın?

```
>>> bilgi = {"işletim sistemi": "GNU", "sistem çekirdeği": "Linux",
... "dağıtım": "Ubuntu GNU/Linux"}

>>> küme = set(bilgi)
```

Küme oluşturmanın son bir yönteminden daha söz edelim. En başta söylediğimiz gibi, listeler, demetler, sözlükler ve karakter dizilerinin aksine kümelerin *[ ]*, *( )*, *{ }*, *‘ ‘* gibi ayırt edici bir işareti yoktur. Ama eğer istersek sözlükleri oluşturmak için kullandığımız özel işaretleri küme oluşturmak

için de kullanabiliriz. Dikkatlice bakın:

```
>>> küme = {'Python', 'C++', 'Ruby', 'PHP'}
```

Gördüğünüz gibi, aslında sözlüklerin ayırt edici işareti olan süslü parantezleri kullanarak ve öğeleri birbirinden virgülle ayırarak da küme adlı veri tipini elde edebiliyoruz. Teyit edelim bunu:

```
>>> type(küme)

<class 'set'>
```

Ancak bu yapıyı kullanarak boş bir küme oluşturamazsınız:

```
>>> küme = {}
```

Bu şekilde oluşturduğunuz şey bir küme değil, sözlük olacaktır:

```
>>> type(küme)

<class 'dict'>
```

Boş bir küme oluşturmak için set() fonksiyonunu kullanmanız gerektiğini biliyorsunuz:

```
>>> küme = set(küme)
>>> type(küme)

<class 'set'>
```

Böylece kümeleri nasıl oluşturacağımızı öğrendik. Eğer oluşturduğunuz kümeyi ekrana yazdırmak isterseniz, ne yapacağınızı biliyorsunuz. Tanımladığınız küme değişkenini kullanmanız yeterli olacaktır:

```
>>> küme

{'işletim sistemi', 'sistem çekirdeği', 'dağıtım'}
```

Bu arada, bir sözlüğü kümeye çevirdiğinizde, elbette sözlüğün yalnızca anahtarları kümeye eklenecektir. Sözlüğün değerleri ise böyle bir işlemin sonucunda ortadan kaybolur.

Eğer bir sözlüğü kümeye çevirirken hem anahtarları hem de değerleri korumak gibi bir niyetiniz varsa şöyle bir şey yazabilirsiniz:

Sözlüğümüz şu:

```
>>> bilgi = {"işletim sistemi": "GNU", "sistem çekirdeği": "Linux",
... "dağıtım": "Ubuntu GNU/Linux"}
```

Bu sözlükteki anahtar-değer çiftlerini bir küme içine, çift öğeli demetler olarak yerleştirebiliriz:

```
>>> liste = [(anahtar, değer) for anahtar, değer in bilgi.items()]
>>> küme = set(liste)
```

Gördüğünüz gibi, liste üreteçlerini kullanarak önce bir liste oluşturuyoruz. Bu liste her bir anahtarı ve değeri tek tek bir demet içine yerleştiriyor. Daha sonra da bu listeyi set() fonksiyonuna göndererek kümemizi oluşturuyoruz.

Kümelerin Yapısı

Bir önceki başlık altında kümelerin nasıl tanımlanacağını inceledik. Gelin şimdi de biraz kümelerin yapısından bahsedelim.

Örneğin şöyle bir küme tanımlayalım:

```
>>> kardiz = "Python Programlama Dili"

>>> küme = set(kardiz)

>>> print(küme)

{'g', 'D', 'a', ' ', 'o', 'n', 'm', 'l', 'i', 'h', 't', 'r', 'P', 'y'}
```

Burada bir şey dikkatinizi çekmiş olmalı. Bir öğeyi küme olarak tanımlayıp ekrana yazdırdığımızda elde ettiğimiz çıktı, o öğe içindeki her bir alt öğeyi tek bir kez içeriyor. Yani mesela “Python Programlama Dili” içinde iki adet “P” karakteri var, ama çıktıda bu iki “P” karakterinin yalnızca biri görünüyor. Buradan anlıyoruz ki, kümeler aynı öğeyi birden fazla tekrar etmez. Bu çok önemli bir özelliktir ve pek çok yerde işimize yarar. Aynı durum karakter dizisi dışında kalan öteki veri tipleri için de geçerlidir. Yani mesela eğer bir listeyi küme haline getiriyorsak, o listedeki öğeler küme içinde yalnızca bir kez geçecektir. Listede aynı öğeden iki-üç tane bulunsa bile, kümemiz bu öğeleri teke indirecektir.

```
>>> liste = ["elma", "armut", "elma", "kebap", "şeker", "armut",
... "çilek", "ağaç", "şeker", "kebap", "şeker"]

>>> for i in set(liste):
...     print(i)
...
ağaç
elma
şeker
kebap
çilek
armut
```

Gördüğünüz gibi, liste içinde birden fazla bulunan öğeler, Python'daki kümeler yardımıyla teke indirilebiliyor.

Öğrendiğimiz bu bilgi sayesinde, daha önce gördüğümüz *count()* metodunu da kullanarak, şöyle bir kod yazabiliriz:

```
>>> liste = ["elma", "armut", "elma", "kiraz",
... "çilek", "kiraz", "elma", "kebap"]


>>> for i in set(liste):
...     print("{} listede {} kez geçiyor!".format(i, liste.count(i)))

kebap listede 1 kez geçiyor!
elma listede 3 kez geçiyor!
kiraz listede 2 kez geçiyor!
armut listede 1 kez geçiyor!
çilek listede 1 kez geçiyor!
```

Burada set(liste) ifadesini kullanarak, liste öğelerini eşşiz ve benzersiz bir hale getirdik.

Kümelerin önemli bir özelliği de, tıpkı sözlükler gibi, herhangi bir şekilde 'öğe sırası' kavramına sahip olmamasıdır.

Dikkatlice bakın:

```
>>> arayüz_takımları = {'Tkinter', 'PyQT', 'PyGobject'}
>>> arayüz_takımları

{'PyGobject', 'PyQT', 'Tkinter'}
```

Sözlüklerde karşılaştığımız durumun aynısının kümeler için de geçerli olduğuna dikkatinizi çekmek isterim. Gördüğünüz gibi, *arayüz_takımları* adlı kümenin öğeleri, öğe tanımlama sırasını çıktıda korumuyor. Biz 'Tkinter' öğesini kümenin ilk sırasına yerleştirmiştik, ama bu öğe çıktıda en sona gitti... Aynen sözlüklerde olduğu gibi, kümelerde de öğelerin tanımlanma sırasına bel bağlayarak herhangi bir işlem yapamazsınız. Bu durumun bir yansıması olarak, küme öğelerine sıralarına göre de erişemezsiniz:

```
>>> arayüz_takımları[0]

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
```

Tıpkı hata mesajında da söylendiği gibi, küme adlı veri tipi açısından öğe sırası diye bir kavram

yoktur...

Esasında tek bir küme pek bir işe yaramaz. Kümeler ancak birden fazla olduğunda bunlarla yararlı işler yapabiliriz. Çünkü kümelerin en önemli özelliği, başka kümelerle karşılaştırılabilme kabiliyetidir. Yani mesela kümelerin kesişimini, birleşimini veya farkını bulabilmek için öncelikle elimizde birden fazla küme olması gerekiyor. İşte biz de şimdi bu tür işlemleri nasıl yapacağımızı öğreneceğiz. O halde hiç vakit kaybetmeden yolumuza devam edelim.

Küme Üreteçleri (*Set Comprehensions*)

Bildiğiniz gibi liste üreteçleri, liste oluşturmanın kısa ve temiz bir yoludur. Aynı şekilde sözlük üreteçleri de sözlük oluşturmanın kısa ve temiz bir yoludur.

İşte liste üreteçlerini ve sözlük üreteçlerini kullanarak nasıl tek satırda ve hızlı bir şekilde listeler ve sözlükler üretebiliyorsak, aynı şekilde küme üreteçlerini kullanarak tek satırda ve hızlı bir şekilde kümeler de üretebiliriz.

Örneğin elimizde şöyle bir liste olduğunu düşünelim:

```
import random

liste = [random.randint(0, 10000) for i in range(1000)]
```

Bu arada, buradaki *random* adlı modüle şimdilik takılmayın. Birkaç bölüm sonra bu modülü inceleyeceğiz. Biz şimdilik *random*'un da tıpkı *sys* ve *os* gibi bir modül olduğunu ve rastgele sayılar üretmemizi sağladığını bilelim yeter. Yukarıdaki kodlarda da bu modül *0* ile *10000* arasında rstgele *1000* adet sayı üretmemizi sağladı.

Şimdi amacımız bu liste içinde yer alan sayılardan, değeri *100*'den küçük olanları bulmak.

Bunun için şu kodları kullanabiliriz:

```
import random

liste = [random.randint(0, 10000) for i in range(1000)]

yüzden_küçük_sayılar = [i for i in liste if i < 100]
print(yüzden_küçük_sayılar)
```

Ancak ortaya çıkan listede aynı sayılardan birkaç tane olabilir. İşte eğer birbirinin aynı olmayan sayılardan oluşmuş bir listeyi hızlı ve pratik bir şekilde elde etmek istiyorsanız küme üreteçlerini kullanabilirsiniz:

```
import random

liste = [random.randint(0, 10000) for i in range(1000)]
```

```
küme = {i for i in liste if i < 100}
print(küme)
```

Gördüğünüz gibi, küme üreteçlerinin sözdizimi, liste ve sözlük üreteçlerinin sözdizimine çok benziyor.

Kümelerin Metotları

Daha önceki veri tiplerinde olduğu gibi, kümelerin de metotları vardır. Artık biz bir veri tipinin metotlarını nasıl listeleyeceğimizi çok iyi biliyoruz. Nasıl liste için list(); demet için tuple(); sözlük için de dict() fonksiyonlarını kullanıyorsak, kümeler için de set() adlı fonksiyondan yararlanacağız:

```
>>> dir(set)

['__and__', '__class__', '__contains__', '__delattr__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__iand__', '__init__', '__ior__', '__isub__', '__iter__',
'__ixor__', '__le__', '__len__', '__lt__', '__ne__', '__new__',
'__or__', '__rand__', '__reduce__', '__reduce_ex__', '__repr__',
'__ror__', '__rsub__', '__rxor__', '__setattr__', '__sizeof__',
'__str__', '__sub__', '__subclasshook__', '__xor__', 'add',
'clear','copy', 'difference', 'difference_update', 'discard',
'intersection', 'intersection_update', 'isdisjoint', 'issubset',
'issuperset', 'pop', 'remove', 'symmetric_difference',
'symmetric_difference_update', 'union', 'update']
```

Hemen işimize yarayacak metotları alalım:

```
>>> for i in dir(set):
...     if "__" not in i:
...         print(i)
...
add
clear
copy
difference
difference_update
discard
intersection
intersection_update
isdisjoint
issubset
issuperset
```

```
pop
remove
symmetric_difference
symmetric_difference_update
union
update
```

Gördüğünüz gibi kümelerin epey metodu var. Bu arada `if "__" not in i` satırında "_" yerine "__" kullandığımıza dikkat edin. Neden? Çünkü eğer sadece "_" kullanırsak *symmetric_difference* ve *symmetric_difference_update* metotları çıktımızda yer almayacaktır.

Unutmadan söyleyelim: Kümeler de, tıpkı listeler ve sözlükler gibi, değiştirilebilir bir veri tipidir.

clear()

Kümelerle ilgili olarak inceleyeceğimiz ilk metot *clear()*. Bu metodu daha önce sözlükleri çalışırken de görmüştük. Sözlüklerde bu metodun görevi sözlüğün içini boşaltmak idi. Burada da aynı vazifeyi görür:

```
>>> km = set("adana")

>>> for i in km:
...     print(i)
...
a
d
n

>>> km.clear()

>>> km
set([])
```

Burada önce "km" adlı bir küme oluşturduk. Daha sonra da *clear()* metodunu kullanarak bu kümenin bütün öğelerini sildik. Artık elimizde boş bir küme var.

copy()

Listeler ve sözlükleri incelerken `copy()` adlı bir metot öğrenmiştik. Bu metot aynı zamanda kümelerle birlikte de kullanılabilir. Üstelik işlevi de aynıdır:

```
>>> km = set("kahramanmaraş")

>>> yedek = km.copy()
```

```
>>> yedek

set(['a', 'r', 'h', 'k', 'm', 'ş', 'n'])

>>> km

set(['a', 'h', 'k', 'm', 'n', 'r', 'ş'])
```

Burada bir şey dikkatinizi çekmiş olmalı. "km" adlı kümeyi "yedek" adıyla kopyaladık, ama bu iki kümenin çıktılarına baktığımız zaman öğe sıralamasının birbirinden farklı olduğunu görüyoruz. Biliyorsunuz, tıpkı sözlüklerde olduğu gibi, kümeler de sırasız veri tipleridir. Bu yüzden, elde ettiğimiz çıktıda öğeler rastgele diziliyor. Dolayısıyla öğelere sıralarına göre erişemiyoruz. Aynen sözlüklerde olduğu gibi...

add()

Kümelerden bahsederken, bunların değiştirilebilir bir veri tipi olduğunu söylemiştik. Dolayısıyla kümeler, üzerlerinde değişiklik yapmamıza müsaade eden metotlar da içerir. Örneğin *add()* bu tür metotlardan biridir. *Add* kelimesi Türkçe'de "eklemek" anlamına gelir. Adından da anlaşılacağı gibi, bu metot yardımıyla kümelerimize yeni öğeler ilave edebileceğiz. Hemen bunun nasıl kullanıldığına bakalım:

```
>>> küme = set(["elma", "armut", "kebap"])

>>> küme.add("çilek")

>>> print(küme)

set(['elma', 'armut', 'kebap', 'çilek'])
```

Gördüğünüz gibi, *add()* metodunu kullanarak, kümemize *çilek* adlı yeni bir öğe ekledik. Eğer kümede zaten varolan bir öğe eklemeye çalışırsak kümede herhangi bir değişiklik olmayacaktır. Çünkü, daha önce de söylediğimiz gibi, kümeler her bir öğeyi tek bir sayıda barındırır.

Eğer bir kümeye birden fazla öğeyi aynı anda eklemek isterseniz *for* döngüsünden yararlanabilirsiniz:

```
>>> yeni = [1,2,3]

>>> for i in yeni:
...     küme.add(i)
...

>>> küme
```

```
set([1, 2, 3, 'elma', 'kebap', 'çilek', 'armut'])
```

Burada yeni adlı listeyi kümeye *for* döngüsü ile ekledik. Ama bu işlemi yapmanın başka bir yolu daha vardır. Bu işlem için Python'da ayrı bir metot bulunur. Bu metodun adı *update()* metodudur. Sırası gelince bu metodu da göreceğiz.

Bu arada, yeri gelmişken kümelerin önemli bir özelliğinden daha söz edelim. Bir kümeye herhangi bir öğe ekleyebilmemiz için, o öğenin değiştirilemeyen (*immutable*) bir veri tipi olması gerekiyor. Bildiğiniz gibi Python'daki şu veri tipleri değiştirilemeyen veri tipleridir:

1. Demetler
2. Sayılar
3. Karakter Dizileri

Şu veri tipleri ise değiştirilebilen veri tipleridir:

1. Listeler
2. Sözlükler
3. Kümeler

Dolayısıyla bir kümeye ancak şu veri tiplerini ekleyebiliriz:

1. Demetler
2. Sayılar
3. Karakter Dizileri

Şu kodları dikkatlice inceleyin:

Önce boş bir küme oluşturalım:

```
>>> küme = set()
```

Bu kümeye bir demet ekleyelim:

```
>>> l = (1,2,3)
>>> küme.add(l)
>>> küme

{(1, 2, 3)}
```

Bir sayı ekleyelim:

```
>>> l = 45
>>> küme.add(l)
>>> küme
```

```
{45, (1, 2, 3)}
```

Bir karakter dizisi ekleyelim:

```
>>> l = 'Jacques Derrida'
>>> küme.add(l)
>>> küme

{'Jacques Derrida', 45, (1, 2, 3)}
```

Yukarıdakiler, değiştirilemeyen veri tipleri olduğu için kümelere eklenebilir.

Bir de şunlara bakalım:

Kümemize bir liste eklemeye çalışalım:

```
>>> l = [1,2,3]
>>> küme.add(l)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Kümemize bir sözlük eklemeye çalışalım:

```
>>> l = {"a": 1, "b": 2, "c": 3}
>>> küme.add(l)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'
```

Kümemize bir küme eklemeye çalışalım:

```
>>> l = {1, 2, 3}
>>> küme.add(l)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
```

Gördüğünüz gibi, tıpkı sözlüklerde olduğu gibi, bir kümeye herhangi bir veri ekleyebilmemiz için o verinin ‘değiştirilemeyen’ bir veri tipi olması gerekiyor.

difference()

Bu metot iki kümenin farkını almamızı sağlar. Örneğin:

```
>>> k1 = set([1, 2, 3, 5])
>>> k2 = set([3, 4, 2, 10])

>>> k1.difference(k2)

set([1, 5])
```

Demek ki k1'in k2'den farkı buymuş. Peki k2'nin k1'den farkını bulmak istersek ne yapacağız?

```
>>> k2.difference(k1)

set([10, 4])
```

Gördüğünüz gibi, birinci kullanımda, k1'de bulunup k2'de bulunmayan öğeleri elde ediyoruz. İkinci kullanımda ise bunun tam tersi. Yani ikinci kullanımda k2'de bulunup k1'de bulunmayan öğeleri alıyoruz.

İsterseniz uzun uzun *difference()* metodunu kullanmak yerine sadece eksi (-) işaretini kullanarak da aynı sonucu elde edebilirsiniz:

```
>>> k1 - k2
```

...veya...

```
>>> k2 - k1
```

Hayır, "*madem eksi işaretini kullanabiliyoruz, o halde artı işaretini de kullanabiliriz!*" gibi bir fikir doğru değildir.

difference_update()

Bu metot, *difference()* metodundan elde edilen sonuca göre bir kümenin güncellenmesini sağlar. Yani? Hemen bir örnek verelim:

```
>>> k1 = set([1, 2, 3])
>>> k2 = set([1, 3, 5])

>>> k1.difference_update(k2)

set([2])
```

```
>>> print(k1)

set([2])

>>> print(k2)

set([1, 3, 5])
```

Gördüğünüz gibi, bu metot k1'in k2'den farkını aldı ve bu farkı kullanarak k1'i yeniden oluşturdu. k1 ile k2 arasındaki tek fark *2* adlı öğe idi. Dolayısıyla *difference_update()* metodunu uyguladığımızda k1'in öğelerinin silinip yerlerine *2* adlı öğenin geldiğini görüyoruz.

discard()

Bir önceki bölümde öğrendiğimiz *add()* metodu yardımıyla, önceden oluşturduğumuz bir kümeye yeni öğeler ekleyebiliyorduk. Bu bölümde öğreneceğimiz *discard()* metodu ise kümeden öğe silmemizi sağlayacak:

```
>>> hayvanlar = set(["kedi", "köpek", "at", "kuş", "inek", "deve"])

>>> hayvanlar.discard("kedi")

>>> print(hayvanlar)

set(['kuş', 'inek', 'deve', 'köpek', 'at'])
```

Eğer küme içinde bulunmayan bir öğe silmeye çalışırsak hiç bir şey olmaz. Yani hata mesajı almayız:

```
>>> hayvanlar.discard("yılan")
```

Burada etkileşimli kabuk sessizce bir alt satıra geçecektir. Bu metodun en önemli özelliği budur. Yani olmayan bir öğeyi silmeye çalıştığımızda hata vermemesi.

remove()

Bu metot da bir önceki bölümde gördüğümüz *discard()* metoduyla aynı işlevi yerine getirir. Eğer bir kümeden öğe silmek istersek *remove()* metodunu da kullanabiliriz:

```
>>> hayvanlar.remove("köpek")
```

Peki *discard()* varken *remove()* metoduna ne gerek var? Ya da tersi.

Bu iki metot aynı işlevi yerine getirse de aralarında önemli bir fark vardır. Hatırlarsanız *discard()* metoduyla, kümede olmayan bir öğeyi silmeye çalışırsak herhangi bir hata mesajı almayacağımızı söylemiştik. Eğer *remove()* metodunu kullanarak, kümede olmayan bir öğeyi silmeye çalışırsak,

discard() metodunun aksine, hata mesajı alırız:

```
>>> hayvanlar.remove("fare")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'fare'
```

Bu iki metot arasındaki bu fark önemli bir farktır. Bazen yazdığınız programlarda, duruma göre her iki özelliğe de ihtiyacınız olabilir.

intersection()

intersection kelimesi Türkçe'de "kesişim" anlamına gelir. Adından da anladığımız gibi, *intersection()* metodu bize iki kümenin kesişim kümesini verecektir:

```
>>> k1 = set([1, 2, 3, 4])
>>> k2 = set([1, 3, 5, 7])

>>> k1.intersection(k2)

set([1, 3])
```

Gördüğünüz gibi, bu metot bize k1 ve k2'nin kesişim kümesini veriyor. Dolayısıyla bu iki küme arasındaki ortak elemanları bulmuş oluyoruz.

Hatırlarsanız, *difference()* metodunu anlatırken, *difference()* kelimesi yerine "-" işaretini de kullanabileceğimiz, söylemiştik. Benzer bir durum *intersection()* metodu için de geçerlidir. İki kümenin kesişimini bulmak için "&" işaretinden yararlanabiliriz:

```
>>> k1 & k2

set([1, 3])
```

Python programcıları genellikle uzun uzun *intersection* yazmak yerine "&" işaretini kullanırlar...

İsterseniz bu metot için örnek bir program verelim. Böylece gerçek hayatta bu metodu nasıl kullanabileceğimizi görmüş oluruz:

```
tr = "şçöğüıŞÇÖĞÜİ"

parola = input("Sisteme giriş için bir parola belirleyin: ")

if set(tr) & set(parola):
    print("Parolanızda Türkçe harfler kullanmayın!")
```

```
else:
    print("Parolanız kabul edildi!")
```

Burada eğer kullanıcı, parola belirlerken içinde Türkçe bir harf geçen bir kelime yazarsa programımız kendisini Türkçe harf kullanmaması konusunda uyaracaktır. Bu kodlarda kümeleri nasıl kullandığımıza dikkat edin. Programda asıl işi yapan kısım şu satırdır:

```
if set(tr) & set(parola):
    print("Parolanızda Türkçe harfler kullanmayın!")
```

Burada aslında şöyle bir şey demiş oluyoruz:

> *Eğer set(tr) ve set(parola) kümelerinin kesişim kümesi boş değilse, kullanıcıya "Parolanızda Türkçe harfler kullanmayın!" uyarısını göster!*

set(tr) ve set(parola) kümelerinin kesişim kümesinin boş olmaması, kullanıcının girdiği kelime içindeki harflerden en az birinin tr adlı değişken içinde geçtiği anlamına gelir. Burada basitçe, tr değişkeni ile parola değişkeni arasındaki ortak öğeleri sorguluyoruz. Eğer kullanıcı herhangi bir Türkçe harf içermeyen bir kelime girerse set(tr) ve set(parola) kümelerinin kesişim kümesi boş olacaktır. İsterseniz küçük bir deneme yapalım:

```
>>> tr = "şçöğüıŞÇÖĞÜİ"

>>> parola = "çilek"

>>> set(tr) & set(parola)

set(['ç'])
```

Burada kullanıcının "çilek" adlı kelimeyi girdiğini varsayıyoruz. Böyle bir durumda set(tr) ve set(parola) kümelerinin kesişim kümesi "ç" harfini içerecek, dolayısıyla da programımız kullanıcıya uyarı mesajı gösterecektir. Eğer kullanıcımız "kalem" gibi Türkçe harf içermeyen bir kelime girerse:

```
>>> tr = "şçöğüıŞÇÖĞÜİ"

>>> parola = "kalem"

>>> set(tr) & set(parola)

set([])
```

Gördüğünüz gibi, elde ettiğimiz küme boş. Dolayısıyla böyle bir durumda programımız kullanıcıya herhangi bir uyarı mesajı göstermeyecektir.

intersection() metodunu pek çok yerde kullanabilirsiniz. Hatta iki dosya arasındaki benzerlikleri bulmak için dahi bu metottan yararlanabilirsiniz. İlerde dosya işlemleri konusunu işlerken bu metottan nasıl yararlanabileceğimizi de anlatacağız.

intersection_update()

Hatırlarsanız *difference_update()* metodunu işlerken şöyle bir şey demiştik:

Bu metot, difference() metodundan elde edilen sonuca göre bir kümenin güncellenmesini sağlar.

İşte *intersection_update* metodu da buna çok benzer bir işlevi yerine getirir. Bu metodun görevi, *intersection()* metodundan elde edilen sonuca göre bir kümenin güncellenmesini sağlamaktır:

```
>>> k1 = set([1, 2, 3])
>>> k2 = set([1, 3, 5])

>>> k1.intersection_update(k2)

set([1, 3])

>>> print(k1)

set([1, 3])

>>> print(k2)

set([1, 3, 5])
```

Gördüğünüz gibi, *intersection_update()* metodu k1'in bütün öğelerini sildi ve yerlerine k1 ve k2'nin kesişim kümesinin elemanlarını koydu.

isdisjoint()

Bu metodun çok basit bir görevi vardır. *isdisjoint()* metodunu kullanarak iki kümenin kesişim kümesinin boş olup olmadığı sorgulayabilirsiniz. Hatırlarsanız aynı işi bir önceki bölümde gördüğümüz *intersection()* metodunu kullanarak da yapabiliyorduk. Ama eğer hayattan tek beklentiniz iki kümenin kesişim kümesinin boş olup olmadığını, yani bu iki kümenin ortak eleman içerip içermediğini öğrenmekse, basitçe *isdisjoint()* metodundan yararlanabilirsiniz:

```
>>> a = set([1, 2, 3])
>>> b = set([2, 4, 6])

>>> a.isdisjoint(b)
```

```
False
```

Gördüğünüz gibi, a ve b kümesinin kesişimi boş olmadığı için, yani bu iki küme ortak en az bir öğe barındırdığı için, *isdisjoint()* metodu False çıktısı veriyor. Burada temel olarak şu soruyu sormuş oluyoruz:

a ve b ayrık kümeler mi?

Python da bize cevap olarak, "*Hayır değil,*" anlamına gelen False çıktısını veriyor... Çünkü a ve b kümelerinin ortak bir elemanı var (*2*).

Bir de şuna bakalım:

```
>>> a = set([1, 3, 5])
>>> b = set([2, 4, 6])

>>> a.isdisjoint(b)

True
```

Burada a ve b kümeleri ortak hiç bir elemana sahip olmadığı için "Doğru" anlamına gelen True çıktısını elde ediyoruz.

issubset()

Bu metot yardımıyla, bir kümenin bütün elemanlarının başka bir küme içinde yer alıp yer almadığını sorgulayabiliriz. Yani bir kümenin, başka bir kümenin alt kümesi olup olmadığını bu metot yardımıyla öğrenebiliriz. Eğer bir küme başka bir kümenin alt kümesi ise bu metot bize True değerini verecek; eğer değilse False çıktısını verecektir:

```
>>> a = set([1, 2, 3])
>>> b = set([0, 1, 2, 3, 4, 5])

>>> a.issubset(b)

True
```

Bu örnekte True çıktısını aldık, çünkü a kümesinin bütün öğeleri b kümesi içinde yer alıyor. Yani a, b'nin alt kümesidir.

issuperset()

Bu metot, bir önceki bölümde gördüğümüz *issubset()* metoduna benzer. Matematik derslerinde

gördüğümüz "kümeler" konusunda hatırladığınız "b kümesi a kümesini kapsar" ifadesini bu metotla gösteriyoruz. Önce bir önceki derste gördüğümüz örneğe bakalım:

```
>>> a = set([1, 2, 3])
>>> b = set([0, 1, 2, 3, 4, 5])

>>> a.issubset(b)

True
```

Buradan, "a kümesi b kümesinin alt kümesidir," sonucuna ulaşıyoruz. Bir de şuna bakalım:

```
>>> a = set([1, 2, 3])
>>> b = set([0, 1, 2, 3, 4, 5])

>>> b.issuperset(a)

True
```

Burada ise, "b kümesi a kümesini kapsar," sonucunu elde ediyoruz. Yani b kümesi a kümesinin bütün elemanlarını içinde barındırıyor.

union()

union() metodu iki kümenin birleşimini almamızı sağlar. Hemen bir örnek verelim:

```
>>> a = set([2, 4, 6, 8])
>>> b = set([1, 3, 5, 7])

>>> a.union(b)

{1, 2, 3, 4, 5, 6, 7, 8}
```

Önceki bölümlerde gördüğümüz bazı metotlarda olduğu gibi, *union()* metodu da bir kısayola sahiptir. *union()* metodu yerine "|" işaretini de kullanabiliriz:

```
>>> a | b
```

union() metodu yerine, bu metodun kısayolu olan "|" işareti Python programcıları tarafından daha sık kullanılır.

update()

Hatırlarsanız *add()* metodunu anlatırken şöyle bir örnek vermiştik:

```
>>> küme = set(["elma", "armut", "kebap"])

>>> yeni = [1, 2, 3]

>>> for i in yeni:
...     küme.add(i)
...

set([1, 2, 3, 'elma', 'kebap', '\x87ilek', 'armut'])
```

Bu örneği verdikten sonra da şöyle bir şey demiştik:

"Burada yeni adlı listeyi kümeye *for* döngüsü ile ekledik. Ama bu işlemi yapmanın başka bir yolu daha vardır. Bu işlem için Python'da ayrı bir metot bulunur."

İşte bu metodu öğrenmenin vakti geldi. Metodumuzun adı *update()*. Bu metot, bir kümeyi güncellememizi sağlar. İsterseniz yukarıdaki örneği, bu metodu kullanarak tekrar yazalım:

```
>>> küme = set(["elma", "armut", "kebap"])

>>> yeni = [1, 2, 3]

>>> küme.update(yeni)

>>> print(küme)

{1, 2, 3, 'elma', 'kebap', 'armut'}
```

Gördüğünüz gibi, *for* döngüsünü kullanmaya gerek kalmadan aynı sonucu elde edebildik.

symmetric_difference()

Daha önceki bölümlerde *difference()* metodunu kullanarak iki küme arasındaki farklı öğeleri bulmayı öğrenmiştik. Örneğin elimizde şöyle iki küme var diyelim:

```
>>> a = set([1, 2, 5])
>>> b = set([1, 4, 5])
```

Eğer a kümesinin b kümesinden farkını bulmak istersek şöyle yapıyoruz:

```
>>> a.difference(b)

set([2])
```

Demek ki a kümesinde bulunup b kümesinde bulunmayan öğe *2* imiş.

Bir de *b* kümesinde bulunup *a* kümesinde bulunmayan öğelere bakalım:

```
>>> b.difference(a)

set([4])
```

Bu da bize "4" çıktısını verdi. Demek ki bu öğe b kümesinde bulunuyor, ama a kümesinde bulunmuyormuş. Peki ya kümelerin ikisinde de bulunmayan öğeleri aynı anda nasıl alacağız? işte bu noktada yardımımıza *symmetric_difference()* adlı metot yetişecek:

```
>>> a.symmetric_difference(b)

set([2, 4])
```

Böylece iki kümede de bulunmayan öğeleri aynı anda almış olduk.

symmetric_difference_update()

Daha önce *difference_update* ve *intersection_update* gibi metotları öğrenmiştik. *symmetric_difference_update()* metodu da bunlara benzer bir işlevi yerine getirir:

```
>>> a = set([1,2, 5])

>>> b = set([1,4, 5])

>>> a.symmetric_difference_update(b)

>>> print(a)

set([2, 4])
```

Gördüğünüz gibi, a kümesinin eski öğeleri gitti, yerlerine *symmetric_difference()* metoduyla elde edilen çıktı geldi. Yani a kümesi, *symmetric_difference()* metodunun sonucuna göre güncellenmiş oldu...

pop()

İnceleyeceğimiz son metot *pop()* metodu olacak. Gerçi bu metot bize hiç yabancı değil. Bu metodu listeler konusundan hatırlıyoruz. Orada öğrendiğimize göre, bu metot listenin bir öğesini silip ekrana basıyordu. Aslında buradaki fonksiyonu da farklı değil. Burada da kümelerin öğelerini silip ekrana basıyor:

```
>>> a = set(["elma", "armut", "kebap"])

>>> a.pop()

'elma'
```

Peki bu metot hangi ölçüte göre kümeden öğe siliyor? Herhangi bir ölçüt yok. Bu metot, küme öğelerini tamamen rastgele siliyor.

Böylelikle Python'da Listeler, Demetler, Sözlükler ve Kümeler konusunu bitirmiş olduk. Bu konuları sık sık tekrar etmek, hiç olmazsa arada sırada göz gezdirmek bazı şeylerin zihnimizde yer etmesi açısından oldukça önemlidir.

Dondurulmuş Kümeler (Frozenset)

Daha önce de söylediğimiz gibi, kümeler üzerinde değişiklik yapabiliyoruz. Zaten kümelerin *add()* ve *remove()* gibi metotlarının olması bu durumu teyit ediyor. Ancak kimi durumlarda, öğeleri üzerinde değişiklik yapılamayan kümelere de ihtiyaç duyabilirsiniz. Hatırlarsanız listeler ve demetler arasında da buna benzer bir ilişki var. Demetler çoğu zaman, üzerinde değişiklik yapılamayan bir liste gibi davranır. İşte Python aynı imkanı bize kümelerde de sağlar. Eğer öğeleri üzerinde değişiklik yapılamayan bir küme oluşturmak isterseniz *set()* yerine *frozenset()* fonksiyonunu kullanabilirsiniz. Dilerseniz hemen bununla ilgili bir örnek verelim:

```
>>> dondurulmus = frozenset(["elma", "armut", "ayva"])
```

Dondurulmuş kümeleri bu şekilde oluşturuyoruz. Şimdi bu dondurulmuş kümenin metotlarına bakalım:

```
>>> dir(dondurulmus)

['__and__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__or__',
'__rand__', '__reduce__', '__reduce_ex__', '__repr__', '__ror__', '__rsub__',
'__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__xor__', 'copy', 'difference', 'intersection',
'isdisjoint', 'issubset', 'issuperset', 'symmetric_difference', 'union']
```

Gördüğünüz gibi, *add()*, *remove()*, *update()* gibi, değişiklik yapmaya yönelik metotlar listede yok.

Dondurulmuş kümeler ile normal kümeler arasında işlev olarak hiçbir fark yoktur. Bu ikisi arasındaki fark, listeler ile demetler arasındaki fark gibidir.

başa dön

Fonksiyonlar

İlk derslerimizden bu yana bir şey özellikle dikkatinizi çekmiş olmalı: İlk andan itibaren hep 'fonksiyon' diye bir kavramdan söz ettik; üstelik yazdığımız kodlarda da bu fonksiyon denen şeyi bolca kullandık. Evet, belki bu kavramı şimdiye dek enine boyuna inceleme fırsatımız hiç olmadı, ama yine de adının fonksiyon olduğunu söylediğimiz pek çok araç tanıdık bu noktaya gelinceye kadar.

Herhalde, 'Fonksiyon denince aklınıza ilk ne geliyor?' diye bir soru sorsam, vereceğiniz cevap print() fonksiyonu olacaktır. Gerçekten de bu fonksiyonu ilk derslerimizden bu yana o kadar sık kullandık ki, fonksiyon denince aklınıza ilk bu fonksiyonun gelmesi gayet doğal.

Elbette öğrendiğimiz tek fonksiyon print() değildi. Bunun dışında type() diye bir fonksiyondan da söz etmiştik. print() kadar olmasa da, type() fonksiyonunu da yazdığımız kodlarda epey kullandık. print() ve type() dışında, fonksiyon olarak str(), int() ve benzeri araçlarla da tanıştık. Bunların dışında pek çok başka fonksiyon da Python'la birlikte hayatımıza girdi.

İşte bu bölümde, en baştan bu yana sıklıkla sözünü ettiğimiz, ama hiçbir zaman tam anlamıyla ele almadığımız bu kavramı daha iyi anlayabilmek için, fonksiyon konusunu ayrıntılı olarak ele alacağız. Bu bölümde amacımız fonksiyonları enine boyuna inceleyerek, okurun bilgi dağarcığında fonksiyonlara ilişkin sağlam bir altyapı oluşturmaktır. Okur, bu bölümü bitirdikten sonra fonksiyonlara ilişkin olarak bilmesi gereken her şeyi öğrenmiş olacak.

Fonksiyon Nedir ve Ne İşe Yarar?

Biz şimdiye dek karşılaştığımız print(), len(), type() ve open() gibi örnekler sayesinde 'fonksiyon' denen şeyi az çok tanıdığımızı söyleyebiliriz. Dolayısıyla fonksiyonun ne demek olduğunu şeklen de olsa biliyoruz ve hatta fonksiyonları kodlarımız içinde etkili bir şekilde kullanabiliyoruz.

İlk derslerimizden bu yana öğrendiğimiz fonksiyonlara şöyle bir bakacak olursak, fonksiyonların görünüşüne ve yapısına dair herhalde şu tespitleri yapabiliriz:

1. Her fonksiyonun bir adı bulunur ve fonksiyonlar sahip oldukları bu adlarla anılır. (print fonksiyonu, open fonksiyonu, type fonksiyonu, input fonksiyonu, len fonksiyonu vb.)
2. Şekil olarak, her fonksiyonun isminin yanında birer parantez işareti bulunur. (open(), print(), input(), len() vb.)
3. Bu parantez işaretlerinin içine, fonksiyonlara işlevsellik kazandıran bazı parametreler yazılır. (open(dosya_adı), print("Merhaba Zalim Dünya!"), len("kahramanmaraş") vb.)
4. Fonksiyonlar farklı sayıda parametre alabilir. Örneğin print() fonksiyonu toplam *256* adet parametre alabilirken, input() fonksiyonu yalnızca tek bir parametre alır.
5. Fonksiyonların isimli ve isimsiz parametreleri vardır. print() fonksiyonundaki *sep*, *end* ve *file* parametreleri isimli parametrelere örnekken, mesela print("Merhaba Dünya!")

kodunda *Merhaba Dünya!* parametresi isimsiz bir parametredir. Aynı şekilde input("Adınız: ") gibi bir kodda *Adınız:* parametresi isimsiz bir parametredir.

6. Fonksiyonların, isimli ve isimsiz parametreleri dışında, bir de varsayılan değerli parametreleri vardır. Örneğin print() fonksiyonunun *sep*, *end* ve *file* parametreleri varsayılan değerli parametrelere birer örnektir. Eğer bir parametrenin varsayılan bir değeri varsa, o parametreye herhangi bir değer vermeden de fonksiyonu kullanabiliriz. Python bu parametrelere, belirli değerleri öntanımlı olarak kendisi atayacaktır. Tabii eğer istersek, varsayılan değerli parametrelere kendimiz de başka birtakım değerler verebiliriz.

Fonksiyon kavramının tam olarak ne olduğunu henüz bilmiyor da olsak, şimdiye kadar öğrendiğimiz fonksiyonlara bakarak fonksiyonlar hakkında yukarıdaki çıkarımları yapabiliyoruz. Demek ki aslında fonksiyonlar hakkında alttan alta pek çok şey öğrenmişiz. O halde, fonksiyonlar hakkında şimdiden bildiklerimize güvenerek, fonksiyon kavramının ne olduğundan ziyade ne işe yaradığı konusuna rahatlıkla eğilebiliriz. Zaten fonksiyonların ne işe yaradığını öğrendikten sonra, fonksiyonların ne olduğunu da hemencecik anlayacaksınız.

Fonksiyonların ne işe yaradığını en genel ve en kaba haliyle tarif etmek istersek şöyle bir tanımlama yapabiliriz:

> **Fonksiyonların görevi, karmaşık işlemleri bir araya toplayarak, bu işlemleri tek adımda yapmamızı sağlamaktır. Fonksiyonlar çoğu zaman, yapmak istediğimiz işlemler için bir şablon vazifesi görür. Fonksiyonları kullanarak, bir veya birkaç adımdan oluşan işlemleri tek bir isim altında toplayabiliriz. Python'daki 'fonksiyon' kavramı başka programlama dillerinde 'rutin' veya 'prosedür' olarak adlandırılır. Gerçekten de fonksiyonlar rutin olarak tekrar edilen görevleri veya prosedürleri tek bir ad/çatı altında toplayan araçlardır.**

Dilerseniz yukarıdaki soyut ifadeleri basit bir örnek üzerinde somutlaştırmaya çalışalım. Örneğin print() fonksiyonunu ele alalım.

Bu fonksiyonun görevini biliyorsunuz: print() fonksiyonunun görevi, kullanıcının girdiği parametreleri ekrana çıktı olarak vermektir. Her ne kadar print() fonksiyonunun görevini, ekrana çıktı vermek olarak tanımlasak da, aslında bu fonksiyon, ekrana çıktı vermenin yanısıra, başka bir takım ilave işlemler de yapar. Yani bu fonksiyon, aslında aldığı parametreleri sadece ekrana çıktı olarak vermekle yetinmez. Örneğin şu komutu inceleyelim:

```
>>> print("Fırat", "Özgül", "1980", "Adana")
```

Burada print() fonksiyonu toplam dört adet parametre alıyor. Fonksiyonumuz, görevi gereği, bu parametreleri ekrana çıktı olarak verecek. Bu komutu çalıştırdığımızda şöyle bir çıktı alıyoruz:

```
Fırat Özgül 1980 Adana
```

Dikkat ederseniz, burada salt bir 'ekrana çıktı verme' işleminden fazlası var. Zira print() fonksiyonu aldığı parametreleri şu şekilde de ekrana verebilirdi:

```
FıratÖzgül1980Adana
```

Veya şu şekilde:

```
F
ı
r
a
t
Ö
z
g
ü
l
1
9
8
0
A
d
a
n
a
```

Neticede bunlar da birer çıktı verme işlemidir. Ama dediğimiz gibi, print() fonksiyonu aldığı parametreleri sadece ekrana çıktı olarak vermekle yetinmiyor. Gelin isterseniz ne demek istediğimizi biraz daha açıklayalım:

print() fonksiyonunun yukarıdaki komutu nasıl algıladığını önceki derslerimizde öğrenmiştik. Yukarıdaki komut Python tarafından şu şekilde algılanıyor:

```
>>> print("Fırat", "Özgül", "1980", "Adana", sep=" ", end="\n",
... file=sys.stdout, flush=False)
```

Yani print() fonksiyonu;

1. Kendisine verilen *"Fırat"*, *"Özgül"*, *"1980"* ve *"Adana"* parametrelerini ekrana basıyor,
2. *sep=" "* parametresinin etkisiyle, bu parametreler arasına birer boşluk ekliyor,
3. *end="\n"* parametresinin etkisiyle, sonuncu parametreyi de ekrana bastıktan sonra bir alt satıra geçiyor,
4. *file=sys.stdout* parametresinin etkisiyle, çıktı konumu olarak komut ekranını kullanıyor. Yani çıktıları ekrana veriyor.
5. *flush=False* parametresinin etkisiyle, çıktılar ekrana gönderilmeden önce tamponda bekletiliyor.

Eğer print() gibi bir fonksiyon olmasaydı, yukarıda listediğimiz bütün bu işlemleri kendimiz yapmak

zorunda kalacaktık. Yani ekranda göstermek istediğimiz ifadeleri ekrana çıktı olarak vermenin yanısıra, bunların ekranda nasıl görüneceğini de tek tek kendimiz elle ayarlamak zorunda kalacaktır. Ekrana çıktı verme ile ilgili pek çok işlem tek bir print() fonksiyonu altında birleştirildiği için, her ihtiyaç duyduğumuzda o işlemleri tek tek bizim yapmamıza gerek kalmıyor.

Aynı şey mesela input() fonksiyonu için de geçerlidir. Bu fonksiyonu kullanarak, programımızı kullanan kişilerle etkileşim içine girebiliyoruz. Tıpkı print() fonksiyonunda olduğu gibi, input() fonksiyonu da aslında alttan alta epey karmaşık işlemler gerçekleştirir. Ama o karmaşık işlemlerin tek bir input() fonksiyonu içinde bir araya getirilmiş olması sayesinde, sadece input() gibi basit bir komut vererek kullanıcılarımızla iletişime geçebiliyoruz.

Bu açıdan bakıldığında fonksiyonlar değişkenlere benzer. Bildiğiniz gibi, her defasında bir değeri tekrar tekrar yazmak yerine bir değişkene atayarak o değere kolayca erişebiliyoruz. Örneğin:

```
>>> kurum = "Sosyal Sigortalar Kurumu"
```

Burada tanımladığımız *kurum* adlı değişken sayesinde, 'Sosyal Sigortalar Kurumu' ifadesini kullanmamız gereken her yerde sadece değişken adını kullanarak, değişkenin tuttuğu değere ulaşabiliyoruz. İşte fonksiyonlar da buna benzer bir işlev görür: Örneğin ekrana bir çıktı vermemiz gereken her yerde, yukarıda verdiğimiz karmaşık adımları tek tek gerçekleştirmeye çalışmak yerine, bu karmaşık ve rutin adımları bir araya getiren print() gibi bir fonksiyondan yararlanarak işlerimizi çok daha kolay bir şekilde halledebiliriz.

Bu anlattıklarımız fonksiyonların ne işe yaradığı konusunda size bir fikir vermiş olabilir. Dilerseniz bu anlattıklarımızı bir örnek aracılığıyla biraz daha somutlaştırmaya çalışalım:

Hatırlarsanız 'Kullanıcıyla Veri Alışverişi' başlıklı bölümde şöyle bir örnek vermiştik:

```
isim    = "Fırat"
soyisim = "Özgül"
işsis   = "Ubuntu"
şehir   = "İstanbul"

print("isim          : ", isim)
print("soyisim       : ", soyisim)
print("işletim sistemi: ", işsis)
print("şehir         : ", şehir)
```

Bu programı çalıştırdığımızda şöyle bir çıktı alıyoruz:

```
isim          :  Fırat
soyisim       :  Özgül
işletim sistemi:  Ubuntu
şehir         :  İstanbul
```

Bu program, belli değerleri kullanarak bir kayıt oluşturma işlemi gerçekleştiriyor. Mesela yukarıdaki örnekte, 'Fırat Özgül' adlı şahsa ait isim, soyisim, işletim sistemi ve şehir bilgilerini alarak, bu kişi için bir kayıt oluşturuyoruz.

Peki 'Fırat Özgül' adlı kişinin yanısıra, 'Mehmet Öztaban' adlı başka bir kişi için de kayıt oluşturmak istersek ne yapacağız?

Aklınıza şöyle bir şey yazmak gelmiş olabilir:

```
isim1    = "Fırat"
soyisim1 = "Özgül"
işsis1   = "Ubuntu"
şehir1   = "İstanbul"

print("isim           : ", isim1)
print("soyisim        : ", soyisim1)
print("işletim sistemi: ", işsis1)
print("şehir          : ", şehir1)

print("-"*30)

isim2    = "Mehmet"
soyisim2 = "Öztaban"
işsis2   = "Debian"
şehir2   = "Ankara"

print("isim           : ", isim2)
print("soyisim        : ", soyisim2)
print("işletim sistemi: ", işsis2)
print("şehir          : ", şehir2)

print("-"*30)
```

Programa her yeni kayıt eklenişinde, her yeni kişi için benzer satırları tekrar tekrar yazabilirsiniz. Peki ama bu yöntem sizce de çok sıkıcı değil mi? Üstelik bir o kadar da hataya açık bir yöntem. Muhtemelen ilk kaydı ekledikten sonra, ikinci kaydı eklerken birinci kayıttaki bilgileri kopyalayıp, bu kopya üzerinden ikinci kaydı oluşturuyorsunuz. Hatta muhtemelen kopyalayıp yapıştırdıktan sonra yeni kaydı düzenlerken bazı hatalar da yapıyor ve düzgün çalışan bir program elde edebilmek için o hataları düzeltmekle de uğraşıyorsunuz.

Bütün bu işleri kolaylaştıracak bir çözüm olsa ve bizi aynı şeyleri tekrar tekrar yazmaktan kurtarsa sizce de çok güzel olmaz mıydı? Mesela tıpkı print() fonksiyonu gibi, kayıt_oluştur() adlı bir fonksiyon olsa, biz sadece gerekli bilgileri bu fonksiyonun parantezleri içine parametre olarak yazsak ve bu fonksiyon bize istediğimiz bilgileri içeren bir kayıt oluştursa ne hoş olurdu, değil mi? Yani

örneğin bahsettiğimiz bu hayali kayıt_oluştur() fonksiyonunu şu şekilde kullanabilseydik...

```
kayıt_oluştur("Mehmet", "Öztaban", "Debian", "Ankara")
```

... ve bu komut bize şu çıktıyı verebilseydi...

```
----------------------------
isim          :  Mehmet
soyisim       :  Öztaban
işletim sistemi:  Debian
şehir         :  Ankara
----------------------------
```

... ne kadar güzel olurdu, değil mi?

İşte böyle bir şey Python'da mümkündür. Nasıl Python geliştiricileri print(), input() ve benzeri fonksiyonları tanımlayıp, karmaşık işlemleri tek adımda yapabilmemiz için bize sunmuş ve böylece bizi her defasında tekerleği yeniden icat etme külfetinden kurtarmışsa, biz de kendi fonksiyonlarımızı tanımlayarak, kendimizi aynı işlemleri tekrar tekrar yapma zahmetinden kurtarabiliriz.

Gelin şimdi bu işi nasıl yapabileceğimizi tartışalım.

Fonksiyon Tanımlamak ve Çağırmak

Bir önceki bölümde, kayıt_oluştur() adlı hayali bir fonksiyondan söz etmiştik. Tasarımıza göre bu fonksiyon şu şekilde kullanılacak:

```
kayıt_oluştur("Ahmet", "Gür", "Pardus", "İzmir")
```

Bu komutu verdiğimizde ise şöyle bir çıktı almayı planlıyoruz:

```
----------------------------
isim          :  Ahmet
soyisim       :  Gür
işletim sistemi:  Pardus
şehir         :  İzmir
----------------------------
```

Dediğimiz gibi, böyle bir şey yapmak Python'la mümkündür. Ancak tabii ki kayıt_oluştur() adlı böyle bir fonksiyonu kullanabilmenin belli ön koşulları var. Nasıl *sayı* adlı bir değişkeni kullanabilmek için öncelikle bu ada sahip bir değişken tanımlamış olmamız gerekiyorsa, aynı şekilde kayıt_oluştur() adlı bir fonksiyonu kullanabilmek için de öncelikle bu ada sahip bir fonksiyonu tanımlamış olmamız gerekiyor. Zira mesela input() ve print() gibi fonksiyonları kullanabiliyor olmamız, Python geliştiricilerinin bu fonksiyonları tanımlayıp dilin içine gömmüş olmaları

sayesindedir.

İşte biz de kayıt_oluştur() adlı fonksiyonu kullanabilmek için bu ada sahip fonksiyonu aşağıdaki şekilde tanımlamalıyız:

```
def kayıt_oluştur(isim, soyisim, işsis, şehir):
    print("-"*30)

    print("isim          : ", isim)
    print("soyisim       : ", soyisim)
    print("işletim sistemi: ", işsis)
    print("şehir         : ", şehir)

    print("-"*30)
```

İlk bakışta bu kodlar size hiçbir şey ifade etmemiş olabilir. Ama hiç endişe etmeyin. Biz birazdan bu satırların ne anlama geldiğini bütün ayrıntılarıyla anlatacağız. Siz şimdilik anlamadığınız kısımları görmezden gelip okumaya devam edin.

Yukarıdaki kodlar yardımıyla fonksiyonumuzu tanımlamış olduk. Artık elimizde, tıpkı print() veya input() gibi, kayıt_oluştur() adlı 'ev yapımı' bir fonksiyon var. Dolayısıyla bu yeni fonksiyonumuzu, daha önce öğrendiğimiz fonksiyonları nasıl kullanıyorsak aynı şekilde kullanabiliriz. Yani aşağıdaki gibi komutlar yazabiliriz:

```
kayıt_oluştur("Fırat", "Özgül", "Ubuntu", "İstanbul")
kayıt_oluştur("Mehmet", "Öztaban", "Debian", "Ankara")
```

Yalnız fonksiyonumuzu tanımlayıp bitirdikten sonra, bu fonksiyonu kullanırken, kodlarımızın hizalamasına dikkat ediyoruz. Fonksiyonu kullanmak için yazdığımız kodları *def* ifadesinin hizasına getiriyoruz. Yani:

```
def kayıt_oluştur(isim, soyisim, işsis, şehir):
    print("-"*30)

    print("isim          : ", isim)
    print("soyisim       : ", soyisim)
    print("işletim sistemi: ", işsis)
    print("şehir         : ", şehir)

    print("-"*30)

kayıt_oluştur("Fırat", "Özgül", "Ubuntu", "İstanbul")
kayıt_oluştur("Mehmet", "Öztaban", "Debian", "Ankara")
```

Yukarıdaki yapıyı kullanarak, istediğiniz sayıda kayıt oluşturabilirsiniz. Mesela:

```
kayıt_oluştur("İlkay", "Kaya", "Mint", "Adana")
kayıt_oluştur("Seda", "Kara", "SuSe", "Erzurum")
```

Gördüğünüz gibi, yukarıdaki yöntem sayesinde kodlarımızdaki tekrar eden kısımlar ortadan kalktı. Yukarıdaki fonksiyonun bize nasıl bir kolaylık sağladığını daha net görebilmek için, fonksiyon kullanarak sadece şu *11* satırla elde ettiğimiz çıktıyı, fonksiyon kullanmadan elde etmeyi deneyebilirsiniz:

```
def kayıt_oluştur(isim, soyisim, işsis, şehir):
    print("-"*30)

    print("isim           : ", isim)
    print("soyisim         : ", soyisim)
    print("işletim sistemi: ", işsis)
    print("şehir          : ", şehir)

    print("-"*30)

kayıt_oluştur("Fırat", "Özgül", "Ubuntu", "İstanbul")
kayıt_oluştur("Mehmet", "Öztaban", "Debian", "Ankara")
kayıt_oluştur("İlkay", "Kaya", "Mint", "Adana")
kayıt_oluştur("Seda", "Kara", "SuSe", "Erzurum")
```

Bu anlattıklarımız size çok karmaşık gelmiş olabilir. Ama endişe etmenize hiç gerek yok. Biraz sonra, yukarıda yazdığımız kodların hepsini didik didik edeceğiz. Ama öncelikle yukarıdaki kod parçasını yapısal olarak bir incelemenizi istiyorum. Fonksiyonu tanımladığımız aşağıdaki kod parçasına şöyle bir baktığınızda neler görüyorsunuz?

```
def kayıt_oluştur(isim, soyisim, işsis, şehir):
    print("-"*30)

    print("isim           : ", isim)
    print("soyisim         : ", soyisim)
    print("işletim sistemi: ", işsis)
    print("şehir          : ", şehir)

    print("-"*30)

kayıt_oluştur("Fırat", "Özgül", "Ubuntu", "İstanbul")
```

Bu kodları incelediğinizde şu noktalar dikkatinizi çekiyor olmalı:

1. Kodlar *def* adlı bir ifade ile başlamış.
2. Bunun ardından ‘kayıt_oluştur’ ifadesini görüyoruz.
3. Bu ifadeyi, içinde birtakım kelimeler barındıran bir parantez çifti izliyor.

4. Parantezin içinde, *isim*, *soyisim*, *işsis* ve *şehir* adlı değerler var.
5. *def* ile başlayan bu satır iki nokta üst üste işareti ile son buluyor.
6. İlk satırın ardından gelen kısım ilk satıra göre girintili bir şekilde yazılmış.
7. kayıt_oluştur("Fırat", "Özgül", "Ubuntu", "İstanbul") satırı önceki satırlara göre girintisiz yazılmış.

Eğer bu kodlara dikkatlice bakacak olursanız, aslında bu kodların topu topu iki parçadan oluştuğunu göreceksiniz. İsterseniz yukarıdaki yapıyı biraz sadeleştirelim:

```
def kayıt_oluştur(parametre1, parametre2, parametre3, parametre4):
    (...)

kayıt_oluştur(parametre1, parametre2, parametre3, parametre4)
```

Bu yapının ilk parçası şudur:

```
def kayıt_oluştur(parametre1, parametre2, parametre3, parametre4):
    (...)
```

İkinci parçası ise şu:

```
kayıt_oluştur(parametre1, parametre2, parametre3, parametre4)
```

Teknik olarak söylemek gerekirse, ilk parçaya 'fonksiyon tanımı' (*function definition*), ikinci parçaya ise 'fonksiyon çağrısı' (*function call*) adı verilir. Dolayısıyla bir fonksiyonun yaşam döngüsü iki aşamadan oluşur. Buna göre bir fonksiyon önce tanımlanır;

```
def kayıt_oluştur(parametre1, parametre2, parametre3, parametre4):
    (...)
```

...sonra da çağrılır;

```
kayıt_oluştur(parametre1, parametre2, parametre3, parametre4)
```

Aslında biz şimdiye kadar gördüğümüz print(), type(), open() vb. fonksiyonlarda bu 'fonksiyon çağrısı' kısmıyla zaten tanışmıştık. Zira şu komut tam anlamıyla bir fonksiyon çağrısıdır (yani bir fonksiyon çağırma işlemidir):

```
print("Fırat", "Özgül", "Adana", 32)
```

Gördüğünüz gibi, yukarıdaki komutun yapı olarak şu komuttan hiçbir farkı yok:

```
kayıt_oluştur("Fırat", "Özgül", "Ubuntu", "İstanbul")
```

Bu iki fonksiyon arasındaki tek fark, print() fonksiyonunu Python geliştiricilerinin; kayıt_oluştur() fonksiyonunu ise sizin tanımlamış olmanızdır.

Elbette bu iki fonksiyon yapı olarak birbirinin aynı olsa da, işlev olarak birbirinden farklıdır. print() fonksiyonunun görevi kendisine parametre olarak verilen değerleri ekrana çıktı vermek iken, kayıt_oluştur() fonksiyonunun görevi kendisine parametre olarak verilen değerleri kullanarak bir kayıt oluşturmaktır.

Bu derse gelinceye kadar öğrendiğimiz print(), type() ve open() gibi fonksiyonlara teknik olarak 'gömülü fonksiyonlar' (*builtin functions*) adı verilir. Bu fonksiyonlara bu adın verilmiş olmasının sebebi, bu fonksiyonların gerçekten de Python programlama dili içine gömülü bir vaziyette olmalarıdır. Dikkat ederseniz kendi yazdığımız fonksiyonları kullanabilmek için öncelikle fonksiyonu tanımlamamız gerekiyor. Gömülü fonksiyonlar ise Python geliştiricileri tarafından halihazırda tanımlanmış olduğu için bunları biz herhangi bir tanımlama işlemi yapmaya gerek kalmadan doğrudan çağırabiliyoruz.

Böylece bir fonksiyonun yapı olarak neye benzediğini üstünkörü de olsa incelemiş olduk. Buraya kadar anlatılan kısımda bazı noktaları anlamakta zorlanmış olabilirsiniz. Eğer öyleyse hiç endişelenmeyin. Bu gayet doğal.

Gelin isterseniz şimdi yukarıda anlattıklarımızın içini doldurmaya çalışalım.

Fonksiyonların Yapısı

İsterseniz biraz da fonksiyonların yapısından söz edelim. Böylelikle ne ile karşı karşıya olduğumuzu anlamak zihninizde biraz daha kolaylaşır.

Dedik ki, bir fonksiyonun ilk parçasına 'fonksiyon tanımı' (*function definition*) adı verilir. Bir fonksiyonu tanımlamak için *def* adlı bir parçacıktan yararlanıyoruz. Örneğin:

```
def bir_fonksiyon():
    (...)
```

Burada *def* parçacığı, tanımladığımız şeyin bir fonksiyon olduğunu gösteriyor. *bir_fonksiyon* ifadesi ise tanımladığımız bu fonksiyonun adıdır. Fonksiyonu tanımladıktan sonra, çağırırken bu adı kullanacağız.

def bir_fonksiyon(): ifadesinin sonundaki iki nokta işaretinden de tahmin edebileceğiniz gibi, sonraki satıra yazacağımız kodlar girintili olacak. Yani mesela:

```
def selamla():
    print("Elveda Zalim Dünya!")
```

Yukarıda selamla() adlı bir fonksiyon tanımlamış olduk. Bu fonksiyonun görevi ekrana *Elveda Zalim Dünya!* çıktısı vermektir.

Bu noktada şöyle bir soru akla geliyor: Acaba fonksiyon gövdesindeki kısım için ne kadarlık bir girinti oluşturacağız?

Girintilemeye ilişkin olarak önceki derslerde bahsettiğimiz bütün kurallar burada da geçerlidir. Fonksiyon gövdesine, *def* ifadesinden itibaren *4* (dört) boşlukluk bir girinti veriyoruz. *def* ifadesinden itibaren girintili olarak yazdığımız kısmın tamamı o fonksiyonun gövdesini oluşturur ve bütünüyle o fonksiyona aittir.

Bu kodlarla yaptığımız şey bir fonksiyon tanımlama işlemidir. Eğer bu kodları bir dosyaya kaydedip çalıştırırsak herhangi bir çıktı almayız. Çünkü henüz fonksiyonumuzu çağırmadık. Bu durumu print(), input() ve benzeri gömülü fonksiyonlara benzetebilirsiniz. Tıpkı yukarıda bizim yaptığımız gibi, gömülü fonksiyonlar da Python geliştiricileri tarafından bir yerlerde tanımlanmış vaziyette dururlar, ama biz bu fonksiyonları yazdığımız programlarda çağırana kadar bu fonksiyonlar çalışmaz.

Daha önce de dediğimiz gibi, bir fonksiyonun yaşam döngüsü iki aşamadan oluşur: Fonksiyon tanımı ve fonksiyon çağrısı. Yukarıda bu döngünün sadece fonksiyon tanımı aşaması mevcut. Unutmayın, bir fonksiyon çağrılmadan asla çalışmaz. Bir fonksiyonun çalışabilmesi için o fonksiyonun tanımlandıktan sonra çağrılması gerekir. Örneğin input() fonksiyonu Python'ın derinliklerinde bir yerlerde tanımlanmış vaziyette durur. Bu fonksiyon, biz onu çağırana kadar, bulunduğu yerde sessizce bekler. Aynı şekilde selamla() adlı fonksiyon da programımız içinde tanımlanmış vaziyette, bizim onu çağıracağımız anı bekliyor. Bu söylediklerimizi destekleyecek açıklayıcı bilgileri biraz sonra vereceğiz. Biz şimdilik fonksiyon tanımı kısmını incelemeye devam edelim.

Bu arada yukarıdaki fonksiyon tanımının yapısına çok dikkat edin. İki nokta üst üste işaretinden sonraki satırda girintili olarak yazılan bütün kodlar (yani fonksiyonun gövde kısmı) fonksiyonun bir parçasıdır. Girintinin dışına çıkıldığı anda fonksiyon tanımlama işlemi de sona erer.

Örneğin:

```
def selamla():
    print("Elveda Zalim Dünya!")

selamla()
```

İşte burada fonksiyonumuzu çağırmış olduk. Dikkat edin! Dediğim gibi, iki nokta üst üste işaretinden sonraki satırda girintili olarak yazılan bütün kodlar fonksiyona aittir. selamla() satırı ise fonksiyon tanımının dışında yer alır. Bu satırla birlikte girintinin dışına çıkıldığı için artık fonksiyon tanımlama safhası sona ermiş oldu.

Biz yukarıdaki örnekte, selamla() adlı fonksiyonu tanımlar tanımlamaz çağırmayı tercih ettik. Ama elbette siz bir fonksiyonu tanımlar tanımlamaz çağırmak zorunda değilsiniz. Yazdığınız bir program içinde fonksiyonlarınızı tanımladıktan sonra, ihtiyacınıza bağlı olarak, programın herhangi başka bir yerinde fonksiyonlarınızı çağırabilirsiniz.

Fonksiyonlarla ilgili söylediklerimizi toparlayacak olursak şöyle bir bilgi listesi ortaya çıkarabiliriz:

1. Python'da kabaca iki tip fonksiyon bulunur. Bunlardan biri gömülü fonksiyonlar (*builtin functions*), öteki ise özel fonksiyonlardır (*custom functions*). Burada 'özel' ifadesi,

'kullanıcının ihtiyaçlarına göre kullanıcı tarafından özel olarak üretilmiş' anlamına gelir.

2. Gömülü fonksiyonlar; Python geliştiricileri tarafından tanımlanıp dilin içine gömülmüş olan print(), open(), type(), str(), int() vb. fonksiyonlardır. Bu fonksiyonlar halihazırda tanımlanıp hizmetimize sunulduğu için bunları biz herhangi bir tanımlama işlemi yapmadan doğrudan kullanabiliriz.
3. Özel fonksiyonlar ise, gömülü fonksiyonların aksine, Python geliştiricileri tarafından değil, bizim tarafımızdan tanımlanmıştır. Bu fonksiyonlar dilin bir parçası olmadığından, bu fonksiyonları kullanabilmek için bunları öncelikle tanımlamamız gerekir.
4. Python'da bir fonksiyonun yaşam döngüsü iki aşamadan oluşur: Tanımlanma ve çağrılma.
5. Bir fonksiyonun çağrılabilmesi (yani kullanılabilmesi) için mutlaka birisi tarafından tanımlanmış olması gerekir.
6. Fonksiyonu tanımlayan kişi Python geliştiricileri olabileceği gibi, siz de olabilirsiniz. Ama neticede ortada bir fonksiyon varsa, bir yerlerde o fonksiyonun tanımı da vardır.
7. Fonksiyon tanımlamak için *def* adlı bir ifadeden yararlanıyoruz. Bu ifadeden sonra, tanımlayacağımız fonksiyonun adını belirleyip iki nokta üst üste işareti koyuyoruz. İki nokta üst üste işaretinden sonra gelen satırlar girintili olarak yazılıyor. Daha önce öğrendiğimiz bütün girintileme kuralları burada da geçerlidir.
8. Fonksiyonun adını belirleyip iki nokta üst üste koyduktan sonra, alt satırda girintili olarak yazdığımız bütün kodlar fonksiyonun gövdesini oluşturur. Doğal olarak, bir fonksiyonun gövdesindeki bütün kodlar o fonksiyona aittir. Girintinin dışına çıkıldığı anda fonksiyon tanımı da sona erer.

Fonksiyonlarla ilgili öğrendiklerimizi toparladığımıza göre, gelin isterseniz fonksiyonlarla ilgili bir örnek yaparak, bu yapıyı daha iyi anlamaya çalışalım:

```
def sistem_bilgisi_göster():
    import sys
    print("\nSistemde kurulu Python'ın;")
    print("\tana sürüm numarası:", sys.version_info.major)
    print("\talt sürüm numarası:", sys.version_info.minor)
    print("\tminik sürüm numarası:", sys.version_info.micro)

    print("\nKullanılan işletim sisteminin;")
    print("\tadı:", sys.platform)
```

Burada sistem_bilgisi_göster() adlı bir fonksiyon tanımladık. Bu fonksiyonun görevi, kullanıcının sistemindeki Python sürümü ve işletim sistemine dair birtakım bilgiler vermektir.

Bu arada, bu kodlarda, daha önceki derslerimizde öğrendiğimiz *sys* modülünden ve bu modül içindeki değişkenlerden yararlandığımızı görüyorsunuz. Bu kodlarda *sys* modülünün içindeki şu araçları kullandık:

1. *version_info.major*: Python'ın ana sürüm numarası (Örn. 3)
2. *version_info.minor*: Python'ın alt sürüm numarası (Örn. 4)
3. *version_info.micro*: Python'ın minik sürüm numarası (Örn. 0)

4. *platform*: Kullanılan işletim sisteminin adı (Örn. ‘win32’ veya ‘linux2’)

Yukarıda tanımladığımız fonksiyonu nasıl çağıracağımızı biliyorsunuz:

```
sistem_bilgisi_göster()
```

Bu fonksiyon tanımı ve çağrısını eksiksiz bir program içinde gösterelim:

```
def sistem_bilgisi_göster():
    import sys
    print("\nSistemde kurulu Python'ın;")
    print("\tana sürüm numarası:", sys.version_info.major)
    print("\talt sürüm numarası:", sys.version_info.minor)
    print("\tminik sürüm numarası:", sys.version_info.micro)

    print("\nKullanılan işletim sisteminin;")
    print("\tadı:", sys.platform)

sistem_bilgisi_göster()
```

Bu kodları bir dosyaya kaydedip çalıştırdığımızda şuna benzer bir çıktı alacağız:

```
Sistemde kurulu Python'ın;
        ana sürüm numarası: 3
        alt sürüm numarası: 3
        minik sürüm numarası: 0

Kullanılan işletim sisteminin;
        adı: linux
```

Demek ki bu kodların çalıştırıldığı sistem Python’ın 3.3.0 sürümünün kurulu olduğu bir GNU/Linux işletim sistemi imiş...

Fonksiyonlar Ne İşe Yarar?

Şimdiye kadar söylediklerimizden ve verdiğimiz örneklerden fonksiyonların ne işe yaradığını anlamış olmalısınız. Ama biz yine de fonksiyonların faydası üzerine birkaç söz daha söyleyelim. Böylece fonksiyonların ne işe yaradığı konusunda aklımızda hiçbir şüphe kalmaz...

İsterseniz bir örnek üzerinden ilerleyelim.

Diyelim ki, bir sayının karesini bulan bir program yazmak istiyoruz. Şimdiye kadarki bilgilerimizi kullanarak şöyle bir şey yazabiliriz:

```
sayı = 12
```

```
çıktı = "{} sayısının karesi {} sayısıdır"
print(çıktı.format(sayı, sayı**2))
```

Yukarıdaki programı çalıştırdığımızda şöyle bir çıktı elde edeceğiz:

```
12 sayısının karesi 144 sayısıdır
```

Gayet güzel. Şimdi şöyle bir durum hayal edin: Diyelim ki büyük bir program içinde, farklı farklı yerlerde yukarıdaki işlemi tekrar tekrar yapmak istiyorsunuz. Böyle bir durumda şöyle bir şey yazmanız gerekebilir:

```
sayı = 12
çıktı = "{} sayısının karesi {} sayısıdır"
print(çıktı.format(sayı, sayı**2))

####programla ilgili başka kodlar###

sayı = 15
print(çıktı.format(sayı, sayı**2))

###programla ilgili başka kodlar###

sayı = 29
print(çıktı.format(sayı, sayı**2))
```

Buradaki sorun, aynı şeyleri tekrar tekrar yazmak zorunda kalmamızdır. Bu küçük örnekte pek belli olmuyor olabilir, ama özellikle büyük programlarda aynı kodların program içinde sürekli olarak tekrarlanması pek çok probleme yol açar. Örneğin kodlarda bir değişiklik yapmak istediğinizde, tekrarlanan kısımları bulup hepsinin üzerinde tek tek değişiklik yapmanız gerekir. Mesela *çıktı* adlı değişkenin içeriğini değiştirmek isterseniz, yaptığınız değişiklik programınızın pek çok kısmını etkileyebilir. Örneğin, *çıktı* değişkenini şu şekle getirdiğinizi düşünün:

```
çıktı = "{} sayısının karesi {}, karekökü {} sayısıdır"
```

Böyle bir durumda, program içinde geçen bütün `print(çıktı.format(sayı, sayı**2))` satırlarını bulup, üçüncü *{}* işaretine ait işlemi parantez içine eklemeniz gerekir. Tahmin edebileceğiniz gibi, son derece sıkıcı, yorucu ve üstelik hata yapmaya açık bir işlemdir bu. İşte bu tür problemlere karşı fonksiyonlar çok iyi bir çözümdür.

Yukarıda bahsettiğimiz kare bulma işlemi için şu şekilde basit bir fonksiyon tanımlayabiliriz:

```
def kare_bul():
    sayı = 12
    çıktı = "{} sayısının karesi {} sayısıdır"
    print(çıktı.format(sayı, sayı**2))
```

Bu fonksiyonu tanımladık. Şimdi de fonksiyonumuzu çağıralım:

```
kare_bul()
```

Kodlarımız tam olarak şöyle görünüyor:

```
def kare_bul():
    sayı = 12
    çıktı = "{} sayısının karesi {} sayısıdır"
    print(çıktı.format(sayı, sayı**2))

kare_bul()
```

Burada fonksiyonumuz `def kare_bul():` satırıyla başlıyor, `print(çıktı.format(sayı, sayı**2))` satırıyla bitiyor. Daha sonra gelen `kare_bul()` kodu, girintinin dışında yer aldığı için fonksiyon tanımına ait değildir.

Bu kodları bir dosyaya kaydedip çalıştırdığımızda alacağımız çıktı şu olacaktır:

```
12 sayısının karesi 144 sayısıdır
```

`kare_bul()` adlı fonksiyonu bir kez tanımladıktan sonra bu fonksiyonu programınız içinde gereken her yerde çağırabilirsiniz:

```
kare_bul()

####programla ilgili başka kodlar###

kare_bul()

###programla ilgili başka kodlar###

kare_bul()
```

Gördüğünüz gibi `kare_bul()` adlı bu fonksiyon bizi pek çok zahmetten kurtarıyor. Ancak bu fonksiyonun bir sorunu var. Bu fonksiyon ekrana yalnızca *12 sayısının karesi 144 sayısıdır* çıktısı verebiliyor. Buradaki problem, fonksiyonun sadece *12* sayısı üzerinde işlem yapabilmesi. Şöyle bir düşününce, bu çıktının ne kadar anlamsız olduğunu, aslında yukarıdaki fonksiyonun tamamen gereksiz bir iş yaptığını rahatlıkla görebiliyoruz. Fonksiyonumuzun adı `kare_bul`. Ama dediğimiz gibi, fonksiyonumuz sadece *12* sayısının karesini söyleyebiliyor. Halbuki mantık olarak fonksiyonumuzun, bütün sayıların karesini söyleyebilmesini beklerdik.

Not

Bu arada, gördüğünüz gibi, yukarıdaki fonksiyon parametresiz bir fonksiyondur. Dolayısıyla bu

fonksiyonu çağırırken parantez içinde herhangi bir değer belirtmiyoruz.

Fonksiyonumuzun gerçek anlamda işlevli bir hale gelebilmesi için sadece tek bir sayıyı değil, bütün sayıları inceleyebiliyor olması gerek. İşte fonksiyonumuza bu yeteneği parametreler aracılığıyla kazandırabiliriz.

Dikkatlice bakın:

```
def kare_bul(sayı):
    çıktı = "{} sayısının karesi {} sayısıdır"
    print(çıktı.format(sayı, sayı**2))
```

Fonksiyona parametre olarak nasıl bir isim verdiğinizin önemi yoktur. Parantez içine parametre olarak istediğiniz kelimeyi yazabilirsiniz. Önemli olan, parantez içinde fonksiyonun kaç parametre alacağını gösteren bir işaret olmasıdır. Mesela yukarıdaki fonksiyonu şöyle de tanımlayabilirdik:

```
def kare_bul(i):
    çıktı = "{} sayısının karesi {} sayısıdır"
    print(çıktı.format(i, i**2))
```

...veya şöyle:

```
def kare_bul(osman):
    çıktı = "{} sayısının karesi {} sayısıdır"
    print(çıktı.format(osman, osman**2))
```

Elbette parametre adı olarak akılda kalıcı ve daha mantıklı bir seçim yapmak işlerinizi kolaylaştıracaktır...

Şimdi de yukarıdaki fonksiyonu çağıralım:

```
kare_bul(9)
```

Bu fonksiyonu çalıştırdığımızda şu çıktıyı alırız:

```
9 sayısının karesi 81 sayısıdır
```

Bu fonksiyona parametre olarak hangi sayıyı verirseniz o sayının karesi hesaplanacaktır. Örneğin:

```
kare_bul(15)
kare_bul(25555)
```

Yine bu fonksiyonu programınız içinde gereken her yerde çağırabilirsiniz:

```
kare_bul(17)
```

```
####programla ilgili başka kodlar###

kare_bul(21)

###programla ilgili başka kodlar###

kare_bul(54354)
```

Fonksiyonu oluşturan kodlarda herhangi bir değişiklik yapmak istediğinizde sadece fonksiyon tanımının gövdesini değiştirmeniz yeterli olacaktır. Örneğin:

```
def kare_bul(sayı):
    çıktı = "{} sayısının karesi {}, karekökü ise {} sayısıdır"
    print(çıktı.format(sayı, sayı**2, sayı**0.5))
```

Bu sayede sadece fonksiyon gövdesinde değişiklik yaparak, programın başka kısımlarını hiç etkilemeden yolumuza devam edebiliyoruz.

Buraya kadar anlattıklarımız, fonksiyonların ne işe yaradığı ve bir program yazarken neden fonksiyonlara ihtiyaç duyacağımız konusunda size bir fikir vermiş olmalı. Eğer hala aklınızda fonksiyonların faydası konusunda bir şüphe kaldıysa, fonksiyonların faydasını anlamanızı sağlayabilmek için size şöyle bir soru sormama izin verin: Acaba 'istihza' kelimesinin kaç karakterden oluştuğunu nasıl buluruz?

'Elbette len() fonksiyonunu kullanarak!' dediğinizi duyar gibiyim. Gerçekten de Python'da bir karakter dizisinin uzunluğunu bulmanın en iyi yolu len() fonksiyonunu kullanmaktır:

```
>>> len("istihza")

7
```

Peki ya Python'da len() diye bir fonksiyon olmasaydı ne yapacaktınız? Böyle bir durumda, karakter dizilerinin uzunluğunu ölçmek için sizin bir yöntem icat etmeniz gerekecekti. Mesela 'istihza' kelimesinin kaç karakterden oluştuğunu bulmak için şöyle bir kod yazacaktınız:

```
c = 0
for s in "istihza":
    c += 1
print(c)
```

Burada önce *c* adlı bir değişken tanımlayıp, bu değişkenin değerini *0* yaptık. Bu değişken, uzunluğunu sorgulamak istediğimiz kelimenin kaç karakterden oluştuğu bilgisini saklayacak.

Ardından bir for döngüsü tanımlıyoruz. Bu döngüde, 'istihza' kelimesindeki her bir karakter için *c*

değişkeninin değerini *1* sayı artırıyoruz. Böylece döngü sonunda *c* değişkeni 'istihza' kelimesi içinde kaç karakter olduğu bilgisini tutmuş oluyor.

Son olarak da *c* değişkeninin nihai değerini ekrana yazdırıyoruz.

Bu kodları çalıştırdığınızda *7* cevabı alacaksınız. Demek ki 'istihza' kelimesinde *7* karakter varmış. Peki 'istihza' kelimesi yerine mesela 'Afyonkarahisar' kelimesi içinde kaç karakter olduğunu hesaplamak isterseniz ne yapacaksınız? Elbette yukarıdaki kodları tekrar yazıp, 'istihza' kelimesini 'Afyonkarahisar' kelimesi ile değiştireceksiniz. Böylece bu kelimenin kaç karakterden oluştuğunu bulmuş olacaksınız. Sorgulamak istediğiniz her kelime için aynı şeyleri yapabilirsiniz...

Ne kadar verimsiz bir yöntem, değil mi?

Halbuki hiç bu tür şeylerle uğraşmaya gerek yok. Eğer Python bize len() fonksiyonu gibi bir fonksiyon vermemiş olsaydı, kendi len() fonksiyonumuzu icat edebilirdik. Dikkatlice bakın:

```
def uzunluk(öğe):
    c = 0
    for s in öğe:
        c += 1
    print(c)
```

Böylece adı *uzunluk* olan bir fonksiyon tanımlamış olduk. Artık bir öğenin uzunluğunu hesaplamak istediğimizde, bütün o kodları her defasında tekrar tekrar yazmak yerine sadece uzunluk() fonksiyonunu kullanabiliriz:

```
uzunluk("istihza")
uzunluk("Afyonkarahisar")
uzunluk("Tarım ve Köyişleri Bakanlığı")
```

Üstelik bu fonksiyon yalnızca karakter dizilerinin değil öteki veri tiplerinin de uzunluğunu hesaplayabilir:

```
liste = ["ahmet", "mehmet", "veli"]
uzunluk(liste)
```

Verdiğimiz bu örnek bize hem gömülü fonksiyonların faydasını, hem de genel olarak fonksiyonların ne işe yaradığını açıkça gösteriyor. Buna göre, len() benzeri gömülü fonksiyonlar tekerleği yeniden icat etme derdinden kurtarıyor bizi. Örneğin Python geliştiricilerinin len() gibi bir fonksiyon tanımlamış olmaları sayesinde, bir karakter dizisinin uzunluğunu hesaplamak için kendi kendimize yöntem icat etmek zorunda kalmıyoruz. Ama eğer kendi yöntemimizi icat etmemiz gerekirse, istediğimiz işlevi yerine getiren bir fonksiyon tanımlamamız da mümkün.

Böylece temel olarak fonksiyonların ne işe yaradığını, neye benzediğini, nasıl tanımlandığını ve nasıl çağrıldığını incelemiş olduk. Şimdi fonksiyonların biraz daha derinine dalmaya başlayabiliriz.

Parametreler ve Argümanlar

Şimdiye kadar yaptığımız örnekler sayesinde aslında parametrelerin neye benzediğini ve ne işe yaradığını öğrenmiştik. Bu bölümde ise sizi 'argüman' adlı bir kavramla tanıştırıp, argüman ile parametre arasındaki benzerlik ve farklılıkları inceleyeceğiz. Bunun yanısıra, parametre kavramını da bu bölümde daha derinlikli bir şekilde ele alacağız.

O halde hemen yola koyulalım.

Parametrenin ne olduğunu biliyorsunuz. Bunlar fonksiyon tanımlarken parantez içinde belirttiğimiz, fonksiyon gövdesinde yapılan işin değişken öğelerini gösteren parçalardır. Mesela:

```
def kopyala(kaynak_dosya, hedef_dizin):
    çıktı = "{} adlı dosya {} adlı dizin içine kopyalandı!"
    print(çıktı.format(kaynak_dosya, hedef_dizin))
```

Burada kopyala() adlı bir fonksiyon tanımladık. Bu fonksiyon toplam iki adet parametre alıyor: *kaynak_dosya* ve *hedef_dizin*. Gördüğünüz gibi, bu iki parametre gerçekten de fonksiyon gövdesinde yapılan işin değişken öğelerini gösteriyor. Bu fonksiyonun üreteceği çıktı, fonksiyonu çağıran kişinin bu iki parametreye vereceği değerlere bağlı olarak şekillenecek.

Bildiğiniz gibi, parametrelere ne ad verdiğinizin hiçbir önemi yok. Elbette parametrenin görevine uygun bir isim vermeniz fonksiyonunuzun okunaklılığını artıracaktır. Ama tabii ki bu fonksiyonu pekala şu parametrelerle de tanımlayabilirdik:

```
def kopyala(a, b):
    çıktı = "{} adlı dosya {} adlı dizin içine kopyalandı!"
    print(çıktı.format(a, b))
```

Burada önemli olan, parametre görevi görecek iki adet kelime bulmak. Bu kelimelerin ne olduğunun önemi yok. Ama tabii ki *kaynak_dosya* ve *hedef_dizin* adları, *a* ve *b* adlarına kıyasla, fonksiyondaki parametrelerin yaptığı işi çok daha iyi tarif ediyor.

Parametre adı belirleme kuralları değişken adı belirleme kurallarıyla aynıdır. Dolayısıyla bir değişken adı belirlerken neye dikkat ediyorsak, parametre adı belirlerken de aynı şeye dikkat etmeliyiz.

Gelin şimdi isterseniz tanımladığınız bu fonksiyonu çağıralım:

```
kopyala("deneme.txt", "/home/istihza/Desktop")
```

Kodlarımız dosya içinde tam olarak şöyle görünüyor:

```
def kopyala(kaynak_dosya, hedef_dizin):
    çıktı = "{} adlı dosya {} adlı dizin içine kopyalandı!"

```

```
    print(çıktı.format(kaynak_dosya, hedef_dizin))

kopyala("deneme.txt", "/home/istihza/Desktop")
```

Bu kodları bir dosyaya kaydedip çalıştırdığımızda şöyle bir çıktı alırız:

```
deneme.txt adlı dosya /home/istihza/Desktop adlı dizin içine kopyalandı!
```

Gördüğünüz gibi, *"deneme.txt"* ve *"/home/istihza/Desktop"* değerleri, *çıktı* adlı karakter dizisinde uygun yerlere yerleştirildi ve ekrana çıktı olarak verildi. İşte burada gördüğünüz bu *"deneme.txt"* ve *"/home/istihza/Desktop"* değerlerine argüman adı verilir. Yani bir fonksiyonu **tanımlarken** belirlediğimiz adlara parametre, aynı fonksiyonu **çağırırken** belirlediğimiz adlara ise argüman deniyor. Dolayısıyla fonksiyon tanımında belirlediğimiz *kaynak_dosya* ve *hedef_dizin* adlı değişkenler birer parametre, fonksiyon çağrısında bu parametrelere karşılık gelen *"deneme.txt"* ve *"/home/istihza/Desktop"* değerleri ise birer argüman oluyor.

Böylece parametre ve argüman arasındaki farkı öğrenmiş olduk. Ancak şunu belirtmekte yarar var: Bu iki kavram genellikle birbirinin yerine kullanılır. Yani bu iki kavram arasındaki, yukarıda açıkladığımız farka pek kimse dikkat etmez. Dolayısıyla pek çok yerde hem parametre hem de argüman için aynı ifadenin kullanıldığını görebilirsiniz. Özellikle Türkçede parametre kelimesi argüman kelimesine kıyasla daha bilinir ve yaygın olduğu için, ayrım yapılmaksızın hem fonksiyon çağrısındaki değerlere, hem de fonksiyon tanımındaki değerlere parametre adı verilir.

Gelelim parametrelerin çeşitlerine...

Python'da parametreler işlevlerine göre farklı kategorilere ayrılır. Gelin şimdi bu kategorileri tek tek inceleyelim.

Sıralı (veya İsimsiz) Parametreler

Python'da şöyle bir fonksiyon tanımlayabileceğimizi biliyoruz:

```
def kayıt_oluştur(isim, soyisim, işsis, şehir):
    print("-"*30)

    print("isim           : ", isim)
    print("soyisim        : ", soyisim)
    print("işletim sistemi: ", işsis)
    print("şehir          : ", şehir)

    print("-"*30)
```

Yukarıda tanımladığımız bu fonksiyonu şu şekilde çağırabiliriz:

```
kayıt_oluştur("Ahmet", "Öz", "Debian", "Ankara")
```

Bu fonksiyonda, yazdığımız parametrelerin sırası büyük önem taşır. Mesela yukarıdaki fonksiyonu şöyle çağırdığımızı düşünün:

```
kayıt_oluştur("Debian", "Ankara", "Öz", "Ahmet")
```

Eğer fonksiyon parametrelerini bu sırayla kullanırsak aldığımız çıktı hatalı olacaktır:

```
------------------------------
isim          :  Debian
soyisim       :  Ankara
işletim sistemi:  Öz
şehir         :  Ahmet
------------------------------
```

Gördüğünüz gibi, isim, soyisim ve öteki bilgiler birbirine karışmış. İşte Python'da, veriliş sırası önem taşıyan bu tür parametrelere 'sıralı parametreler' (veya isimsiz parametreler) adı verilir.

İsimli Parametreler

Bir önceki bölümde verdiğimiz şu örneği yeniden ele alalım:

```
def kayıt_oluştur(isim, soyisim, işsis, şehir):
    print("-"*30)

    print("isim          : ", isim)
    print("soyisim       : ", soyisim)
    print("işletim sistemi: ", işsis)
    print("şehir         : ", şehir)

    print("-"*30)
```

Bu fonksiyonu çağırırken parametrelerin sırasını doğru vermenin, alacağımız çıktının düzgün olması bakımından büyük önem taşıdığını biliyoruz. Ancak özellikle parametre sayısının çok olduğu fonksiyonlarda parametre sırasını akılda tutmak zor olabilir. Böyle durumlarda parametreleri isimleri ile birlikte kullanmayı tercih edebiliriz:

```
kayıt_oluştur(soyisim="Öz", isim="Ahmet", işsis="Debian", şehir= "Ankara")
```

Böylece fonksiyon parametrelerini istediğimiz sıra ile kullanabiliriz. Ancak burada dikkat etmemiz gereken bazı noktalar var. Python'da isimli bir parametrenin ardından sıralı bir parametre gelemez. Yani şu kullanım yanlıştır:

```
kayıt_oluştur(soyisim="Öz", isim="Ahmet", "Debian", "Ankara")
```

Bu kodlar bize şu hatayı verir:

```
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
```

Bu yüzden, eğer isimli parametreler kullanacaksak, isimli parametrelerden sonra sıralı parametre kullanmamaya dikkat ediyoruz.

Varsayılan Değerli Parametreler

Şimdiye kadar karşılaştığımız fonksiyonlarda bir şey dikkatinizi çekmiş olmalı. Mesela print() fonksiyonunu ele alalım. Bildiğiniz gibi, bu fonksiyonu en basit şekilde şöyle kullanıyoruz:

```
print("Fırat", "Özgül")
```

Evet, print() fonksiyonunu bu şekilde kullanabiliyoruz, ancak bildiğiniz gibi, aslında bu fonksiyonun bazı özel parametreleri de var. Daha önceki derslerimizden hatırlayacağınız gibi, biz yukarıdaki komutu verdiğimizde aslında Python bunu şu şekilde algılıyor:

```
print("Fırat", "Özgül", sep=" ", end="\n", file=sys.stdout, flush=False)
```

Yani biz görmesek de aslında her print() çağrısı *sep*, *end*, *file* ve *flush* parametrelerini de içeriyor. Biz bu özel parametreleri kullanmasak da, yazdığımız kod düzgün bir şekilde çalışır. Bunun nedeni, *sep*, *end*, *file* ve *flush* parametrelerinin öntanımlı olarak birtakım değerlere sahip olmasıdır. Yani biz bu parametrelere kendimiz bir değer atamazsak Python bu parametrelere kendi belirlediği bazı öntanımlı değerleri atayacaktır. Dolayısıyla, eğer biz başka bir değer yazmazsak, *sep* parametresi *" "* değerine, *end* parametresi *"n"* değerine, *file* parametresi *sys.stdout* değerine, *flush* parametresi ise *False* değerine sahip olacaktır. İşte bu tür parametrelere Python'da 'varsayılan değerli parametreler' adı verilir. Peki biz kendimiz varsayılan değerli parametreler içeren fonksiyonları nasıl tanımlayabiliriz?

Şu örneğe dikkatlice bakın:

```
def kur(kurulum_dizini="/usr/bin/"):
    print("Program {} dizinine kuruldu!".format(kurulum_dizini))
```

Burada kur() adlı bir fonksiyon tanımladık. Bu fonksiyonun görevi, yazdığımız bir programı, kullanıcının bilgisayarındaki bir dizine kurmak ve programın hangi dizine kurulduğu konusunda kullanıcıyı bilgilendirmek. Bu fonksiyonu şu şekilde çağırabiliriz:

```
kur()
```

Eğer kur() fonksiyonunu böyle çağırırsak bize şu çıktıyı verecektir:

```
Program /usr/bin/ dizinine kuruldu!
```

Gördüğünüz gibi, kur() fonksiyonunun *kurulum_dizini* adlı bir parametresi var. Biz fonksiyonu

tanımlarken, bu parametreye bir varsayılan değer atadık (*/usr/bin/*). Böylece kur() fonksiyonu parametresiz olarak çağrıldığında bu varsayılan değer devreye girdi. Eğer biz bu değeri değiştirmek istersek, mesela programımızın "C:\Users\firat" dizinine kurulmasını istersek, kur() fonksiyonunu şöyle çağırmalıyız:

```
kur("C:\\Users\\firat")
```

kur() fonksiyonunu yukarıdaki gibi çağırdığımızda Python bize şöyle bir çıktı verir:

```
Program C:\Users\firat dizinine kuruldu!
```

Bu örnek size, varsayılan değerli parametreler belirlemenin ne kadar faydalı olabileceğini göstermiş olmalı. Mesela bir program yazdığınızı düşünün. Programınızı indiren kullanıcılar, yukarıdaki gibi bir varsayılan değerli parametre belirlemiş olmanız sayesinde programınızı nereye kuracaklarını belirlemek zorunda kalmadan bir sonraki kurulum adımına geçebiliyorlar...

Elbette eğer isterseniz kullanıcılarınızı bir kurulum dizini belirlemeye zorlamak da isteyebilirsiniz. Bunun için yine varsayılan değerli parametrelerden yararlanabilirsiniz:

```
def kur(kurulum_dizini=''):
    if not kurulum_dizini:
        print("Lütfen programı hangi dizine kurmak istediğinizi belirtin!")
    else:
        print("Program {} dizinine kuruldu!".format(kurulum_dizini))
```

Bu defa *kurulum_dizini* parametresinin varsayılan değerini boş bir karakter dizisi olarak belirledik. Eğer bu parametrenin değeri boş bir karakter dizisi olursa, kullanıcı herhangi bir kurulum dizini belirtmemiş demektir. Eğer kullanıcı herhangi bir kurulum dizini belirtmezse *kurulum_dizini* parametresinin bool değeri *False* olacaktır. Bu özelliği dikkate alarak fonksiyon gövdesinde şu kodları kullanabiliyoruz:

```
if not kurulum_dizini:
    print("Lütfen programı hangi dizine kurmak istediğinizi belirtin!")
```

Böylece, *kurulum_dizini* parametresinin bool değeri *False* olursa kullanıcılarımıza şöyle bir uyarı gösteriyoruz:

```
"Lütfen programı hangi dizine kurmak istediğinizi belirtin!"
```

Dolayısıyla kuruluma başlayabilmek için kur() fonksiyonunun şöyle çalıştırılmasını zorunlu tutuyoruz:

```
kur("C:\\Users\\istihza")
```

Buna benzer durumlarla pek çok kez karşılaşmış olmalısınız. Özellikle programların kurulmasını sağlayan '*setup*' betiklerinde her aşama için bir varsayılan değer belirlenip, kullanıcının sadece

'*Next*' tuşlarına basarak sağlıklı bir kurulum yapması sağlanabiliyor. Eğer kullanıcı varsayılan değerlerin dışında birtakım değerler belirlemek isterse, yukarıda örneğini verdiğimiz yapı kullanıcıya böyle bir özgürlük de sağlıyor.

Rastgele Sayıda İsimsiz Parametre Belirleme

Şimdiye kadar öğrendiğimiz pek çok fonksiyonun toplam kaç parametre alabileceği bellidir. Örneğin input() fonksiyonu yalnızca tek bir parametre alabilir. Eğer bu fonksiyona birden fazla parametre verirsek Python bize bir hata mesajı gösterecektir. Aynı şekilde mesela pow() fonksiyonunun da kaç parametre alabileceği bellidir. Ama örneğin print() fonksiyonuna verebileceğimiz parametre sayısı (teknik olarak 256 ile sınırlı olsa da) pratik olarak neredeyse sınırsızdır.

Peki acaba biz kendimiz, sınırsız parametre alabilen fonksiyonlar üretebilir miyiz?

Bu sorunun cevabı 'evet' olacaktır. Şimdi şu örneğe dikkatlice bakın:

```
def fonksiyon(*parametreler):
    print(parametreler)

fonksiyon(1, 2, 3, 4, 5)
```

Bu kodları çalıştırdığımızda şu çıktıyı alacağız:

```
(1, 2, 3, 4, 5)
```

Gördüğünüz gibi, fonksiyon tanımı içinde kullandığımız * işareti sayesinde fonksiyonumuzun pratik olarak sınırsız sayıda parametre kabul etmesini sağlayabiliyoruz. Bu arada, bu tür fonksiyonların alabileceği parametre sayısı, dediğimiz gibi, pratikte sınırsızdır, ama teknik olarak bu sayı 256 adedi geçemez.

Yukarıdaki kodların verdiği çıktının bir demet olduğuna dikkatinizi çekmek isterim. Bu bilgiye sahip olduktan sonra, bu tür fonksiyonları demet işleme kurallarına göre istediğiniz şekilde manipüle edebilirsiniz.

Peki böyle bir fonksiyon tanımlamak ne işimize yarar?

Mesela bu yapıyı kullanarak şöyle bir fonksiyon yazabilirsiniz:

```
def çarp(*sayılar):
    sonuç = 1
    for i in sayılar:
        sonuç *= i
    print(sonuç)
```

Bu fonksiyon kendisine verilen bütün parametreleri birbiriyle çarpar. Örneğin:

```
çarp(1, 2, 3, 4)
```

Bu kodun çıktısı *24* olacaktır. Gördüğünüz gibi, fonksiyonumuza istediğimiz sayıda parametre vererek bu sayıların birbiriyle çarpılmasını sağlayabiliyoruz.

Aslında burada kullandığımız * işareti size hiç yabancı değil. Hatırlarsanız print() fonksiyonundan bahsederken şuna benzer bir kullanım örneği vermiştik:

```
>>> print(*TBMM, sep='.')

T.B.M.M
```

Burada * işareti, eklendiği parametreyi öğelerine ayırıyor. *sep* parametresi ise * işaretinin birbirinden ayırdığı öğelerin arasına birer '.' karakteri ekliyor.

Bu işaretin etkilerini şu örneklerde daha net görebilirsiniz:

```
>>> liste = ["Ahmet", "Mehmet", "Veli"]
>>> print(*liste)

Ahmet Mehmet Veli

>>> sözlük = {"a": 1, "b": 2}
>>> print(*sözlük)

a b
```

Gördüğünüz gibi, * işareti herhangi bir öğeyi alıp, bunu parçalarına ayırıyor. İşte bu * işaretini fonksiyon tanımlarken kullandığımızda ise bu işlemin tam tersi gerçekleşiyor. Yani fonksiyon tanımında parametrenin soluna * getirdiğimizde, bu fonksiyon çağrılırken verilen argümanlar tek bir değişken içinde bir demet olarak toplanıyor. Zaten bu konunun başında verdiğimiz şu örnekte de bu durum açıkça görünüyor:

```
def fonksiyon(*parametreler):
    print(parametreler)

fonksiyon(1, 2, 3, 4, 5)
```

Bu fonksiyonu çağırdığımızda şu çıktı veriliyor:

```
(1, 2, 3, 4, 5)
```

Aynen söylediğimiz gibi, fonksiyon() adlı fonksiyona argüman olarak verdiğimiz her bir öğenin (*1*, *2*, *3*, *4*, *5*) tek bir demet içinde toplandığını görüyorsunuz.

Yıldızlı parametreler, tanımladığınız fonksiyonun parametre sayısını herhangi bir şekilde sınırlamak istemediğiniz durumlarda çok işinize yarar.

Elbette * işaretiyle birlikte kullanacağınız parametrenin adı olarak, Python'ın değişken adlandırma kurallarına uygun bütün kelimeleri belirleyebilirsiniz. Mesela biz yukarıda 'parametreler' adını tercih ettik. Ama Python dünyasında * işaretiyle birlikte kullanılacak parametrenin adı geleneksel olarak, 'argümanlar' anlamında 'args'tır. Yani Python programcıları genellikle yukarıdaki gibi bir fonksiyonu şöyle tanımlar:

```
def fonksiyon(*args):
    ...
```

* işareti ile birlikte kullanılacak parametrenin adını 'args' yapmak bir zorunluluk olmamakla birlikte, başka Python programcılarının kodlarınızı daha kolay anlayabilmesi açısından bu geleneği devam ettirmenizi tavsiye ederim. Yazdığımız kodlarda Python programlama dilinin geleneklerine bağlı kalmak çoğunlukla iyi bir alışkanlıktır.

Rastgele Sayıda İsimli Parametre Belirleme

Bir önceki başlık altında, fonksiyon tanımlarken rastgele sayıda isimsiz parametrelerin nasıl belirleneceğini tartıştık. Aynı bu şekilde, rastgele sayıda **isimli** parametre belirlemek de mümkündür.

Örneğin:

```
def fonksiyon(**parametreler):
    print(parametreler)

fonksiyon(isim="Ahmet", soyisim="Öz", meslek="Mühendis", şehir="Ankara")
```

Bu kodları çalıştırdığımızda şöyle bir çıktı alıyoruz:

```
{'şehir': 'Ankara', 'isim': 'Ahmet', 'soyisim': 'Öz', 'meslek': 'Mühendis'}
```

Gördüğünüz gibi, fonksiyonu tanımlarken parametremizin sol tarafına yerleştirdiğimiz ** işareti, bu fonksiyonu çağırırken yazdığımız isimli parametrelerin bize bir sözlük olarak verilmesini sağlıyor. Bu yapının bize bir sözlük verdiğini bildikten sonra, bunu sözlük veri tipinin kuralları çerçevesinde istediğimiz şekilde evirip çevirebiliriz.

Peki bu araç ne işimize yarar?

Hatırlarsanız bu bölümün en başında kayıt_oluştur() adlı şöyle bir fonksiyon tanımlamıştık:

```
def kayıt_oluştur(isim, soyisim, işsis, şehir):
    print("-"*30)

    print("isim          : ", isim)
```

```
    print("soyisim       : ", soyisim)
    print("işletim sistemi: ", işsis)
    print("şehir         : ", şehir)

    print("-"*30)
```

Bu fonksiyon bize toplam dört adet parametre kullanarak, isim, soyisim, işletim sistemi ve şehir bilgilerinden meydana gelen bir kayıt oluşturma imkanı sağlıyor. Bu fonksiyonda kullanıcının girebileceği bilgiler sınırlı. Ama bir de şöyle bir fonksiyon yazdığımızı düşünün:

```
def kayıt_oluştur(**bilgiler):
    print("-"*30)

    for anahtar, değer in bilgiler.items():
        print("{:<10}: {}".format(anahtar, değer))

    print("-"*30)

kayıt_oluştur(ad="Fırat", soyad="Özgül", şehir="İstanbul", tel="05333213232")
```

Bu fonksiyonu çalıştırdığımızda şu çıktıyı alacağız:

```
tel       : 05333213232
ad        : Fırat
şehir     : İstanbul
soyad     : Özgül
```

Gördüğünüz gibi, ** işaretlerini kullanmamız sayesinde hem adlarını hem de değerlerini kendimiz belirlediğimiz bir kişi veritabanı oluşturma imkanı elde ediyoruz. Üstelik bu veritabanının, kişiye ait kaç farklı bilgi içereceğini de tamamen kendimiz belirleyebiliyoruz.

Tıpkı * işaretlerinin betimlediği parametrenin geleneksel olarak 'args' şeklinde adlandırılması gibi, ** işaretlerinin betimlediği parametre de geleneksel olarak 'kwargs' şeklinde adlandırılır. Dolayısıyla yukarıdaki gibi bir fonksiyonu Python programcıları şöyle tanımlar:

```
def kayıt_oluştur(**kwargs):
    ...
```

** işaretli parametreler pek çok farklı durumda işinize yarayabilir veya işinizi kolaylaştırabilir. Mesela * ve ** işaretlerini kullanarak şöyle bir program yazabilirsiniz:

```
def karşılık_bul(*args, **kwargs):
    for sözcük in args:
        if sözcük in kwargs:
            print("{} = {}".format(sözcük, kwargs[sözcük]))
```

```
        else:
            print("{} kelimesi sözlükte yok!".format(sözcük))


sözlük = {"kitap"      : "book",
          "bilgisayar" : "computer",
          "programlama": "programming"}

karşılık_bul("kitap", "bilgisayar", "programlama", "fonksiyon", **sözlük)
```

Burada tanımladığımız karşılık_bul() adlı fonksiyon, kendisine verilen parametreleri (**args*), bir sözlük içinde arayarak (***sözlük*) karşılıklarını bize çıktı olarak veriyor. Eğer verilen parametre sözlükte yoksa, ilgili kelimenin sözlükte bulunmadığı konusunda da bizi bilgilendiriyor.

karşılık_bul() adlı fonksiyonu nasıl tanımladığımıza çok dikkat edin. Parametre listesi içinde belirttiğimiz **args* ifadesi sayesinde, fonksiyonu kullanacak kişiye, istediği sayıda isimsiz parametre girme imkanı tanıyoruz. ***kwargs* parametresi ise kullanıcıya istediği sayıda isimli parametre girme olanağı veriyor.

Esasında yukarıdaki kod **args* ve ***kwargs* yapıları açısından ucuz bir örnektir. Bu yapılar için daha nitelikli bir örnek verelim...

Bildiğiniz gibi print() fonksiyonu sınırsız sayıda isimsiz parametre ve buna ek olarak birkaç tane de isimli parametre alıyor. Bu fonksiyonun alabildiği isimli parametrelerin *sep*, *end*, *file* ve *flush* adlı parametreler olduğunu biliyorsunuz. Yine bildiğiniz gibi, *sep* parametresi print() fonksiyonuna verilen isimsiz parametrelerin her birinin arasına hangi karakterin geleceğini; *end* parametresi ise bu parametrelerin en sonuna hangi karakterin geleceğini belirliyor. Bizim amacımız bu fonksiyona bir de *start* adında isimli bir parametre ekleyerek print() fonksiyonunun işlevini genişleten başka bir fonksiyon yazmak. Bu yeni parametre, karakter dizilerinin **en başına** hangi karakterin geleceğini belirleyecek.

Şimdi bu amacımızı gerçekleştirecek kodlarımızı yazalım:

```
def bas(*args, start='', **kwargs):
    for öğe in args:
        print(start+öğe, **kwargs)

bas('öğe1', 'öğe2', 'öğe3', start="#.")
```

print() fonksiyonunun işlevini genişleten yeni fonksiyonumuzun adı bas(). Bu fonksiyon her bakımdan print() fonksiyonu ile aynı işlevi görecek. Ancak bas() fonksiyonu, print() fonksiyonuna ek olarak, sahip olduğu *start* adlı bir isimli parametre sayesinde, kendisine verilen parametrelerin **en başına** istediğimiz herhangi bir karakteri eklemek olanağı da verecek bize.

bas() fonksiyonunun ilk parametresi olan **args* sayesinde kullanıcıya istediği kadar parametre verme imkanı tanıyoruz. Daha sonra da ilave *start* parametresini tanımlıyoruz. Bu parametrenin öntanımlı değeri boş bir karakter dizisi. Yani eğer kullanıcı bu parametrenin değerine herhangi bir şey yazmazsa, **args* kapsamında verilen parametreler üzerinde hiçbir değişiklik yapmıyoruz. Bunun ardından gelen ***kwargs* parametresi ise print() fonksiyonunun halihazırda sahip olduğu *sep*, *end*, *file* ve *flush* parametrelerinin bas() fonksiyonunda da aynı şekilde kullanılmasını sağlıyor. ***kwargs* şeklinde bir tanımlama sayesinde, print() fonksiyonunun isimli parametrelerini tek tek belirtip tanımlamak zorunda kalmıyoruz:

```
def bas(*args, start='', **kwargs):
    for öğe in args:
        print(start+öğe, **kwargs)

f = open("te.txt", "w")

bas('öğe1', 'öğe2', 'öğe3', start="#.", end="", file=f)
```

Eğer elimizde ***kwargs* gibi bir imkan olmasaydı yukarıdaki fonksiyonu şu şekilde tanımlamamız gerekirdi:

```
import sys

def bas(*args, start='', sep=' ', end='\n', file=sys.stdout, flush=False):
    for öğe in args:
        print(start+öğe, sep=sep, end=end, file=file, flush=flush)
```

Gördüğünüz gibi, print() fonksiyonunun bütün isimli parametrelerini ve bunların öntanımlı değerlerini tanımlamak zorunda kaldık. Eğer günün birinde Python geliştiricileri print() fonksiyonuna bir başka isimli parametre daha eklerse, yukarıdaki fonksiyonu ilgili yeniliğe göre elden geçirmemiz gerekir. Ama ***kwargs* yapısını kullandığımızda, print() fonksiyonuna Python geliştiricilerince eklenecek bütün parametreler bizim fonksiyonumuza da otomatik olarak yansıyacaktır...

return Deyimi

Bu bölümde return adlı bir deyimden söz edeceğiz. Özellikle Python programlama dilini öğrenmeye yeni başlayanlar bu deyimin ne işe yaradığını anlamakta zorlanabiliyor. Biz burada bu deyimi anlaşılır hale getirebilmek için elimizden geleni yapacağız. Öncelikle çok basit bir örnek verelim:

```
def ismin_ne():
    isim = input("ismin ne? ")
    print(isim)
```

Bu çok basit bir fonksiyon. Bu fonksiyonu nasıl çağıracağımızı biliyoruz:

```
ismin_ne()
```

Fonksiyonu bu şekilde çağırdıktan sonra, fonksiyon tanımında yer alan input() fonksiyonu sayesinde kullanıcıya ismi sorulacak ve verdiği cevap ekrana basılacaktır.

Yukarıdaki fonksiyonun tek işlevi kullanıcıdan aldığı isim bilgisini ekrana basmaktır. Aldığınız bu veriyi başka yerlerde kullanamazsınız. Bu fonksiyonu çağırdığınız anda kullanıcıya ismi sorulacak ve alınan cevap ekrana basılacaktır. Ancak siz, tanımladığınız fonksiyonların tek görevinin bir veriyi ekrana basmak olmasını istemeyebilirsiniz.

Örneğin yukarıdaki fonksiyon yardımıyla kullanıcıdan ismini aldıktan sonra, bu isim bilgisini başka bir karakter dizisi içinde kullanmak isteyebilirsiniz. Diyelim ki amacınız ismin_ne() fonksiyonuyla aldığınız ismi şu karakter dizisi içine aşağıdaki şekilde yerleştirmek:

```
Merhaba Fırat. Nasılsın?
```

Bildiğimiz yöntemi kullanarak bu amacımızı gerçekleştirmeye çalışalım:

```
print("Merhaba {}. Nasılsın?".format(ismin_ne()))
```

Buradan şöyle bir çıktı alıyoruz:

```
ismin ne? Fırat
Fırat
Merhaba None. Nasılsın?
```

Gördüğünüz gibi, istediğimiz şeyi elde edemiyoruz. Çünkü dediğimiz gibi, yukarıdaki fonksiyonun tek görevi kullanıcıdan aldığı çıktıyı ekrana basmaktır. Bu fonksiyondan gelen çıktıyı başka yerde kullanamayız. Eğer kullanmaya çalışırsak yukarıdaki gibi hiç beklemediğimiz bir sonuç alırız.

Bu arada, çıktıda *None* diye bir şey gördüğünüze dikkat edin. Yukarıdaki fonksiyonu şu şekilde çağırarak bunu daha net görebilirsiniz:

```
print(ismin_ne())
```

Buradan şu çıktıyı alıyoruz:

```
ismin ne? Fırat
Fırat
None
```

Bu çıktının ne anlama geldiğini birazdan açıklayacağız. Ama öncelikle başka bir konudan söz edelim.

Biraz önce söylediğimiz gibi, yukarıda tanımladığımız ismin_ne() adlı fonksiyonun tek görevi kullanıcıdan aldığı isim bilgisini ekrana basmaktır. Şimdi bu fonksiyonu bir de şöyle tanımlayalım:

```
def ismin_ne():
    isim = input("ismin ne? ")
    return isim
```

Şimdi de bu fonksiyonu çağıralım:

```
ismin_ne()
```

Gördüğünüz gibi, fonksiyonu çağırdığımızda yalnızca fonksiyon gövdesindeki input() fonksiyonu çalıştı, ama bu fonksiyondan gelen veri ekrana çıktı olarak verilmedi. Çünkü biz burada herhangi bir ekrana basma ('print') işlemi yapmadık. Yaptığımız tek şey *isim* adlı değişkeni 'döndürmek'.

Peki bu ne anlama geliyor?

return kelimesi İngilizcede 'iade etmek, geri vermek, döndürmek' gibi anlamlar taşır. İşte yukarıdaki örnekte de return deyiminin yaptığı iş budur. Yani bu deyim bize fonksiyondan bir değer 'döndürür'.

Eğer tanımladığımız bir fonksiyonda return deyimini kullanarak herhangi bir değer döndürmezsek, Python fonksiyondan hususi bir değerin döndürülmediğini göstermek için 'None' adlı bir değer döndürür... İşte yukarıda tanımladığımız ilk ismin_ne() fonksiyonunu print(ismin_ne()) şeklinde çağırdığımızda ekranda *None* değerinin görünmesinin nedeni budur.

Peki bir fonksiyon içinde herhangi bir veriyi ekrana basmayıp return deyimi yardımıyla döndürmemizin bize ne faydası var?

Aslında bunun cevabı çok açık. Bir fonksiyon içinde bir değeri döndürmek yerine ekrana bastığınızda o fonksiyonun işlevini alabildiğine kısıtlamış oluyorsunuz. Fonksiyonunuzun tek işlevi bir değeri ekrana basmak oluyor. Şu örnekte de gösterdiğimiz gibi, bu değeri daha sonra başka ortamlarda kullanamıyoruz:

```
def ismin_ne():
    isim = input("ismin ne? ")
    print(isim)

print("Merhaba {}. Nasılsın?".format(ismin_ne()))
```

Ama eğer, mesela yukarıdaki fonksiyonda *isim* değişkenini basmak yerine döndürürsek işler değişir:

```
def ismin_ne():
    isim = input("ismin ne? ")
    return isim

print("Merhaba {}. Nasılsın?".format(ismin_ne()))
```

Bu kodları çalıştırdığımızda şu çıktıyı alıyoruz:

```
ismin ne? Fırat
Merhaba Fırat. Nasılsın?
```

Gördüğünüz gibi, istediğimiz çıktıyı rahatlıkla elde ettik. ismin_ne() adlı fonksiyondan *isim* değerini döndürmüş olmamız sayesinde bu değerle istediğimiz işlemi gerçekleştirebiliyoruz. Yani bu değeri sadece ekrana basmakla sınırlamıyoruz kendimizi. Hatta fonksiyondan döndürdüğümüz değeri başka bir değişkene atama imkanına dahi sahibiz bu şekilde:

```
ad = ismin_ne()
print(ad)
```

Eğer fonksiyondan değer döndürmek yerine bu değeri ekrana basmayı tercih etseydik yukarıdaki işlemi yapamazdık.

return deyimiyle ilgili son bir şey daha söyleyelim...

Bu deyim, içinde bulunduğu fonksiyonun çalışma sürecini kesintiye uğratır. Yani return deyimini kullandığınız satırdan sonra gelen hiçbir kod çalışmaz. Basit bir örnek verelim:

```
def fonk():
    print(3)
    return
    print(5)

fonk()
```

Bu kodları çalıştırdığınızda yalnızca print(3) satırının çalıştığını, print(5) satırına ise hiç ulaşılmadığını göreceksiniz. İşte bu durumun sebebi, Python'ın kodları return satırından itibaren okumayı bırakmasıdır. Bu özellikten çeşitli şekillerde yararlanabilirsiniz. Örneğin:

```
def fonk(n):
    if n < 0:
        return 'eksi değerli sayı olmaz!'
    else:
        return n

f = fonk(-5)
print(f)
```

Burada eğer fonksiyona parametre olarak eksi değerli bir sayı verilirse Python bize bir uyarı verecek ve fonksiyonun çalışmasını durduracaktır.

Örnek bir Uygulama

Gelin isterseniz buraya kadar öğrendiklerimizi kullanarak örnek bir uygulama yazalım. Bir yandan da yeni şeyler öğrenerek bilgimize bilgi katalım.

Amacımız belli miktarda ve belli aralıkta rastgele sayılar üreten bir program yazmak. Örneğin programımız şu şekilde altı adet rastgele sayı üretebilecek:

```
103, 298, 152, 24, 91, 285
```

Ancak programımız bu sayıları üretirken her sayıdan yalnızca bir adet üretecek. Yani aynı seride bir sayıdan birden fazla bulunamayacak.

Dilerseniz öncelikle kodlarımızı görelim:

```
import random

def sayı_üret(başlangıç=0, bitiş=500, adet=6):
    sayılar = set()

    while len(sayılar) < adet:
        sayılar.add(random.randrange(başlangıç, bitiş))

    return sayılar
```

Esasında bu kodların (neredeyse) tamamını anlayabilecek kadar Python bilgisine sahipsiniz. Burada anlamamış olabileceğiniz tek şey *random* modülüdür. O yüzden gelin isterseniz bu modülden biraz söz edelim.

Biz henüz modül kavramını bilmiyoruz. Ama buraya gelene kadar birkaç konu altında modüllerle ilgili bazı örnekler de yapmadık değil. Örneğin şimdiye kadar yazdığımız programlardan öğrendiğimiz kadarıyla Python'da *os* ve *sys* adlı iki modülün bulunduğunu, bu modüllerin içinde, program yazarken işimize yarayacak pek çok değişken ve fonksiyon bulunduğunu ve bu fonksiyonları programlarımızda kullanabilmek için ilkin bu modülleri içe aktarmamız gerektiğini biliyoruz. İşte tıpkı *os* ve *sys* gibi, *random* da Python programlama dili bünyesinde bulunan modüllerden biridir. Bu modülün içinde, rastgele sayılar üretmemizi sağlayacak bazı fonksiyonlar bulunur. İşte randrange() de bu fonksiyonlardan biridir. Dilerseniz bu fonksiyonun nasıl kullanıldığını anlamak için etkileşimli kabukta birkaç deneme çalışması yapalım.

random modülünün içindeki araçları kullanabilmek için öncelikle bu modülü içe aktarmalıyız:

```
>>> import random
```

Acaba bu modülün içinde neler varmış?

```
>>> dir(random)

['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST',
```

```
'SystemRandom', 'TWOPI', '_BuiltinMethodType', '_MethodType', '_Sequence',
'_Set', '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
'__initializing__', '__loader__', '__name__', '__package__', '_acos', '_ceil',
'_cos', '_e', '_exp', '_inst', '_log', '_pi', '_random', '_sha512', '_sin',
'_sqrt', '_test', '_test_generator', '_urandom', '_warn', 'betavariate',
'choice', 'expovariate', 'gammavariate', 'gauss', 'getrandbits', 'getstate',
'lognormvariate', 'normalvariate', 'paretovariate', 'randint', 'random',
'randrange', 'sample', 'seed', 'setstate', 'shuffle', 'triangular', 'uniform',
'vonmisesvariate', 'weibullvariate']
```

Gördüğünüz gibi bu modülün içinde epey araç var. Gelin isterseniz bu araçlardan en sık kullanılanlarını tanıyalım.

Örneğin *random* modülü içinde bulunan sample() adlı fonksiyon herhangi bir dizi içinden istediğimiz sayıda rastgele numune almamızı sağlar:

```
>>> liste = ["ahmet", "mehmet", "sevgi", "sevim", "selin", "zeynep", "selim"]
>>> random.sample(liste, 2)

['sevim', 'ahmet']
```

Gördüğünüz gibi, yedi kişilik bir isim listesinden *2* adet rastgele numune aldık. Aynı işlemi tekrarlayalım:

```
>>> random.sample(liste, 2)

['sevgi', 'zeynep']

>>> random.sample(liste, 5)

['selin', 'zeynep', 'ahmet', 'selim', 'mehmet']
```

Numune alma işlemi tamamen rastgeledir. Ayrıca gördüğünüz gibi, listeden istediğimiz sayıda numune alabiliyoruz.

random modülü içinde bulunan shuffle() adlı başka bir fonksiyon, bir dizi içindeki öğelerin sırasını rastgele bir şekilde karıştırmamızı sağlar:

```
>>> liste = ["ahmet", "mehmet", "sevgi", "sevim", "selin", "zeynep", "selim"]
>>> random.shuffle(liste)
```

shuffle() fonksiyonu liste öğelerini yine aynı liste içinde değiştirdi. Değişikliği görmek için listeyi ekrana basabilirsiniz:

```
>>> liste
```

```
['selim', 'selin', 'ahmet', 'mehmet', 'sevim', 'sevgi', 'zeynep']
```

random modülü içinde bulunan bir başka fonksiyon ise randrange() fonksiyonudur. Bu fonksiyon, belli bir aralıkta rastgele sayılar üretmemizi sağlar:

```
>>> random.randrange(0, 500)

156
```

Burada *0* ile *500* arasında rastgele bir sayı ürettik.

Gördüğünüz gibi *random* son derece faydalı olabilecek bir modüldür. Dilerseniz şimdi *random* fonksiyonunu bir kenara bırakıp kodlarımıza geri dönelim:

```
import random

def sayı_üret(başlangıç=0, bitiş=500, adet=6):
    sayılar = set()

    while len(sayılar) < adet:
        sayılar.add(random.randrange(başlangıç, bitiş))

    return sayılar
```

Burada ilk satırın ne iş yaptığını öğrendik. Bu satır yardımıyla *random* modülünü içe aktarıyoruz.

Sonraki satırda fonksiyonumuzu tanımlamaya başlıyoruz:

```
def sayı_üret(başlangıç=0, bitiş=500, adet=6):
    ...
```

Fonksiyonumuzun adı *sayı_üret*. Bu fonksiyon toplam üç farklı parametre alıyor. Bunlar *başlangıç*, *bitiş* ve *adet*. Dikkat ederseniz bu parametrelerin her birinin bir varsayılan değeri var. Dolayısıyla sayı_üret() fonksiyonu parametresiz olarak çağrıldığında bu üç parametre öntanımlı değerlerine sahip olacaktır.

Gelelim fonksiyon gövdesine...

İlk olarak *sayılar* adlı bir küme tanımlıyoruz.

Bildiğiniz gibi, kümeler içinde öğeler her zaman tektir. Yani bir küme içinde aynı öğeden yalnızca bir adet bulunabilir. Kümelerin bu özelliği bizim yazdığımız program için oldukça uygun. Çünkü biz de ürettiğimiz rastgele sayıların benzersiz olmasını istiyoruz. Bu benzersizliği sağlayabilecek en uygun veri tipi kümelerdir.

Bir sonraki satırda bir while döngüsü görüyoruz:

```
while len(sayılar) < adet:
    sayılar.add(random.randrange(başlangıç, bitiş))
```

Bu döngüye göre, *sayılar* değişkeninin uzunluğu *adet* parametresinin değerinden az olduğu müddetçe, *sayılar* adlı değişkene *başlangıç* ve *bitiş* parametrelerinin gösterdiği değerler arasından rastgele sayılar eklemeye devam edeceğiz. Örneğin kullanıcı fonksiyonumuzu parametresiz olarak çağırdıysa, yukarıdaki döngü şu şekilde işleyecektir:

```
while len(sayılar) < 6:
    sayılar.add(random.randrange(0, 500))
```

Buna göre, *sayılar* değişkeninin uzunluğu *6*'dan az olduğu müddetçe bu değişkene *0* ile *500* arasında rastgele sayılar eklemeye devam edeceğiz. Böylelikle *sayılar* değişkeni içinde birbirinden farklı toplam *6* sayı olmuş olacak.

Fonksiyonun son satırında ise şu kodu görüyoruz:

```
return sayılar
```

Bu kod yardımıyla, belirtilen miktardaki sayıları tutan *sayılar* adlı değişkeni fonksiyondan döndürüyoruz. Yani fonksiyonumuz dış dünyaya *sayılar* adlı bir değişken veriyor... Bu değişkeni bu şekilde döndürdükten sonra istediğimiz gibi kullanabiliriz. Mesela:

```
for i in range(100):
    print(sayı_üret())
```

Buradan şuna benzer bir çıktı alacaksınız:

```
{34, 144, 211, 468, 58, 286}
{41, 170, 395, 113, 178, 29}
{161, 195, 452, 271, 212, 324}
{1, 328, 461, 398, 464, 220}
{356, 489, 12, 114, 329, 472}
{320, 34, 238, 176, 243, 149}
{364, 304, 434, 403, 217, 63}
{452, 392, 175, 464, 81, 467}
{36, 230, 21, 440, 287, 415}
{292, 391, 145, 182, 440, 223}
{386, 38, 309, 377, 59, 277}
{0, 2, 42, 400, 404, 60}
{48, 482, 393, 80, 116, 407}
{483, 136, 431, 35, 344, 381}
...
```

Gördüğünüz gibi, sayı_üret() fonksiyonunu kullanarak, her biri *6* öğeden oluşan *100* adet sayı listesi elde ettik. Biz yukarıda bu fonksiyonu parametresiz olarak çalıştırdığımız için, Python *başlangıç*, *bitiş* ve *adet* parametrelerinin öntanımlı değerlerini kullandı (sırasıyla *0*, *500* ve *6*).

İstersek biz fonksiyonumuzu farklı parametrelerle çağırabiliriz:

```
print(sayı_üret(0, 100, 10))
```

Bu kodlar bize *0* ile *100* arasından *10* adet rastgele sayı seçer:

```
{3, 4, 9, 11, 13, 47, 50, 53, 54, 61}
```

Eğer çıktının küme parantezleri arasında görünmesini istemiyorsanız elbette çıktıyı keyfinize göre biçimlendirebilirsiniz:

```
print(*sayı_üret(100, 1500, 20), sep='-')
```

Bu şekilde, *100* ile *1500* arası sayılardan rastgele *20* adet seçip her bir sayının arasına bir tane - işareti yerleştirdik:

```
352-1251-1366-1381-1350-330-203-842-269-285-816-658-643-308-1174-152-594-522-1214-959
```

Fonksiyonların Kapsamı ve global Deyimi

Elimizde şöyle bir kod olduğunu düşünelim:

```
x = 0

def fonk():
    x = 1
    return x
```

Bu kodlarda, fonksiyonun dışında *x* adlı bir değişken var. Fonksiyonun içinde de yine *x* adını taşıyan başka bir değişken var. Fonksiyonumuzun görevi bu *x* değişkenini döndürmek.

Bu noktada size şöyle bir soru sormama izin verin: Acaba fonksiyon içinde tanımladığımız *x* değişkeni, fonksiyon dışındaki *x* değişkeninin değerini değiştiriyor mu? Bu sorunun cevabını şu kodlarla verelim:

```
x = 0

def fonk():
    x = 1
    return x
```

```
print('fonksiyon içindeki x: ', fonk())
print('fonksiyon dışındaki x: ', x)
```

Bu kodları çalıştırdığımızda şu çıktıyı alacağız:

```
fonksiyon içindeki x:  1
fonksiyon dışındaki x:  0
```

Gördüğünüz gibi fonksiyon içindeki ve fonksiyon dışındaki aynı adlı değişkenler birbirine karışmıyor. Bunun sebebi, Python'daki 'isim alanı' (*namespace*) adlı bir kavramdır.

Peki isim alanı ne demek?

Python'da değişkenlerin, fonksiyonların ve daha sonra göreceğiniz gibi sınıfların bir kapsamı vardır. Bu kapsama Python'da 'isim alanı' adı verilir. Dolayısıyla Python'da her nesnenin, geçerli ve etkin olduğu bir isim alanı bulunur. Örneğin yukarıdaki kodlarda fonksiyon dışındaki *x* değişkeni ana isim alanında yer alan 'global' bir değişkendir. Fonksiyon içindeki *x* değişkeni ise fonk() değişkeninin isim alanı içinde yer alan 'lokal' bir değişkendir. Bu iki değişken, adları aynı da olsa, birbirlerinden farklı iki nesnedir.

Bir de şu örneklere bakalım:

```
x = []
print('x\'in ilk hali:', x)

def değiştir():
    print('x\'i değiştiriyoruz...')
    x.append(1)
    return x

değiştir()
print('x\'in son hali: ', x)
```

Burada ise daha farklı bir durum söz konusu. Fonksiyon içinde append() metodunu kullanarak yaptığımız ekleme işlemi fonksiyon dışındaki listeyi de etkiledi. Peki ama bu nasıl oluyor?

Python herhangi bir nesneye göndermede bulunduğumuzda, yani o nesnenin değerini talep ettiğimizde aradığımız nesneyi ilk önce mevcut isim alanı içinde arar. Eğer aranan nesneyi mevcut isim alanı içinde bulamazsa yukarıya doğru bütün isim alanlarını tek tek kontrol eder.

Birkaç örnek verelim:

```
def fonk():
    print(x)
```

```
fonk()
```

Tahmin edebileceğiniz gibi, bu kodlar şu hatayı verecektir:

```
Traceback (most recent call last):
  File "deneme.py", line 4, in <module>
    fonk()
  File "deneme.py", line 2, in fonk
    print(x)
NameError: global name 'x' is not defined
```

Bu hatanın sebebi, *x* adlı bir değişkenin tanımlanmamış olmasıdır. Bu hatayı gidermek için şöyle bir kod yazabiliriz:

```
x = 0

def fonk():
    print(x)

fonk()
```

Bu kod global alandaki *x* değişkeninin değerini verecektir.

Yukarıdaki örnekte, biz print() ile *x*'in değerini sorguladığımızda Python öncelikle fonk() adlı fonksiyonun isim alanına baktı. Orada *x*'i bulamayınca bu kez global alana yönelip, orada bulduğu *x*'in değerini yazdırdı.

Bu durumu daha net anlayabilmek için şu kodları inceleyelim:

```
x = 0

def fonk():
    x = 10
    print(x)

fonk()
print(x)
```

Bu kodları çalıştırdığımızda *10* çıktısını alırız. Çünkü Python, dediğimiz gibi, öncelikle mevcut isim alanını kontrol ediyor. *x* değişkenini mevcut isim alanında bulduğu için de global alana bakmasına gerek kalmıyor.

Yalnız burada dikkat etmemiz gereken bazı şeyler var.

Dediğimiz gibi, global isim alanındaki nesnelerin değerini lokal isim alanlarından sorgulayabiliyoruz. Ancak istediğimiz şey global isim alanındaki nesnelerin değerini değiştirmekse bazı kavramlar arasındaki farkları iyi anlamamız gerekiyor.

Python'da bir nesnenin değerini değiştirmekle, o nesneyi yeniden tanımlamak farklı kavramlardır.

Eğer bir nesne değiştirilebilir bir nesne ise, o nesnenin değerini, lokal isim alanlarından değiştirebilirsiniz:

```
x = set()

def fonk():
    x.add(10)
    return x

print(fonk())
```

Ama eğer bir nesne değiştirilemez bir nesne ise, o nesnenin değerini zaten normalde de değiştiremezsiniz. Değiştirmiş gibi yapmak için ise o nesneyi yeniden tanımlamanız gerektiğini biliyorsunuz:

```
>>> isim = 'Fırat'
>>> isim += ' Özgül'
>>> print(isim)

Fırat Özgül
```

Burada yaptığımız şey, karakter dizisinin değerini değiştirmekten ziyade bu karakter dizisini yeniden tanımlamaktır. Çünkü bildiğiniz gibi karakter dizileri değiştirilemeyen veri tipleridir.

İşte karakter dizileri gibi değiştirilemeyen nesneleri, lokal isim alanlarında değiştiremeyeceğiniz gibi, yeniden tanımlayamazsınız da...

```
isim = 'Fırat'

def fonk():
    isim += ' Özgül'
    return isim

print(fonk())
```

Bu kodları çalıştırdığınızda Python size bir hata mesajı gösterecektir.

Aynı durum değiştirilebilir nesneler için de geçerlidir:

```
isim_listesi = []

def fonk():
    isim_listesi += ['Fırat Özgül', 'Orçun Kunek']
    return isim_listesi

print(fonk())
```

Değiştirilebilen bir veri tipi olan listeleri, fonksiyon içinde yeniden tanımlayamazsınız. Ancak tabii isterseniz listeleri değişikliğe uğratabilirsiniz:

```
isim_listesi = []

def fonk():
    isim_listesi.extend(['Fırat Özgül', 'Orçun Kunek'])
    return isim_listesi

print(fonk())
```

Bu kodlar düzgün bir şekilde çalışıp, fonksiyon dışındaki *isim_listesi* adlı listeyi değişikliğe uğratacaktır. Ancak şu kodlar hata verecektir:

```
isim_listesi = []

def fonk():
    isim_listesi += ['Fırat Özgül', 'Orçun Kunek']
    return isim_listesi

print(fonk())
```

İşte Python programlama dili bu tür durumlar için çözüm olacak bir araç sunar bize. Bu aracın adı *global*.

Gelin isterseniz bu *global* adlı deyimin nasıl kullanılacağına bakalım önce...

Şu kodların hata vereceğini biliyorsunuz:

```
isim = 'Fırat'

def fonk():
    isim += ' Özgül'
    return isim

print(fonk())
```

Ama bu kodlara şöyle bir ekleme yaparsanız işler değişir:

```
isim = 'Fırat'

def fonk():
    global isim
    isim += ' Özgül'
    return isim

print(fonk())
```

Burada fonk() adlı fonksiyonun ilk satırında şöyle bir kod görüyoruz:

```
global isim
```

İşte bu satır, *isim* adlı değişkenin global alana taşınmasını sağlıyor. Böylece global alanda bulunan *isim* adlı değişkeni değişikliğe uğratabiliyoruz.

global deyimi her ne kadar ilk bakışta çok faydalı bir araçmış gibi görünse de aslında programlarımızda genellikle bu deyimi kullanmaktan kaçınmamız iyi bir fikir olacaktır. Çünkü bu deyim aslında global alanı kirletmemize neden oluyor. Global değişkenlerin lokal isim alanlarında değişikliğe uğratılması, eğer dikkatsiz davranırsanız programlarınızın hatalı çalışmasına yol açabilir.

başa dön

Gömülü Fonksiyonlar

Bu bölümde, daha önce de birkaç kez bahsettiğimiz ve çokça örneğini gördüğümüz bir kavramdan söz edeceğiz. Bu kavramın adı ‘gömülü fonksiyonlar’.

Esasında biz buraya gelene kadar Python’da pek çok gömülü fonksiyon gördük. Dolayısıyla aslında görünüş olarak bunların neye benzediğini biliyoruz. Örneğin daha önceki derslerimizde gördüğümüz print() gömülü bir fonksiyondur. Aynı şekilde open(), type(), len(), pow(), bin() ve şimdiye kadar tanıştığımız öteki bütün fonksiyonlar birer gömülü fonksiyondur.

Gömülü fonksiyonlar İngilizcede *builtin functions* olarak adlandırılır. Bu fonksiyonlar gerçekten de dile gömülü vaziyettedirler. Bildiğiniz gibi, bir fonksiyonu kullanabilmemiz için o fonksiyonu tanımlamamız gerekir. İşte gömülü fonksiyonlar, bizim tanımlamamıza gerek kalmadan, Python geliştiricileri tarafından önceden tanımlanıp dile gömülmüş ve hizmetimize sunulmuş faydalı birtakım araçlardır.

İşte bu bölümde biz de bu gömülü fonksiyonları tek tek ve ayrıntılı olarak inceleyeceğiz. Dediğimiz gibi, bunlardan bir kısmını halihazırda görmüştünüz. Ama biz bütünlük açısından, önceden ele almış olduğumuz bu fonksiyonlara da kısaca değinmeden geçmeyeceğiz. Böylelikle hem yeni fonksiyonlar öğrenmiş olacağız hem de önceden öğrendiğimiz fonksiyonlarla birlikte yeni fonksiyonları da derli toplu bir şekilde görme imkanımız olacak.

Bu bölümde elbette birtakım fonksiyonları salt art arda sıralamakla yetinmeyeceğiz. Python’daki gömülü fonksiyonları incelerken bir yandan da Python programlama dilindeki çok önemli bazı kavramları ele alacağız.

İlk olarak abs() adlı bir fonksiyonla başlıyoruz gömülü fonksiyonları incelemeye...

abs()

İngilizcede ‘mutlak’ anlamına gelen *absolute* adlı bir kelime bulunur. İşte bu fonksiyonun adı da bu kelimeden gelir. Fonksiyonumuzun görevi de isminin anlamına yakındır. abs() fonksiyonunu bir sayının mutlak değerini elde etmek için kullanıyoruz.

Peki ‘mutlak değer’ ne anlama geliyor. Esasında siz bu kavrama matematik derslerinden aşinasınız. Ama bilmeyenler veya unutmuş olanlar için tekrar edelim. ‘Mutlak değer’ bir sayının *0*‘a olan uzaklığıdır. Örneğin *20* sayısının *0* sayısına olan uzaklığı 20’dir. Dolayısıyla *20* sayısının mutlak değeri 20’dir. Aynı şekilde *-20* sayısının da *0* sayısına uzaklığı 20’dir. Yani, *-20* sayısının da mutlak değeri 20’dir.

İşte abs() fonksiyonu bize bir sayının mutlak değerinin ne olduğunu söyler:

```
>>> abs(-20)
```

```
20

>>> abs(20)

20

>>> abs(20.0)

20.0
```

Mutlak değer kavramı yalnızca tamsayılar ve kayan noktalı sayılar için değil, aynı zamanda karmaşık sayılar için de geçerlidir. Dolayısıyla abs() fonksiyonunu kullanarak karmaşık sayıların da mutlak değerini hesaplayabiliriz:

```
>>> abs(20+3j)

20.223748416156685
```

Gördüğünüz gibi bu fonksiyon yalnızca tek bir parametre alıyor ve bu parametrenin mutlak değerini döndürüyor.

round()

round() fonksiyonu bir sayıyı belli ölçütlere göre yukarı veya aşağı doğru yuvarlamamızı sağlar. Basit birkaç örnek verelim:

```
>>> round(12.4)

12

>>> round(12.7)

13
```

Gördüğünüz gibi bu fonksiyon, kayan noktalı sayıları en yakın tam sayıya doğru yuvarlıyor.

Ancak burada dikkat etmemiz gereken bir nokta var.

Şu örnekleri bir inceleyelim:

```
>>> round(1.5)

2
```

```
>>> round(12.5)

12
```

Gördüğünüz gibi, fonksiyonumuz *1.5* sayısını yukarı doğru, *12.5* sayısını ise aşağı doğru yuvarladı. Bunun sebebi, kayan noktalı bir sayının üst ve alt tam sayılara olan uzaklığının birbirine eşit olduğu durumlarda Python'ın çift sayıya doğru yuvarlama yapmayı tercih etmesidir. Mesela yukarıdaki örneklerde *1.5* sayısı hem *1* sayısına, hem de *2* sayısına eşit uzaklıkta bulunuyor. İşte Python bu durumda, bir çift sayı olan *2* sayısına doğru yuvarlamayı tercih edecektir.

round() fonksiyonu toplam iki parametre alır. İlk parametre, yuvarlanacak sayının kendisidir. Yuvarlama hassasiyetini belirlemek için ise ikinci bir parametreden yararlanabiliriz.

Örneğin *22* sayısını *7*'ye böldüğümüzde normalde şöyle bir çıktı elde ederiz:

```
>>> 22/7

3.142857142857143
```

round() fonksiyonunu tek parametre ile kullandığımızda bu fonksiyon yukarıdaki sayıyı şu şekilde yuvarlayacaktır:

```
>>> round(22/7)

3
```

İşte biz round() fonksiyonuna ikinci bir parametre daha vererek, yuvarlama hassasiyetini kontrol edebiliriz.

Aşağıdaki örnekleri dikkatlice inceleyin:

```
>>> round(22/7)

3

>>> round(22/7, 0)

3.0

>>> round(22/7, 1)

3.1

>>> round(22/7, 2)
```

```
3.14

>>> round(22/7, 3)

3.143

>>> round(22/7, 4)

3.1429
```

Gördüğünüz gibi, round() fonksiyonuna verdiğimiz ikinci parametre, yuvarlama işleminin ne kadar hassas olacağını belirliyor.

all()

All kelimesi Türkçede 'hepsi' anlamına gelir. Bu fonksiyonun görevi de bu anlamı çağrıştırır. all() fonksiyonunun görevi, bir dizi içinde bulunan bütün değerler *True* ise *True* değeri, eğer bu değerlerden herhangi biri *False* ise de *False* değeri döndürmektir.

Örneğin elimizde şöyle bir liste olduğunu varsayalım:

```
>>> liste = [1, 2, 3, 4]
```

Şimdi all() fonksiyonunu bu liste üzerine uygulayalım:

```
>>> all(liste)

True
```

Bildiğiniz gibi, *0* hariç bütün sayıların bool değeri *True*'dur. Yukarıdaki listede *False* değeri verebilecek herhangi bir değer bulunmadığından, all() fonksiyonu bu liste için *True* değerini veriyor. Bir de şuna bakalım:

```
>>> liste = [0, 1, 2, 3, 4]
>>> all(liste)

False
```

Dediğimiz gibi, all() fonksiyonu ancak dizi içindeki bütün değerlerin bool değeri *True* ise *True* çıktısı verecektir.

Son bir örnek daha verelim:

```
>>> liste = ['ahmet', 'mehmet', '']
```

```
>>> all(liste)

False
```

Listede *False* değerine sahip bir boş karakter dizisi bulunduğu için all() fonksiyonu *False* çıktısı veriyor.

Bu fonksiyonu her türlü kodun bool değerlerini test etmek için kullanabilirsiniz. Mesela bu fonksiyonu kullanarak, bir nesnenin listelenen özelliklerin hepsine sahip olup olmadığını denetleyebilirsiniz:

```
>>> a = 3
>>> t1 = a == 3        #sayı 3 mü?
>>> t2 = a < 4         #sayı 4'ten küçük mü?
>>> t3 = a % 2 == 1    #sayı bir tek sayı mı?
>>> all([t1, t2, t3])  #sayı bu özelliklerin hepsine sahip mi?

True
```

Eğer sayımız bu özelliklerin birine bile sahip değilse, all() fonksiyonu *False* çıktısı verecektir.

any()

Any kelimesi İngilizcede 'herhangi bir' anlamına gelir. İşte any() fonksiyonunun görevi de, bir dizi içindeki bütün değerlerden en az biri *True* ise *True* çıktısı vermektir.

Örneğin:

```
>>> liste = ['ahmet', 'mehmet', '']
>>> any(liste)

True
```

any() fonksiyonunun *True* çıktısı verebilmesi için listede yalnızca bir adet *True* değerli öğe olması yeterlidir. Bu fonksiyonun *False* çıktısı verebilmesi için dizi içindeki bütün öğelerin bool değerinin *False* olması gerekir:

```
>>> l = ['', 0, [], (), set(), dict()]
>>> any(l)

False
```

İçi boş veri tiplerinin bool değerinin *False* olduğunu biliyorsunuz.

Tıpkı all() fonksiyonunda olduğu gibi, any() fonksiyonunu da, bir grup nesnenin bool değerlerini denetlemek amacıyla kullanabilirsiniz.

ascii()

Bu fonksiyon, bir nesnenin ekrana basılabilir halini verir bize. Dilerseniz bu fonksiyonun yaptığı işi tanımlamak yerine bunu bir örnek üzerinden anlatmaya çalışalım:

```
>>> a = 'istihza'
>>> print(ascii(a))

'istihza'
```

Bu fonksiyonun, print() fonksiyonundan farklı olarak, çıktıya tırnak işaretlerini de eklediğine dikkat edin.

ascii() fonksiyonunun tam olarak ne yaptığını daha iyi anlamak için herhalde şu örnek daha faydalı olacaktır.

Dikkatlice bakın:

```
>>> print('\n')
```

Bu komutu verdiğimizde, *n* kaçış dizisinin etkisiyle yeni satıra geçileceğini biliyorsunuz.

Bir de şuna bakın:

```
>>> print(ascii('\n'))

'\n'
```

Gördüğünüz gibi, ascii() fonksiyonu, satır başı kaçış dizisinin görevini yapmasını sağlamak yerine bu kaçış dizisinin ekrana basılabilir halini veriyor bize.

Ayrıca bu fonksiyon, karakter dizileri içindeki Türkçe karakterlerin de UNICODE temsillerini döndürür. Örneğin:

```
>>> a = 'ışık'
>>> print(ascii(a))
'\u0131\u015f\u0131k'
```

Bunu daha net şu şekilde görebiliriz:

```
>>> for i in a:
...     print(ascii(i))
```

```
...
'\u0131'
'\u015f'
'\u0131'
'k'
```

Gördüğünüz gibi, ascii() fonksiyonu ASCII olmayan karakterlerle karşılaştığında bunların karakter temsilleri yerine UNICODE temsillerini (veya onaltılık sayma düzenindeki karşılıklarını) veriyor.

Son olarak şu örneğe bakalım:

```
>>> liste = ['elma', 'armut', 'erik']
>>> temsil = ascii(liste)
>>> print(temsil)

['elma', 'armut', 'erik']
```

Burada listemiz ascii() fonksiyonuna parametre olarak verildikten sonra artık liste olma özelliğini yitirip bir karakter dizisi haline gelir. Bunu denetleyelim:

```
>>> print(type(temsil))
<class 'str'>

>>> temsil[0]

'['
```

Gördüğünüz gibi, ascii() fonksiyonu listeyi alıp, bunu ekrana basılabilir bir bütün haline getiriyor. Elbette bunun için de, kendisine verilen parametreyi bir karakter dizisine dönüştürüyor.

repr()

repr() fonksiyonunun yaptığı iş, biraz önce gördüğümüz ascii() fonksiyonunun yaptığı işe çok benzer. Bu iki fonksiyon, ASCII olmayan karakterlere muameleleri açısından birbirinden ayrılır.

Hatırlarsanız ascii() fonksiyonu ASCII olmayan karakterlerle karşılaştığında bunların UNICODE (veya onaltılık) temsillerini gösteriyordu:

```
>>> ascii('şeker')

'"\\u015feker"'
```

repr() fonksiyonu ise ASCII olmayan karakterlerle karşılaşsa bile, bize çıktı olarak bunların da karakter karşılıklarını gösterir:

```
>>> repr('şeker')

"'şeker'"
```

Geri kalan özellikleri bakımından repr() ve ascii() fonksiyonları birbiriyle aynıdır.

bool()

Bu fonksiyon bir nesnenin bool değerini verir:

```
>>> bool(0)

False

>>> bool(1)

True

>>> bool([])

False
```

bin()

Bu fonksiyon, bir sayının ikili düzendeki karşılığını verir:

```
>>> bin(12)

'0b1100'
```

Bu fonksiyonun verdiği çıktının bir sayı değil, karakter dizisi olduğuna dikkat etmelisiniz.

bytes()

Bu fonksiyon *bytes* türünde nesneler oluşturmak için kullanılır. Bu fonksiyonu 'bayt' adlı veri tipini incelerken ayrıntılı olarak ele almıştık. Gelin isterseniz burada da bu fonksiyona şöyle bir değinelim.

Dediğimiz gibi, bytes() adlı fonksiyon, *bytes* türünde veriler oluşturmaya yarar. Bu fonksiyon işlev olarak, daha önce öğrendiğimiz list(), str(), int(), set(), dict() gibi fonksiyonlara çok benzer. Tıpkı bu fonksiyonlar gibi, bytes() fonksiyonunun görevi de farklı veri tiplerini 'bayt' adlı veri tipine dönüştürmektir.

Bu fonksiyon, kendisine verilen parametrelerin türüne bağlı olarak birbirinden farklı sonuçlar ortaya

çıkarır. Örneğin eğer bu fonksiyona parametre olarak bir tam sayı verecek olursanız, bu fonksiyon size o tam sayı miktarınca bir bayt nesnesi verecektir. Gelin isterseniz bu durumu örnekler üzerinde göstermeye çalışalım:

```
>>> bytes(10)

b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

Yukarıdaki komut bize, her bir öğesinin değeri *0* olan 10 baytlık bir veri döndürdü:

```
>>> a = bytes(10)

>>> a

b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'

>>> a[0]

0

>>> a[1]

0

>>> a[2]

0
```

Gördüğünüz gibi, bytes(10) komutuyla oluşturduğumuz *a* değişkeni içinde toplam 10 adet bayt var ve bu baytların her birinin değeri 0.

Yukarıda, bytes() fonksiyonuna bir tam sayı değerli parametre verdiğimizde nasıl bir sonuç alacağımızı öğrendik. Peki biz bu fonksiyona parametre olarak bir karakter dizisi verirsek ne olur?

Hemen görelim:

```
>>> bytes('istihza')

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: string argument without an encoding
```

Bu fonksiyona karakter dizilerini doğrudan parametre olarak veremeyiz. Eğer verirsek yukarıdaki gibi bir hata alırız. Peki acaba bu hatayı almamızın nedeni ne olabilir?

Dediğimiz gibi, bytes() fonksiyonu, çeşitli veri tiplerini bayta dönüştürmeye yarar. Ancak bildiğiniz gibi, bayta dönüştürme işlemi her kod çözücü tarafından farklı biçimde yapılır. Örneğin:

```
>>> 'ışık'.encode('utf-8')

b'\xc4\xb1\xc5\x9f\xc4\xb1k'

>>> 'ışık'.encode('cp857')

b'\x8d\x9f\x8dk'

>>> 'ışık'.encode('cp1254')

b'\xfd\xfe\xfdk'
```

Dolayısıyla, bytes() fonksiyonunun bir karakter dizisini bayta çevirirken nasıl davranması gerektiğini anlayabilmesi için, bayta dönüştürme işlemini hangi kod çözücü ile yapmak istediğimizi açıkça belirtmemiz gerekir:

```
>>> bytes('ışık', 'utf-8')

b'\xc4\xb1\xc5\x9f\xc4\xb1k'

>>> bytes('ışık', 'cp1254')

b'\xfd\xfe\xfdk'

>>> bytes('ışık', 'cp857')

b'\x8d\x9f\x8dk'
```

Gördüğünüz gibi, bytes() fonksiyonuna parametre olarak bir karakter dizisi verebilmek için, bu karakter dizisi ile birlikte bir kod çözücü de belirtmemiz gerekiyor. Böylece bytes() fonksiyonu kendisine verdiğimiz karakter dizisini, belirttiğimiz kod çözücünün kurallarına göre bayta dönüştürüyor.

Bu arada, çıktıda görünen ‘b’ harflerinin, elimizdeki verinin bir bayt olduğunu gösteren bir işaret olduğunu biliyorsunuz.

Ayrıca, bytes() fonksiyonuna verdiğimiz ikinci parametrenin isminin *encoding* olduğunu ve bu parametreyi isimli bir parametre olarak da kullanabileceğimizi belirtelim:

```
>>> bytes('istihza', encoding='ascii')
```

Bu noktada size şöyle bir soru sorayım: Acaba bytes() fonksiyonuna ilk parametre olarak verdiğimiz karakter dizisi, ikinci parametrede belirttiğimiz kod çözücü tarafından tanınmazsa ne olur?

Cevabı tahmin edebilirsiniz: Böyle bir durumda elbette Python bize bir hata mesajı gösterir:

```
>>> bytes('şeker', 'ascii')

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character '\u015f' in position 0:
 ordinal not in range(128)
```

... veya:

```
>>> bytes('€', 'cp857')

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Python33\lib\encodings\cp857.py", line 12, in encode
    return codecs.charmap_encode(input,errors,encoding_map)
UnicodeEncodeError: 'charmap' codec can't encode character '\u20ac' in position
0: character maps to <undefined>
```

'ş' harfi 'ASCII' karakter kümesinde; '€' işareti ise 'CP857' adlı karakter kümesinde tanımlanmamış birer karakter olduğu için, ilgili kod çözücüler bu karakterleri çözüp bayta dönüştüremiyor. Yazdığımız kodların bu tür durumlarda tamamen çökmesini engellemek için, önceki derslerimizde de çeşitli vesilelerle öğrenmiş olduğumuz *errors* adlı bir parametreden yararlanabiliriz:

```
>>> bytes('ışık', encoding='ascii', errors='replace')

b'???k'

>>> bytes('şeker', encoding='ascii', errors='replace')

b'?eker'

>>> bytes('€', encoding='cp857', errors='replace')

b'?'

>>> bytes('€', encoding='cp857', errors='ignore')

b''
```

```
>>> bytes('€', encoding='cp857', errors='xmlcharrefreplace')

b'&#8364;'

>>> bytes('şeker', encoding='cp857', errors='xmlcharrefreplace')

b'\x9feker'
```

Gördüğünüz gibi, *errors* parametresine verdiğimiz çeşitli değerler yardımıyla, bytes() fonksiyonunun, *encoding* parametresinde belirtilen kod çözücü ile çözülemeyen karakterlerle karşılaştığında nasıl davranacağını belirleyebiliyoruz.

errors parametresine verdiğimiz bütün bu değerleri önceki derslerimizde öğrenmiştik. Dolayısıyla yukarıda gösterdiğimiz kodları rahatlıkla anlayabilecek kadar Python bilgisine sahibiz.

Son olarak, bytes() fonksiyonuna parametre olarak 0-256 arası sayılardan oluşan diziler de verebiliriz:

```
>>> bytes([65, 10, 12, 11, 15, 66])

b'A\n\x0c\x0b\x0fB'
```

Bu yapı içinde Python, *0* ile *128* arası sayılar için standart ASCII tablosunu, *128* ile *256* arası sayılar için ise Latin-1 karakter kümesini temel alarak sayıları birer bayta dönüştürecektir.

bytearray()

Bildiğiniz gibi baytlar değiştirilemeyen bir veri tipidir. Dolayısıyla bir bayt veri tipi üzerinde herhangi bir değişiklik yapamayız. Örneğin bir baytın herhangi bir öğesini başka bir değerle değiştiremeyiz:

```
>>> a = bytes('istihza', 'ascii')
>>> a[0]

105

>>> a[0] = 106

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment
```

Ama eğer hem baytlarla çalışmak, hem de bu baytların üzerinde değişiklik yapabilmek isterseniz baytlar yerine bayt dizileri ile çalışabilirsiniz. İşte bunun için bytearray() adlı bir fonksiyondan

yararlanıyoruz.

Yaptıkları iş bakımından bytearray() ve bytes() fonksiyonları birbirlerine çok benzer. Bu ikisi arasındaki tek fark, bytearray() ile oluşturulan veri tipinin, bytes() ile oluşturulan veri tipinin aksine, değiştirilebilir nitelikte olmasıdır:

```
>>> a = bytearray('adana', 'ascii')

>>> a

bytearray(b'adana')

>>> a[0] = 65

>>> a

bytearray(b'Adana')
```

chr()

Bu fonksiyon, kendisine parametre olarak verilen bir tam sayının karakter karşılığını döndürür. Örneğin:

```
>>> chr(10)

'\n'
```

Bildiğiniz gibi *10* sayısının karakter karşılığı satır başı karakteridir. Bir de şuna bakalım:

```
>>> chr(65)

'A'
```

65 sayısının karakter karşılığı ise ‘A’ harfidir.

Bu fonksiyon sayıları karakterlere dönüştürürken ASCII sistemini değil, UNICODE sistemini temel alır. Dolayısıyla bu fonksiyon ile 128 (veya 255) üstü sayıları da dönüştürebiliriz. Örneğin:

```
>>> chr(305)

'ı'
```

list()

Bu fonksiyon iki farklı amaç için kullanılabilir:

1. Liste tipinde bir veri oluşturmak
2. Farklı veri tiplerini liste adlı veri tipine dönüştürmek

Birinci amaç için bu fonksiyonu şu şekilde kullanıyoruz:

```
>>> l = list()
```

Böylece liste tipinde bir veri oluşturmuş olduk.

Dediğimiz gibi list() fonksiyonunu, farklı tipteki verileri listeye dönüştürmek için de kullanabiliriz. Örneğin:

```
>>> list('istihza')

['i', 's', 't', 'i', 'h', 'z', 'a']
```

Burada *'istihza'* adlı karakter dizisini bir listeye dönüştürdük.

Elbette bu fonksiyonu kullanarak başka veri tiplerini de listeye dönüştürebiliriz. Örneğin bir sözlüğü, bu fonksiyon yardımıyla kolayca listeye dönüştürebiliriz:

```
>>> s = {'elma': 44, 'armut': 10, 'erik': 100}
>>> list(s)

['armut', 'erik', 'elma']
```

Bir sözlük listeye dönüştürülürken, elbette sözlüğün anahtarları dikkate alınacaktır. Eğer siz sözlüğün anahtarlarından değil de değerlerinde bir liste oluşturmak isterseniz şöyle bir kod yazabilirsiniz:

```
>>> list(s.values())

[10, 100, 44]
```

set()

set() fonksiyonu list() fonksiyonuna çok benzer. Bu fonksiyon da tıpkı list() fonksiyonu gibi, veri tipleri arasında dönüştürme işlemleri gerçekleştirmek için kullanılabilir. set() fonksiyonunun görevi farklı veri tiplerini kümeye dönüştürmektir:

```
>>> k = set()
```

Burada boş bir küme oluşturduk. Şimdi de mesela bir karakter dizisini kümeye dönüştürelim:

```
>>> i = 'istihza'
>>> set(i)

{'t', 's', 'z', 'a', 'i', 'h'}
```

tuple()

tuple() fonksiyonu da, tıpkı list(), set() ve benzerleri gibi bir dönüştürücü fonksiyondur. Bu fonksiyon farklı veri tiplerini demete dönüştürür:

```
>>> tuple('a')

('a',)
```

frozenset()

Bu fonksiyonu kullanarak farklı veri tiplerini dondurulmuş kümeye dönüştürebilirsiniz:

```
>>> s = set('istihza')
>>> df = frozenset(s)
>>> df

frozenset({'t', 's', 'a', 'z', 'i', 'h'})
```

complex()

Sayılardan söz ederken, eğer matematikle çok fazla içli dışlı değilseniz pek karşılaşmayacağınız, 'karmaşık sayı' adlı bir sayı türünden de bahsetmiştik. Karmaşık sayılar, bir gerçek, bir de sanal kısımdan oluşan sayılardır.

Karmaşık sayılar Python'da 'complex' ifadesiyle gösteriliyor. Mesela şu bir karmaşık sayıdır:

```
>>> 12+0j
```

İşte eğer herhangi bir sayıyı karmaşık sayıya dönüştürmeniz gerekirse complex() adlı bir fonksiyondan yararlanabilirsiniz. Örneğin:

```
>>> complex(15)

(15+0j)
```

Böyle bir kod yazdığımızda, verdiğimiz parametre karmaşık sayının gerçek kısmını oluşturacak, sanal kısım ise *0* olarak kabul edilecektir. Elbette isterseniz sanal kısmı kendiniz de belirleyebilirsiniz:

```
>>> complex(15, 2)

(15+2j)
```

float()

Bu fonksiyonu, sayıları veya karakter dizilerini kayan noktalı sayıya dönüştürmek için kullanıyoruz:

```
>>> float('134')

134.0

>>> float(12)

12.0
```

int()

Bu fonksiyon birkaç farklı amaç için kullanılabilir. int() fonksiyonunun en temel görevi, bir karakter dizisi veya kayan noktalı sayıyı (eğer mümkünse) tam sayıya dönüştürmektir:

```
>>> int('10')

10

>>> int(12.4)

12
```

Bunun dışında bu fonksiyonu, herhangi bir sayma sisteminde temsil edilen bir sayıyı onlu sayma sistemine dönüştürmek için de kullanabiliriz. Örneğin:

```
>>> int('12', 8)

10
```

Burada, sekizli sayma sistemine ait sayı değerli bir karakter dizisi olan *'12'`yi onlu sayma sistemine dönüştürdük ve böylece `10* sayısını elde ettik.

int() fonksiyonunu sayma sistemleri arasında dönüştürme işlemlerinde kullanabilmek için ilk parametrenin bir karakter dizisi olması gerektiğine dikkat ediyoruz.

Bu arada, int('12', 8) komutununun *12* sayısını sekizli sayma sistemine dönüştürmediğine dikkat edin. Bu komutun yaptığı iş sekizli sayma sistemindeki *12* sayısını onlu sayma sistemine dönüştürmektir.

int() fonksiyonunun bu kullanımıyla ilgili bir örnek daha verelim:

```
>>> int('4cf', 16)

1231
```

Burada da, onaltılı sayma sistemine ait bir sayı olan *4cf*'yi onlu sayma sistemine çevirdik ve *1231* sayısını elde ettik. *4cf* sayısını int() fonksiyonuna parametre olarak verirken bunu karakter dizisi şeklinde yazmayı unutmuyoruz. Aksi halde Python bize bir hata mesajı gösterecektir.

str()

Bu fonksiyonun, farklı veri tiplerini karakter dizisine dönüştürmek için kullanıldığını biliyorsunuz. Örneğin:

```
>>> str(12)

'12'
```

Burada *12* sayısını bir karakter dizisine dönüştürdük. Şimdi de bir baytı karakter dizisine dönüştürelim:

```
>>> bayt = b'istihza'
```

Bayt nesnemizi tanımladık. Şimdi bunu bir karakter dizisine dönüştürelim:

```
>>> kardiz = str(bayt, encoding='utf-8')
>>> print(kardiz)

istihza
```

Gördüğünüz gibi, bir baytı karakter dizisine dönüştürmek için str() fonksiyonuna *encoding* adlı bir parametre veriyoruz. Fonksiyonumuz, bu parametrede hangi kodlama biçimi belirtildiyse, baytları bu kodlama biçiminin kurallarına göre bir karakter dizisine dönüştürüyor.

Tahmin edebileceğiniz gibi, belirttiğiniz kodlama biçiminin herhangi bir baytı karakter dizisine dönüştüremediği durumlara karşı bir *errors* parametresi de verebiliriz str() fonksiyonuna. Örneğin elimizde bayt tipinde şöyle bir veri olduğunu varsayalım:

```
>>> bayt = bytes('kadın', encoding='utf-8')
>>> print(bayt)

b'kad\xc4\xb1n'
```

Şimdi bu bayt veri tipini bir karakter dizisine dönüştürmeye çalışalım:

```
>>> kardiz = str(bayt, encoding='ascii')

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc4 in position 3: ordinal
not in range(128)
```

ASCII adlı kod çözücü, b'kadın' içindeki baytlardan birini tanıyamadığı için bize bir hata mesajı gösterdi. Bildiğiniz gibi ASCII 128'den büyük baytları dönüştüremez. İşte bu tür durumlara karşı *errors* parametresinden yararlanabilirsiniz:

```
>>> kardiz = str(bayt, encoding='ascii', errors='ignore')
>>> print(kardiz)

kadn
```

errors parametresine verdiğimiz 'ignore' değeri sayesinde Python bize hata mesajı göstermek yerine, ASCII ile çözülemeyen baytı görmezden geldi. *errors* parametresinin hangi değerleri alabileceğini önceki derslerimizden hatırlıyor olmalısınız.

dict()

Bu fonksiyon, farklı veri tiplerinden sözlükler üretmemizi sağlar. Örneğin bu fonksiyonu kullanarak boş bir sözlük oluşturabiliriz:

```
>>> s = dict()
```

Bu fonksiyon, değişkenlerden sözlükler oluşturmamızı da sağlar:

```
>>> s = dict(a=1, b=2, c=3)

{'a': 1, 'b': 2, 'c': 3}
```

dict() fonksiyonuna parametre olarak iç içe geçmiş listeler veya demetler vererek de sözlük üretebiliriz:

```
>>> öğeler = (['a', 1], ['b', 2], ['c', 3])
>>> dict(öğeler)

{'a': 1, 'b': 2, 'c': 3}
```

callable()

Bu fonksiyon, bir nesnenin 'çağrılabilir' olup olmadığını denetler. Peki hangi nesneler çağrılabilir

özelliktedir. Mesela fonksiyonlar çağrılabilir nesnelerdir. Değişkenler ise çağrılabilir nesneler değildir.

Birkaç örnek verelim bununla ilgili:

```
>>> callable(open)

True
```

Python'ın open() adlı bir fonksiyonu olduğu için, doğal olarak callable() fonksiyonu *True* çıktısı veriyor.

Bir de şuna bakalım:

```
>>> import sys
>>> callable(sys.version)

False
```

Burada da *sys* modülü içindeki *version* adlı nesnenin çağrılabilir özellikte olup olmadığını sorguladık. Daha önceki derslerimizde de gördüğünüz gibi, *sys* modülü içindeki *version* adlı araç bir fonksiyon değil, değişkendir. Dolayısıyla bu değişken callable(sys.version) sorgusuna *False* yanıtı verir.

ord()

Bu fonksiyon, bir karakterin karşılık geldiği ondalık sayıyı verir. Örneğin:

```
>>> ord('a')

97

>>> ord('ı')

305
```

oct()

Bu fonksiyon, bir sayıyı sekizli düzendeki karşılığına çevirmemizi sağlar:

```
>>> oct(10)

'0o12'
```

hex()

Bu fonksiyon, bir sayıyı onaltılı düzendeki karşılığına çevirmemizi sağlar:

```
>>> hex(305)

'Ox131'
```

Yalnız hem oct() hem de hex() fonksiyonlarında dikkat etmemiz gereken şey, bu fonksiyonların parametre olarak bir sayı alıp, çıktı olarak bir karakter dizisi veriyor olmasıdır.

eval(), exec(), globals(), locals(), compile()

Bu bölümde beş farklı fonksiyonu bir arada inceleyeceğiz. Bu fonksiyonları birlikte ele almamızın nedeni bunları birbiriyle yakından bağlantılı olması.

Burada işleyeceğimiz bu beş fonksiyon şunlardan oluşuyor:

1. eval()
2. exec()
3. globals()
4. locals()
5. compile()

Ancak bu fonksiyonlardan söz etmeye başlamadan önce Python'daki iki önemli kavramı açıklığa kavuşturmamız gerekiyor: Bu kavramlar şunlar:

1. ifade
2. deyim

Öncelikle 'ifade' kavramından başlayalım.

İngilizcede *expression* denen 'ifadeler', bir değer üretmek için kullanılan kod parçalarıdır. Karakter dizileri, sayılar, işleçler, öteki veri tipleri, liste üreteçleri, sözlük üreteçleri, küme üreteçleri, fonksiyonlar hep birer ifadedir. Örneğin:

```
>>> 5

>>> 23 + 4

>>> [i for i in range(10)]

>>> len([1, 2, 3])
```

İngilizcede *statement* olarak adlandırılan 'deyimler' ise ifadeleri de kapsayan daha geniş bir kavramdır. Buna göre bütün ifadeler aynı zamanda birer deyimdir. Daha doğrusu, ifadelerin bir araya gelmesi ile deyimler oluşturulabilir.

Deyimlere birkaç örnek verelim:

```
>>> a = 5

>>> if a:
...     print(a)

>>> for i in range(10):
...     print(i)
```

Python programlama dilinde deyimlerle ifadeleri ayırt etmenin kolay bir yolu da eval() fonksiyonundan yararlanmaktır. Eğer deyim mi yoksa ifade mi olduğundan emin olamadığınız bir şeyi eval() fonksiyonuna parametre olarak verdiğinizde hata almıyorsanız o parametre bir ifadedir. Eğer hata alıyorsanız o parametre bir deyimdir. Çünkü eval() fonksiyonuna parametre olarak yalnızca ifadeler verilebilir.

Birkaç örnek verelim:

```
>>> eval('a = 5')

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1
    a = 5
      ^
SyntaxError: invalid syntax
```

Gördüğünüz gibi, eval() fonksiyonu bize bir hata mesajı verdi. Çünkü a = 5 kodu bir deyimdir. Unutmayın, Python'da bütün değer atama işlemleri birer deyimdir. Dolayısıyla eval() fonksiyonu bu deyimi parametre olarak alamaz.

Bir de şuna bakalım:

```
>>> eval('5 + 25')

30
```

Bu defa hata almadık. Çünkü eval() fonksiyonuna, olması gerektiği gibi, parametre olarak bir ifade verdik. Bildiğiniz gibi, 5 + 25 kodu bir ifadedir.

Dediğimiz gibi, eval() fonksiyonu deyimleri parametre olarak alamaz. Ama exec() fonksiyonu alabilir:

```
>>> exec('a = 5')
```

Bu şekilde, değeri *5* olan *a* adlı bir değişken oluşturmuş olduk. İsterseniz kontrol edelim:

```
>>> print(a)

5
```

Gördüğünüz gibi, exec() fonksiyonu, mevcut isim alanı içinde *a* adlı bir değişken oluşturdu. Yalnız elbette mevcut isim alanı içinde yeni değişkenler ve yeni değerler oluştururken dikkatli olmamız gerektiğini biliyorsunuz. Zira mesela yukarıdaki komutu vermeden önce mevcut isim alanında zaten *a* adlı bir değişken varsa, o değişkenin değeri değişecektir:

```
>>> a = 20
```

Elimizde, değeri *20* olan *a* adlı bir değişken var. Şimdi exec() fonksiyonu yardımıyla *a* değişkeninin de içinde yer aldığı mevcut isim alanına müdahale ediyoruz:

```
>>> exec('a = 10')
```

Böylece *a* değişkeninin eski değerini silmiş olduk. Kontrol edelim:

```
>>> print(a)

10
```

Bu tür durumlarda, exec() ile oluşturduğunuz değişkenleri global isim alanına değil de, farklı bir isim alanına göndermeyi tercih edebilirsiniz. Peki ama bunu nasıl yapacağız?

Python programlama dilinde isim alanları sözlük tipinde bir veridir. Örneğin global isim alanı basit bir sözlükten ibarettir.

Global isim alanını gösteren sözlükte hangi anahtar ve değerlerin olduğunu görmek için globals() adlı bir fonksiyonu kullanabilirsiniz:

```
>>> globals()
```

Bu fonksiyonu çalıştırdığımızda şuna benzer bir çıktı alırız:

```
{'__doc__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
'__name__': '__main__', '__package__': None, '__builtins__': <module 'builtins'>}
```

Gördüğünüz gibi, elimizdeki şey gerçekten de bir sözlük. Dolayısıyla bir sözlük ile ne yapabilirsek

bu sözlükle de aynı şeyi yapabiliriz...

'globals' adlı bu sözlüğün içeriği, o anda global isim alanında bulunan nesnelere göre farklılık gösterecektir. Örneğin:

```
>>> x = 10
```

şeklinde *10* değerine sahip bir *x* nesnesi tanımladıktan sonra globals() fonksiyonunu tekrar çalıştırırsanız global isim alanına bu nesnenin de eklenmiş olduğunu görürsünüz.

Dediğimiz gibi, globals() fonksiyonundan dönen nesne bir sözlüktür. Bu sözlüğe, herhangi bir sözlüğe veri ekler gibi değer de ekleyebilirsiniz:

```
>>> globals()['z'] = 23
```

Bu şekilde global isim alanına *z* adlı bir değişken eklemiş oldunuz:

```
>>> z

23
```

Yalnız, Python programlama dili bize bu şekilde global isim alanına nesne ekleme imkanı verse de, biz mecbur değilsek bu yöntemi kullanmaktan kaçınmalıyız. Çünkü bu şekilde sıradışı bir yöntemle değişken tanımladığımız için aslında global isim alanını, nerden geldiğini kestirmenin güç olduğu değerlerle 'kirletmiş' oluyoruz.

Bildiğiniz gibi, Python'da global isim alanı dışında bir de lokal isim alanı bulunur. Lokal isim alanlarının, fonksiyonlara (ve ileride göreceğimiz gibi sınıflara) ait bir isim alanı olduğunu biliyorsunuz. İşte bu isim alanlarına ulaşmak için de locals() adlı bir fonksiyondan yararlanacağız:

```
def fonksiyon(param1, param2):
    x = 10
    print(locals())

fonksiyon(10, 20)
```

Bu fonksiyonu çalıştırdığınızda şu çıktıyı alacaksınız:

```
{'param2': 20, 'param1': 10, 'x': 10}
```

Gördüğünüz gibi, locals() fonksiyonu gerçekten de bize fonksiyon() adlı fonksiyon içindeki lokal değerleri veriyor.

globals() ve locals() fonksiyonlarının ne işe yaradığını incelediğimize göre exec() fonksiyonunu anlatırken kaldığımız yere dönebiliriz.

Ne diyorduk?

Elimizde, değeri *20* olan *a* adlı bir değişken vardı:

```
>>> a = 20
```

exec() fonksiyonu yardımıyla *a* değişkeninin de içinde yer aldığı mevcut isim alanına müdahale edelim:

```
>>> exec('a = 3')
```

Bu şekilde *a* değişkeninin varolan değerini silmiş olduk:

```
>>> print(a)

3
```

Dediğimiz gibi, bu tür durumlarda, exec() ile oluşturduğunuz değişkenleri global isim alanı yerine farklı bir isim alanına göndermeyi tercih etmemiz daha uygun olacaktır. Python'da isim alanlarının basit bir sözlük olduğunu öğrendiğimize göre, exec() ile oluşturduğumuz değişkenleri global isim alanı yerine nasıl farklı bir isim alanına göndereceğimizi görebiliriz.

Önce yeni bir isim alanı oluşturalım:

```
>>> ia = {}
```

Şimdi exec() ile oluşturacağımız değerleri bu isim alanına gönderebiliriz:

```
>>> exec('a = 3', ia)
```

Böylece global isim alanındaki *a* değişkeninin değerine dokunmamış olduk:

```
>>> a

20
```

Yeni oluşturduğumuz değer ise *ia* adlı yeni isim alanına gitti:

```
>>> ia['a']

3
```

copyright()

Bu fonksiyon yardımıyla Python'ın telif haklarına ilişkin bilgilere erişebilirsiniz:

```
>>> copyright()

Copyright (c) 2001-2012 Python Software Foundation.
All Rights Reserved.

Copyright (c) 2000 BeOpen.com.
All Rights Reserved.

Copyright (c) 1995-2001 Corporation for National Research Initiatives.
All Rights Reserved.

Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.
```

credits()

Bu fonksiyon, Python programlama diline katkıda bulunanlara teşekkür içeren küçük bir metni ekrana çıktı olarak verir:

```
>>> credits()

Thanks to CWI, CNRI, BeOpen.com, Zope Corporation and a cast of thousands
for supporting Python development.  See www.python.org for more information.
```

license()

Bu fonksiyon yardımıyla Python'ın lisansına ilişkin epey ayrıntılı metinlere ulaşabilirsiniz.

dir()

Eğer dir() fonksiyonunu parametresiz olarak kullanırsak, mevcut isim alanındaki öğeleri bir liste halinde elde ederiz:

```
>>> dir()

['__builtins__', '__doc__', '__loader__', '__name__', '__package__']
```

Bu bakımdan dir() fonksiyonu globals() ve locals() fonksiyonlarına benzer. Ancak onlardan farkı, dir() fonksiyonunun çıktı olarak bir liste, globals() ve locals() fonksiyonlarının ise birer sözlük vermesidir.

Ayrıca dir() fonksiyonunu kullanarak nesnelerin metot ve niteliklerini içeren bir listeye ulaşabileceğimizi de biliyorsunuz. Örneğin bu fonksiyonu kullanarak farklı veri tiplerinin metot ve

niteliklerini listeleyebiliriz:

```
>>> dir('')
>>> dir([])
>>> dir({})
```

divmod()

Bu fonksiyonun işlevini bir örnek üzerinden göstermeye çalışalım:

```
>>> divmod(10, 2)

(5, 0)
```

Gördüğünüz gibi divmod(10, 2) komutu bize iki öğeli bir demet veriyor. Bu demetin ilk öğesi, divmod() fonksiyonuna verilen ilk parametrenin ikinci parametreye bölünmesi işleminin sonucudur. Demetimizin ikinci öğesi ise, ilk parametrenin ikinci parametreye bölünmesi işleminden kalan sayıdır. Yani demetin ilk parametresi bölme işleminin 'bölüm' kısmını, ikinci öğesi ise 'kalan' kısmını verir.

Bu fonksiyonun bölme işlemininin sonucunu tamsayı cinsinden verdiğine dikkat ediyoruz:

```
>>> divmod(10, 3)

(3, 1)
```

10 sayısı *3* sayısına bölündüğünde tamsayı cinsinden sonuç *3*'tür. Bu bölme işleminin kalanı ise *1*'dir.

enumerate()

İngilizcede *enumerate* kelimesi 'numaralandırmak' anlamına gelir. enumerate() fonksiyonunun görevi de kelimenin bu anlamıyla aynıdır. Yani bu fonksiyonu kullanarak nesneleri numaralandırabiliriz.

Bu fonksiyon bize bir 'enumerate' nesnesi verir:

```
>>> enumerate('istihza')

<class 'enumerate'>
```

Bu nesnenin içeriğine nasıl erişebileceğimizi biliyorsunuz:

Nesneyi bir listeye çevirebiliriz:

```
>>> list(enumerate('istihza'))

[(0, 'i'), (1, 's'), (2, 't'), (3, 'i'), (4, 'h'), (5, 'z'), (6, 'a')]
```

veya:

```
>>> [i for i in enumerate('istihza')]

[(0, 'i'), (1, 's'), (2, 't'), (3, 'i'), (4, 'h'), (5, 'z'), (6, 'a')]
```

print() fonksiyonuna yıldızlı parametre olarak verebiliriz:

```
>>> print(*enumerate('istihza'))

(0, 'i') (1, 's') (2, 't') (3, 'i') (4, 'h') (5, 'z') (6, 'a')
```

veya nesne üzerinde bir döngü kurabiliriz:

```
>>> for i in enumerate('istihza'):
...     print(i)
...
(0, 'i')
(1, 's')
(2, 't')
(3, 'i')
(4, 'h')
(5, 'z')
(6, 'a')
```

Gördüğünüz gibi, 'enumerate' nesnesi bize her koşulda iki öğeli demetler veriyor. Bu demetlerin herbir öğesine nasıl ulaşabileceğimizi de biliyor olmalısınız:

```
>>> for sıra, öğe in enumerate('istihza'):
...     print("{}. {:>2}".format(sıra, öğe))
...
0.  i
1.  s
2.  t
3.  i
4.  h
5.  z
6.  a
```

Örneklerden de gördüğünüz gibi, enumerate() fonksiyonu bize bir dizi içindeki öğelerin sırasını ve öğenin kendisini içeren bir demet veriyor. Dikkat ettiyseniz, her zaman olduğu gibi, Python burada da

saymaya 0'dan başlıyor. Yani enumerate() fonksiyonunun ürettiği öğe sıralamasında ilk öğenin sırası 0 oluyor. Elbette eğer isterseniz enumerate() fonksiyonunun saymaya kaçtan başlayacağını kendiniz de belirleyebilirsiniz:

```
>>> for sıra, öğe in enumerate('istihza', 1):
...     print("{}. {:>2}".format(sıra, öğe))
...
1.  i
2.  s
3.  t
4.  i
5.  h
6.  z
7.  a
```

enumerate() fonksiyonuna verdiğimiz ikinci parametre saymaya kaçtan başlanacağını gösteriyor. Eğer bu fonksiyonu ikinci parametre olmadan kullanırsak Python bizim saymaya 0'dan başlamak istediğimizi varsayacaktır.

exit()

Bu fonksiyon, o anda çalışan programdan çıkmanızı sağlar. Eğer bu komutu etkileşimli kabukta verirseniz o anda açık olan oturum kapanacaktır.

help()

help() fonksiyonu gömülü fonksiyonlar içinde en faydalı fonksiyonların başında gelir. Bu fonksiyonu kullanarak Python programlama diline ait öğelere ilişkin yardım belgelerine ulaşabiliriz. Örneğin:

```
>>> help(dir)
```

Bu komutu verdiğimizde dir() fonksiyonunun ne işe yaradığını gösteren İngilizce bir belgeye ulaşırız. Gördüğünüz gibi, hakkında bilgi edinmek istediğimiz öğeyi help() fonksiyonuna parametre olarak vererek ilgili yardım dosyasına erişebiliyoruz.

Eğer bu fonksiyonu parametresiz olarak kullanırsak 'etkileşimli yardım' denen ekrana ulaşırız:

```
>>> help()

Welcome to Python 3.3!  This is the interactive help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.3/tutorial/.
```

```
Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help>
```

Bu ekranda, hakkında bilgi edinmek istediğiniz öğeyi help> ibaresinden hemen sonra, boşluk bırakmadan yazarak öğeye ilişkin bilgilere ulaşabilirsiniz:

```
help> dir
```

Etkileşimli yardım ekranından çıkmak için ‘q’ harfine basabilirsiniz.

id()

Python’da her nesnenin bir kimliğinin olduğunu biliyorsunuz. Kimlik işleçlerini incelediğimiz konuda bundan bir miktar bahsetmiş ve orada id() adlı bir fonksiyondan söz etmiştik.

Orada şöyle bir örnek vermiştik:

```
>>> a = 50
>>> id(a)

505494576

>>> kardiz = "Elveda Zalim Dünya!"
>>> id(kardiz)

14461728
```

Orada söylediğimiz ve yukarıdaki örneklerden de bir kez daha gördüğünüz gibi, Python’daki her nesnenin kimliği eşşiz, tek ve benzersizdir.

input()

Bu fonksiyonun ne işe yaradığını gayet iyi biliyorsunuz. input() adlı bu gömülü fonksiyonu kullanarak kullanıcı ile veri alışverişinde bulunabiliyoruz.

format()

Bu gömülü fonksiyonun görevi, daha önce karakter dizilerini işlerken, karakter dizilerinin bir metodu olarak öğrendiğimiz format() metodununa benzer bir şekilde, karakter dizilerini biçimlendirmektir. Ancak format() fonksiyonu, daha önce öğrendiğimiz format() metoduna göre daha dar kapsamlıdır. format() metodunu kullanarak oldukça karmaşık karakter dizisi biçimlendirme işlemlerini gerçekleştirebiliriz, ama birazdan inceleyeceğimiz format() gömülü fonksiyonu yalnızca tek bir değeri biçimlendirmek için kullanılır.

Basit bir örnek verelim:

```
>>> format(12, '.2f')
```

‘12.00’

Yukarıdaki ifadeyi daha önce gördüğümüz format() metodu ile şu şekilde yazabiliriz:

```
>>> '{:.2f}'.format(12)

'12.00'
```

filter()

Bu gömülü fonksiyon yardımıyla dizi niteliği taşıyan nesneler içindeki öğeler üzerinde belirli bir ölçüte göre bir süzme işlemi uygulayabiliriz. Dilerseniz filter() fonksiyonunun görevini bir örnek üzerinden anlamaya çalışalım.

Diyelim ki elimizde şöyle bir liste var:

```
>>> [400, 176, 64, 175, 355, 13, 207, 298, 397, 386, 31, 120, 120, 236,
    241, 123, 249, 364, 292, 153]
```

Amacımız bu liste içindeki tek sayıları süzmek.

Daha önce öğrendiğimiz yöntemleri kullanarak bu görevi şu şekilde yerine getirebiliriz:

```
>>> for i in l:
...     if i % 2 == 1:
...             print(i)
...
175
355
13
207
```

```
397
31
241
123
249
153
```

Hatta eğer istersek liste üreteçlerini kullanarak aynı işlemi daha kısa bir yoldan da halledebiliriz:

```
>>> [i for i in l if i % 2 == 1]

[175, 355, 13, 207, 397, 31, 241, 123, 249, 153]
```

İşte Python, yukarıdaki işlemi yapabilmemiz için bize üçüncü bir yol daha sunar. Bu üçüncü yolun adı filter() fonksiyonudur. Dikkatlice bakın:

```
def tek(sayı):
    return sayı % 2 == 1

print(*filter(tek, l))
```

Dilerseniz bu kodları daha iyi anlayabilmek için filter() fonksiyonuna biraz daha yakından bakalım...

filter() fonksiyonu toplam iki parametre alır. Bu parametrelerden ilki ölçütü belirleyen fonksiyon, ikincisi ise bu ölçütün uygulanacağı öğedir. Yukarıdaki örneğe baktığımızda, tek() adlı fonksiyonun, *l* adlı öğe üzerine uygulandığını görüyoruz.

Yukarıdaki örnekte ilk olarak tek() adlı bir fonksiyon tanımladık:

```
def tek(sayı):
    return sayı % 2 == 1
```

Bu fonksiyonun görevi, kendisine parametre olarak verilen bir sayının tek sayı olup olmadığını sorgulamak. Eğer verilen parametre bir tek sayı ise fonksiyonumuz *True* değerini, tek sayı değilse *False* değerini döndürecektir. İsterseniz fonksiyonumuzu test edelim:

```
print(tek(12))
```

12 sayısı bir tek sayı olmadığı için fonksiyonumuz bize *False* çıktısı verir.

Bir de şuna bakalım:

```
print(tek(117))
```

117 sayısı ise bir tek sayıdır. Bu nedenle fonksiyonumuz bize *True* değerini verecektir.

İşte biz bu fonksiyonu, filter() fonksiyonu yardımıyla şu liste üzerine uygulayacağız:

```
l = [400, 176, 64, 175, 355, 13, 207, 298, 397, 386, 31,
    120, 120, 236, 241, 123, 249, 364, 292, 153]
```

Dediğimiz gibi, filter() fonksiyonu, dizi özelliği taşıyan nesneler üzerinde belli bir ölçüte göre filtreleme işlemi yapmamızı sağlar. Biz de biraz önce tanımladığımız tek() adlı fonksiyonu *l* adlı bu listeye uygulayarak liste içindeki tek sayıları filtreleyeceğiz.

filter() fonksiyonunu çalıştıralım:

```
>>> filter(tek, l)
```

Burada filter() fonksiyonuna ilk parametre olarak tek() fonksiyonunu verdik. İkinci parametremiz ise bu fonksiyonu uygulayacağımız *l* adlı liste. Amacımız, *l* adlı liste üzerine tek() fonksiyonunu uygulayarak, bu liste içindeki öğelerden *True* çıktısı verenleri (yani tek sayıları) ayıklamak.

Şimdi de yukarıdaki koddan aldığımız çıktıya bakalım:

```
<filter object at 0x00F74F30>
```

Gördüğünüz gibi, bu fonksiyonu bu şekilde kullandığımızda elde ettiğimiz şey bir 'filtre nesnesi'. Bu nesne içindeki öğeleri görebilmek için ne yapabileceğimizi biliyorsunuz:

```
>>> list(filter(tek, l))

[175, 355, 13, 207, 397, 31, 241, 123, 249, 153]
```

veya:

```
>>> print(*filter(tek, l))

175 355 13 207 397 31 241 123 249 153
```

ya da:

```
>>> [i for i in filter(tek, l)]

[175, 355, 13, 207, 397, 31, 241, 123, 249, 153]
```

Gördüğünüz gibi, gerçekten de *l* adlı liste içindeki bütün tek sayılar süzüldü.

Gelin isterseniz filter() fonksiyonunu biraz daha iyi anlayabilmek için basit bir çalışma yapalım.

Elimizde bir sınıftaki öğrencilerin Matematik sınavından aldığı notları içeren bir sözlük var:

```
notlar = {'Ahmet'   : 60,
          'Sinan'   : 50,
          'Mehmet'  : 45,
          'Ceren'   : 87,
          'Selen'   : 99,
          'Cem'     : 98,
          'Can'     : 51,
          'Kezban'  : 100,
          'Hakan'   : 66,
          'Mahmut'  : 80}
```

Okulda kullanılan not sistemine göre yukarıdaki notları şu şekilde sınıflandırabiliyoruz:

```
def not_durumu(n):
    if n in range(0, 50): return 'F'
    if n in range(50, 70): return 'D'
    if n in range(70, 80): return 'C'
    if n in range(80, 90): return 'B'
    if n in range(90, 101): return 'A'
```

Buna göre mesela print(not_durumu(54)) gibi bir kod yazdığımızda bu notun karşılık geldiği 'D' sayısını alabiliyoruz. Peki biz bu notları belli bir ölçüte göre süzmek, ayıklamak istersek ne yapabiliriz? Örneğin notu 70'ten yukarı olan öğrencileri listelemek istersek nasıl bir kod yazabiliriz?

İşte böyle bir durumda filter() adlı gömülü fonksiyonu kullanabiliriz:

```
notlar = {'Ahmet'   : 60,
          'Sinan'   : 50,
          'Mehmet'  : 45,
          'Ceren'   : 87,
          'Selen'   : 99,
          'Cem'     : 98,
          'Can'     : 51,
          'Kezban'  : 100,
          'Hakan'   : 66,
          'Mahmut'  : 80}

def süz(n):
    return n >= 70

print(*filter(süz, notlar.values()))
```

Gördüğünüz gibi, filter() fonksiyonu, süz() adlı fonksiyon ile belirlediğimiz ölçütü *notlar* adlı sözlüğün değerleri üzerine tek tek uygulamamızı sağlıyor.

hash()

Bu fonksiyon, belirli türdeki nesnelere bir karma değeri vermemizi sağlar. Örneğin:

```
>>> hash('istihza')

-840510580

>>> hash('python')

212829695
```

Ancak bu fonksiyonun ürettiği çıktı aynı nesne için bütün sistemlerde aynı olmayabilir. Yani örneğin yukarıdaki hash('istihza') komutu 32 bitlik ve 64 bitlik işletim sistemlerinde birbirinden farklı sonuçlar verebilir. Ayrıca bu fonksiyonun ürettiği karma değerlerinin birbiriyle çakışma ihtimali de yüksektir. Dolayısıyla bu fonksiyonu kullanarak, mesela parola girişleri için karma değeri üretmek doğru olmaz.

isinstance()

Hatırlarsanız daha ilk derslerimizde öğrendiğimiz type() adlı bir fonksiyon vardı. Bu fonksiyonu bir nesnenin hangi veri tipinde olduğunu tespit etmek için kullanıyorduk:

```
>>> type('istihza')

<class 'str'>
```

İşte buna benzer şekilde, tip denetimi için kullanabileceğimiz bir fonksiyon daha var. Bu fonksiyonun adı isinstance().

Bu fonksiyonu şöyle kullanıyoruz:

```
>>> isinstance('istihza', str)

True
```

Gördüğünüz gibi 'istihza' gerçekten bir karakter dizisi (str) olduğu için komutumuz *True* çıktısı veriyor.

Bir de şuna bakalım:

```
>>> isinstance('istihza', list)

False
```

'istihza' bir liste (list) olmadığı için komutumuz bu kez *False* çıktısı verdi.

len()

Bu fonksiyon yardımıyla nesnelerin uzunluklarını hesaplayabileceğimizi biliyorsunuz:

```
>>> len('istihza')

7

>>> l = [1, 4, 5, 3, 2, 9, 10]
>>> len(l)

7
```

map()

Diyelim ki elimizde şöyle bir liste var:

```
>>> l = [1, 4, 5, 4, 2, 9, 10]
```

Amacımız bu liste içindeki her öğenin karesini hesaplamak. Bunun için şöyle bir yol izleyebiliriz:

```
>>> for i in l:
...     i ** 2
...
1
16
25
16
4
81
100
```

Böylece, istediğimiz gibi, bütün öğelerin karesini bulmuş olduk. Bu tür bir işlemi yapmanın bir başka yolu da map() adlı bir gömülü fonksiyondan yararlanmaktır. Dikkatlice bakın:

```
>>> def karesi(n):
...     return n ** 2
...
```

Burada bir *n* sayısının karesini hesaplayan bir fonksiyon tanımladık. Şimdi bu fonksiyonu *l* listesinin bütün öğeleri üzerine uygulayacağız:

```
>>> list(map(karesi, l))

[1, 16, 25, 16, 4, 81, 100]
```

max()

max() gömülü fonksiyonunun görevi, bir dizi içindeki en büyük öğeyi vermektir. Bu fonksiyon birkaç farklı parametre alır ve verdiği çıktı, aldığı parametrelerin türüne ve sayısına bağlı olarak değişiklik gösterebilir.

Bu fonksiyonu en basit şu şekilde kullanabilirsiniz:

```
>>> max(1, 2, 3)

3
```

max() fonksiyonu yukarıdaki şekilde çalıştırıldığında, kendisine verilen parametreler arasında en büyük olanı bulacaktır. Yukarıdaki parametrelerden en büyüğü *3* olduğu için de yukarıdaki komut *3* çıktısı verecektir.

Yukarıdaki kodların sağladığı etkiyi şu şekilde de elde edebiliriz:

```
>>> liste = [1, 2, 3]
>>> max(liste)

3
```

max() fonksiyonu yukarıda gösterildiği gibi birtakım isimsiz parametrelerle birlikte *key* adlı isimli bir parametre de alır. Bu parametre yardımıyla max() fonksiyonunun 'en büyük' kavramını hangi ölçüte göre seçeceğini belirleyebiliriz. Örneğin:

```
>>> isimler = ['ahmet', 'can', 'mehmet', 'selin', 'abdullah', 'kezban']
>>> max(isimler, key=len)

'abdullah'
```

max() fonksiyonu öntanımlı olarak, 'en büyük' kavramını sayısal büyüklük üzerinden değerlendirir. Yani herhangi bir *key* parametresi kullanılmadığında, bu fonksiyon otomatik olarak bir dizi içindeki en büyük sayıyı bulur. Ancak eğer biz istersek, yukarıdaki örnekte olduğu gibi, 'en büyük' kavramının uzunluk cinsinden değerlendirilmesini de sağlayabiliriz.

Yukarıdaki örnekte elimizde şöyle bir liste var:

```
>>> isimler = ['ahmet', 'can', 'mehmet', 'selin', 'abdullah', 'kezban']
```

Amacımız bu liste içindeki isimler arasından, en fazla harf içerenini bulmak. Bildiğiniz gibi Python'da bir karakter dizisinin uzunluğunu belirlemek için len() adlı bir fonksiyondan yararlanıyoruz. İşte aşağıdaki kod yardımıyla da max() fonksiyonunun 'en büyük' ölçütünü len() fonksiyonu üzerinden değerlendirmesini sağlıyoruz:

```
>>> max(isimler, key=len)
```

Bu arada *key* fonksiyonuna len() fonksiyonunu parantezsiz olarak verdiğimize dikkat edin.

Gelin isterseniz max() fonksiyonunu biraz daha iyi anlamak için ufak bir çalışma yapalım.

Diyelim ki elimizde şöyle bir sözlük var:

```
askerler = {'ahmet'    : 'onbaşı',
            'mehmet'   : 'teğmen',
            'ali'      : 'yüzbaşı',
            'cevat'    : 'albay',
            'berkay'   : 'üsteğmen',
            'mahmut'   : 'binbaşı'}
```

Amacımız bu sözlük içindeki en yüksek askeri rütbeyi bulmak. İşte bunun için max() fonksiyonundan yararlanabiliriz.

Bildiğiniz gibi, max() fonksiyonu ölçüt olarak sayısal büyüklüğü göz önüne alıyor. Elbette askeri rütbeleri böyle bir ölçüte göre sıralamak pek mümkün değil. Ama eğer şöyle bir fonksiyon yazarsak işler değişir:

```
def en_yüksek_rütbe(rütbe):
    rütbeler = {'er'       : 0,
                'onbaşı'   : 1,
                'çavuş'    : 2,
                'asteğmen' : 3,
                'teğmen'   : 4,
                'üsteğmen' : 5,
                'yüzbaşı'  : 6,
                'binbaşı'  : 7,
                'yarbay'   : 8,
                'albay'    : 9}

    return rütbeler[rütbe]
```

Burada, rütbelerin her birine bir sayı verdik. En küçük rütbe en düşük sayıya, en yüksek rütbe ise en büyük sayıya sahip. Fonksiyonumuz bir adet parametre alıyor. Bu parametrenin adı *rütbe*. Yazdığımız fonksiyon, kendisine parametre olarak verilecek rütbeyi *rütbeler* adlı sözlükte arayıp, bu rütbeye

karşılık gelen sayıyı döndürecek.

Bu bilgileri kullanarak kodlarımızın son halini düzenleyelim:

```
def en_yüksek_rütbe(rütbe):
    rütbeler = {'er'        : 0,
                'onbaşı'    : 1,
                'çavuş'     : 2,
                'asteğmen'  : 3,
                'teğmen'    : 4,
                'üsteğmen'  : 5,
                'yüzbaşı'   : 6,
                'binbaşı'   : 7,
                'yarbay'    : 8,
                'albay'     : 9}

    return rütbeler[rütbe]

askerler = {'ahmet': 'onbaşı',
            'mehmet': 'teğmen',
            'ali': 'yüzbaşı',
            'cevat': 'albay',
            'berkay': 'üsteğmen',
            'mahmut': 'binbaşı'}
```

Artık max() fonksiyonunu *askerler* adlı sözlük üzerine uygulayabiliriz:

```
print(max(askerler.values(), key=en_yüksek_rütbe))
```

Böylece *askerler* adlı sözlüğün değerleri en_yüksek_rütbe() fonksiyonunun sunduğu ölçüt üzerinden ele alınacak ve en büyük sayı değerine sahip olan rütbe çıktı olarak verilecektir.

Yukarıdaki kodlar problemimizi çözüyor. Peki ama ya biz rütbe ile birlikte bu rütbeyi taşıyan askerin adını da öğrenmek istersek nasıl bir kod yazacağız?

Bunun için de şöyle bir kod yazabiliriz:

```
for k, v in askerler.items():
    if askerler[k] in max(askerler.values(), key=en_yüksek_rütbe):
        print(v, k)
```

Eğer isterseniz burada *in* işleci yerine == işlecini de kullanabilirsiniz:

```
for k, v in askerler.items():
    if askerler[k] in max(askerler.values(), key=en_yüksek_rütbe):
```

```
    print(v, k)
```

min()

min() fonksiyonu max() fonksiyonunun tam tersini yapar. Bildiğiniz gibi max() fonksiyonu bir dizi içindeki en büyük öğeyi buluyordu. İşte min() fonksiyonu da bir dizi içindeki en küçük öğeyi bulur. Bu fonksiyonun kullanımı max() ile aynıdır.

open()

Bu fonksiyon herhangi bir dosyayı açmak veya oluşturmak için kullanılır. Eğer dosyanın açılması veya oluşturulması esnasında bir hata ortaya çıkarsa IOError türünde bir hata mesajı verilir.

Bu fonksiyonun formülü şudur:

```
>>> open(dosya_adi, mode='r', buffering=-1, encoding=None,
...     errors=None, newline=None, closefd=True, opener=None)
```

Gördüğünüz gibi, bu fonksiyon pek çok farklı parametre alabiliyor. Biz şimdiye kadar bu parametrelerin yalnızca en sık kullanılanlarını işlemiştik. Şimdi ise geri kalan parametrelerin ne işe yaradığını da ele alacağız.

Yukarıdaki formülden de görebileceğiniz gibi, open() fonksiyonunun ilk parametresi *dosya_adi*'dır. Yani açmak veya oluşturmak istediğimiz dosya adını bu parametre ile belirtiyoruz:

```
>>> open('falanca_dosya.txt')
```

Elbette eğer açmak istediğiniz dosya, o anda içinde bulunduğunuz dizinde değilse dosya adı olarak, o dosyanın tam adresini yazmanız gerekir. Mesela:

```
>>> open('/home/istihza/Desktop/dosya.txt')
```

Bu arada, dosya adresini yazarken ters taksim yerine düz taksim işaretlerini kullanmak daha doğru olacaktır. Bu taksim türü hem Windows'ta hem de GNU/Linux'ta çalışır. Ancak eğer ters taksim işaretlerini kullanacaksanız, dosya yolu içindeki sinsi kaçış dizilerine karşı dikkatli olmalısınız:

```
>>> f = open('test\nisan.txt')

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 22] Invalid argument: 'test\nisan.txt'
```

Burada problemin *nisan.txt* adlı dosyanın ilk harfi ile, bundan önce gelen ters taksim işaretinin birleşerek tesadüfen bir kaçış dizisi oluşturması olduğunu biliyorsunuz. Bu tür hatalara karşı ters

taksim yerine düz taksim işaretlerini kullanabileceğiniz gibi *r* adlı kaçış dizisinden de yararlanabilirsiniz:

```
f = open(r'test\nisan.txt')
```

open() fonksiyonunun ikinci parametresi olan *mode*'un da ne olduğunu biliyorsunuz. Bu parametre yardımıyla, herhangi bir dosyayı hangi kipte açmak istediğimizi belirtebiliyoruz.

Bildiğiniz gibi, eğer *mode* parametresine herhangi bir değer vermezseniz Python ilgili dosyayı okuma kipinde açacaktır.

Bu parametreye verebileceğiniz değerleri şöyle özetleyebiliriz:

Karakter	Anlamı
'r'	Okuma kipidir. Öntanımlı değer budur.
'w'	Yazma kipidir. Eğer belirtilen adda dosya zaten varsa o dosya silinir.
'x'	Yeni bir dosya oluşturulup yazma kipinde açılır.
'a'	Dosya ekleme kipinde açılır. Bu kip ile, varolan bir dosyanın sonuna eklemeler yapılabilir.
'b'	Dosyaları ikili kipte açmak için kullanılır.
't'	Dosyaları metin kipinde açmak için kullanılır. Öntanımlı değerdir.
'+'	Aynı dosya üzerinde hem okuma hem de yazma işlemleri yapılmasını sağlar.

open() fonksiyonunun alabileceği bir başka parametre de *buffering* parametresidir. Bildiğiniz gibi, open() fonksiyonuyla bir dosyayı açıp bu dosyaya veri girdiğimizde bu veriler önce tampona alınacak, dosya kapandıktan sonra ise tamponda bekletilen veriler dosyaya işlenecektir. İşte bu *buffering* parametresi yardımıyla bu tampona alma işleminin nasıl yürüyeceğini belirleyebiliriz.

Eğer dosyaya işlenecek verilerin tampona alınmadan doğrudan dosyaya işlenmesini isterseniz *buffering* değerini *0* olarak belirlersiniz. Yalnız bu değer sadece ikili kipte etkindir. Yani bir dosyayı eğer metin kipinde açıyorsanız *buffering* parametresinin değerini *0* yapamazsınız.

Eğer dosyaya veri işlerken tampona alınan verilerin satır satır dosyaya eklenmesini isterseniz *buffering* değerini *1* olarak belirlersiniz. Bunun nasıl çalıştığını anlamak için şu örneği dikkatlice inceleyin:

```
>>> f = open('ni.txt', 'w', buffering=1)
>>> f.write('birinci satır\n')

14

>>> f.write('ikinci satır\n')

13
```

```
>>> f.write('aaa')

3

>>> f.write('\n')

1
```

Burada her write() komutundan sonra *ni.txt* adlı dosyayı açıp bakarsanız, şu durumu görürsünüz:

#. f.write('birinci satır\n') komutuyla dosyaya bir satırlık veri ekledik ve bu veri dosyaya anında işlendi.

#. f.write('ikinci satır\n') komutuyla dosyaya bir satırlık başka bir veri daha ekledik ve bu veri de dosyaya anında işlendi.

#. f.write('aaa') komutuyla eklenen veri satır değil. Çünkü satır sonuna işaret eden satır başı kaçış dizisini kullanmadık.

#. f.write('\n') komutuyla satır başı kaçış dizisini eklediğimiz anda bir önceki karakter dizisi ('aaa') de dosyaya eklenecektir.

Ancak *buffering* parametresi bu *1* değerini yalnızca metin kipinde alabilir. Bu kısıtlamayı da aklımızın bir kenarına not edelim...

0 ve *1* dışında *buffering* parametresine 1'den büyük bir değer verdiğinizde ise tampon boyutunun ne kadar olacağını kendiniz belirlemiş olursunuz.

Yalnız çoğu durumda *buffering* parametresine herhangi bir özel değer atamanız gerekmeyecektir. Bu parametreye herhangi bir değer atamadığınızda, kullandığınız işletim sistemi tampona alma işlemlerininin nasıl yürütüleceğine ve tampon boyutuna kendisi karar verecektir. İşletim sisteminin sizin yerinize verdiği bu karar da çoğunlukla istediğiniz şey olacaktır... Eğer kendi sisteminizde öntanımlı tampon boyutunun ne olduğunu merak ediyorsanız şu komutları kullanabilirsiniz:

```
>>> import io
>>> io.DEFAULT_BUFFER_SIZE
```

Çoğu sistemde bu değer 4096 ve 8192 bayt olacaktır.

open() fonksiyonunun alabileceği bir başka parametre de *encoding* parametresidir. Bu parametre, dosyanın hangi karakter kodlaması ile açılacağını belirler. Örneğin bir dosyayı 'UTF-8' karakter kodlaması ile açmak için şu komutu kullanıyoruz:

```
>>> f = open('dosya', encoding='utf-8')
```

Üzerinde işlem yaptığınız dosyalarda özellikle Türkçe karakter sorunları yaşamak istemiyorsanız, bir dosyayı açarken mutlaka *encoding* parametresinin değerini de ayarlamanızı tavsiye ederim.

Bir dosyayı açarken veya okurken herhangi bir karakter kodlama hatası ile karşılaştığınızda Python'ın ne tepki vermesi gerektiğini ise *errors* adlı parametre yardımıyla belirleyebilirsiniz.

Eğer bu parametreye *strict* değerini verirseniz karakter kodlama hataları programınızın ValueError türünde bir hata vererek çalışmayı kesmesine neden olacaktır. Bu parametreye herhangi bir değer vermediğinizde de Python sanki *strict* değerini vermişsiniz gibi davranır.

Eğer *errors* parametresine *ignore* değerini verirseniz kodlama hataları görmezden gelinecek, bu hataya sebep olan karakter silinecektir. Yalnız bu değerin veri kaybına yol açma ihtimalini de göz önünde bulundurmalısınız.

Eğer *errors* parametresine *replace* değerini verirseniz kodlama hatasına yol açan karater '?' veya 'ufffd' karakterleri ile değiştirilecektir.

open() fonksiyonunun kabul ettiği bir başka parametre de *newline* adlı parametredir. Peki bu parametre ne işe yarar?

Windows ve GNU/Linux işletim sistemleri satır sonlarını birbirlerinden farklı şekilde gösterir. GNU/Linux'ta yazılmış dosyalarda satır sonları *\n* karakteri ile gösterilirken, Windows'ta yazılmış dosyalarda satır sonunda *\r\n* karakterleri bulunur. Eğer Windows ve GNU/Linux sistemleri arasında dosya alışverişi yapıyorsanız kimi durumlarda bu farklılık çeşitli sorunların ortaya çıkmasına yol açabilir. İşte dosyalarınızın hangi satır sonu karakterine sahip olacağını yukarıda bahsettiğimiz *newline* adlı parametre ile belirleyebilirsiniz. Örneğin:

```
>>> f = open('dosya', newline='\n')
```

Bu şekilde dosyanız hangi işletim sisteminde olursa olsun satır sonlarında *\n* karakterine sahip olacaktır.

Dosyaların metotlarını incelerseniz o listede fileno() adlı bir metodun olduğunu göreceksiniz. Bu metot, bize bir dosyanın 'dosya tanımlayıcısını' (*file descriptor*) verir. Dosya tanımlayıcıları, dosyaya işaret eden pozitif tam sayılardır. *0*, *1* ve *2* sayıları standart girdi, standart çıktı ve standart hata dosyalarına ayrılmış olduğu için, sizin açtığınız ve üzerinde işlem yaptığınız dosyaların tanımlayıcıları *2* sayısından büyük olacaktır.

Bir örnek verelim:

```
>>> f = open('ni.txt')
>>> f.fileno()

3
```

İşte burada gördüğünüz sayı, *ni.txt* adlı dosyanın 'dosya tanımlayıcısıdır. Her dosyanın dosya

tanımlayıcısı benzersizdir:

```
>>> g = open('zi.txt')
>>> g.fileno()

4
```

Python'da bir dosyayı open() fonksiyonuyla açarken dosya adını vermenin yanısıra, dosyanın tanımlayıcısını da kullanabilirsiniz:

```
>>> z = open(4)
```

veya:

```
>>> z = open(g.fileno())
```

Bu sayede, eğer isterseniz, elinizdeki dosyalarla daha ileri düzeyli işlemler yapabilirsiniz. Bir örnek verelim.

Dediğimiz gibi, bir dosyanın tanımlayıcısı tek ve benzersizdir. Farklı dosyalar aynı tanımlayıcılara sahip olmaz:

```
>>> a = open('aaa.txt')
>>> a.fileno()

3

>>> b = open('bbb.txt')
>>> b.fileno()

4
```

Şimdi şu örneklere bakın:

```
>>> c = open(b.fileno(), closefd=False)
```

Bu şekilde *b* adlı dosyanın tanımlayıcısını kullanarak, aynı dosyayı bir de *c* adıyla açtık. Ancak burada kullandığımız *closefd=False* parametresine dikkat edin. Normalde dosyayı kapattığımızda dosyanın tanımlayıcısı serbest kalır ve başka bir dosya açıldığında bu tanımlayıcı yeni dosyaya atanır. Ama *closefd* parametresine False değeri verdiğimizde dosya kapansa bile, o dosyaya ait dosya tanımlayıcısı varolmaya devam edecektir.

pow()

Daha önceki derslerimizde pek çok kez örneklerini verdiğimiz bu fonksiyon İngilizcedeki *power*

(kuvvet) kelimesinin kısaltmasından oluşur. Adının anlamına uygun olarak, bu fonksiyonu bir sayının kuvvetlerini hesaplamak için kullanıyoruz.

Bu fonksiyon en temel şekilde şöyle kullanılır:

```
>>> pow(2, 3)

8
```

Bu komutla *2* sayısının 3. kuvvetini hesaplamış oluyoruz.

pow() fonksiyonu toplamda üç farklı parametre alır. İlk iki parametrenin ne olduğunu yukarıda örnekledik. Üçüncü parametre ise kuvvet hesaplaması sonucu elde edilen sayının modülüsünü hesaplayabilmemizi sağlar. Yani:

```
>>> pow(2, 3, 2)

0
```

Burada yaptığımız şey şu: Öncelikle *2* sayısının 3. kuvvetini hesapladık. Elde ettiğimiz sayı *8*. Ardından da bu sayının *2*'ye bölünmesi işleminden kalan sayıyı elde ettik. Yani *0*. Çünkü bildiğiniz gibi 8 % 2 işleminin sonucu *0*'dır. Dolayısıyla yukarıdaki komut şuna eşdeğerdir:

```
>>> (2 ** 3) % 2

0
```

Ancak önceki derslerimizde de söylediğimiz gibi, pow() fonksiyonu çoğunlukla yanlızca ilk iki parametresi ile birlikte kullanılır:

```
>>> pow(12, 2)

144
```

print()

print() fonksiyonunu artık gayet iyi tanıyoruz. Bu fonksiyonu, bildiğiniz gibi, kullanıcılarımıza birtakım mesajlar göstermek için kullanıyoruz.

Kullanımını daha önce ayrıntılı bir şekilde anlatmış olduğumuz bu fonksiyonu şu şekilde formüle edebiliriz:

```
print(deg1, deg2, deg3, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

Burada;

degx: Çıktı verilecek değerlerin ne olduğunu belirtir. Buraya 256 adete kadar değer yazabilirsiniz.

sep: Çıktı verilirken değerlerin arasına hangi karakterin yerleştirileceğini belirtir. Bu değer öntanımlı olarak boşluk karakteridir.

end: Çıktı verilecek son değerin ardından hangi karakterin iliştirileceğini belirtir. Bu değer öntanımlı olarak satır başı (*n*) karakteridir.

file: Çıktıların hangi dosyaya yazılacağını belirtir. Öntanımlı olarak bu parametrenin değeri *sys.stdout*'tur. Yani print() fonksiyonu çıktılarını öntanımlı olarak standart çıktı konumuna gönderir.

flush: Bildiğiniz gibi, herhangi bir dosyaya yazma işlemi sırasında dosyaya yazılacak değerler öncelikle tampona alınır. İşlem tamamlandıktan sonra tampondaki bu değerler topluca dosyaya aktarılır. İşte bu parametre, değerleri tampona almadan doğrudan dosyaya gönderebilmemizi sağlar. Bu parametrenin öntanımlı değeri *False*'tur. Yani değerler dosyaya yazılmadan önce öntanımlı olarak öncelikle tampona gider. Ama eğer biz bu parametrenin değerini *True* olarak değiştirirsek, değerler doğrudan dosyaya yazılır.

quit()

Bu fonksiyonu programdan çıkmak için kullanıyoruz. Eğer bu fonksiyonu etkileşimli kabukta verecek olursanız etkileşimli kabuk kapanacaktır.

range()

Bu fonksiyonu belli bir aralıktaki sayıları listelemek için kullanıyoruz. Yani mesela *0* ile *10* arası sayıların listesini almak istersek şöyle bir komut yazabiliriz:

```
>>> l = range(0, 10)
```

Ancak burada dikkat etmemiz gereken bir özellik var: Bu fonksiyon aslında doğrudan herhangi bir sayı listesi oluşturmaz. Yukarıda *l* değişkenine atadığımız komutu ekrana yazdırırsak bunu daha net görebilirsiniz:

```
>>> print(l)

range(0, 10)
```

Bir de bu verinin tipine bakalım:

```
>>> type(l)

<class 'range'>
```

Gördüğünüz gibi, elimizdeki şey aslında bir sayı listesi değil, bir 'range' (aralık) nesnesidir. Biz bu

nesneyi istersek başka veri tiplerine dönüştürebiliriz. Mesela bunu bir listeye dönüştürelim:

```
>>> list(l)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

veya bir demete:

```
>>> tuple(l)

(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

ya da bir kümeye veya dondurulmuş kümeye:

```
>>> set(l) #küme

{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

>>> frozenset(l) #dondurulmuş küme

frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9})
```

Bu 'range' nesnesini istediğiniz veri tipine dönüştürdükten sonra, dönüştürdüğünüz veri tipinin kuralları çerçevesinde elinizdeki veriyi işleyebilirsiniz.

range() fonksiyonundan elde ettiğiniz 'range' nesnesinin içeriğini elde etmek için bunu başka bir veri tipine dönüştürmenin yanısıra, bu nesne üzerinde bir for döngüsü de kurabilirsiniz:

```
>>> for i in range(10):
...     print(i)
...
0
1
2
3
4
5
6
7
8
9
```

Ya da yıldızlı parametreler yardımıyla bu nesneyi print() fonksiyonuna göndererek, bu nesneyi istediğiniz gibi evirip çevirebilirsiniz:

```
>>> print(*range(10), sep=', ')

0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Esasında, yukarıda nasıl kullanılacağına dair bazı örnekler verdiğimiz bu range() fonksiyonunu temel olarak şu şekilde formüle edebiliriz:

```
range(başlangıç_değer, bitiş_değeri, atlama_değeri)
```

Aşağıdaki örneği tekrar ele alalım:

```
>>> range(0, 10)
```

Burada *0* başlangıç değeri, *10* ise bitiş değeridir. Buna göre oluşturulacak sayılar *0* ile *10* arasında olacaktır. Yalnız burada üretilecek sayı listesinde *0* sayısının dahil, *10* sayısının ise hariç olduğunu unutmuyoruz. Yani bu komutun bize vereceği ilk sayı *0*; son sayı ise *9* olacaktır.

range() fonksiyonunda başlangıç değerinin öntanımlı değeri *0*'dır. Dolayısıyla istersek biz yukarıdaki komutu şöyle de yazabiliriz:

```
>>> range(10)
```

Böylece Python bizim range(0, 10) komutunu kastettiğimizi varsayacaktır. Elbette eğer başlangıç değerinin *0* dışında bir değer olmasını istiyorsanız bunu özellikle belirtmeniz gerekir:

```
>>> range(10, 100)
```

Bu komut bize *10* ile (*10* dahil) *100* arası (*100* hariç) sayıları içeren bir 'range' nesnesi verecektir.

Yukarıda verdiğimiz formülden de göreceğiniz gibi, *başlangıç_değer* ve *bitiş_değer* dışında range() fonksiyonu üçüncü bir parametre daha alabiliyor. Bu parametreye *atlama_değeri* adı verdik. Bu parametreyi şöyle kullanıyoruz:

```
>>> list(range(0, 10, 2))

[0, 2, 4, 6, 8]
```

Gördüğünüz gibi, range() fonksiyonuna üçüncü parametre olarak verdiğimiz *2* sayısı, *0* ile *10* arası sayıların ikişer ikişer atlanarak üretilmesini sağladı.

reversed()

Diyelim ki elimizde şöyle bir liste var:

```
>>> isimler = ['ahmet', 'mehmet', 'veli', 'ayşe', 'çiğdem', 'ışık']
```

Eğer bu listedeki isimleri ters çevirmek, yani şöyle bir liste elde etmek isterseniz:

```
['ışık', 'çiğdem', 'ayşe', 'veli', 'mehmet', 'ahmet']
```

... ne yapmanız gerektiğini biliyorsunuz. Bu amaç için liste dilimleme yöntemlerinden yararlanabilirsiniz:

```
>>> isimler[::-1]

['ışık', 'çiğdem', 'ayşe', 'veli', 'mehmet', 'ahmet']
```

İşte aynı işlevi reversed() adlı bir fonksiyon yardımıyla da yerine getirebilirsiniz:

```
>>> reversed(isimler)

<list_reverseiterator object at 0x00EB9710>
```

Gördüğünüz gibi, tıpkı range() fonksiyonunda olduğu gibi, reversed() fonksiyonu da bize ürettiği öğelerin kendisi yerine, bir 'nesne' veriyor. Ama tabii ki bu bizim için bir sorun değil. Biz bu nesnenin içeriğini nasıl elde edebileceğimizi gayet iyi biliyoruz:

```
>>> list(reversed(isimler))

['ışık', 'çiğdem', 'ayşe', 'veli', 'mehmet', 'ahmet']
```

range() fonksiyonunu anlatırken sözünü ettiğimiz içerik elde etme yöntemlerini reversed() fonksiyonuna da uygulayabilirsiniz.

sorted()

Bu metot, daha önceki derslerimizden de bildiğiniz gibi, bir dizi içindeki öğeleri belirli bir ölçüte göre sıraya dizmemizi sağlıyor. Bununla ilgili çok basit bir örnek verelim:

```
>>> sorted('ahmet')

['a', 'e', 'h', 'm', 't']
```

Bu kodlar yardımıyla ahmet adlı karakter dizisi içindeki harfleri alfabe sırasına dizdik.

Elbette bu fonksiyonu sadece karakter dizileri üzerine uygulamıyoruz. sorted() adlı fonksiyon, dizi özelliği taşıyan her türlü nesne üzerine uygulanabilir. Mesela demetlerin ve listelerin bir dizi olduğunu biliyoruz. Dolayısıyla:

```
>>> sorted(('elma', 'armut', 'kiraz', 'badem'))
```

```
['armut', 'badem', 'elma', 'kiraz']

>>> sorted(['elma', 'armut', 'kiraz', 'badem'])
['armut', 'badem', 'elma', 'kiraz']
```

sorted() fonksiyonuna hangi türde bir veri tipi verirseniz verin, aldığınız çıktı her zaman bir liste olacaktır. Bunu unutmayın.

Gördüğünüz gibi, sorted() fonksiyonu nesneler üzerinde bir sıralama işlemi gerçekleştiriyor. Ancak bu fonksiyonun bir problemi var.

Dikkatlice bakın:

```
>>> isimler = ['ahmet', 'çiğdem', 'ışık', 'şebnem', 'zeynep', 'selin']
>>> sorted(isimler)

['ahmet', 'selin', 'zeynep', 'çiğdem', 'ışık', 'şebnem']
```

Bu fonksiyon, Türkçe karakter içeren öğeleri düzgün sıralayamaz.

Bu sorunu *kısmen* çözebilmek için locale adlı bir modül içindeki strxfrm() adlı bir fonksiyondan yararlanabilirsiniz:

```
>>> import locale
```

Henüz modülleri öğrenmemiş de olsak, bir modülü kullanabilmek için öncelikle o modülü 'içe aktarmamız' gerektiğini artık biliyorsunuz. Bu işlemi import adlı bir komut yardımıyla yaptığımızı da biliyorsunuz.

Şimdi de yerelimizi (*locale*) 'Türkçe' olarak ayarlayalım:

```
>>> locale.setlocale(locale.LC_ALL, 'tr_TR') #GNU/Linux
>>> locale.setlocale(locale.LC_ALL, 'Turkish_Turkey.1254') #Windows
```

Bu işlemleri yaptıktan sonra, sorted() fonksiyonunun *key* adlı bir parametresini kullanarak ve yine locale modülünün strxfrm() adlı fonksiyonundan faydalanarak Türkçe karakterler içeren listemizi sıralamayı deneyebiliriz:

```
>>> sorted(isimler, key=locale.strxfrm)

['ahmet', 'çiğdem', 'ışık', 'selin', 'şebnem', 'zeynep']
```

locale modülü içinde bulunan strxfrm() adlı fonksiyon, karakter dizilerinin, o anda etkin yerel neyse, ona göre muamele görmesini sağlar. Biz yukarıda yerelimizi Türkçe olarak ayarladığımız için strxfrm() fonksiyonu, sorted() ile yapılan alfabe sırasına dizme işleminin Türkçenin kurallarına

göre yapılmasını sağlıyor.

Ancak bu yöntemin de sorunlu olduğunu bir süre sonra kendiniz de farkedeceksiniz. Mesela şu örneği inceleyin:

```
>>> sorted('afgdhkıi', key=locale.strxfrm)

['a', 'd', 'f', 'g', 'h', 'i', 'ı', 'k']
```

Gördüğünüz gibi, listede 'i' harfi 'ı' harfinden önce geliyor. Ama aslında bunun tersi olmalıydı.

İşte böyle bir durumda, kendi sıralama mekanizmamızı kendimiz icat etmeliyiz. Peki ama nasıl?

Bilgisayarlar farklı dillerdeki karakterleri her zaman doğru sıralayamasa da, sayıları her zaman doğru sıralar:

```
>>> sorted([10, 9, 4, 14, 20])

[4, 9, 10, 14, 20]
```

Bilgisayarların bu özelliğinden ve Python'daki sözlük veri tipinden yararlanarak kendi sıralama mekanizmamızı rahatlıkla icat edebiliriz.

Öncelikle harflerimizi yazalım:

```
>>> harfler = "abcçdefgğhıijklmnoöprsştuüvyz"
```

Burada Türk alfabesini oluşturan harfleri sırasına göre dizdik. Şimdi bu harflerin her birine bir sayı vereceğiz:

```
>>> çevrim = {'a': 0, 'b': 1, 'c': 2, 'ç': 3, 'd': 4, 'e': 5, 'f': 6,
           'g': 7, 'ğ': 8, 'h': 9, 'ı': 10, 'i': 11, 'j': 12, 'k': 13,
           'l': 14, 'm': 15, 'n': 16, 'o': 17, 'ö': 18, 'p': 19, 'r': 20,
           's': 21, 'ş': 22, 't': 23, 'u': 24, 'ü': 25, 'v': 26, 'y': 27,
           'z': 28}
```

Yalnız böyle her harfe karşılık gelen sayıyı elle yazmak yorucu olacaktır. Bu işlemi daha kolay bir şekilde yapabilmek için farklı teknikleri kullanabilirsiniz. Mesela daha önce öğrendiğimiz sözlük üreteçlerinden yararlanabilirsiniz:

```
>>> çevrim = {i: harfler.index(i) for i in harfler}
```

Bu şekilde *harfler* değişkeni içindeki herbir harfin bir anahtar; bu harflerin *harfler* değişkeni içindeki sırasını gösteren herbir sayının ise bir değer olduğu bir sözlük oluşturmuş olduk.

Şimdi isim listemizi alalım karşımıza:

```
isimler = ["ahmet", "ışık", "ismail", "çiğdem", "can", "şule"]
```

Normal bir sorted() işleminin yanlış sonuç döndüreceğini biliyoruz:

```
>>> sorted(isimler)

['ahmet', 'can', 'ismail', 'çiğdem', 'ışık', 'şule']
```

Aynı şekilde *key* parametresine locale.strxfrm değerinin verilmesi de yetersiz kalacaktır:

```
>>> sorted(isimler, key=locale.strxfrm)

['ahmet', 'can', 'çiğdem', 'ismail', 'ışık', 'şule']
```

Ama biraz önce oluşturduğumuz *çevrim* anahtarını kullanırsak durum bambaşka olacaktır:

```
>>> sorted(isimler, key=lambda x: çevrim.get(x[0]))

['ahmet', 'can', 'çiğdem', 'ışık', 'ismail', 'şule']
```

Ancak aslında burada da oldukça sinsi bir problem var. Bu metot ile kelime listesini oluşturan kelimeleri yalnızca ilk harflerine göre sıralıyoruz (x[0]). Peki ya aynı liste içinde ilk harfleri aynı olup, sonraki harflerde farklılaşan kelimeler varsa ne olacak? Yani mesela bu metot acaba 'ismail' ve 'iskender' kelimelerini doğru bir şekilde sıralayabilir mi? Bakalım:

```
harfler = "abcçdefgğhıijklmnoöprsştuüvyz"
çevrim = {i: harfler.index(i) for i in harfler}

isimler = ["ahmet", "ışık", "ismail", "çiğdem", "can", "şule", "iskender"]

print(sorted(isimler, key=lambda x: çevrim.get(x[0])))
```

Buradan şu çıktıyı alıyoruz:

```
['ahmet', 'can', 'çiğdem', 'ışık', 'ismail', 'iskender', 'şule']
```

Gördüğünüz gibi 'ismail' ve 'iskender' kelimeleri doğru bir şekilde sıralanmadı; 'iskender' kelimesinin 'ismail' kelimesinden önce gelmesi gerekiyordu...

Demek ki şimdiye kadar kullandığımız bütün sıralama yöntemlerinin bir eksiği varmış. O halde başka bir metot bulmaya çalışalım.

Dikkatlice bakın:

```
harfler = 'abcçdefgğhıijklmnoöprsştuüvyz'
çevrim = {i: harfler.index(i) for i in harfler}
```

```
def sırala(kelime):
    return ([çevrim.get(kelime[i]) for i in range(len(kelime))])

isimler = ['ahmet', 'can', 'iskender', 'cigdem',
           'ismet', 'ismail', 'ismit', 'çiğdem',
           'ismıt', 'ışık', 'şule']

print(*sorted(isimler, key=sırala), sep='\n')
```

Gelin bu kodları biraz inceleyelim.

Burada ilk gördüğümüz kodlar şunlar:

```
harfler = 'abcçdefgğhıijklmnoöprsştuüvyz'
çevrim = {i: harfler.index(i) for i in harfler}
```

Esasında siz bu kodların anlamını biliyorsunuz. Önceki derslerimizde de aynı kodları birkaç kez kullanmıştık. Yalnız biz burada, örnek olması açısından, *harfler* değişkeni için değer olarak yalnızca küçük harfleri kullandık. Bu kodları daha kapsamlı bir program içinde kullanacaksanız bu değişkenin uygun yerlerine mesela büyük harfleri ve noktalama işaretleriyle sayıları da eklemek isteyebilirsiniz.

Sonraki satırlarda sırala() adlı bir fonksiyon tanımladık:

```
def sırala(kelime):
    return ([çevrim.get(kelime[i]) for i in range(len(kelime))])
```

Burada liste üreteçlerinden yararlandığımıza dikkatinizi çekmek isterim:

```
[çevrim.get(kelime[i]) for i in range(len(kelime))]
```

Bu kod yardımıyla *kelime* içinde geçen herbir harfi *çevrim* adlı sözlükte sorgulayarak, sözlükte ilgili harfe karşılık gelen sayıyı buluyoruz.

Aslında bu kodları daha iyi anlayabilmek için Python'daki sorted() fonksiyonunun mantığını biraz daha derinlemesine incelememiz gerekiyor. Gelin şimdi bu inceleme işini yapalım:

Diyelim ki elimizde şöyle bir liste var:

```
elemanlar = [('ahmet',      33,   'karataş'),
             ('mehmet',     45,   'arpaçbahşiş'),
             ('sevda',      24,   'arsuz'),
             ('arzu',       40,   'siverek'),
             ('abdullah',   30,   'payas'),
             ('ilknur',     40,   'kilis'),
```

```
          ('abdurrezzak', 40,    'bolvadin')]
```

Bu liste, her biri ‘isim’, ‘yaş’ ve ‘memleket’ bilgilerini içeren üç öğeli birer demetten oluşuyor. Eğer biz bu liste üzerine sorted() fonksiyonunu uygularsak:

```
print(*sorted(elemanlar), sep='\n')
```

Python elemanları demetlerin ilk öğesine göre sıralayacaktır. Yani isme göre.

Peki ya biz bu elemanları yaşa göre sıralamak istersek ne yapacağız?

Bu amacı gerçekleştirmek için şöyle bir kod yazabiliriz:

```
def sırala(liste):
   return liste[1]

elemanlar = [('ahmet',       33,    'karataş'),
          ('mehmet',     45,    'arpaçbahşiş'),
          ('sevda',      24,    'arsuz'),
          ('arzu',       40,    'siverek'),
          ('abdullah',   30,    'payas'),
          ('ilknur',     40,    'kilis'),
          ('abdurrezzak', 40,    'bolvadin')]

print(*sorted(elemanlar, key=sırala), sep='\n')
```

Bu örnek bize *key* parametresinin de ne işe yaradığını açık seçik gösteriyor. Eğer Python’ın kendi sıralama yönteminin dışında bir sıralama yöntemi uygulayacaksak, bu sıralama yönteminin ne olduğunu bir fonksiyon yardımıyla tarif edip bunu *key* parametresine değer olarak veriyoruz. Örneğin biz yukarıdaki Python’ın *elemanlar* adlı listeyi ilk sütuna (‘isim’ sütunu) göre değil, ikinci sütuna (‘yaş’ sütunu) göre sıralamasını istedik. Bunun için de şöyle bir fonksiyon yazdık:

```
def sırala(liste):
   return liste[1]
```

Bu fonksiyon, kendisine parametre olarak verilen nesnenin ikinci öğesini döndürüyor. İşte biz sorted() fonksiyonunun *key* parametresine bu fonksiyonu verdiğimizde Python sıralama işleminde *elemanlar* listesinin ikinci öğesini dikkate alacaktır. Eğer Python’ın sıralama işleminde mesela üçüncü sütunu dikkate almasını isterseniz sırala() fonksiyonunu şöyle yazabilirsiniz:

```
def sırala(liste):
   return liste[2]
```

Gördüğünüz gibi, *elemanlar* listesinin ikinci sütununda değeri aynı olan öğeler var. Mesela ‘arzu’, ‘ilknur’ ve ‘abdurrezzak’ 40 yaşında. Python bu öğeleri sıralarken, bunların listede geçtiği sırayı

dikkate alacaktır. Ama bazen biz sıralamanın böyle olmasını istemeyebiliriz. Mesela bizim istediğimiz şey, değeri aynı olan öğeler için üçüncü sütunun (veya birinci sütunun) dikkate alınması olabilir. İşte bunun için de sırala() fonksiyonunu şu şekilde tanımlayabiliriz:

```
def sırala(liste):
    return (liste[1], liste[2])
```

Gördüğünüz gibi burada sırala() fonksiyonu bize iki öğeli bir demet döndürüyor.

Kodlarımız tam olarak şöyle görünecek:

```
def sırala(liste):
    return (liste[1], liste[2])

elemanlar = [('ahmet',      33,   'karataş'),
             ('mehmet',     45,   'arpaçbahşiş'),
             ('sevda',      24,   'arsuz'),
             ('arzu',       40,   'siverek'),
             ('abdullah',   30,   'payas'),
             ('ilknur',     40,   'kilis'),
             ('abdurrezzak', 40,   'bolvadin')]

print(*sorted(elemanlar, key=sırala), sep='\n')
```

Kodlarımızı böyle yazdığımızda Python listeyi ilk olarak ikinci sütundaki ‘yaş’ değerlerine göre sıralar. Değeri aynı olan öğelerle karşılaştığında ise üçüncü sütundaki ‘memleket’ değerlerine bakar ve sıralamayı ona göre yapar.

Bütün bu açıklamalardan sonra yukarıdaki şu kodları daha iyi anlıyor olmalısınız:

```
harfler = 'abcçdefgğhıijklmnoöprsştuüvyz'
çevrim = {i: harfler.index(i) for i in harfler}

def sırala(kelime):
    return ([çevrim.get(kelime[i]) for i in range(len(kelime))])

isimler = ['ahmet', 'can', 'iskender', 'cigdem',
           'ismet', 'ismail', 'ismit', 'çiğdem',
           'ismıt', 'ışık', 'şule']

print(*sorted(isimler, key=sırala), sep='\n')
```

Biz yine de her şeyin iyiden iyine anlaşıldığından emin olmak için durumu kısaca açıklayalım. Öncelikle ilgili fonksiyonu önümüze alalım:

```
def sırala(kelime):
    return ([çevrim.get(kelime[i]) for i in range(len(kelime))])
```

Burada yaptığımız şey biraz önce yaptığımız şeyle tamamen aynı aslında. Tek fark, Python'ın sıralamada kullanmasını istediğimiz öğeleri tek tek elle yazmak yerine, bunları bir liste üreteci yardımıyla otomatik olarak belirlemek.

Eğer yukarıdaki kodları şöyle yazsaydık:

```
def sırala(kelime):
    return (çevrim.get(kelime[0]))
```

Bu durumda Python sıralamada kelimelerin yalnızca ilk harflerini dikkate alacaktı. İlk harfi aynı olan kelimeleri ise bu yüzden düzgün sıralayamayacaktı. Elbette Python'ın önce ilk harfe, sonra ikinci harfe, sonra da üçüncü harfe bakmasını sağlayabiliriz:

```
def sırala(kelime):
    return (çevrim.get(kelime[0]), çevrim.get(kelime[1]), çevrim.get(kelime[2]))
```

Ancak bu yöntemin uygulanabilir ve pratik olmadığı ortada. Kendi kendinize bazı denemeler yaparak bunu kendiniz de rahatlıkla görebilirsiniz.

Python'ın, sıralama yaparken kelimelerin önce ilk harflerini, sonra ikinci, sonra üçüncü, vb. harflerini karşılaştırmasını sağlamanın en uygun yolu şu olacaktır:

```
def sırala(kelime):
    return ([çevrim.get(kelime[i]) for i in range(len(kelime))])
```

Gördüğünüz gibi, burada kelimelerdeki harflerin sırasını tek tek elle yazmak yerine, bunu bir `for` döngüsü içinde otomatik olarak yaptırıyoruz. Dolayısıyla `sırala()` fonksiyonuna verilen parametrenin mesela *ahmet* olduğu bir durumda yukarıdaki fonksiyon şu demeti döndürüyor:

```
def sırala('ahmet'):
    return (çevrim.get('ahmet'[0]),
            çevrim.get('ahmet'[1]),
            çevrim.get('ahmet'[2]),
            çevrim.get('ahmet'[3]),
            çevrim.get('ahmet'[4]))
```

Mesela 'can' için ise şunu:

```
def sırala('can'):
    return (çevrim.get('can'[0]),
            çevrim.get('can'[1]),
            çevrim.get('can'[2]))
```

Böylece Python, hangi uzunlukta bir isimle karşılaşırsa karşılaşsın, sıralama işlemini düzgün bir şekilde gerçekleştirebiliyor.

Bu bölümde Python'da sıralama konusunu epey ayrıntılı bir şekilde ele aldık.

Not

'Sıralama' konusuna ilişkin bir tartışma için http://www.istihza.com/forum/viewtopic.php?f=25&t=1523 adresindeki konuyu inceleyebilirsiniz.

slice()

Bildiğiniz gibi, birtakım öğelerden oluşan bir nesnenin yalnızca belli kısımlarını ayırıp alma işlemine 'dilimleme' adı veriliyor. Örneğin elimizde şöyle bir liste olduğunu düşünelim:

```
>>> l = ['ahmet', 'mehmet', 'ayşe', 'senem', 'salih']
```

5 öğeli bu listenin yalnızca ilk iki öğesini almak, yani dilimlemek için şu yapıyı kullanıyoruz:

```
>>> l[0:2]

['ahmet', 'mehmet']
```

Dilimleme işleminin şöyle bir formülden oluştuğunu biliyoruz:

```
l[başlangıç:bitiş:atlama_değeri]
```

Başlangıç parametresinin öntanımlı değeri *0* olduğu için yukarıdaki kodu şöyle de yazabilirdik:

```
>>> l[:2]

['ahmet', 'mehmet']
```

Aynı listenin, ilk öğeden itibaren sonuna kadar olan bütün öğelerini almak için ise şunu yazıyoruz:

```
>>> l[1:]
```

Eğer bu listeyi, öğelerini ikişer ikişer atlayarak dilimlemek istersek de şu yolu takip ediyoruz:

```
>>> l[::2]

['ahmet', 'ayşe', 'salih']
```

Bu örnekte başlangıç ve bitiş parametrelerinin öntanımlı değerlerini kullandık. O yüzden o kısımları

boş bıraktık. Öğeleri ikişer ikişer atlayabilmek için ise atlama_değeri olarak *2* sayısını kullandık.

İşte yukarıdakine benzer dilimleme işlemleri için slice() adlı bir gömülü fonksiyondan da yararlanabiliriz. Dikkatlice bakın:

Listemiz şu:

```
>>> l = ['ahmet', 'mehmet', 'ayşe', 'senem', 'salih']
```

Bir 'dilimleme' (*slice*) nesnesi oluşturuyoruz:

```
>>> dl = slice(0, 3)
```

Bu nesneyi liste üzerine uyguluyoruz:

```
>>> l[dl]

['ahmet', 'mehmet', 'ayşe']
```

Gördüğünüz gibi, slice() fonksiyonunu yukarıda iki parametre ile kullandık. Tahmin edebileceğiniz gibi, bu fonksiyonunu formülü şu şekildedir:

```
slice(başlangıç, bitiş, atlama)
```

sum()

Bu fonksiyonun temel görevi, bir dizi içindeki değerlerin toplamını bulmaktır. Örneğin:

```
>>> l = [1, 2, 3]
>>> sum(l)

6
```

Bu fonksiyon genellikle yukarıdaki gibi tek parametreyle kullanılır. Ama aslında bu fonksiyon ikinci bir parametre daha alır. Dikkatlice bakın:

```
>>> l = [1, 2, 3]
>>> sum(l, 10)

16
```

Gördüğünüz gibi, Python sum() fonksiyonuna verilen ikinci parametreyi, birinci parametredeki toplam değerin üzerine ekliyor.

type()

type() fonksiyonunun görevi bir nesnenin hangi veri tipine ait olduğunu söylemektir. Bu fonksiyonu artık yakından tanıyorsunuz:

```
>>> type('elma')

<class 'str'>
```

zip()

Gelin isterseniz bu fonksiyonu bir örnek üzerinden açıklamaya çalışalım.

Diyelim ki elimizde şöyle iki farklı liste var:

```
>>> a1 = ['a', 'b', 'c']
>>> a2 = ['d', 'e', 'f']
```

Eğer bu listelerin öğelerini birbirleriyle eşleştirmek istersek zip() fonksiyonundan yararlanabiliriz.

Dikkatlice bakın:

```
>>> zip(a1, a2)

<zip object at 0x00FD0BE8>
```

Gördüğünüz gibi, yukarıdaki kod bize bir 'zip' nesnesi veriyor. Bu nesnenin öğelerine nasıl ulaşabileceğinizi biliyorsunuz:

```
>>> print(*zip(a1, a2))

('a', 'd') ('b', 'e') ('c', 'f')

>>> list(zip(a1, a2))

[('a', 'd'), ('b', 'e'), ('c', 'f')]

>>> for a, b in zip(a1, a2):
...     print(a, b)
...
a d
b e
c f
```

Yukarıdaki çıktıları incelediğimizde, ilk listenin ilk öğesinin, ikinci listenin ilk öğesiyle; ilk listenin ikinci öğesinin, ikinci listenin ikinci öğesiyle; ilk listenin üçüncü öğesinin ise, ikinci listenin üçüncü

öğesiyle eşleştiğini görüyoruz.

Bu özellikten pek çok farklı şekilde yararlanabilirsiniz. Örneğin:

```
>>> isimler = ['ahmet', 'mehmet', 'zeynep', 'ilker']
>>> yaşlar = [25, 40, 35, 20]
>>> for i, y in zip(isimler, yaşlar):
...     print('isim: {} / yaş: {}'.format(i, y))
...
isim: ahmet / yaş: 25
isim: mehmet / yaş: 40
isim: zeynep / yaş: 35
isim: ilker / yaş: 20
```

Burada *isimler* ve *yaşlar* adlı listelerin öğelerini zip() fonksiyonu yardımıyla birbirleriyle eşleştirdik.

vars()

Bu fonksiyon, mevcut isim alanı içindeki metot, fonksiyon ve nitelikleri listeler. Eğer bu fonksiyonu parametresiz olarak kullanırsak, daha önce gördüğümüz locals() fonksiyonuyla aynı çıktıyı elde ederiz:

```
>>> vars()

{'__builtins__': <module 'builtins' (built-in)>, '__name__': '__main__',
 '__package__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
 '__doc__': None}
```

Bu fonksiyonu, nesnelerin metotlarını ve niteliklerini öğrenmek için de kullanabilirsiniz:

```
>>> vars(str)
>>> vars(list)
>>> vars(dict)
```

Yukarıda sırasıyla karakter dizilerinin, listelerin ve sözlüklerin metotlarını listeledik. Bu yönüyle vars() fonksiyonu dir() fonksiyonuna benzer.

Böylece Python'daki gömülü fonksiyonları tek tek incelemiş olduk. Bu bölümde incelemediğimiz gömülü fonksiyonlar şunlar:

1. memoryview()
2. iter()
3. next()

4. object()
5. property()
6. staticmethod()
7. super()
8. getattr()
9. hasattr()
10. delattr()
11. classmethod()
12. issubclass()
13. setattr()

Bu fonksiyonları, ilerleyen derslerle birlikte Python bilgimiz biraz daha arttığında ele alacağız.

başa dön

İleri Düzey Fonksiyonlar

Buraya gelinceye kadar fonksiyonlara ilişkin epey söz söyledik. Artık Python programlama dilinde fonksiyonlara dair hemen her şeyi bildiğimizi rahatlıkla söyleyebiliriz. Zira bu noktaya kadar hem fonksiyonların temel (ve orta düzey) özelliklerini öğrendik, hem de 'gömülü fonksiyon' kavramını ve gömülü fonksiyonların kendisini bütün ayrıntılarıyla inceledik. Dolayısıyla yazdığımız kodlarda fonksiyonları oldukça verimli bir şekilde kullanabilecek kadar fonksiyon bilgisine sahibiz artık.

Dediğimiz gibi,fonksiyonlara ilişkin en temel bilgileri edindik. Ancak fonksiyonlara dair henüz bilmediğimiz şeyler de var. Ama artık Python programlama dilinde geldiğimiz aşamayı dikkate alarak ileriye doğru bir adım daha atabilir, fonksiyonlara dair ileri düzey sayılabilecek konulardan da söz edebiliriz.

Bu bölümde sırasıyla şu konuları inceleyeceğiz:

1. Lambda Fonksiyonları (*Lambda Functions*)
2. Özyinelemeli Fonksiyonlar (*Recursive Functions*)
3. Bezeyiciler (*Decorators*)
4. Kapalı Fonksiyonlar (*Closures*)

Yukarıdaki listeden de görebileceğiniz gibi, burada inceleyeceğimiz ilk konu 'lambda fonksiyonları' olacak. O halde hiç vakit kaybetmeden yola koyulalım.

Lambda Fonksiyonları

Şimdiye kadar Python programlama dilinde fonksiyon tanımlamak için hep *def* adlı bir ifadeden yararlanmıştık. Bu bölümde ise Python programlama dilinde fonksiyon tanımlamamızı sağlayacak, tıpkı *def* gibi bir ifadeden daha söz edeceğiz. Fonksiyon tanımlamamızı sağlayan bu yeni ifaye *lambda* denir. Bu ifade ile oluşturulan fonksiyonlara ise 'lambda fonksiyonları'...

Bildiğiniz gibi Python'da bir fonksiyonu *def* ifadesi yardımıyla şöyle tanımlıyoruz:

```
>>> def fonk(param1, param2):
...     return param1 + param2
```

Bu fonksiyon, kendisine verilen parametreleri birbiriyle toplayıp bize bunların toplamını döndürüyor:

```
>>> fonk(2, 4)

6
```

Peki aynı işlemi lambda fonksiyonları yardımıyla yapmak istersek nasıl bir yol izleyeceğiz?

Dikkatlice bakın:

```
>>> fonk = lambda param1, param2: param1 + param2
```

İşte burada tanımladığımız şey bir lambda fonksiyonudur. Bu lambda fonksiyonunu da tıpkı biraz önce tanımladığımız def fonksiyonu gibi kullanabiliriz:

```
>>> fonk(2, 4)

6
```

Gördüğünüz gibi lambda fonksiyonlarını tanımlamak ve kullanmak hiç de zor değil.

Lambda fonksiyonlarının neye benzediğinden temel olarak bahsettiğimize göre artık biraz daha derine inebiliriz.

Lambda fonksiyonları Python programlama dilinin ileri düzey fonksiyonlarından biridir. Yukarıdaki örnek yardımıyla bu lambda fonksiyonlarının nasıl bir şey olduğunu gördük. Esasında biz buraya gelene kadar bu lambda fonksiyonlarını hiç görmemiş de değiliz. Hatırlarsanız daha önceki derslerimizde şöyle bir örnek kod yazmıştık:

```
harfler = "abcçdefgğhıijklmnoöprsştuüvyz"
çevrim = {i: harfler.index(i) for i in harfler}

isimler = ["ahmet", "ışık", "ismail", "çiğdem", "can", "şule", "iskender"]

print(sorted(isimler, key=lambda x: çevrim.get(x[0])))
```

Burada sorted() fonksiyonunun *key* parametresi içinde kullandığımız ifade bir lambda fonksiyonudur:

```
lambda x: çevrim.get(x[0])
```

Peki lambda fonksiyonları nedir ve ne işe yarar?

Lambda fonksiyonlarını, bir fonksiyonun işlevselliğine ihtiyaç duyduğumuz, ama konum olarak bir fonksiyon tanımlayamayacağımız veya fonksiyon tanımlamanın zor ya da meşakkatli olduğu durumlarda kullanabiliriz. Yukarıdaki örnek kod, bu tanıma iyi bir örnektir: sorted() fonksiyonunun *key* parametresi bizden bir fonksiyon tanımı bekler. Ancak biz elbette oraya *def* ifadesini kullanarak doğrudan bir fonksiyon tanımlayamayız. Ama *def* yerine *lambda* ifadesi yardımıyla *key* parametresi için bir lambda fonksiyonu tanımlayabiliriz.

Eğer yukarıdaki kodları 'normal' bir fonksiyonla yazmak isteseydik şu kodları kullanabilirdik:

```
harfler = "abcçdefgğhıijklmnoöprsştuüvyz"
çevrim = {i: harfler.index(i) for i in harfler}

isimler = ["ahmet", "ışık", "ismail", "çiğdem", "can", "şule", "iskender"]
```

```
def sırala(eleman):
    return çevrim.get(eleman[0])

print(sorted(isimler, key=sırala))
```

Burada lambda fonksiyonu kullanmak yerine, sırala() adlı bir fonksiyon kullandık.

Eğer yukarıda 'lambda' ile yazdığımız örneği sırala() fonksiyonu ile yazdığımız örnekle kıyaslarsanız lambda fonksiyonlarında hangi parçanın neye karşılık geldiğini veya ne anlama sahip olduğunu rahatlıkla anlayabilirsiniz.

Gelin bir örnek daha verelim:

Diyelim ki bir sayının çift sayı olup olmadığını denetleyen bir fonksiyon yazmak istiyorsunuz. Bunun için şöyle bir fonksiyon tanımlayabileceğimizi biliyorsunuz:

```
def çift_mi(sayı):
    return sayı % 2 == 0
```

Eğer çift_mi() fonksiyonuna parametre olarak verilen bir sayı çift ise fonksiyonumuz *True* çıktısı verecektir:

```
print(çift_mi(100))
```

```
True
```

Aksi halde *False* çıktısı alırız:

```
print(çift_mi(99))
```

```
False
```

İşte yukarıdaki etkiyi lambda fonksiyonları yardımıyla da elde edebiliriz.

Dikkatlice bakın:

```
>>> çift_mi = lambda sayı: sayı % 2 == 0
>>> çift_mi(100)

True

>>> çift_mi(99)

False
```

Başka bir örnek daha verelim. Diyelim ki bir liste içindeki bütün sayıların karesini hesaplamak istiyoruz. Elimizdeki liste şu:

```
>>> l = [2, 5, 10, 23, 3, 6]
```

Bu listedeki sayıların her birinin karesini hesaplamak için şöyle bir şey yazabiliriz:

```
>>> for i in l:
...     print(i**2)

4
25
100
529
9
36
```

Veya şöyle bir şey:

```
>>> [i**2 for i in l]

[4, 25, 100, 529, 9, 36]
```

Ya da map() fonksiyonuyla birlikte lambda'yı kullanarak şu kodu yazabiliriz:

```
>>> print(*map(lambda sayı: sayı ** 2, l))

4 25 100 529 9 36
```

Son örnekte verdiğimiz lambda'lı kodu normal bir fonksiyon tanımlayarak şöyle de yazabilirdik:

```
>>> def karesi(sayı):
...     return sayı ** 2
...
>>> print(*map(karesi, l))

4 25 100 529 9 36
```

Sözün özü, mesela şu kod:

```
lambda x: x + 10
```

Türkçede şu anlama gelir:

```
'x' adlı bir parametre alan bir lambda fonksiyonu tanımla. Bu fonksiyon, bu
```

```
'x parametresine 10 sayısını eklesin.
```

Biz yukarıdaki örneklerde lambda fonksiyonunu tek bir parametre ile tanımladık. Ama elbette lambda fonksiyonlarının birden fazla parametre de alabileceğini de biliyorsunuz.

Örneğin:

```
>>>  birleştir = lambda ifade, birleştirici: birleştirici.join(ifade.split())
```

Burada lambda fonksiyonumuz toplam iki farklı parametre alıyor: Bunlardan ilki *ifade*, ikincisi ise *birleştirici*. Fonksiyonumuzun gövdesinde *ifade* parametresine split() metodunu uyguladıktan sonra, elde ettiğimiz parçaları *birleştirici* parametresinin değerini kullanarak birbirleriyle birleştiriyoruz. Yani:

```
>>> birleştir('istanbul büyükşehir belediyesi', '-')

'istanbul-büyükşehir-belediyesi'
```

Eğer aynı işlevi 'normal' bir fonksiyon yardımıyla elde etmek isteseydik şöyle bir şey yazabilirdik:

```
>>> def birleştir(ifade, birleştirici):
...     return birleştirici.join(ifade.split())
...
>>> birleştir('istanbul büyükşehir belediyesi', '-')

'istanbul-büyükşehir-belediyesi'
```

Yukarıdaki örneklerin dışında, lambda fonksiyonları özellikle grafik arayüz çalışmaları yaparken işinize yarayabilir. Örneğin:

```
import tkinter
import tkinter.ttk as ttk

pen = tkinter.Tk()

btn = ttk.Button(text='merhaba', command=lambda: print('merhaba'))
btn.pack(padx=20, pady=20)

pen.mainloop()
```

Not

Bu kodlardan hiçbir şey anlamamış olabilirsiniz. Endişe etmeyin. Burada amacımız size sadece lambda fonksiyonlarının kullanımını göstermek. Bu kodlarda yalnızca lambda fonksiyonuna odaklanmanız şimdilik yeterli olacaktır. Eğer bu kodları çalıştıramadıysanız

http://www.istihza.com/forum adresinde sorununuzu dile getirebilirsiniz.

Bu kodları çalıştırıp 'deneme' düğmesine bastığınızda komut satırında 'merhaba' çıktısı görünecektir. Tkinter'de fonksiyonların *command* parametresi bizden parametresiz bir fonksiyon girmemizi bekler. Ancak bazen, elde etmek istediğimiz işlevsellik için oraya parametreli bir fonksiyon yazmak durumunda kalabiliriz. İşte bunun gibi durumlarda lambda fonksiyonları faydalı olabilir. Elbette yukarıdaki kodları şöyle de yazabilirdik:

```
import tkinter
import tkinter.ttk as ttk

pen = tkinter.Tk()

def merhaba():
    print('merhaba')

btn = ttk.Button(text='merhaba', command=merhaba)
btn.pack(padx=20, pady=20)

pen.mainloop()
```

Burada da lambda yerine isimli bir fonksiyon tanımlayıp, *command* parametresine doğrudan bu fonksiyonu verdik.

Bütün bu örneklerden gördüğünüz gibi, lambda fonksiyonları son derece pratik araçlardır. Normal, isimli fonksiyonlarla elde ettiğimiz işlevselliği, lambda fonksiyonları yardımıyla çok daha kısa bir şekilde elde edebiliriz. Ancak lambda fonksiyonları normal fonksiyonlara göre biraz daha okunaksız yapılardır. O yüzden, eğer lambda fonksiyonlarını kullanmaya mecbur değilseniz, bunların yerine normal fonksiyonları veya yerine göre liste üreteçlerini tercih edebilirsiniz.

Özyinelemeli (*Recursive*) Fonksiyonlar

Bu bölümde, lambda fonksiyonlarının ardından, yine Python'ın ileri düzey konularından biri olan 'özyinelemeli fonksiyonlar'dan söz edeceğiz. İngilizcede *recursive functions* olarak adlandırılan özyinelemeli fonksiyonların, Python programlama dilinin anlaması en zor konularından biri olduğu söylenir. Ama bu söylenti sizi hiç endişelendirmesin. Zira biz burada bu çapraşık görünen konuyu size olabildiğince basit ve anlaşılır bir şekilde sunmak için elimizden gelen bütün çabayı göstereceğiz.

O halde hemen başlayalım...

Şimdiye kadar Python'da pek çok fonksiyon gördük. Bu fonksiyonlar kimi zaman Python programcılarınca tanımlanıp dile entegre edilmiş 'gömülü fonksiyonlar' (*builtin functions*) olarak, kimi zamansa o anda içinde bulunduğumuz duruma ve ihtiyaçlarımıza göre bizzat kendimizin

tanımladığı ‘el yapımı fonksiyonlar’ (*custom functions*) olarak çıktı karşımıza.

Şimdiye kadar öğrendiğimiz bütün bu fonksiyonların ortak bir noktası vardı. Bu ortak nokta, şu ana kadar fonksiyonları kullanarak yaptığımız örneklerden de gördüğünüz gibi, bu fonksiyonlar yardımıyla başka fonksiyonları çağırabiliyor olmamız. Örneğin:

```
def selamla(kim):
    print('merhaba', kim)
```

Burada selamla() adlı bir fonksiyon tanımladık. Gördüğünüz gibi bu fonksiyon print() adlı başka bir fonksiyonu çağırıyor. Burada sıradışı bir şey yok. Dediğimiz gibi, şimdiye kadar zaten hep böyle fonksiyonlar görmüştük.

Python fonksiyonları, yukarıdaki örnekte de gördüğünüz gibi, nasıl başka fonksiyonları çağırabiliyorsa, aynı şekilde, istenirse, kendi kendilerini de çağırabilirler. İşte bu tür fonksiyonlara Python programlama dilinde ‘kendi kendilerini yineleyen’, veya daha teknik bir dille ifade etmek gerekirse ‘özyinelemeli’ (*recursive*) fonksiyonlar adı verilir.

Çok basit bir örnek verelim. Diyelim ki, kendisine parametre olarak verilen bir karakter dizisi içindeki karakterleri teker teker azaltarak ekrana basan bir fonksiyon yazmak istiyorsunuz. Yani mesela elinizde ‘istihza’ adlı bir karakter dizisi var. Sizin amacınız bu karakter dizisini şu şekilde basan bir fonksiyon yazmak:

```
istihza
stihza
tihza
ihza
hza
za
a
```

Elbette bu işi yapacak bir fonksiyonu, daha önce öğrendiğiniz döngüler ve başka yapılar yardımıyla rahatlıkla yazabilirsiniz. Ama isterseniz aynı işi özyinelemeli fonksiyonlar yardımıyla da yapabilirsiniz.

Şimdi şu kodlara dikkatlice bakın:

```
def azalt(s):
    if len(s) < 1:
        return s
    else:
        print(s)
        return azalt(s[1:])

print(azalt('istihza'))
```

Bu kodlar bize yukarıda bahsettiğimiz çıktıyı verecek:

```
istihza
stihza
tihza
ihza
hza
za
a
```

Fonksiyonumuzu yazıp çalıştırdığımıza ve bu fonksiyonun bize nasıl bir çıktı verdiğini gördüğümüze göre fonksiyonu açıklamaya geçebiliriz.

Bu fonksiyon ilk bakışta daha önce öğrendiğimiz fonksiyonlardan çok da farklı görünmüyor aslında. Ama eğer fonksiyonun son kısmına bakacak olursanız, bu fonksiyonu daha önce öğrendiğimiz fonksiyonlardan ayıran şu satırı görürsünüz:

```
return azalt(s[1:])
```

Gördüğünüz gibi, burada azalt() fonksiyonu içinde yine azalt() fonksiyonunu çağırıyoruz. Böylece fonksiyonumuz sürekli olarak kendi kendini yineliyor. Yani aynı fonksiyonu tekrar tekrar uyguluyor.

Peki ama bunu nasıl yapıyor?

Nasıl bir durumla karşı karşıya olduğumuzu daha iyi anlamak için yukarıdaki kodları şu şekilde yazalım:

```
def azalt(s):
    if len(s) < 1:
        return s
    else:
        print(list(s))
        return azalt(s[1:])
```

Burada fonksiyonun her yinelenişinde, özyinelemeli fonksiyona parametre olarak giden karakter dizisinin nasıl değiştiğini birazcık daha net olarak görebilmek için karakter dizisi içindeki karakterleri bir liste haline getirip ekrana basıyoruz:

```
print(list(s))
```

Bu kodları çalıştırdığımızda şu çıktıyı alacağız:

```
['i', 's', 't', 'i', 'h', 'z', 'a']
['s', 't', 'i', 'h', 'z', 'a']
['t', 'i', 'h', 'z', 'a']
['i', 'h', 'z', 'a']
```

```
['h', 'z', 'a']
['z', 'a']
['a']
```

Yukarıdaki çıktının ilk satırında gördüğünüz gibi, fonksiyon ilk çağrıldığında listede ‘istihza’ karakter dizisini oluşturan bütün harfler var. Yani fonksiyonumuz ilk çalışmada parametre olarak karakter dizisinin tamamını alıyor. Ancak fonksiyonun her yinelenişinde listedeki harfler birer birer düşüyor. Böylece özyinelemeli fonksiyonumuz parametre olarak karakter dizisinin her defasında bir eksiltilmiş biçimini alıyor.

Yukarıdaki sözünü ettiğimiz düşmenin yönü karakter dizisinin başından sonuna doğru. Yani her defasında, elde kalan karakter dizisinin ilk harfi düşüyor. Düşme yönünün böyle olması bizim kodları yazış şeklimizden kaynaklanıyor. Eğer bu kodları şöyle yazsaydık:

```
def azalt(s):
    if len(s) < 1:
        return s
    else:
        print(list(s))
        return azalt(s[:-1])
```

Harflerin düşme yönü sondan başa doğru olacaktı:

```
['i', 's', 't', 'i', 'h', 'z', 'a']
['i', 's', 't', 'i', 'h', 'z']
['i', 's', 't', 'i', 'h']
['i', 's', 't', 'i']
['i', 's', 't']
['i', 's']
['i']
```

Burada, bir önceki koddaki azalt(s[1:]) satırını azalt(s[:-1]) şeklinde değiştirdiğimize dikkat edin.

Fonksiyonun nasıl işlediğini daha iyi anlamak için, ‘istihza’ karakter dizisinin son harfinin her yineleniş esnasındaki konumunun nasıl değiştiğini de izleyebilirsiniz:

```
n = 0

def azalt(s):
    global n
    if len(s) < 1:
        return s
    else:
        n += 1
        print('{} harfinin {}. çalışmadaki konumu: {}'.format('a', n, s.index('a')))
```

```
        return azalt(s[1:])
```

Bu kodlar şu çıktıyı verir:

```
a harfinin 1. çalışmadaki konumu: 6
a harfinin 2. çalışmadaki konumu: 5
a harfinin 3. çalışmadaki konumu: 4
a harfinin 4. çalışmadaki konumu: 3
a harfinin 5. çalışmadaki konumu: 2
a harfinin 6. çalışmadaki konumu: 1
a harfinin 7. çalışmadaki konumu: 0
```

Gördüğünüz gibi ‘istihza’ kelimesinin en sonunda bulunan ‘a’ harfi her defasında baş tarafa doğru ilerliyor.

Aynı şekilde, kodları daha iyi anlayabilmek için, fonksiyona parametre olarak verdiğimiz ‘istihza’ kelimesinin her yinelemede ne kadar uzunluğa sahip olduğunu da takip edebilirsiniz:

```
def azalt(s):
    if len(s) < 1:
        return s
    else:
        print(len(s))
        return azalt(s[:-1])
```

Bu fonksiyonu ‘istihza’ karakter dizisine uyguladığımızda bize şu çıktıyı veriyor:

```
7
6
5
4
3
2
1
```

Gördüğünüz gibi, fonksiyonun kendini her yineleyişinde karakter dizimiz küçülüyor.

Bu durum bize özyinelemeli fonksiyonlar hakkında çok önemli bir bilgi veriyor esasında:

Özyinelemeli fonksiyonlar; büyük bir problemin çözülebilmesi için, o problemin, problemin bütününü temsil eden daha küçük bir parçası üzerinde işlem yapabilmemizi sağlayan fonksiyonlardır.

Yukarıdaki örnekte de bu ilkeyi uyguluyoruz. Yani biz ‘istihza’ karakter dizisinin öncelikle yalnızca ilk karakterini düşürüyoruz:

```
s[1:]
```

Daha sonra da bu yöntemi özyinelemeli bir şekilde uyguladığımızda, ‘istihza’ karakter dizisinin her defasında daha küçük bir parçası bu yöntemden etkileniyor:

```
azalt(s[1:])
```

Yani fonksiyonumuz ilk olarak ‘istihza’ karakter dizisinin ilk harfi olan ‘i’ harfini düşürüyor. Sonra ‘stihza’ kelimesinin ilk harfi olan ‘s’ harfini düşürüyor. Ardından ‘tihza’ kelimesinin ilk harfi olan ‘t’ harfini düşürüyor ve kelime tükenene kadar bu işlemi devam ettiriyor.

Peki ama bunu nasıl yapıyor?

Şimdi yukarıdaki fonksiyondaki şu kısma dikkatlice bakın:

```
if len(s) < 1:
    return s
```

İşte burada özyinelemeli fonksiyonumuzun, karakter dizisi üzerinde ne kadar derine inmesi gerektiğini belirliyoruz. Buna göre, karakter dizisinin uzunluğu 1’in altına düştüğünde eldeki karakter dizisini döndürüyoruz. Yani karakter dizisinin uzunluğu 1’in altına düştüğünde elde kalan karakter dizisi boş bir karakter dizisi olduğu için o boş karakter dizisini döndürüyoruz. Eğer istersek elbette bu durumda başka bir şey de döndürebiliriz:

```
def azalt(s):
    if len(s) < 1:
        return 'bitti!'
    else:
        print(s)
        return azalt(s[1:])
```

İşte `if len(s) < 1:` bloğunun bulunduğu bu kodlara ‘dip nokta’ adı veriyoruz. Fonksiyonumuzun yinelene yinelene (veya başka bir ifadeyle ‘dibe ine ine’) geleceği en son nokta burasıdır. Eğer bu dip noktayı belirtmezsek fonksiyonumuz, tıpkı dipsiz bir kuyuya düşmüş gibi, sürekli daha derine inmeye çalışacak, sonunda da hata verecektir. Ne demek istediğimizi daha iyi anlamak için kodlarımızı şöyle yazalım:

```
def azalt(s):
    print(s)
    return azalt(s[1:])
```

Gördüğünüz gibi burada herhangi bir dip nokta belirtmedik. Bu kodları çalıştırdığımızda Python bize şöyle bir hata mesajı verecek:

```
RuntimeError: maximum recursion depth exceeded
```

Yani:

```
ÇalışmaZamanıHatası: Azami özyineleme derinliği aşıldı
```

Dediğimiz gibi, özyinelemeli fonksiyonlar her yinelenişte sorunun (yani üzerinde işlem yapılan parametrenin) biraz daha derinine iner. Ancak bu derine inmenin de bir sınırı vardır. Bu sınırın ne olduğunu şu kodlar yardımıyla öğrenebilirsiniz:

```
>>> import sys
>>> sys.getrecursionlimit()
```

İşte biz özyinelemeli fonksiyonlarımızda dip noktayı mutlaka belirterek, Python'ın fonksiyonu yinelerken ne kadar derine inip nerede duracağını belirlemiş oluyoruz.

Şimdi son kez, yukarıdaki örnek fonksiyonu, özyineleme mantığını çok daha iyi anlamanızı sağlayacak bir şekilde yeniden yazacağız. Dikkatlice bakın:

```
def azalt(s):
    if len(s) < 1:
        return s
    else:
        print('özyineleme sürecine girerken:', s)
        azalt(s[1:])
        print('özyineleme sürecinden çıkarken:', s)

azalt('istihza')
```

Burada, fonksiyon kendini yinelemeye başlamadan hemen önce bir print() satırı yerleştirerek *s* değişkeninin durumunu takip ediyoruz:

```
print('özyineleme sürecine girerken:', s)
```

Aynı işlemi bir de fonksiyonun kendini yinelemeye başlamasının hemen ardından yapıyoruz:

```
print('özyineleme sürecinden çıkarken:', s)
```

Yukarıdaki kodlar bize şu çıktıyı verecek:

```
özyineleme sürecine girerken: istihza
özyineleme sürecine girerken: stihza
özyineleme sürecine girerken: tihza
özyineleme sürecine girerken: ihza
özyineleme sürecine girerken: hza
özyineleme sürecine girerken: za
özyineleme sürecine girerken: a
özyineleme sürecinden çıkarken: a
özyineleme sürecinden çıkarken: za
```

```
özyineleme sürecinden çıkarken: hza
özyineleme sürecinden çıkarken: ihza
özyineleme sürecinden çıkarken: tihza
özyineleme sürecinden çıkarken: stihza
özyineleme sürecinden çıkarken: istihza
```

Gördüğünüz gibi fonksiyon özyineleme sürecine girerken düşürdüğü her bir karakteri, özyineleme sürecinden çıkarken yeniden döndürüyor. Bu, özyinelemeli fonksiyonların önemli bir özelliğidir. Mesela bu özellikten yararlanarak şöyle bir kod yazabilirsiniz:

```
def ters_çevir(s):
    if len(s) < 1:
        return s
    else:
        ters_çevir(s[1:])
        print(s[0])

ters_çevir('istihza')
```

Yazdığımız bu kodda ters_çevir() fonksiyonu, kendisine verilen parametreyi ters çevirecektir. Yani yukarıdaki kod bize şu çıktıyı verir:

```
a
z
h
i
t
s
i
```

Burada yaptığımız şey çok basit: Yukarıda da söylediğimiz gibi, özyinelemeli fonksiyonlar, özyineleme sürecine girerken yaptığı işi, özyineleme sürecinden çıkarken tersine çevirir. İşte biz de bu özellikten yararlandık. Fonksiyonun kendini yinelediği noktanın çıkışına bir print() fonksiyonu yerleştirip, geri dönen karakterlerin ilk harfini ekrana bastık. Böylece *s* adlı parametrenin tersini elde etmiş olduk.

Ancak eğer yukarıdaki kodları bu şekilde yazarsak, fonksiyondan dönen değeri her yerde kullanamayız. Mesela yukarıdaki fonksiyonu aşağıdaki gibi kullanamayız:

```
def ters_çevir(s):
    if len(s) < 1:
        return s
    else:
        ters_çevir(s[1:])
```

```
        print(s[0])

kelime = input('kelime girin: ')
print('Girdiğiniz kelimenin tersi: {}'.format(ters_çevir('istihza')))
```

Fonksiyonumuzun daha kullanışlı olabilmesi için kodlarımızı şöyle yazabiliriz:

```
def ters_çevir(s):
    if len(s) < 1:
        return s
    else:
        return ters_çevir(s[1:]) + s[0]

kelime = input('kelime girin: ')
print('Girdiğiniz kelimenin tersi: {}'.format(ters_çevir('istihza')))
```

Burada bizim amacımızı gerçekleştirmemizi sağlayan satır şu:

```
return ters_çevir(s[1:]) + s[0]
```

İlk bakışta bu satırın nasıl çalıştığını anlamak zor gelebilir. Ama aslında son derece basit bir mantığı var bu kodların. Şöyle düşünün: ters_çevir() fonksiyonunu özyinelemeli olarak işlettiğimizde, yani şu kodu yazdığımızda:

```
return ters_çevir(s[1:])
```

...döndürülecek son değer boş bir karakter dizisidir. İşte biz özyinelemeden çıkılırken geri dönen karakterlerin ilk harflerini bu boş karakter dizisine ekliyoruz ve böylece girdiğimiz karakter dizisinin ters halini elde etmiş oluyoruz.

Yukarıdaki işlevin aynısını, özyinelemeli fonksiyonunuzu şöyle yazarak da elde edebilirdiniz:

```
def ters_çevir(s):
    if not s:
        return s
    else:
        return s[-1] + ters_çevir(s[:-1])

print(ters_çevir('istihza'))
```

Burada aynı iş için farklı bir yaklaşım benimsedik. İlk olarak, dip noktasını şu şekilde belirledik:

```
if not s:
    return s
```

Bildiğiniz gibi, boş veri tiplerinin bool değeri False'tur. Dolayısıyla özyineleme sırasında *s* parametresinin uzunluğunun 1'in altına düşmesi, *s* parametresinin içinin boşaldığını gösterir. Yani o anda *s* parametresinin bool değeri False olur. Biz de yukarıda bu durumdan faydalandık.

Bir önceki kodlara göre bir başka farklılık da şu satırda:

```
return s[-1] + ters_çevir(s[:-1])
```

Burada benimsediğimiz yaklaşımın özü şu: Bildiğiniz gibi bir karakter dizisini ters çevirmek istediğimizde öncelikle bu karakter dizisinin en son karakterini alıp en başa yerleştiririz. Yani mesela elimizdeki karakter dizisi 'istihza' ise, bu karakter dizisini ters çevirmenin ilk adımı bunun en son karakteri olan 'a' harfini alıp en başa koymaktır. Daha sonra da geri kalan harfleri tek tek tersten buna ekleriz:

```
düz:  istihza
ters: a + z + h + i + t + s + i
```

İşte yukarıdaki fonksiyonda da yaptığımız şey tam anlamıyla budur.

Önce karakter dizisinin son harfini en başa koyuyoruz:

```
return s[-1]
```

Ardından da buna geri kalan harfleri tek tek tersten ekliyoruz:

```
return s[-1] + ters_çevir(s[:-1])
```

Özyinelemeli fonksiyonlara ilişkin olarak yukarıda tek bir örnek üzerinde epey açıklama yaptık. Bu örnek ve açıklamalar, özyinelemeli fonksiyonların nasıl çalıştığı konusunda size epey fikir vermiş olmalı. Ancak elbette bu fonksiyonları tek bir örnek yardımıyla tamamen anlayamamış olabilirsiniz. O yüzden gelin isterseniz bir örnek daha verelim. Mesela bu kez de basit bir sayaç yapalım:

```
def sayaç(sayı, sınır):
    print(sayı)
    if sayı == sınır:
        return 'bitti!'
    else:
        return sayaç(sayı+1, sınır)
```

Not

Bu fonksiyonun yaptığı işi elbette başka şekillerde çok daha kolay bir şekilde halledebilirdik. Bu örneği burada vermemizin amacı yalnızca özyinelemeli fonksiyonların nasıl işlediğini göstermek. Yoksa böyle bir işi özyinelemeli fonksiyonlarla yapmanızı beklemiyoruz.

Yukarıdaki fonksiyona dikkatlice bakarsanız aslında yaptığı işi çok basit bir şekilde gerçekleştirdiğini göreceksiniz.

Burada öncelikle sayaç() adlı bir fonksiyon tanımladık. Bu fonksiyon toplam iki farklı parametre alıyor: *sayı* ve *sınır*.

Buna göre fonksiyonumuzu şöyle kullanıyoruz:

```
print(sayaç(0, 100))
```

Burada *sayı* parametresine verdiğimiz *0* değeri sayacımızın saymaya kaçtan başlayacağını gösteriyor. *sınır* parametresine verdiğimiz *100* değeri ise kaça kadar sayılacağını gösteriyor. Buna göre biz *0*'dan *100*'e kadar olan sayıları sayıyoruz...

Gelin şimdi biraz fonksiyonumuzu inceleyelim.

İlk olarak şu satırı görüyoruz fonksiyon gövdesinde:

```
print(sayı)
```

Bu satır, özyinelemeli fonksiyonun her yinelenişinde *sayı* parametresinin durumunu ekrana basacak.

Sonraki iki satırda ise şu kodları görüyoruz:

```
if sayı == sınır:
    return 'bitti!'
```

Bu bizim 'dip nokta' adını verdiğimiz şey. Fonksiyonumuz yalnızca bu noktaya kadar yineleyecek, bu noktanın ilerisine geçmeyecektir. Yani *sayı* parametresinin değeri *sınır* parametresinin değerine ulaştığında özyineleme işlemi de sona erecek. Eğer böyle bir dip nokta belirtmezsek fonksiyonumuz sonsuza kadar kendini yinelemeye çalışacak, daha önce sözünü ettiğimiz 'özyineleme limiti' nedeniyle de belli bir aşamadan sonra hata verip çökecektir.

Sonraki satırlarda ise şu kodları görüyoruz:

```
else:
    return sayaç(sayı+1, sınır)
```

Bu satırlar, bir önceki aşamada belirttiğimiz dip noktaya ulaşılana kadar fonksiyonumuzun hangi işlemleri yapacağını gösteriyor. Buna göre, fonksiyonun her yinelenişinde *sayı* parametresinin değerini 1 sayı artırıyoruz.

Fonksiyonumuzu sayaç(0, 100) gibi bir komutla çalıştırdığımızı düşünürsek, fonksiyonun ilk çalışmasında *0* olan sayı değeri sonraki yinelemede *1*, sonraki yinelemede *2*, sonraki yinelemede ise *3* olacak ve bu durum *sınır* değer olan *100*'e varılana kadar devam edecektir. *sayı* parametresinin değeri *100* olduğunda ise dip nokta olarak verdiğimiz ölçüt devreye girecek ve fonksiyonun kendi

kendisini yinelemesi işlemine son verilecektir.

Biz yukarıdaki örnekte yukarıya doğru sayan bir fonksiyon yazdık. Eğer yukarıdan aşağıya doğru sayan bir sayaç yapmak isterseniz yukarıdaki fonksiyonu şu şekle getirebilirsiniz:

```
def sayaç(sayı, sınır):
    print(sayı)
    if sayı == sınır:
        return 'bitti!'
    else:
        return sayaç(sayı-1, sınır)

print(sayaç(100, 0))
```

Burada, önceki fonksiyonda + olan işleci - işlecine çevirdik:

```
return sayaç(sayı-1, sınır)
```

Fonksiyonumuzu çağırırken de elbette *sayı* parametresinin değerini *100* olarak, *sınır* parametresinin değerini ise *0* olarak belirledik.

Bu arada, daha önce de bahsettiğimiz gibi, özyinelemeli fonksiyonlar, özyinelemeye başlarken döndürdükleri değeri, özyineleme işleminin sonunda tek tek geri döndürür. Bu özelliği göz önünde bulundurarak yukarıdaki fonksiyonu şu şekilde de yazabilirdiniz:

```
def sayaç(sayı, sınır):
    if sayı == sınır:
        return 'bitti!'
    else:
        sayaç(sayı+1, sınır)
        print(sayı)

print(sayaç(0, 10))
```

Dikkat ederseniz burada print(sayı) satırını özyineleme işlevinin çıkışına yerleştirdik. Böylece *0*'dan *10*'a kadar olan sayıları tersten elde ettik. Ancak tabii ki yukarıdaki anlamlı bir kod yazım tarzı değil. Çünkü fonksiyonumuzun yazım tarzıyla yaptığı iş birbiriyle çok ilgisiz. Sayıları yukarı doğru saymak üzere tasarlandığı belli olan bu kodlar, yalnızca bir print() fonksiyonunun özyineleme çıkışına yerleştirilmesi sayesinde yaptığı işi yapıyor...

Yukarıda verdiğimiz örnekler sayesinde artık özyinelemeli fonksiyonlar hakkında en azından fikir sahibi olduğumuzu söyleyebiliriz. Gelin isterseniz şimdi özyinelemeli fonksiyonlarla ilgili (biraz daha mantıklı) bir örnek vererek bu çetrefilli konuyu zihnimizde netleştirmeye çalışalım.

Bu defaki örneğimizde iç içe geçmiş listeleri tek katmanlı bir liste haline getireceğiz. Yani elimizde

şöyle bir liste olduğunu varsayarsak:

```
l = [1, 2, 3, [4, 5, 6], [7, 8, 9, [10, 11], 12], 13, 14]
```

Yazacağımız kodlar bu listeyi şu hale getirecek:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Bu amacı gerçekleştirebilmek için şöyle bir fonksiyon yazalım:

```
def düz_liste_yap(liste):
    if not isinstance(liste, list):
        return [liste]
    elif not liste:
        return []
    else:
        return düz_liste_yap(liste[0]) + düz_liste_yap(liste[1:])

l = [1, 2, 3, [4, 5, 6], [7, 8, 9, [10, 11], 12], 13, 14]

print(düz_liste_yap(l))
```

Bu fonksiyonu yukarıdaki iç içe geçmiş listeye uyguladığınızda istediğiniz sonucu aldığınızı göreceksiniz.

İlk bakışta yukarıdaki kodları anlamak biraz zor gelmiş olabilir. Ama endişe etmenize gerek yok. Zira biz bu kodları olabildiğince ayrıntılı bir şekilde açıklayacağız.

İlk olarak dip noktamızı tanımlıyoruz her zamanki gibi:

```
if not isinstance(liste, list):
    return [liste]
```

Fonksiyonumuzun temel çalışma prensibine göre liste içindeki bütün öğeleri tek tek alıp başka bir liste içinde toplayacağız. Eğer liste elemanları üzerinde ilerlerken karşımıza liste olmayan bir eleman çıkarsa bu elemanı [liste] koduyla bir listeye dönüştüreceğiz.

Önceki örneklerden farklı olarak, bu kez kodlarımızda iki farklı dip noktası kontrolü görüyoruz. İlkini yukarıda açıkladık. İkinci dip noktamız şu:

```
elif not liste:
    return []
```

Burada yaptığımız şey şu: Eğer özyineleme esnasında boş bir liste ile karşılaşırsak, tekrar boş bir liste döndürüyoruz. Peki ama neden?

Bildiğiniz gibi boş bir listenin 0. elemanı olmaz. Yani boş bir liste üzerinde şu işlemi yapamayız:

```
>>> a = []
>>> a[0]

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Gördüğünüz gibi, boş bir liste üzerinde indeksleme işlemi yapmaya kalkıştığımızda hata alıyoruz. Şimdi durumu daha iyi anlayabilmek için isterseniz yukarıdaki kodları bir de ikinci dip noktası kontrolü olmadan yazmayı deneyelim:

```
def düz_liste_yap(liste):
    if not isinstance(liste, list):
        return [liste]
    else:
        return düz_liste_yap(liste[0]) + düz_liste_yap(liste[1:])

l = [1, 2, 3, [4, 5, 6], [7, 8, 9, [10, 11], 12], 13, 14]

print(düz_liste_yap(l))
```

Bu kodları çalıştırdığımızda şu hata mesajıyla karşılaşıyoruz:

```
Traceback (most recent call last):
  File "deneme.py", line 9, in <module>
    print(düz_liste_yap(l))
  File "deneme.py", line 5, in düz_liste_yap
    return düz_liste_yap(liste[0]) + düz_liste_yap(liste[1:])
  File "deneme.py", line 5, in düz_liste_yap
    return düz_liste_yap(liste[0]) + düz_liste_yap(liste[1:])
  File "deneme.py", line 5, in düz_liste_yap
    return düz_liste_yap(liste[0]) + düz_liste_yap(liste[1:])
  File "deneme.py", line 5, in düz_liste_yap
    return düz_liste_yap(liste[0]) + düz_liste_yap(liste[1:])
  File "deneme.py", line 5, in düz_liste_yap
    return düz_liste_yap(liste[0]) + düz_liste_yap(liste[1:])
  File "deneme.py", line 5, in düz_liste_yap
    return düz_liste_yap(liste[0]) + düz_liste_yap(liste[1:])
  File "deneme.py", line 5, in düz_liste_yap
    return düz_liste_yap(liste[0]) + düz_liste_yap(liste[1:])
  File "deneme.py", line 5, in düz_liste_yap
    return düz_liste_yap(liste[0]) + düz_liste_yap(liste[1:])
```

```
IndexError: list index out of range
```

Gördüğünüz gibi, biraz önce boş bir liste üzerinde indeksleme yapmaya çalıştığımızda aldığımız hatanın aynısı bu. Çünkü kodlarımızın else bloğuna bakarsanız liste üzerinde indeksleme yaptığımızı görürsünüz:

```
return düz_liste_yap(liste[0]) + düz_liste_yap(liste[1:])
```

Elbette boş bir liste liste[0] veya liste[1:] gibi sorgulamalara IndexError tipinde bir hata mesajıyla cevap verecektir. İşte böyle bir durumda hata almamak için şu kodları yazıyoruz:

```
elif not liste:
    return []
```

Böylece özyineleme esnasında boş bir listeyle karşılaştığımızda bu listeyi şu şekle dönüştürüyoruz:

```
[[]]
```

Böyle bir yapı üzerinde indeksleme yapılabilir:

```
>>> a = [[]]
>>> a[0]

[]
```

Dip noktaya ulaşılana kadar yapılacak işlemler ise şunlar:

```
return düz_liste_yap(liste[0]) + düz_liste_yap(liste[1:])
```

Yani listenin ilk öğesine, geri kalan öğeleri teker teker ekliyoruz.

Gelin bir örnek daha verelim:

```
def topla(sayilar):
    if len(sayilar) < 1:
        return 0
    else:
        ilk, son = sayilar[0], sayilar[1:]
        return ilk+topla(son)
```

Bu fonksiyonun görevi, kendisine liste olarak verilen sayıları birbiriyle toplamak. Biz bu işi başka yöntemlerle de yapabileceğimizi biliyoruz, ama bizim burada amacımız özyinelemeli fonksiyonları anlamak. O yüzden sayıları birbiriyle toplama işlemini bir de bu şekilde yapmaya çalışacağız.

Elimizde şöyle bir liste olduğunu varsayalım:

```
liste = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Böyle bir durumda fonksiyonumuz *55* çıktısı verir.

Gelelim bu fonksiyonu açıklamaya...

Her zamanki gibi ilk olarak dip noktamızı tanımlıyoruz:

```
if len(sayilar) < 1:
    return 0
```

Buna göre *sayilar* adlı listenin uzunluğu 1'in altına düşünce *0* değerini döndürüyoruz. Burada *0* değerini döndürmemizin nedeni, listede öğe kalmadığında programımızın hata vermesini önlemek. Eğer *0* dışında başka bir sayı döndürürsek bu sayı toplama işleminin sonucuna etki edecektir. Toplama işleminin sonucunu etkilemeyecek tek sayı *0* olduğu için biz de bu sayıyı döndürüyoruz.

Taban noktaya varılıncaya kadar yapılacak işlemler ise şunlar:

```
ilk, son = sayilar[0], sayilar[1:]
return ilk+topla(son)
```

Burada amacımız, listenin ilk sayısı ile listenin geri kalan öğelerini tek tek birbiriyle toplamak. Bunun için *sayilar* adlı listenin ilk öğesini, listenin geri kalanından ayırıyoruz ve ilk öğeyi *ilk*; geri kalan öğeleri ise *son* adlı bir değişkene gönderiyoruz:

```
ilk, son = sayilar[0], sayilar[1:]
```

Sonra da ilk öğeyi, geri kalan liste öğeleri ile tek tek topluyoruz. Bunun için de topla() fonksiyonunun kendisini *son* adlı değişken içinde tutulan liste öğelerine özyinelemeli olarak uyguluyoruz:

```
return ilk+topla(son)
```

Böylece liste içindeki bütün öğelerin toplam değerini elde etmiş oluyoruz.

Bu arada, yeri gelmişken Python programlama dilinin pratik bir özelliğinden söz edelim. Gördüğünüz gibi sayıların ilk öğesini geri kalan öğelerden ayırmak için şöyle bir kod yazdık:

```
ilk, son = sayilar[0], sayilar[1:]
```

Aslında aynı işi çok daha pratik bir şekilde de halledebilirdik. Dikkatlice bakın:

```
ilk, *son = sayilar
```

Böylece *sayilar* değişkenin ilk öğesi *ilk* değişkeninde, geri kalan öğeleri ise *son* değişkeninde tutulacaktır. İlerleyen derslerde 'Tekrarlayıcılar' (*Iterators*) konusunu işlerken bu yapıdan daha ayrıntılı bir şekilde söz edeceğiz.

Bezeyiciler (*Decorators*)

Kapalı Fonksiyonlar (*Closures*)

başa dön

Katkıda Bulunanlar

Bu sayfada, Python3 belgelerine herhangi bir şekilde katkıda bulunanların listesi bir tablo halinde yer alıyor. Bu tabloda kimlerin hangi konuya ne şekilde katkıda bulunduğunu görebilirsiniz.

Lütfen siz de belgelerde gördüğünüz hataları kistihza [at] yahoo [nokta] com adresine iletmekten çekinmeyin. Katkılarınız, bu belgelerin hem daha az hata içermesini hem de daha çok kişiye ulaşmasını sağlayacaktır.

Katkıda Bulunan	**Konu**	**Açıklama**
Barbaros Akkurt	*Temel Komut Satırı Bilgisi*	`echo $HOME` komutunun, hatalı olarak *C:\Users\falanca* şeklinde gösterilen çıktısı */home/istihza* olarak düzeltildi.
Şebnem Duyar	*Kaçış Dizileri*	Yanlışlıkla 'dahtaa' şeklinde yazılan 'daha' kelimesi düzeltildi.
Şebnem Duyar	*İşleçler*	Bölme işleminde yanlışlıkla *30* olarak gösterilen değer *3* olarak değiştirildi.
Şebnem Duyar	*İşleçler*	*23 + 5* işleminin yanlışlıkla *27* olarak gösterilen sonucu *28* olarak düzeltildi.
Onur Eker	*Etkileşimli Python*	Satırdaki kayma düzeltildi.
Onur Eker	*İşleçler*	*b* değişkeninin *23* olarak verilen değeri *10* olarak düzeltildi.
Onur Eker	*Karakter Dizilerinin Metotları (Devamı)*	`count()` örneğinin açıklamasında yanlışlıkla *2* olarak belirtilen karakter sayısı *1* olarak düzeltildi.
Emre Erözgün	*Kullanıcıyla Veri Alışverişi*	Yanlışlıkla `type()` ile yazılan örnek `int()` ile yeniden yazıldı.
Emre Erözgün	*Döngüler (Loops)*	Tamsayıya çevrilmesi unutulan öğeler sayıya çevrildi.
Tayfun Yaşar	*Sayma Sistemleri*	Koyulması unutulan küme parantezleri karakter dizisi içine yerleştirildi.
Metin Hırçın	*Temel Program Kaydetme ve Çalıştırma Mantığı*	Yanlışlıkla 'etkileşimli kabul' şeklinde yazılan ifade 'etkileşimli kabuk' olarak düzeltildi.
Metin Hırçın	*Karakter Dizilerinin Metotları (Devamı)*	Yanlışlıkla 'içidnde' şeklinde yazılan ifade 'içinde' olarak düzeltildi.
Metin Hırçın	*Karakter Dizilerinin Metotları*	Yanlışlıkla 'görünmüyür' şeklinde yazılan ifade 'görünmüyor' olarak

	(Devamı)	düzeltildi.
Metin Hırçın	*Listeler ve Demetler*	Yanlışlıkla ‘öğrendikce’ şeklinde yazılan ifade ‘öğrendikçe’ olarak düzeltildi.
Metin Hırçın	*Baytlar (Bytes) ve Bayt Dizileri (Bytearrays)*	Yanlışlıkla ‘dizilerinne’ şeklinde yazılan ifade ‘dizilerine’ olarak düzeltildi.
Metin Hırçın	*Sözlükler*	Yanlışlıkla birbirinden ‘,’ işareti ile ayrılan anahtar-değer çiftleri ‘:’ işareti ile ayrıldı.
Metin Hırçın	*Gömülü Fonksiyonlar*	Yanlışlıkla ‘yanınıra’ şeklinde yazılan ifade ‘yanısıra’ olarak düzeltildi.
Emre Erözgün	*Karakter Dizilerinin Metotları (Devamı)*	Hatalı çıktı veren *count()* örneği düzeltildi.
Ahmet Öztekin	*Kaçış Dizileri*	Yanlışlıkla ‘yukarı’ şeklinde yazılan ifade ‘yukarıda’ olarak düzeltildi.
Ahmet Öztekin	*Karakter Dizilerinin Metotları (Devamı)*	*rjust()* metodunun yanlış yazılan çıktısı düzeltildi.
Ahmet Öztekin	*Listeler ve Demetler*	*li1* yerine yanlışlıkla *l1* şeklinde yazılan değişken adı düzeltildi.

başa dön