Resmi Python 3 Kılavuzu

Sürüm 3

Guido van Rossum
ve Python geliştirme ekibi
(Çeviren: Fırat Özgül)

15.12.2015

İÇİNDEKİLER

Python öğrenmesi kolay, güçlü bir programlama dilidir. Verimli, yüksek seviyeli veri yapılarına sahip bu dil, nesne tabanlı programlamayı da basit ama etkili bir tarzda ele alır. Python'ın zarif söz dizimi ve dinamik veri tipleri, dilin yorumlanan yapısı ile birleştiğinde, onu pek çok alanda ve çoğu platformda, betik yazma ve hızlı uygulama geliştirme konusunda ideal bir dil haline getirir.

Python'ın yorumlayıcısı ve kapsamlı standart kütüphanesi https://www.python.org/ adresindeki Python web sitesinden belli başlı bütün platformlar için kaynak kod veya ikili paket halinde ücretsiz olarak indirilip özgürce dağıtılabilir. Adı geçen web sitesi aynı zamanda pek çok ücretsiz üçüncü şahıs Python modüllerini, programları, araçları ve ilave belgeleri barındırmanın yanısıra, bunlara nerelerden ulaşabileceğinizi gösteren bağlantılar da içerir.

Python'ın yorumlayıcısı, C veya C++ dillerinde (ya da C'den çağrılabilen başka dillerde) yazılmış fonksiyonlar ve veri tipleri aracılığıyla kolayca genişletilebilir. Python aynı zamanda özelleştirilebilen uygulamalar için bir eklenti dili olmaya da uygundur.

Bu kılavuzda Python dilinin ve sisteminin temel kavramları ile özellikleri, okura teknik olmayan bir dille sunulmuştur. Öğrendiklerinizi deneyebilmek için elinizin altında bir Python yorumlayıcısının bulunması elbette iyi olur, ama buradaki bütün örnekler kendi içinde bağımsızdır. Bu sayede bu kılavuzu internete veya bilgisayara ihtiyaç duymadan da okuyabilirsiniz.

Standart nesnelerin ve modüllerin açıklamaları için Standart Python Kütüphanesi'ne bakabilirsiniz. Python Dil Referansı'nda bu dil daha teknik bir şekilde tanımlanmıştır. C veya C++ dillerinde uzantı yazabilmek için Python Yorumlayıcısının Genişletilmesi ve Başka Uygulamalara İliştirilmesi ile Python/C API Referans Kılavuzu başlıklı belgeleri okuyabilirsiniz. Ayrıca etrafta Python'ı derinlemesine ele alan çeşitli kitaplar da bulunmaktadır.

Bu kılavuzda, dilin tek tek her özelliğini, hatta yaygın kullanılan özelliklerinin hepsini, tüm ayrıntılarıyla ele almak gayesi güdülmemiştir. Ama bu kılavuzda Python'ın en dikkate değer özelliklerinin pek çoğu sunulmuş olup, bu kılavuz size dilin havası ve tarzı hakkında epeyce fikir verecektir. Bu kılavuzu bitirdikten sonra Python modüllerini ve programlarını okuyup yazabilmenin yanısıra, Standart Python Kütüphanesi'nde ele alınan çeşitli kütüphane modülleri hakkında daha fazla bilgi edinmeye de hazır hale geleceksiniz.

Ayrıca sözlük kısmı da göz atmaya değerdir.

BÖLÜM 1

İştahınızı Kabartalım

Eğer bilgisayarlarla bolca haşır neşir oluyorsanız, otomatikleştirmek isteyeceğiniz birtakım işlerle günün birinde mutlaka karşılaşırsınız. Örneğin çok sayıda metin dosyası üzerinde bir arama-değiştirme işlemi gerçekleştirmek veya bir dizi fotoğraf dosyasını karmaşık bir şekilde adlandırıp yeniden düzenlemek isteyebilirsiniz. Belki de kendi ihtiyaçlarınıza özgü ufak bir veritabanı ya da belli bir amaca yönelik grafik arayüzlü uygulama veya basit bir oyun yazmak istiyorsunuzdur.

Eğer siz profesyonel bir yazılım geliştirici iseniz, belki de çeşitli C/C++/Java kütüphaneleri ile çalışmanız gerekiyor, ancak o bildik yazma/derleme/test etme/yeniden derleme döngüsünü çok hantal buluyorsunuzdur. Belki de yukarıda bahsi geçen kütüphane için bir test takımı yazıyorsunuzdur, ama test kodlarını yazmak size angarya geliyordur. Ya da belki, bir eklenti dili kullanabilecek bir program yazmışsınızdır, ama uygulamanız için baştan aşağı yeni bir eklenti dili tasarlayıp gerçeklemek istemiyorsunuzdur.

Eğer öyleyse Python tam size göre bir dildir.

Yukarıda bahsi geçen işlerin bazıları için Unix kabuk betikleri veya Windows toplu iş dosyaları yazabilirsiniz, ancak kabuk betikleri daha ziyade dosyaları bir yerden bir yere taşımaya ve metin verileri üzerinde değişiklik yapmaya yarar; bunlar grafik arayüzlü uygulamalar veya oyunlar için pek elverişli değildir. Elbette bu işler için C/C++/Java programları da yazabilirsiniz, ancak programın daha ilk taslağını çıkarmak bile çok fazla geliştirme çalışmasına girişmenizi gerektirecektir. Python'ın ise kullanımı daha kolaydır; Windows, Mac, OS X ve Unix işletim sistemlerinde çalışır; işinizi daha çabuk bir şekilde halletmenize yardımcı olur.

Python'ın kullanımı basit olsa da, büyük programlar için, kabuk betikleri veya toplu iş dosyalarının sunabileceğinden çok daha fazla yapı ve destek sunan dört dörtlük bir programlama dilidir. Öte yandan Python C'ye kıyasla daha fazla hata denetimine imkan tanır ve *epey yüksek seviyeli bir dil* olarak, bünyesinde esnek diziler ve sözlükler gibi yüksek seviyeli veri tiplerini barındırır. Python, sahip olduğu daha genel amaçlı veri tipleri sayesinde Awk ya da hatta Perl'e göre daha geniş bir problem sahasına hitap etmekle birlikte, pek çok şey Python'da en az bu programlama dillerindeki kadar kolaydır.

Python, programlarınızı, başka Python programlarında yeniden kullanabileceğiniz birtakım modüllere bölmenize de izin verir. Python'da, programlarınız için temel olarak alabileceğiniz (veya Python ile programlamayı öğrenmeye başlamada örnek olarak kullanabileceğiniz) çok sayıda hazır modül bulunur. Bunlar arasında dosya giriş-çıkışı, sistem çağrıları ve soket desteğinin yanısıra, Tk gibi grafik kullanıcı arayüzü takımları için arayüzler sunan modüller dahi vardır.

Python yorumlanan bir dildir; bu sayede, herhangi bir derleme ve bağlama işlemi de gerekmediği için, program geliştirme esnasında zamandan epey tasarruf edebilirsiniz.

Yorumlayıcıyı etkileşimli olarak kullanabilirsiniz. Bu da dilin özelliklerine ilişkin deneme çalışmalarını kolayca yapabilmenizi, tek kullanımlık programlar yazabilmenizi veya aşağıdan-yukarıya (bottom-up) program geliştirme sırasında fonksiyonlarınızı test edebilmenizi sağlar. Python aynı zamanda kullanışlı bir hesap makinesi olma işlevi de görür.

Python, programlarınızı daha öz ve daha okunaklı bir şekilde yazabilmenizi sağlar. Python ile yazılan programlar C, C++ veya Java ile yazılmış muadillerinden genellikle çok daha kısadır. Bunun çeşitli sebepleri vardır:

- Yüksek seviyeli veri tipleri sayesinde karmaşık işlemleri tek bir deyimle ifade edebilirsiniz;
- Deyimler, başlangıç ve bitiş parantezleri yerine girintileme ile gruplanır;
- Değişkenleri veya argümanları önceden bildirmenize gerek yoktur.

Python *genişletilebilir* bir dildir: Eğer C biliyorsanız, gerek kritik işlemleri azami hızla gerçekleştirmek, gerekse Python programlarını yalnızca ikili formda mevcut bulunan kütüphanelere (mesela üreticiye özgü bir grafik kütüphanesine) bağlamak amacıyla Python yorumlayıcısına yeni bir gömülü fonksiyon veya modül eklemek basit bir iştir. Artık bu dile iyiden iyiye ısındıktan sonra, Python'ın yorumlayıcısını C ile yazılmış bir uygulamaya bağlayabilir, bunu o uygulama için bir eklenti veya komut dili olarak kullanabilirsiniz.

Bu arada bu dil, adını BBC'de yayımlanan "Monty Python's Flying Circus" adlı gösteriden alır; sürüngenlerle bir ilgisi yoktur. Dolayısıyla belgelerinizde Monty Python skeçlerine atıfta bulunmanıza izin veriyoruz; hatta izin vermekle kalmıyor, bunu teşvik de ediyoruz!

Artık hepiniz Python'ı merak ettiğinize göre, bu dili biraz daha ayrıntılı bir şekilde incelemeye başlayabiliriz. Bir dili öğrenmenin en iyi yolu onu kullanmak olduğu için, kılavuzu okurken sizi Python yorumlayıcısını kurcalamaya davet ediyoruz.

Bir sonraki bölümde yorumlayıcıyı kullanmanın inceliklerini göstereceğiz. Vereceğimiz bilgiler oldukça yavan olsa da bunlar daha sonra sunulacak örnekleri deneyebilmeniz açısından önem taşıyor.

Kılavuzun geri kalanında, basit ifadeler, deyimler ve veri tipleri ile başlanarak, fonksiyonlar ve modüller de ele alındıktan sonra nihayet istisnalar ve kullanıcı tarafından tanımlanan sınıflar gibi ileri düzey kavramlara da değinilerek, Python dili ve sisteminin çeşitli özellikleri örnekler yoluyla sunulacaktır.

BÖLÜM 2

Python Yorumlayıcısının Kullanımı

2.1 Yorumlayıcının Başlatılması

Python yorumlayıcısı, mevcut olduğu makinelerde, genellikle `/usr/local/bin/python3.5` dizininde kuruludur. Unix kabuğunun arama yoluna `/usr/local/bin` satırını eklemeniz halinde; kabukta

```
python3.5
```

komutunu vererek yorumlayıcıyı başlatabilirsiniz.[1] Yorumlayıcının yer aldığı dizinin hangisi olduğu kurulum sırasındaki bir seçenekle belirlendiği için, başka konumlar da mümkündür. Bu dizinin hangisi olabileceğini etrafınızda Python bilen birilerine veya sistem yöneticinize sorabilirsiniz. (Örn. `/usr/local/python` da yaygın bir alternatif konumdur.)

Windows makinelerinde Python genellikle `C:\Python35` dizinine kurulur, ama kurulum programını çalıştırdığınızda siz bunu değiştirebilirsiniz. Python'ı kurduğunuz dizini yola eklemek için DOS komut satırında aşağıdaki komutu verebilirsiniz:

```
set path=%path%;C:\python35
```

Birincil komut satırında dosya sonu karakterinin (Unix'te `Control-D`, Windows'ta `Control-Z`) girilmesi, yorumlayıcının *sıfır çıkış durumu* ile (yani hatasız bir şekilde) kapanmasını sağlar. Eğer bu işe yaramazsa, yorumlayıcıdan çıkmak için şu komutu verebilirsiniz: `quit()`.

Yorumlayıcı, readline yazılımını destekleyen sistemlerde etkileşimli düzenleme, önceki kodları getirme ve kod tamamlama gibi satır düzenleme özelliklerine sahiptir. Komut satırı düzenleme özelliğinin var olup olmadığını anlamanın herhalde en hızlı yolu, karşılaştığınız ilk Python komut satırında Control-P tuşlarına basmak olacaktır. Eğer bir bip sesi duyarsanız komut satırı düzenleme özelliği etkin demektir. Tuşların kullanımına ilişkin giriş mahiyetindeki bilgiler için *Etkileşimli Girdi Düzenleme ve Eski Kodları Getirme* başlıklı eke bakabilirsiniz. Eğer hiçbir şey olmazsa veya çıktıda `^P` görürseniz, komut satırı düzenleme özelliğinden faydalanamayacaksınız demektir; bu durumda yapabileceğiniz tek şey, klavyedeki geri tuşunu kullanarak mevcut satırdan karakterleri silebilmek olacaktır.

Yorumlayıcının çalışma şekli bir bakıma Unix kabuğuna benzer: Yorumlayıcı, standart girdi bir tty cihazına bağlıyken çalıştırıldığında, komutları etkileşimli olarak okur ve yürütür; standart girdi olarak bir dosya adı veya dosya argümanı ile çalıştırıldığında ise, o dosyadaki *betiği* okur ve yürütür.

[1] Unix'te, Python 3.x yorumlayıcısının çalıştırılabilir dosyası öntanımlı olarak `python` adıyla kurulmaz. Bunun amacı, sistemde aynı anda kurulu olabilecek Python 2.x çalıştırılabilir dosyasıyla herhangi bir çakışmayı engellemektir.

Yorumlayıcıyı başlatmanın ikinci bir yolu da `python -c komut [arg] ...` yapısını kullanmaktır. Böylece, kabuktaki `-c` seçeneğine benzer şekilde, *komut* içindeki deyim(ler) yürütülür. Python deyimleri genellikle boşluklar ya da kabuk için özel anlamı olan başka karakterler içerdiğinden, *komutu* bütünüyle tek tırnak içine alsanız iyi olur.

Bazı Python modülleri aynı zamanda betik olarak da kullanılabilir. Bu modüller `python -m modül [arg]...` şeklinde çalıştırılabilir. Böylece *modülün* kaynak dosyası, sanki dosyanın tam konumunu ve adını komut satırına yazmışsınız gibi yürütülecektir.

Bir betik dosyası kullanılırken, bazen betiği çalıştırdıktan hemen sonra etkileşimli kipe geçmek faydalı olabilir. Bu iş betik adından önce `-i` seçeneği getirilerek yapılabilir.

Komut satırına ilişkin bütün seçenekler Komut Satırı ve Çevresi adresinde açıklanmıştır.

2.1.1 Argüman Atama

Yorumlayıcıya gönderilen betik adı ve ilave argümanlar, eğer yorumlayıcı bunları tanıdıysa, karakter dizilerinden oluşan bir listeye dönüştürülüp `sys` modülü içindeki `argv` değişkenine atanır. Adı geçen bu listeye `import sys` komutuyla erişebilirsiniz. Bu listenin uzunluğu en az birdir. Yani yorumlayıcı herhangi bir betik adı ve argüman olmadan çalıştırıldığında `sys.argv[0]` boş bir karakter dizisi olacaktır. Betik adı (yani standart girdi) `'-'` olarak verildiğinde, `sys.argv[0]` `'-'` olarak ayarlanır. `-c` *komut* yapısı kullanıldığında `sys.argv[0]` `'-c'` olarak ayarlanır. `-m` *modül* yapısı kullanıldığında ise `sys.argv[0]` modülün tam yolu ve tam adı olarak ayarlanır. `-c` *komut* veya `-m` *modül* yapısından sonra getirilen seçenekler Python yorumlayıcısının seçenek işleme mekanizması tarafından dikkate alınmasa da, bunlar yine de komut veya modül tarafından kullanılmak üzere `sys.argv` listesine yerleştirilir.

2.1.2 Etkileşimli Kip

Komutlar bir tty cihazından okunuyorsa, yorumlayıcı *etkileşimli kipte* demektir. Bu kipte komutlarımızı genellikle üç adet büyüktür işareti ile (>>>) gösterilen *birincil komut isteminden* hemen sonra yazıyoruz. Devam satırları için ise, öntanımlı olarak üç adet nokta ile (...) gösterilen *ikincil komut istemi* devreye girecektir. Yorumlayıcı, birincil komut istemini ekrana basmadan önce, sürüm numarası ve telif hakkına dair birtakım bilgiler içeren bir karşılama mesajı gösterir:

```
$ python3.5
Python 3.5 (default, Sep 16 2015, 09:25:04)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Devam satırlarına ise, birden fazla satır içeren bir yapıya girilirken ihtiyaç duyulur. Bir örnek olarak, şu `if` deyimine bir göz atalım:

```
>>> dünya_düzdür = True
>>> if dünya_düzdür:
...     print("Dikkat edin de düşmeyin!")
...
Dikkat edin de düşmeyin!
```

Etkileşimli kip hakkında daha fazla bilgi için *Etkileşimli Kip* bölümüne bakınız.

2.2 Yorumlayıcı ve Çevresi

2.2.1 Kaynak Dosyalarının Dil Kodlaması

Python kaynak dosyaları öntanımlı olarak UTF-8 ile kodlanmış gibi işlem görür. Bu dil kodlamasında, dünyadaki çoğu dilin karakterleri karakter dizilerinde, tanım adlarında ve yorumlarda aynı anda kullanılabilir. Ancak standart kütüphanede tanım adları için yalnızca ASCII karakterler kullanılır. Taşınabilir olması istenen her kodda uyulması gereken bir kaidedir bu. Bütün bu karakterleri düzgün bir şekilde görüntüleyebilmek için, kullandığınız metin düzenleyicinin de, dosyanın UTF-8 olduğunu anlayabilmesi ve dosyadaki bütün karakterleri destekleyen bir yazı tipi kullanması gerekir.

Bunun yanısıra, kaynak dosyaları için farklı bir dil kodlaması belirtmek de mümkündür. Kaynak dosyasının dil kodlamasını tanımlamak için, `#!` ile başlayan başka bir özel yorum satırı daha eklemeniz gerekir:

```
# -*- coding: dil_kodlamasinin_adi -*-
```

Bu bildirim sayesinde kaynak dosyadaki her şey, UTF-8 yerine, belirttiğiniz *dil kodlamasına* sahipmiş gibi değerlendirilecektir. Kullanabileceğiniz dil kodlamalarının listesine Python Kütüphane Referansı'ndaki `codecs` bölümünden ulaşabilirsiniz.

Örneğin, eğer tercih ettiğiniz metin düzenleyici UTF-8 ile kodlanmış dosyaları desteklemiyorsa ve başka bir dil kodlaması kullanmanızı dayatıyorsa (mesela Windows-1252), şu satırı yazabilirsiniz:

```
# -*- coding: cp-1252 -*-
```

Böylece Windows-1252 adlı karakter kümesi içindeki bütün karakterleri kaynak dosyanızda kullanabilirsiniz. Dil kodlamasını ayarlayan bu özel yorum satırı, dosya içindeki *ilk veya ikinci* satır olmak zorundadır.

BÖLÜM 3

Python'a Teknik Olmayan bir Giriş

Aşağıdaki örneklerde neyin komut, neyin bu komutun çıktısı olduğunu, komut istemcilerinin (>>> ve ...) varlığı veya yokluğu ile ayırt edebilirsiniz: Örneği denemek için, komut istemcisini ekranda gördükten sonra, örnek kodun tamamını komut istemcisinin hemen ardından yazmanız gerekiyor. Komut istemcisinin olmadığı satırlarsa yorumlayıcının verdiği çıktıları temsil ediyor. Herhangi bir örnekte, eğer satır başında yalnızca ikincil komut istemcisini görüyorsanız tekrar ENTER tuşuna basın. Bu şekilde, birden fazla satıra yayılan komutları sona erdirmiş oluyoruz.

Bu kılavuzdaki örneklerin pek çoğunda, hatta etkileşimli komut istemcisinde bile, yorum satırları göreceksiniz. Python'da yorumlar diyez karakteri (#) ile başlar ve satır sonuna kadar devam eder. Yorumlar satır başında yer alabileceği gibi, boşluk ya da koddan sonra da gelebilir, ama karakter dizilerinin içinde yer alamaz. Diyez işareti bir karakter dizisinin içinde geçtiğinde, alelade bir diyez işareti olarak işlem görecektir. Yorumlar, yalnızca yazdığınız kodları açıklama amacı taşıdığı, Python tarafından işleme alınmadığı için, aşağıdaki örnekleri denerken yorumları yazmasanız da olur.

Birkaç örnek verelim:

```
# bu ilk yorumumuz
varan = 1  # bu da ikinci yorumumuz
           # ... bu da üçüncüsü!.
metin = "# Tırnak içinde gösterildiği için bu bir yorum değildir."
```

3.1 Python'ı Hesap Makinesi Olarak Kullanmak

Gelin şimdi birkaç basit Python komutu deneyelim. Yorumlayıcıyı başlatalım ve birincil komut istemcisinin (>>>) görüntülenmesini bekleyelim. (Çok sürmez.)

3.1.1 Sayılar

Yorumlayıcıyı basit bir hesap makinesi niyetine kullanabilirsiniz. Yorumlayıcı, ekrana yazdığınız ifadenin sonucunu size çıktı olarak verecektir. Python'da ifadelerin söz dizimi gayet yalındır: +, -, * ve / işleçleri tıpkı başka dillerdeki gibi çalışır (örneğin Pascal veya C). Gruplama işlemleri için ise parantezlerden (()) yararlanabilirsiniz. Mesela:

```
>>> 2 + 2
4
>>> 50 - 5*6
```

```
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # bölme her zaman kayan noktalı bir sayı döndürür
1.6
```

Tam sayılar (örn. `2`, `4`, `20`) `int` tipindeyken, ondalık kısmı olan sayılar ise (örn. `5.0`, `1.6`) `float` tipindedir. Kılavuzun ilerleyen bölümlerinde sayı tiplerine ilişkin daha fazla bilgi vereceğiz.

Bölme işlemi (`/`) her zaman 'float' tipinde bir değer döndürür. taban bölme işlemi yaparak (ondalık kısmın atılması suretiyle) tam sayı sonuç elde etmek için `//` işlecini kullanabilirsiniz. Bölme işleminden kalanı hesaplamak için ise `%` işleci kullanılır:

```
>>> 17 / 3  # klasik bölme işlemi 'float' döndürür.
5.666666666666667
>>>
>>> 17 // 3  # taban bölme işleminde ondalık kısım atılır
5
>>> 17 % 3  # % işleci bölme işleminden kalan sayıyı verir
2
>>> 5 * 3 + 2  # sonuç * bölen + kalan
17
```

Python'da bir sayının kuvvetlerini hesaplamak için `**` işlecini kullanabilirsiniz[1]:

```
>>> 5 ** 2  # 5'in karesi
25
>>> 2 ** 7  # 2'nin 7. kuvveti
128
```

Eşittir işareti (`=`) bir değişkene değer atamak için kullanılır. Sonrasında, herhangi bir çıktı verilmeden alt satıra geçilir:

```
>>> genişlik = 20
>>> yükseklik = 5 * 9
>>> genişlik * yükseklik
900
```

Eğer bir değişken 'tanımlı' değilse (yani kendisine bir değer atanmamışsa), bunu kullanmaya çalışmak hata almanıza yol açacaktır:

```
>>> n  # tanımsız bir değişkene erişmeye çalışıyoruz.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Python'da kayan noktalı sayılar bütünüyle desteklenir: Eğer bir işlecin işlenenleri birbirinden farklı tipte ise, tam sayı değerli işlenenler kayan noktalı sayıya dönüştürülür:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Etkileşimli kipte ekrana en son basılan ifade `_` değişkenine atanır. Bu sayede Python'ı hesap

[1] `**` işleci `-` işlecine göre daha yüksek bir önceliğe sahip olduğu için, `-3**2` işlemi `-(3**2)` şeklinde yorumlanacak ve dolayısıyla bu işlem `-9` sonucunu verecektir. Bunu önleyip `9` sonucunu elde etmek isterseniz, işlemi `(-3)**2` şeklinde yazabilirsiniz.

makinesi olarak kullanırken, işlemlerin sürekliliğini sağlamanız bir nebze kolaylaşacaktır. Örneğin:

```
>>> vergi = 12.5 / 100
>>> fiyat = 100.50
>>> fiyat * vergi
12.5625
>>> fiyat + _
113.0625
>>> round(_, 2)
113.06
```

Ancak bu değişkene salt okunur muamelesi yapmalısınız. Yani bu değişkene kendiniz bir değer atamamalısınız. Bu şekilde, asıl _ ile aynı adı taşıyan başka bir lokal değişken oluşturmuş olursunuz. Bu da, _ adlı gömülü değişkenin sahip olduğu özel işlevin kaybolmasına yol açar.

Python; `int` ve `float` veri tiplerine ek olarak, `Decimal` ve `Fraction` gibi başka sayı tiplerini de destekler. Python ayrıca karmaşık sayıları da öntanımlı olarak destekler. Python'da karmaşık sayıların sanal kısımlarını göstermek için `j` veya `J` soneki kullanılır (örn. `3+5j`).

3.1.2 Karakter Dizileri

Python, sayıların yanısıra, karakter dizileri üzerinde de işlem yapabilir. Karakter dizilerini birkaç farklı şekilde gösterebilirsiniz. Karakter dizileri tek tırnak (`'...'`) veya çift tırnak (`"..."`) içine alınabilir. Her iki kullanım da aynı neticeyi verecektir[2]. Tırnak işaretlerini 'kurtarabilmek' (yani karakter dizileri içinde tırnak işaretlerini kullanabilmek) için \ işaretinden yararlanabilirsiniz.

```
>>> 'adana mersin'  # tek tırnak işaretleri
'adana mersin'
>>> 'Mersin\'e'  # tek tırnağı \' ile kurtarıyoruz...
"Mersin'e"
>>> "Mersin'e"  # ...tek tırnak yerine çift tırnak da olur
"Mersin'e"
>>> '"Evet," dedi.'
'"Evet," dedi.'
>>> "\"Evet,\" dedi."
'"Evet," dedi.'
>>>  '"Adana\'ya," dedi.'
'"Adana\'ya," dedi.'
```

Yorumlayıcı etkileşimli kipteyken karakter dizisi çıktıları tırnak içinde gösterilir; özel karakterler ise önlerine ters taksim işaretleri getirilerek kurtarılır. Bazen girilen değer çıkan değerden farklı görünse de (çıktıda kullanılan tırnak işaretleri girilenden farklı olabilir), iki karakter dizisi birbirine eşdeğerdir. Eğer karakter dizisi içinde tek tırnak işaretleri geçiyor, ama çift tırnak işaretleri geçmiyorsa karakter dizisini çift tırnak ile tanımlıyoruz. Tersi durumda ise karakter dizisini tek tırnak işaretlerini kullanarak tanımlıyoruz. `print()` fonksiyonu, karakter dizisini içine alan tırnakları atarak ve kurtarılan karakterlerle özel karakterleri de ekrana basarak daha okunaklı bir çıktı üretilmesini sağlar.

[2] Başka dillerin aksine, `\n` gibi özel karakterler, tek tırnakla (`'...'`) kullanıldığında da, çift tırnakla (`"..."`) kullanıldığında da aynı anlama sahip olacaktır. Bu ikisi arasındaki tek fark, tek tırnak kullanırken `"` işaretini kurtarmak zorunda olmamanız, ama `\'` işaretini kurtarmak zorunda olmanız; çift tırnak kullanırken ise `'` işaretini kurtarmak zorunda olmamanız, ama `\"` işaretini kurtarmak zorunda olmanızdır.

```
>>>  '"Adana\'ya," dedi.'
'"Adana\'ya," dedi.'
>>> print('"Adana\'ya," dedi.')
"Adana'ya," dedi.
>>> s = 'İlk satır.\nİkinci satır.'  # \n alt satıra geçirir
>>> s  # print() yoksa, \n çıktıda görünür
'İlk satır.\nİkinci satır.'
>>> print(s)  # print() varsa, \n alt satıra geçirir
İlk satır.
İkinci satır.
```

Eğer önünde \ işareti taşıyan karakterlerin özel karakter olarak işlem görmesini istemiyorsanız, ilk tırnak işaretinden önce `r` karakterini getirerek 'ham karakter dizilerinden' yararlanabilirsiniz.

```
>>> print('C:\kullanıcı\nihat')  # burada \n satır başı anlamında!
C:\kullanıcı
ihat
>>> print(r'C:\kullanıcı\nihat')  # tırnaktan önceki r'ye dikkat
C:\kullanıcı\nihat
```

Karakter dizileri birden fazla satıra da yayılabilir. Bunun için üç tırnak işaretlerinden yararlanabilirsiniz: """..."""" veya '''...'''. Satır boşlukları karakter dizisi çıktılarına otomatik olarak yansır, ama bunu engellemek için satır sonuna bir adet \ işareti yerleştirebilirsiniz. Aşağıdaki örneğe bakalım:

```
print("""\
Kullanım: falanca [SEÇENEKLER]
     -h                        Kullanıma ilişkin bu mesajı görüntüler
     -H bilgisayaradı          Bağlanılacak bilgisayarın adı
""")
```

Bu örnek aşağıdaki çıktıyı üretir (ilk satırdaki boşluğun çıktıda görünmediğine dikkat edin):

```
Kullanım: falanca [SEÇENEKLER]
     -h                        Kullanıma ilişkin bu mesajı görüntüler
     -H bilgisayaradı          Bağlanılacak bilgisayarın adı
```

Karakter dizileri, + işleci yardımıyla birleştirilebilir (yani birbirine eklenebilir) ve * işleci ile yinelenebilir:

```
>>> # önce 2 kez 'an', ardından da bir kez 'as'
>>> 2 * 'an' + 'as'
'ananas'
```

Art arda gelen iki veya daha fazla *karakter dizisi* (yani tırnak içine alınmış diziler) otomatik olarak birleştirilir.

```
>>> 'Py' 'thon'
'Python'
```

Ancak bu özellik yalnızca karakter dizileri için geçerlidir. Değişkenler veya ifadeler için değil:

```
>>> önek = 'Py'
>>> önek 'thon'  # bir değişken ile bir karakter dizisi birleştirilemez
  ...
SyntaxError: invalid syntax
>>> ('an' * 2) 'as'
```

```
  ...
SyntaxError: invalid syntax
```

Eğer değişkenleri birbiriyle ya da değişkenleri karakter dizileriyle birleştirmek istiyorsanız + işlecini kullanın:

```
>>> önek + 'thon'
'Python'
```

Bu durum özellikle uzun karakter dizilerini bölmek istediğinizde işinize yarayacaktır:

```
>>> metin = ('Birkaç karakter dizisini birleştirmek için '
             'bunları parantez içine alın.')
>>> metin
'Birkaç karakter dizisini birleştirmek için bunları parantez içine alın.'
```

Karakter dizileri, ilk karakterin sıra numarası 0 olacak şekilde *indekslenebilir* (karakterlerine ayrılabilir). Tek bir karakter için 'char' benzeri ayrı bir veri tipi yoktur. Tek bir karakter, uzunluğu 1 olan bir karakter dizisi muamelesi görecektir:

```
>>> kelime = 'Python'
>>> kelime[0]  # 0 konumundaki karakter
'P'
>>> kelime[5]  # 5 konumundaki karakter
'n'
```

Saymaya sağdan başlamak için sıra numarası olarak eksi sayılar da kullanılabilir:

```
>>> kelime[-1]  # son karakter
'n'
>>> kelime[-2]  # sondan bir önceki karakter
'o'
>>> kelime[-6]
'P'
```

-0 ile 0 aynı şey olduğu için, eksi değerli sıra numaraları -1'den başlar.

İndekslemeye ek olarak, *dilimleme* de desteklenir. İndeksleme tek tek karakterleri almaya yararken, *dilimleme* karakter dizisinin belli bir bölümünü almanızı sağlar:

```
>>> kelime[0:2]  # 0'dan (0 dahil) 2 konumuna kadar karakterler (2 hariç)
'Py'
>>> kelime[2:5]  # 2'den (2 dahil) 5 konumuna kadar karakterler (5 hariç)
'tho'
```

Burada başlangıç değerinin her zaman dahil olduğuna, bitiş değerinin ise dışarıda kaldığına dikkat edin. Bu şekilde `s[:i] + s[i:]` her zaman `s` değerine eşit olacaktır:

```
>>> kelime[:2] + kelime[2:]
'Python'
>>> kelime[:4] + kelime[4:]
'Python'
```

Dilimleme indekslerinin, kullanım kolaylığı sağlayan bazı öntanımlı değerleri vardır: Eğer ilk indeksi yazmazsanız, bu indeksin varsayılan değeri sıfır olacaktır, eğer ikinci indeksi yazmazsanız bunun değeri de öntanımlı olarak dilimlenen karakter dizisinin uzunluğu olacaktır.

```
>>> kelime[:2]  # başlangıçtan 2 konumuna kadar karakterler (2 hariç)
'Py'
>>> kelime[4:]  # 4 konumundan (4 dahil) en sona kadar karakterler
'on'
>>> kelime[-2:] # sondan bir öncekinden (dahil) en sona kadar karakterler
'on'
```

Dilimlemenin nasıl çalıştığını aklınızda tutmak için, ilk karakterin sol kenarının 0 olduğunu varsayarak, sıra numaralarının karakterlerin *ara kısmına* işaret ettiğini düşünebilirsiniz. Bu durumda, *n* karaktere sahip bir karakter dizisinin son karakterinin sağ kenarı *n* sıra numarasına sahip olacaktır. Örneğin:

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

İlk sıradaki sayılar, karakter dizisi içinde 0'dan 6'ya kadar olan konumları verir; ikinci sıra ise bunlara karşılık gelen eksi değerli konumları gösterir. *i* 'den *j* 'ye kadar olan dilim, sırasıyla *i* ve *j* olarak işaretlenen kenarlar arasındaki bütün karakterleri içerir.

Artı değerli sıra numaralarının her ikisi de karakter dizisinin sınırları içinde ise, bu sıra numaralarının farkı, dilimin uzunluğunu verir. Örneğin `kelime[1:3]` ifadesinin uzunluğu 2 olacaktır.

Karakter dizisinin uzunluğunu aşan bir sıra numarası kullanmaya çalışırsanız yorumlayıcı hata verecektir:

```
>>> kelime[42]  # kelimede yalnızca 6 karakter var
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Ama dilimleme işleminde, aralık dışında kalan sıra numaraları kullanıldığında bu durum sessizce geçiştirilecektir:

```
>>> kelime[4:42]
'on'
>>> kelime[42:]
''
```

Python'da karakter dizileri üzerinde değişiklik yapılamaz; bunlar değiştirilemeyen veri tipleridir. Bu yüzden, karakter dizisi içinde sıra numarası belirtilen bir konuma değer atamaya çalışmak hata ile sonuçlanacaktır:

```
>>> kelime[0] = 'J'
  ...
TypeError: 'str' object does not support item assignment
>>> kelime[2:] = 'py'
  ...
TypeError: 'str' object does not support item assignment
```

Eğer karakter dizisini değiştirmek istiyorsanız, yapmanız gereken şey yeni bir karakter dizisi oluşturmaktır:

```
>>> 'J' + kelime[1:]
'Jython'
>>> kelime[:2] + 'py'
'Pypy'
```

`len()` adlı gömülü bir fonksiyon, bize karakter dizilerinin uzunluğunu verir:

```
>>> s = 'çekoslavakyalılaştıramadıklarımız'
>>> len(s)
33
```

Ayrıca bkz.:

Metin Dizisi Tipi - str Karakter dizileri *dizi tiplerine* örnektir. Bunlar bu tiplerin desteklediği ortak işlemleri destekler.

Karakter Dizisi Metotları Karakter dizileri, temel dönüştürme ve arama işlemlerine ilişkin çok sayıda metoda sahiptir.

Karakter Dizisi Biçimlendirme Burada, `str.format()` metodu ile karakter dizisi biçimlendirme işlemlerine ilişkin bilgiler verilmiştir.

printf Tarzı Karakter Dizisi Biçimlendirme `%` işlecinin sol taraftaki işleneninin karakter dizileri ve Unicode dizileri olduğu durumlarda kullanılan eski biçimlendirme işlemleri burada daha ayrıntılı olarak ele alınmıştır.

3.1.3 Listeler

Python, çeşitli değerleri gruplandırmak için kullanılan birtakım *birleşik* veri tiplerini de tanır. Bunların en yeteneklisi *listeler* olup, bunlar köşeli parantezler içinde birbirinden virgülle ayrılmış değerler (öğeler) dizisi şeklinde gösterilir. Listeler farklı tipteki öğeleri barındırabilir, ama genellikle öğelerin hepsi aynı tiptendir.

```
>>> kareler = [1, 4, 9, 16, 25]
>>> kareler
[1, 4, 9, 16, 25]
```

Karakter dizilerinde (ve öteki bütün gömülü dizi tiplerinde) olduğu gibi listeler de indekslenip dilimlenebilir:

```
>>> kareler[0]  # indeksleme öğeyi döndürür
1
>>> kareler[-1]
25
>>> kareler[-3:]  # dilimleme yeni bir liste döndürür
[9, 16, 25]
```

Bütün dilimleme işlemleri, istenen öğeleri içeren yeni bir liste döndürecektir. Yani aşağıdaki dilim, listenin yeni bir kopyasını (sığ kopya) döndürür:

```
>>> kareler[:]
[1, 4, 9, 16, 25]
```

Listeler ayrıca birleştirme gibi işlemleri de destekler:

```
>>> kareler + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Değiştirilemeyen bir veri tipi olan karakter dizilerinin aksine listeler değiştirilebilen bir veri tipidir. Yani bunların içeriğini değiştirmek mümkündür:

```
>>> küpler = [1, 8, 27, 65, 125]  # burada bir hata var
>>> 4 ** 3  # 4 sayısının küpü 65 değil, 64'tür!
64
>>> küpler[3] = 64  # şimdi yanlış değeri değiştiriyoruz
>>> küpler
[1, 8, 27, 64, 125]
```

Ayrıca, `append()` *metodunu* kullanarak (ilerde metotlardan daha ayrıntılı söz edeceğiz) listenin sonuna yeni öğeler de ekleyebilirsiniz:

```
>>> küpler.append(216)  # 6'nın küpünü ekliyoruz
>>> küpler.append(7 ** 3)  # şimdi de 7'nin küpünü
>>> küpler
[1, 8, 27, 64, 125, 216, 343]
```

Dilimlenmiş kısımlara da değer atayabilirsiniz. Bu işlem listenin boyutunu değiştirebileceği gibi, listeyi tamamen temizleyebilir de:

```
>>> harfler = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> harfler
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # bazı değerleri değiştiriyoruz
>>> harfler[2:5] = ['C', 'D', 'E']
>>> harfler
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # şimdi de bunları kaldırıyoruz
>>> harfler[2:5] = []
>>> harfler
['a', 'b', 'f', 'g']
>>> # bütün öğelerin yerine boş bir liste yerleştirerek listeyi temizliyoruz
>>> harfler[:] = []
>>> harfler
[]
```

Gömülü `len()` fonksiyonu aynı zamanda listelere de uygulanabilir:

```
>>> harfler = ['a', 'b', 'c', 'd']
>>> len(harfler)
4
```

Listeleri iç içe geçirmek (yani başka listeleri içeren listeler oluşturmak) de mümkündür. Örneğin:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3.2 Programlama Yolunda İlk Adımlar

Elbette Python'ı, iki ile ikiyi toplamaktan daha karmaşık işlemler için de kullanabiliriz. Mesela *Fibonacci* serilerinin ilk alt-dizilerini şu şekilde yazabiliriz:

```
>>> # Fibonacci serileri:
... # iki öğenin toplamı bir sonraki öğeyi verir
... a, b = 0, 1
>>> while b < 10:
...     print(b)
...     a, b = b, a+b
...
1
1
2
3
5
8
```

Bu örnekle birlikte bazı yeni özellikler görüyoruz.

- İlk satır *çoklu atamaya* bir örnektir: `a` ve `b` değişkenleri aynı anda 0 ve 1 değerlerini üstleniyor. Son satırda bu özellikten yine yararlanıyoruz. Gördüğünüz gibi, herhangi bir atama işlemi gerçekleşmeden önce sağ taraftaki ifadeler işleme alınıyor. Sağ taraftaki ifadeler soldan sağa doğru işlem görüyor.
- `while` döngüsü, koşul (bizim örneğimizde `b < 10`) true olduğu müddetçe yürütülmeye devam ediyor. Python'da, tıpkı C'deki gibi, sıfır harici bütün tam sayı değerler true'dur. Sıfır ise false'tur. Ayrıca koşul, bir karakter dizisi veya liste değeri de olabilir. Hatta bu her türlü dizi olabilir. Sıfır harici bir uzunluğa sahip her şey true, boş diziler ise false'tur. Bu örnekte kullanılan test basit bir karşılaştırma işlemidir. Standart karşılaştırma işleçleri C ile aynı şekilde yazılır: < (küçüktür), > (büyüktür), == (eşittir), <= (küçük eşittir), >= (büyük eşittir) ve != (eşit değildir).
- Döngünün *gövdesi girintili bir şekilde* yazılır: Python'da deyimleri gruplandırma yöntemi girintilemedir. Etkileşimli komut satırında her bir girintilenmiş satır için, sekme tuşuna veya boşluk tuşuna/tuşlarına basmanız gerekir. Gerçek hayatta bir metin düzenleyici yardımıyla daha karmaşık Python kodları yazacaksınız. Bütün kaliteli metin düzenleyicilerde otomatik girintileme özelliği bulunur. Etkileşimli kabukta bir birleşik deyim yazdığınızda, kodun tamamlandığını göstermek için ENTER tuşuna basarak boş satır bırakmalısınız (çünkü ayrıştırıcı (parser), koda ait son satırın hangisi olduğunu kestiremez). Basit bir kod bloğu içindeki her bir satır aynı miktarda girintilenmelidir.
- `print()` fonksiyonu, kendisine verilen argüman(lar)ın değerini ekrana basar. Bu fonksiyon, birden fazla argümanı, kayan noktalı değerleri ve karakter dizilerini ekrana basarken (daha önce verdiğimiz hesap makinesi örneklerinde olduğu gibi) bunları dümdüz ekrana basmakla yetinmez, bu değerlere birbirinden farklı şekilde muamele eder. Karakter dizileri tırnaklar olmadan basılır ve öğeler arasına birer boşluk yerleştirilir. Böylece bu değerleri göze hitap edecek şekilde biçimlendirebilirsiniz. Örneğin:

```
>>> i = 256*256
>>> print('i şu değere sahiptir:', i)
i şu değere sahiptir: 65536
```

Çıktı sonunda satır başına geçilmesini önlemek veya çıktıyı farklı bir karakter dizisi ile sonlandırmak için *end* denen bir isimli argümandan yararlanabilirsiniz:

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print(b, end=',')
...     a, b = b, a+b
...
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

BÖLÜM 4

Başka Kontrol Akışı Araçları

Python, daha önce gösterdiğimiz `while` deyiminin yanısıra, bazı farklılıkları olmakla birlikte, başka dillerden aşina olduğumuz kontrol akışı deyimlerine de yer verir.

4.1 `if` Deyimleri

Herhalde yukarıda andığımız bu deyimlerin en bilineni `if` deyimleridir. Örneğin:

```
>>> x = int(input("Lütfen bir tamsayı girin: "))
Lütfen bir tamsayı girin: 42
>>> if x < 0:
...     x = 0
...     print('Eksi değer sıfıra dönüştürüldü')
... elif x == 0:
...     print('Sıfır')
... elif x == 1:
...     print('Tek')
... else:
...     print('Çok')
...
Çok
```

Kodlarınız içinde `elif` kısmı hiç olmayabileceği gibi, birden fazla `elif` de kullanabilirsiniz. `else` kısmı ise tercihe bağlıdır. `elif` ismi 'else if' ifadesinin kısaltması olup, sizi haddinden fazla girintileme yapmaktan kurtarır. `if ... elif ... elif ...` takımı, başka dillerde bulunan `switch` veya `case` deyimlerinin yerini tutar.

4.2 `for` Deyimleri

Python'daki `for` deyimi, C veya Pascal'dan biliyor olabileceğiniz benzerlerinden biraz farklıdır. Python'daki `for` deyimi, (Pascal'daki gibi) daima sayılardan oluşan bir aritmetik dizi üzerinde yürümek veya (C'deki gibi) kullanıcıya hem bir yürüme basamağı hem de durma koşulu belirleme imkanı vermek yerine, herhangi bir diziye (liste veya karakter dizisi) ait öğelerin dizi içinde geçtikleri sırayı gözeterek, bu öğeler üzerinde yürür. Örneğin:

```
>>> # Birkaç karakter dizisini ölçelim:
... kelimeler = ['arnavut', 'şevket', 'çipetpet']
>>> for k in kelimeler:
...     print(k, len(k))
```

```
...
arnavut 7
şevket 6
çipetpet 8
```

Döngü içindeyken, üzerinde yürüdüğünüz dizide değişiklik yapmanız gerekirse (örneğin belirli öğeleri ikilemek isterseniz), ilk önce dizinin bir kopyasını almanızı öneririz. Dizi üzerinde yürüme işlemi dizinin otomatik olarak bir kopyasını almaz. Dilimleme, özellikle bu tür işlemleri bir hayli kolaylaştırır:

```
>>> for k in kelimeler[:]:  # Listenin dilimlenmiş bir kopyası üzerinde döngü kuruyoruz.
...     if len(k) > 7:
...         kelimeler.insert(0, k)
...
>>> kelimeler
['çipetpet', 'arnavut', 'şevket', 'çipetpet']
```

4.3 `range()` Fonksiyonu

Eğer bir sayı dizisi üzerinde yürümeniz gerekirse gömülü `range()` fonksiyonu işinizi kolaylaştıracaktır. Bu fonksiyon yardımıyla aritmetik diziler üretebilirsiniz:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

Fonksiyonda parametre olarak belirttiğiniz bitiş değeri hiçbir zaman üretilen diziye girmez. Yani `range(10)` komutu, uzunluğu 10 olan bir dizinin her bir öğesinin geçerli sıra numarasını gösterecek şekilde, 0'dan başlayarak 10 adet değer üretir. Sayı aralığını farklı bir sayı ile başlatmak veya farklı bir artış (ya da hatta azalış) miktarı belirtmek de mümkündür. Bu miktara 'basamak' dendiği de olur:

```
range(5, 10)
   5'ten 9'a kadar

range(0, 10, 3)
   0, 3, 6, 9

range(-10, -100, -30)
  -10, -40, -70
```

Bir dizideki öğelerin sıra numaraları üzerinde yürümek için `range()` ve `len()` fonksiyonlarını şu şekilde birleştirebilirsiniz:

```
>>> a = ['Ayşeciğin', 'küçücük', 'bir', 'kuzucuğu', 'varmış']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Ayşeciğin
1 küçücük
```

```
2 bir
3 kuzucuğu
4 varmış
```

Ancak buna benzer çoğu durumda `enumerate()` fonksiyonunu kullanmak daha pratik olacaktır (bkz. *Döngü Kurma Teknikleri*).

Eğer sadece aralığı ekrana basacak olursanız, gördüğünüz şey sizi şaşırtabilir:

```
>>> print(range(10))
range(0, 10)
```

`range()` fonksiyonunun döndürdüğü nesne pek çok yönden sanki bir listeymiş gibi davransa da, aslında liste değildir. Bu, ilgili dizinin, üzerinde yürüdüğünüzde elde edeceğiniz bütün öğelerini içeren bir nesnedir, ama aslında bu nesne listeyi gerçekten oluşturmaz, böylece bellekten tasarruf sağlar.

Biz bu tür nesnelere *üzerinde yürünebilir* nesneler diyoruz. Bu nesneler, öğe stoğu tükenene kadar öğelerini peş peşe alabilecekleri bir şey bekleyen fonksiyonlar ve yapılar için uygun bir hedeftir. `for` deyiminin, nesneler üzerinde yürüme işlemi yapabilmemizi sağlayan bir *yürüyücü* (*iterator*) olduğunu görmüştük. Bu tür nesnelere bir başka örnek de `list()` fonksiyonudur. Bu fonksiyon, üzerinde yürünebilir nesnelerden listeler oluşturur:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

İlerleyen sayfalarda, üzerinde yürünebilir nesneler döndüren ve bu nesneleri argüman olarak alan daha başka fonksiyonlar da göreceğiz.

4.4 Döngülerde `break` ve `continue` Deyimleri ve `else` Cümlecikleri

`break` deyimi, tıpkı C'de olduğu gibi, doğrudan doğruya içinde bulunduğu `for` veya `while` döngüsünden çıkılmasını sağlar.

Döngü deyimlerinde bir `else` cümleciği de bulunabilir. Bu cümlecik, listenin son öğesine ulaşılmasıyla (`for` ile) veya koşulun false olmasıyla (`while` ile) döngü sona erdiğinde yürütülür. Ancak döngü `break` deyimi ile sona erdirilmişse bu cümlecik devreye girmez. Asal sayıları bulan aşağıdaki döngüde bu duruma bir örnek görüyorsunuz:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'eşittir', x, '*', n//x)
...             break
...     else:
...         # çarpan bulunamadığında döngü sona eriyor
...         print(n, 'asal sayıdır')
...
2 asal sayıdır
3 asal sayıdır
4 eşittir 2 * 2
5 asal sayıdır
6 eşittir 2 * 3
7 asal sayıdır
```

```
8 eşittir 2 * 4
9 eşittir 3 * 3
```

(Yanlış görmediniz. Bu doğru bir kodlamadır. Dikkatli bakın: burada `else` cümleciği `for` döngüsüne aittir, `if` deyimine **değil**.)

`else` cümlecikleri bir döngü içinde kullanıldıklarında, `if` deyimlerindeki `else` cümleciklerine kıyasla `try` deyimlerindeki `else` cümleciklerine daha çok benzerlik gösterir. `try` deyimlerindeki `else` cümlecikleri herhangi bir istisna oluşmadığı durumlarda çalışırken, döngülerdeki `else` cümlecikleri ise herhangi bir `break` durumu oluşmadığında çalışır. `try` deyimi ve istisnalara ilişkin daha fazla bilgi için bkz. *İstisnaların Yakalanması*.

C'den esinlenen `continue` deyimi, döngüde bir sonraki tura geçilmesini sağlar:

```
>>> for sayı in range(2, 10):
...     if sayı % 2 == 0:
...         print("Bir çift sayıya rastlandı:", sayı)
...         continue
...     print("Bir sayıya rastlandı:", sayı)
Bir çift sayıya rastlandı: 2
Bir sayıya rastlandı: 3
Bir çift sayıya rastlandı: 4
Bir sayıya rastlandı: 5
Bir çift sayıya rastlandı: 6
Bir sayıya rastlandı: 7
Bir çift sayıya rastlandı: 8
Bir sayıya rastlandı: 9
```

4.5 `pass` Deyimleri

`pass` deyimi hiçbir şey yapmaz. Yazdığınız kodların söz diziminin bir deyim kullanılmasını gerektirdiği, ama programınızın herhangi bir şey yapılmasını gerektirmediği durumlarda bunu kullanabilirsiniz. Mesela:

```
>>> while True:
...     pass  # Klavyeden kesme sinyali (Ctrl+C) bekliyoruz
...
```

`pass` deyimi sıklıkla, asgari sınıf yapıları oluşturmak için kullanılmaktadır:

```
>>> class BenimBoşSınıfım:
...     pass
...
```

`pass` deyimi bunun dışında, yeni bir kod üzerinde çalışılırken fonksiyon veya koşullu yapıların gövdeleri içinde bir yer tutucu olarak da kullanılabilir. Bu sayede, kodlarınızın daha soyut kısımları üzerinde kafa yorabilirsiniz. `pass` deyimi Python tarafından sessizce görmezden gelinir:

```
>>> def initlog(*arglar):
...     pass   # Bu kısmı daha sonra yazarız!
...
```

4.6 Fonksiyonların Tanımlanması

Sınırı kullanıcı tarafından belirlenecek şekilde Fibonacci dizilerini ekrana döken bir fonksiyon oluşturalım:

```
>>> def fib(n):    # n'e kadar olan Fibonacci dizilerini yazdırıyoruz
...     """n'e kadar olan Fibonacci dizilerini ekrana basar."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Biraz önce tanımladığımız bu fonksiyonu çağıralım şimdi de:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

`def` sözcüğü fonksiyonun *tanımlanma sürecini* başlatır. Sonra da fonksiyonun adı ve kabul edilecek parametrelerin listesi parantez içinde belirtilir. Fonksiyonun gövdesini oluşturan deyimler bir sonraki satıra girintili bir şekilde yazılır.

İsterseniz fonksiyon gövdesinin ilk deyimi olarak bir karakter dizisi de kullanabilirsiniz. Bu karakter dizisi, fonksiyonun belgelendirme dizisi (ya da *belge dizisi*) olacaktır. (Belge dizilerine ilişkin daha fazla bilgiyi *Belgelendirme Dizileri* kısmında bulabilirsiniz). Belge dizilerinden otomatik olarak çevrimiçi veya basılı belgeler üreten ya da kullanıcıların kodlara etkileşimli bir biçimde göz atabilmesine imkan tanıyan çeşitli araçlar bulunmaktadır. Yazdığınız kodlarda belge dizilerini kullanmanız iyi bir alışkanlıktır. Dolayısıyla bunu kendinize adet edinin.

Bir fonksiyon *yürütüldüğünde*, bu fonksiyonun lokal değişkenlerini barındıran yeni bir simge tablosu oluşur. Daha açık bir dille ifade etmek gerekirse, bir fonksiyon içindeki bütün değişken atama işlemleri, bu değişkenlerin değerini lokal simge tablosunda depolayacaktır. Bu değişkenlerin değerlerine erişmek gerektiğinde ilk olarak lokal simge tablosuna, daha sonra dış taraftaki fonksiyonların lokal simge tablolarına, sonra global simge tablosuna, en son da gömülü isimlerin bulunduğu tabloya bakılacaktır. Global değişkenlere (bunları `global` deyimi ile belirtmedikçe), doğrudan fonksiyon içinden yeni değer atayamazsınız, ama bu değişkenlerin varolan değerini fonksiyon içinde kullanabilirsiniz.

Fonksiyon çağrılırken kullanılan asıl parametreler (argümanlar), fonksiyonun çağrılması esnasında, çağrılan fonksiyonun lokal simge tablosuna dahil edilir. Dolayısıyla argümanlar *değer ile çağırma* esasına göre atanır (burada *değer* her zaman *nesneye atıfta* bulunur, nesnenin değerine değil)[1]. Bir fonksiyon başka bir fonksiyonu çağırdığında, bu çağırma işlemi için yeni bir simge tablosu oluşturulur.

Fonksiyonun tanımlanması, bu fonksiyonun adını mevcut simge tablosuna işler. Fonksiyon adının değeri, yorumlayıcı tarafından, kullanıcı tanımlı bir fonksiyon olarak tanınan bir veri tipine sahiptir. Bu değer başka bir isme atanabilir, böylece bu isim de bir fonksiyon olarak kullanılabilir. Yukarıda bahsi geçen yeniden adlandırmaya ilişkin genel mekanizma şöyle işler:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
```

[1] Aslında *nesneye atıf yoluyla çağırma* daha iyi bir tanım olurdu, çünkü eğer verilen değer değiştirilebilir bir nesne ise, fonksiyonu çağıran yapı, çağrılan yapının nesne üzerinde yaptığı bütün değişiklikleri görecektir (mesela listeye yeni eklenen öğeler).

```
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Eğer başka dillerden geliyorsanız, `fib`'in bir fonksiyon olmadığını, çünkü bunun herhangi bir değer döndürmediğini, dolayısıyla bir prosedür olduğunu savunabilirsiniz. Aslında `return` deyimi içermeyen fonksiyonlar bile bir değer döndürür. Gerçi oldukça yavan bir değerdir bu. Bu değere `None` adı verilir (bu gömülü bir isimdir). Eğer döndürülecek tek değer `None` ise, yorumlayıcı normalde bu değeri size göstermez. Ama eğer gerçekten bu değeri görmek isterseniz `print()` fonksiyonunu kullanabilirsiniz:

```
>>> fib(0)
>>> print(fib(0))
None
```

Fibonacci serilerindeki sayıları ekrana basmak yerine bunları bir liste halinde döndüren bir fonksiyon yazmak da basit bir iştir:

```
>>> def fib2(n): # n'e kadar olan Fibonacci dizilerini döndürüyoruz
...     """n'e kadar olan Fibonacci dizilerini içeren bir liste döndürür."""
...     sonuç = []
...     a, b = 0, 1
...     while a < n:
...         sonuç.append(a)    # aşağıya bakın
...         a, b = b, a+b
...     return sonuç
...
>>> f100 = fib2(100)    # fonksiyonu çağırıyoruz
>>> f100                # sonucu yazdırıyoruz
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Bu örnek, her zaman olduğu gibi, Python'a ilişkin bazı yeni bilgiler içeriyor:

- `return` deyimi fonksiyondan bir değer döndürür. `return` deyimi, eğer herhangi bir argüman almadan, tek başına kullanılırsa `None` değerini döndürür. Fonksiyondan `return` olmadan çıkıldığında da `None` döndürülür.
- `sonuç.append(a)` deyiminde, `sonuç` adlı liste nesnesinin bir *metodunu* çağırmış oluyoruz. Metotlar, nesnelere 'ait' fonksiyonlar olup `nesne.metotadı` şeklinde adlandırılır. Burada `nesne` herhangi bir nesne olabilir (bu bir ifade de olabilir) ve `metotadı` da nesnenin ait olduğu veri tipi içinde tanımlanmış herhangi bir metodun adıdır. Farklı veri tipleri için farklı metotlar tanımlanmıştır. Farklı veri tiplerinin metotları aynı ada sahip olabilir. Bu isimler birbirine karışmaz. (*Sınıfları* kullanarak kendi nesne tiplerinizi ve metotlarınızı tanımlamanız da mümkündür. Bunun için *Sınıflar* bölümüne bakın) Örnekte gösterilen `append()` metodu liste nesneleri için tanımlanmıştır; bu metot listenin en sonuna yeni bir öğe ekler. Bu örnekte bu metot `sonuç = sonuç + [a]` ifadesine eşdeğerdir, ama ondan daha hızlı çalışır.

4.7 Fonksiyonlarının Tanımlanmasına Dair Başka Bilgiler

Değişken sayıda argümana sahip fonksiyonlar tanımlamak da mümkündür. Bunun üç farklı yöntemi bulunur (bu farklı yöntemleri birlikte de kullanabilirsiniz).

4.7.1 Öntanımlı Argüman Değerleri

En kullanışlı yöntem bir veya daha fazla argüman için öntanımlı bir değer belirtmektir. Böylece izin verilenden daha az argümanla çağrılabilecek bir fonksiyon oluşturabilirsiniz. Mesela:

```
def tamam_mı(soru, deneme=4, sitem='Evet veya hayır lütfen!'):
    while True:
        onay = input(soru)
        if onay in ('e', 'hıhı', 'evet'):
            return True
        if onay in ('h', 'cık', 'yok', 'hayır'):
            return False
        deneme = deneme - 1
        if deneme < 0:
            raise OSError('dik kafalı kullanıcı')
        print(sitem)
```

Bu fonksiyon çeşitli şekillerde çağrılabilir:

- Yalnızca zorunlu argüman belirtilerek: `tamam_mı('Gerçekten çıkmak istiyor musunuz?')`
- Tercihe bağlı argümanlardan biri belirtilerek: `tamam_mı('Dosyanın üzerine yazılsın mı?', 2)`
- Hatta argümanların tamamı belirtilerek: `tamam_mı('Dosyanın üzerine yazılsın mı?', 2, 'evet mi diyorsun hayır mı!')`

Bu örnekte ayrıca `in` sözcüğünü de görüyoruz. Bu sözcük bir dizinin belirli bir değeri içerip içermediğini denetler.

Öntanımlı değerler, fonksiyon tanımlandığı anda, *tanımın* etki alanı içinde işlem görür. Yani

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

Bu kod `5` değerini basacaktır.

Önemli uyarı: Öntanımlı değer yalnızca bir kez işlem görür. Eğer öntanımlı değer, liste, sözlük veya çoğu sınıfın örneği gibi değiştirilebilen bir nesne ise farklı bir durum ortaya çıkar. Mesela aşağıdaki fonksiyon, art arda gelen çağrılarda kendisine verilen argümanları biriktirir:

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

Bu kodlar şu çıktıyı verir:

```
[1]
[1, 2]
[1, 2, 3]
```

Eğer öntanımlı değerin art arda gelen çağrılar arasında paylaşılmasını istemiyorsanız, fonksiyonu yukarıdaki yerine şöyle yazabilirsiniz:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.7.2 İsimli Argümanlar

Fonksiyonlar aynı zamanda, `isimli_arg=değer` şeklinde isimli argümanlar kullanılarak da çağrılabilir. Örnek olarak aşağıdaki fonksiyonu ele alalım:

```
def papağan(voltaj, durum='mefta olmuş', olay='vınlayamaz.', cins='Norveç Mavisi'):
    print("-- Bak arkadaş!", voltaj, "volt elektrik bile verseler", end=' ')
    print("bu papağan bir yere", olay)
    print("-- Ne tatlı bi tüy yumağı değil mi şu", cins)
    print("-- Yahu sen ne diyorsun, bu kuş", durum, "!")
```

Bu fonksiyon bir adet zorunlu argümana (`voltaj`), üç adet de tercihe bağlı argümana sahiptir (`durum`, `olay` ve `cins`). Bu fonksiyon şu yöntemlerden herhangi biri kullanılarak çağrılabilir:

```
papağan(1000) # 1 tane konumlu argüman
papağan(voltaj=1000) # 1 tane isimli argüman
papağan(voltaj=1000000, olay='VINLAYAMAZ.') # 2 tane isimli argüman
papağan(olay='VINLAYAMAZ.', voltaj=1000000) # 2 tane isimli argüman
papağan('bir milyon', 'tahtalı köyü boylamış', 'zıplayamaz.')  # 3 tane konumlu argüman
papağan('bin', durum='nalları dikmiş')  # 1 tane konumlu, 1 tane isimli argüman
```

Ancak aşağıdaki bütün çağrılar geçersizdir:

```
papağan() # zorunlu argüman belirtilmemiş
papağan(voltaj=5.0, 'ölü') # isimli bir argümanın ardından isimsiz bir argüman getirilmiş
papağan(110, voltaj=220) # aynı argüman için birden fazla değer verilmiş
papağan(aktör='John Cleese') # bilinmeyen bir isimli argüman kullanılmış
```

Fonksiyonlar çağrılırken, isimli argümanlar konumlu argümanlardan sonra getirilmelidir. Belirtilen bütün isimli argümanlar, fonksiyonun kabul ettiği argümanlarla eşleşmelidir (örn. `papağan` fonksiyonunda `aktör` geçerli bir argüman değildir). İsimli argümanların sırası önemli değildir. Aynı şey zorunlu argümanlar için de geçerlidir (örn. `papağan(voltaj=1000)` ifadesi de doğrudur). Hiçbir argümana birden fazla değer atanamaz. Bu kısıtlama nedeniyle hata veren bir örnek şöyle olabilir:

```
>>> def fonksiyon(a):
...     pass
...
>>> fonksiyon(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

Fonksiyonun argüman listesinin en sonunda `**isim` biçimli bir parametre varsa, bu parametre, konumlu parametreler dışarıda kalacak şekilde, bütün isimli argümanları bir sözlük içinde toplar (bkz. Eşleştirme Tipleri - dict). Yukarıda bahsettiğimiz şey, ayrıca `*isim` biçimli bir başka parametre çeşidiyle birlikte de kullanılabilir (bunu bir sonraki alt bölümde açıklayacağız). `*isim` parametresi, parametre listesinde normal bir şekilde tanımlanan parametrelerin sağındaki konumlu argümanları bir demet içinde toplar. (`*isim`, `**isim`'den önce getirilmelidir.) Örneğin, eğer şöyle bir fonksiyon tanımlayacak olursak:

```
def peynirci(tür, *argümanlar, **isimli_argümanlar):
    print("-- Acaba sizde", tür, "var mı ?")
    print("-- Maalesef hiç", tür, "kalmadı")
    for arg in argümanlar:
        print(arg)
    print("-" * 40)
    anahtarlar = sorted(isimli_argümanlar.keys())
    for anahtar in anahtarlar:
        print(anahtar, ":", isimli_argümanlar[anahtar])
```

Bu fonksiyon şöyle çağrılabilir:

```
peynirci("Limburger peyniri", "Beyefendi, bu çok cıvık.",
           "Evet, beyefendi, evet, gerçekten, ama GERÇEKTEN, cıvık.",
           dükkan_sahibi="Michael Palin",
           müşteri="John Cleese",
           skeç="Peynir Dükkanı Skeci")
```

Buradan şu çıktıyı alırız:

```
-- Acaba sizde hiç Limburger peyniri var mı?
-- Maalesef, hiç Limburger peyniri kalmadı
Beyefendi, bu çok cıvık.
Evet, beyefendi, evet, gerçekten, ama GERÇEKTEN, cıvık."
----------------------------------------
müşteri: "John Cleese",
dükkan_sahibi: "Michael Palin",
skeç: "Peynir Dükkanı Skeci")
```

İsimli argümanları oluşturan isim listesinin, isimli argüman sözlüğünün `keys()` metodu yardımıyla alfabe sırasına dizildikten sonra ekrana basıldığına dikkat edin. Eğer böyle yapmazsak, argümanların ekrana basılış sırası belirsiz olacaktır.

4.7.3 Sınırsız Argüman Listeleri

Son olarak, en nadir kullanılan seçenek, sınırsız sayıda argümanla çağrılabilecek fonksiyonlar tanımlamaktır. Bu argümanlar bir demet içine yerleştirilecektir (bkz. *Demetler ve Diziler*). Sanırsız sayıdaki argümanları tanımlamadan önce başka hiç bir argüman belirtmeyebileceğiniz gibi, bir veya daha fazla sayıda normal argüman da belirtebilirsiniz.

```
def birden_fazla_öğeyi_yazdır(dosya, ayraç, *argümanlar):
    dosya.write(ayraç.join(argümanlar))
```

Normalde bu `değişken sayılı` argümanlar, fonksiyona verilen bütün sahipsiz argümanları toparladığı için normal parametre listesinin sonunda yer alır. `*argümanlar` parametresinden sonra, 'yalnızca isimleriyle çağrılabilecek' argümanlar getirebilirsiniz, yani bu parametreden

sonra getireceğiniz argümanları konumlu argüman olarak değil, sadece isimli argüman olarak kullanabilirsiniz.

```
>>> def birleştir(*argümanlar, ayraç="/"):
...     return ayraç.join(argümanlar)
...
>>> birleştir("dünya", "mars", "venüs")
'dünya/mars/venüs'
>>> birleştir("dünya", "mars", "venüs", ayraç=".")
'dünya.mars.venüs'
```

4.7.4 Argüman Listelerinin Çözülmesi

Tersi bir durum, argümanların halihazırda bir liste veya demet içinde bulunduğu, ama bunların, konumlu argümanların ayrı ayrı bulunmasını gerektiren bir fonksiyon çağrısında kullanılmak üzere çözülmesi gerektiği durumlarda ortaya çıkar. Örneğin, gömülü `range()` fonksiyonu ayrı birer *başlangıç* ve *bitiş* argümanı bekler. Eğer bunlar ayrı olarak mevcut değilse, argümanları liste veya demetten çıkarmak için fonksiyonunuzu `*` işlecini kullanarak yazabilirsiniz:

```
>>> list(range(3, 6)) # fonksiyonu normalde ayrı argümanlarla böyle çağırıyoruz
[3, 4, 5]
>>> argümanlar = [3, 6]
>>> list(range(*argümanlar)) # argüman listesini çözüp fonksiyonu çağırıyoruz
[3, 4, 5]
```

Aynı şekilde sözlüklerin de `**` işleci yardımıyla isimli argümanları sunması sağlanabilir:

```
>>> def papağan(voltaj, durum='mefta olmuş', olay='vınlayamaz.'):
...     print("-- ", voltaj, "volt verseler", end=' ')
...     print("bu papağan bir yere", olay)
...     print("Bu kuş", durum, "!")
...
>>> d = {"voltaj": "Dört milyon", "durum": "kuyruğu titretmiş", "olay": "VINLAYAMAZ."}
>>> papağan(**d)
-- Dört milyon volt verseler bu papağan bir yere VINLAYAMAZ. Bu kuş kuyruğu titretmiş !
```

4.7.5 Lambda İfadeleri

`lambda` sözcüğü kullanılarak küçük, isimsiz fonksiyonlar oluşturulabilir. Mesela şu fonksiyon iki argümanın toplamını döndürür: `lambda a,b:a+b`. Lambda fonksiyonları, fonksiyon nesnelerinin gerektiği her yerde kullanılabilir. Bunlar söz dizimleri gereği tek bir ifade ile sınırlıdır. Anlam bakımından ise, normal bir fonksiyon tanımı yerine kullanılabilecek bir söz dizimi çeşnisi olmaktan ibaretlerdir. İç içe fonksiyon tanımlarında olduğu gibi, lambda fonksiyonları da, içinde bulundukları etki alanındaki değişkenlere atıfta bulunabilir:

```
>>> def artır(n)
...     return lambda x: x + n
...
>>> f = artır(42)
>>> f(0)
42
>>> f(1)
43
```

Yukarıdaki örnekte, fonksiyon döndürmek için lambda ifadelerinden yararlanıyoruz. Küçük bir fonksiyonu argüman olarak kullanmak gerektiğinde de lambda ifadelerinden yararlanabilirsiniz:

```
>>> çiftler = [(1, 'bir'), (2, 'iki'), (3, 'üç'), (4, 'dört')]
>>> çiftler.sort(key=lambda çift: çift[1])
>>> çiftler
[(1, 'bir'), (4, 'dört'), (2, 'iki'), (3, 'üç')]
```

4.7.6 Belgelendirme Dizileri

Belgelendirme dizilerinin içeriği ve biçimine ilişkin bazı yerleşmiş kurallardan söz edelim şimdi de.

Kodlarınızın ilk satırı her zaman, nesnenin amacına dair kısa ve net bir özet içermelidir. Daha kısa olması için, (eğer isim, fonksiyonun işleyişini tarif eden bir fiil değilse) burada nesnenin adı veya tipi açıkça belirtilmemelidir, zira bunlara başka vasıtalarla da erişebiliyoruz. Bu satır büyük harfle başlamalı ve nokta ile sona ermelidir.

Eğer belgelendirme dizisinde birden fazla satır varsa ikinci satır boş bırakılmalı, böylece özet kısmı, açıklamanın geri kalanından görüntü itibariyle ayrılmalıdır. Sonraki satırlar, nesnenin çağrılmasına ilişkin kaideleri, yan etkilerini, vb. tarif eden bir veya daha fazla paragraftan oluşmalıdır.

Python'ın ayrıştırıcısı (parser), birden fazla satırdan oluşan karakter dizilerinden girintileri çıkarmaz, bu yüzden belgelendirme dizilerini işleyen araçlar, eğer gerekiyorsa bu girintileri kendileri çıkarmalıdır. Bu işlem aşağıdaki kaideye göre gerçekleştirilir. Karakter dizisinin ilk satırından *sonra* gelen ilk boş olmayan satır, bütün belgelendirme dizisinin girintileme miktarını belirler. (İlk satırı kullanamayız, çünkü bu satır genellikle karakter dizisinin açılış tırnaklarına bitişiktir. Bu yüzden bunun girintileme yapısı karakter dizisine bakılarak anlaşılamaz.) Bu girintilemenin boşluk "eşdeğeri", karakter dizisinin bütün satırlarının başlangıcından itibaren çıkarılıp atılır. Daha az girintilenmiş satır olmamalıdır, ama eğer olursa bunların önündeki bütün boşluklar çıkarılmalıdır. Sekmelerin genişletilmesinin ardından, boşlukların birbirine eşit olup olmadığı test edilmelidir (normalde 8 vuruşluk boşluk temel alınır).

Aşağıda birden fazla satırdan oluşan bir belge dizisi örneği görüyorsunuz:

```
>>> def fonksiyonum():
...     """Hiçbir şey yapmıyoruz, sadece belgelendiriyoruz..
...
...     Yo, yo, gerçekten hiçbir şey yapmıyoruz.
...     """
...     pass
...
>>> print(fonksiyonum.__doc__)
Hiçbir şey yapmıyoruz, sadece belgelendiriyoruz..

    Yo, yo, gerçekten hiçbir şey yapmıyoruz.
```

4.7.7 Fonksiyon Açıklamaları

Fonksiyon açıklamaları, kulllanıcı tarafından tanımlanan fonksiyonlar tarafından kullanılan, tamamen isteğe bağlı, zorunlu olmayan metaveri bilgileridir (daha fazla bilgi için bkz. **PEP**

484).

Açıklamalar, fonksiyonun `__annotations__` niteliği içinde bir sözlük olarak tutulur ve fonksiyonun başka kısımları üzerinde herhangi bir etkiye sahip değildir. Parametre açıklamaları, parametre adının ardından getirilen iki nokta üst üste işaretinden sonra, açıklamanın değerini işleten bir ifade şeklinde tanımlanır. Dönüş açıklamaları `->` işaretinin ardından, parametre listesi ile `def` deyiminin bitişini gösteren iki nokta üst üste işareti arasına getirilen bir ifade şeklinde tanımlanır. Aşağıdaki örnekte bir konumlu argüman, bir isimli argüman ve açıklamalı bir dönüş değeri bulunmaktadır:

```
>>> def f(jambon: str, yumurtalar: str = 'yumurtalar') -> str:
...     print("Açıklamalar:", f.__annotations__)
...     print("Argümanlar:", jambon, yumurtalar)
...     return jambon + ' ve ' + yumurtalar
...
>>> f('sucuk')
Açıklamalar: {'jambon': <class 'str'>, 'dönüş değeri': <class 'str'>, 'yumurtalar': <class 'str'>}
Argümanlar: sucuk yumurtalar
'sucuk ve yumurtalar'
```

4.8 Perde Arası: Kodlama Tarzı

Daha uzun, daha karmaşık Python kodları yazmaya hazır hale gelmek üzere olduğunuza göre, *kodlama tarzından* bahsetmenin şimdi tam sırası. Çoğu dil farklı tarzlar kullanılarak yazılabilir (veya daha özlü bir ifadeyle, *biçimlendirilebilir*). Ancak bu tarzların bazıları diğerlerinden daha okunaklıdır. Kodlarınızı başkaları için daha okunaklı hale getirmeye çalışmak her zaman doğru bir düşünce şeklidir. Güzel bir kodlama tarzı benimsemek bu konuda size epey yararlı olacaktır.

Python için, **PEP 8** pek çok proje tarafından benimsenen bir üslup kılavuzu halini almıştır. Bu kılavuzca teşvik edilen kodlama tarzı bir hayli okunaklı ve göze hoş görünen bir tarzdır. Her Python geliştiricisi bir noktada bu kılavuzu okumalıdır. Burada biz sizin için bu kılavuzun önemli noktalarını ayıkladık:

- 4 boşlukluk girintileme kullanın, sekme kullanmaktan kaçının.

  4 boşluk, az girinti (daha fazla iç içe geçmiş yapılar üretilmesini kolaylaştırır) ile fazla girinti (daha okunaklıdır) arasında makul bir orta yoldur. Sekmeler kafa karışıklığına yol açtığı için en iyisi sekmelerden kaçınmaktır.

- Satırlar 79 karakteri geçmeyecek şekilde ayarlanmalıdır.

  Bu, küçük ekranlarda çalışan kullanıcılar açısından faydalıdır ve geniş ekranlarda da birden fazla kod dosyasının yan yana açılabilmesini olanaklı kılar.

- Fonksiyonlar ile sınıfları ve fonksiyonlar içindeki büyük kod bloklarını ayırmak için boş satırlardan yararlanın.

- Mümkünse yorumlarınızı ayrı satırlarda gösterin.

- Belge dizilerini kullanın.

- İşleçler arasında ve virgüllerden sonra boşluk koyun, ama parantezli yapıların içinde boşluk kullanmayın: `a = f(1, 2) + g(3, 4)`.

- Sınıflarınızı ve fonksiyonlarınızı tutarlı bir şekilde isimlendirin. Kabul görmüş uygulama, sınıf adlarının `İlkHarfleriniBüyükYazarken`, fonksiyon ve metot adlarının bütün harflerini küçük yazmak ve `kelimeler_arasına_alt_çizgi_yerleştirmektir`. Metotların ilk argümanı için her zaman `self` adını kullanın (sınıflar ve metotlar hakkında daha fazla bilgi için bkz. *Sınıflara İlk Bakış*).
- Eğer yazdığınız kodların uluslararası ortamlarda kullanılmasını hedefliyorsanız nadir karakter kodlamalarını kullanmayın. Python'ın öntanımlı karakter kodlaması olan UTF-8 ve hatta saf ASCII her durumda en iyi sonuçları verecektir.
- Aynı şekilde, farklı bir dil konuşan insanların, kodlarınızı en ufak bir okuma veya düzenleme ihtimali varsa tanım adlarında ASCII dışı karakterler kullanmayın.

BÖLÜM 5

Veri Yapıları

Bu bölümde hem önceden öğrenmiş olduğumuz bazı konuları daha ayrıntılı bir şekilde ele alacağız, hem de bunlara yeni birtakım bilgiler ilave edeceğiz.

5.1 Listeler Hakkında Daha Fazlası

Liste veri tipinin, önceki bölümlerde gördüklerimiz dışında başka metotları da bulunur. Liste nesnelerinin bütün metotlarını şöyle sıralayabiliriz:

`liste.append(`*x*`)`
: Listenin en sonuna öğe ekler. Şununla aynı işi yapar: `a[len(a):] = [x]`.

`liste.extend(`*L*`)`
: Kendisine parametre olarak verilen bir listedeki bütün öğeleri ilk listeye ekleyerek o listeyi genişletir. Şununla aynı işi yapar: `a[len(a):] = L`.

`liste.insert(`*i*, *x*`)`
: Bir öğeyi, belirtilen konuma yerleştirir. İlk argüman, önüne yeni bir öğe yerleştirilecek liste öğesinin sırasını gösterir. Dolayısıyla `a.insert(0, x)` komutu öğeyi listenin en başına yerleştirir. `a.insert(len(a), x)` komutu da `a.append(x)` ile aynı işi yapar.

`liste.remove(`*x*`)`
: Listede geçen, *x* değerine sahip ilk öğeyi listeden çıkarır. Eğer bu değere sahip bir öğe yoksa bir hata mesajı gösterilir.

`liste.pop(`[*i*]`)`
: Belirtilen konumdaki öğeyi listeden çıkarır ve çıkarılan bu öğeyi döndürür. Eğer herhangi bir sıra numarası belirtilmezse, `a.pop()` komutu listenin son öğesini listeden çıkaracak ve bu öğeyi döndürecektir. (Metodun formülündeki *i* harfini içine alan köşeli parantezler ilgili parametrenin tercihe bağlı olduğunu gösterir, oraya sizin de köşeli parantez koymanız gerektiğini değil. Bu gösterimle Python Kütüphane Referansı'nda sıklıkla karşılaşacaksınız.)

`liste.clear()`
: Listedeki bütün öğeleri listeden çıkarır. Şuna eşdeğerdir: `del a[:]`.

`liste.index(`*x*`)`
: Listede, *x* değerine sahip ilk öğenin sırasını döndürür. Eğer bu değere sahip bir öğe yoksa bir hata mesajı gösterilir.

`liste.count(`*x*`)`
: *x* değerinin listede kaç kez geçtiğini döndürür.

`list.sort(`*key=None*, *reverse=False*`)`
Liste öğelerini, aynı liste üzerinde alfabe sırasına dizer (argümanları kullanarak, sıralama ölçütlerini belirleyebilirsiniz. Argümanların açıklamaları için `sorted()` fonksiyonunu inceleyebilirsiniz).

`liste.reverse()`
Liste öğelerinin sırasını, aynı liste üzerinde ters çevirir.

`liste.copy()`
Listenin sığ kopyasını döndürür. Şuna eşdeğerdir: `a[:]`.

Liste metotlarının çoğunu bir arada gösteren bir örnek verelim:

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
>>> a.pop()
1234.5
>>> a
[-1, 1, 66.25, 333, 333]
```

`insert`, `remove` veya `sort` gibi, liste üzerinde sadece değişiklik yapan metotların ekrana herhangi bir dönüş değeri basmadığını farketmişsinizdir. Bu metotlar öntanımlı `None` değerini döndürür[1]. Python'daki bütün değiştirilebilir veri yapıları için bir tasarım ilkesidir bu.

5.1.1 Listelerin Yığın Olarak Kullanılması

Liste metotları sayesinde listeler, eklenen son öğenin erişilen ilk öğe olduğu ('son giren ilk çıkar') birer yığın olarak kolayca kullanılabilir. Yığının en üstüne bir öğe eklemek için `append()` metodunu kullanıyoruz. Yığının en üstündeki öğeye erişmek için ise, `pop()` metodunu öğe sırasını açıkça belirtmeden kullanıyoruz. Mesela:

```
>>> yığın = [3, 4, 5]
>>> yığın.append(6)
>>> yığın.append(7)
>>> yığın
[3, 4, 5, 6, 7]
>>> yığın.pop()
7
```

[1] Başka dillerde, değişikliğe uğrayan nesne de döndürülüyor olabilir, böylece `d->insert("a")->remove("b")->sort();` şeklinde metotlar birbirine zincirlenebilir.

```
>>> yığın
[3, 4, 5, 6]
>>> yığın.pop()
6
>>> yığın.pop()
5
>>> yığın
[3, 4]
```

5.1.2 Listelerin Kuyruk Olarak Kullanılması

Listeleri, eklenen ilk öğenin erişilen ilk öğe olduğu ('ilk giren ilk çıkar') bir kuyruk olarak kullanmak da mümkün olsa bile, listeler bu amaç için pek verimli değildir. Listenin sonuna öğe ekleme ve listenin sonundan öğe atma işlemleri hızlı olsa da, listenin başına öğe yerleştirme veya listenin başından öğe atma işlemleri yavaştır (çünkü bu durumda öteki bütün öğelerin bir sıra kaydırılması gerekir).

Bir kuyruk oluşturmak için, `collections.deque` metodu kullanılabilir. Bu metot, listelerin her iki tarafında hızlı bir şekilde öğe ekleme ve atma işlemleri yapılmasını sağlayacak şekilde tasarlanmıştır. Mesela:

```
>>> from collections import deque
>>> kuyruk = deque(["Eric", "John", "Michael"])
>>> kuyruk.append("Terry")           # Terry geldi
>>> kuyruk.append("Graham")          # Graham geldi
>>> kuyruk.popleft()                 # Şimdi ilk gelen öğe gidiyor
'Eric'
>>> kuyruk.popleft()                 # Şimdi de ikinci gelen öğe gidiyor
'John'
>>> kuyruk                           # Geliş sıralarına göre kuyruğun geri kalan öğeleri
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3 Liste Üreticiler

Liste üreticileri kullanarak kısa yoldan listeler oluşturabiliriz. Liste üreticilerin yaygın kullanıldığı uygulama alanı, her öğesinin, başka bir dizinin veya üzerinde yürünebilir nesnenin her bir öğesine uygulanan bazı işlemlerin sonucu olduğu yeni listeler üretmek veya belirli bir koşulu karşılayan öğelerden meydana gelen bir altdizi oluşturmaktır.

Örneğin, sayıların karesinden oluşan bir liste oluşturmak istediğimizi varsayalım:

```
>>> kareler = []
>>> for x in range(10):
...     kareler.append(x**2)
...
>>> kareler
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Bu işlemin, döngü tamamlandıktan sonra da varlığını sürdüren `x` adlı bir değişken oluşturduğuna (veya bu adı taşıyan bir değişkenin üzerine yazdığına) dikkat edin. Karelerin listesini, herhangi bir olumsuz etkiye yol açmayacak şekilde şöyle hesaplayabiliriz:

```
kareler = list(map(lambda x: x**2, range(10)))
```

Bunun eşdeğeri şudur:

```
kareler = [x**2 for x in range(10)]
```

Son yazdığımız kod daha kısa ve okunaklıdır.

Liste üreticilerde köşeli parantez içinde bir ifade, bunun ardından bir `for` cümleciği, sonra da arzu edilen sayıda `for` ya da `if` cümleciği bulunur. Sonuç; ifadenin, kendisini takip eden `for` ve `if` cümleciklerinin gerektirdiği şekilde işletilmesinin ardından ortaya çıkan yeni bir liste olacaktır. Örneğin aşağıdaki liste üretici, iki listenin aynı olmayan öğelerini bir araya getiriyor:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

bu kodlar şuna eşdeğerdir:

```
>>> bileşimler = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             bileşimler.append((x, y))
...
>>> bileşimler
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

`for` ve `if` deyimlerinin her iki kod parçasında da aynı sırada yer aldığına dikkat edin.

Eğer ifade bir demet ise (mesela bir önceki örnekteki `(x, y)`), bunun parantez içinde gösterilmesi gerekir.

```
>>> vek = [-4, -2, 0, 2, 4]
>>> # değerlerin iki katından oluşan yeni bir liste oluşturalım.
>>> [x*2 for x in vek]
[-8, -4, 0, 4, 8]
>>> # listeyi süzerek eksi değerli sayıları atalım
>>> [x for x in vek if x >= 0]
[0, 2, 4]
>>> # bütün öğelere bir fonksiyon uygulayalım
>>> [abs(x) for x in vek]
[4, 2, 0, 2, 4]
>>> # her bir öğe üzerine bir metot uygulayalım
>>> tazemeyveler = ['  muz', '  ahududu ', 'böğürtlen  ']
>>> [meyve.strip() for meyve in tazemeyveler]
 ['muz', 'ahududu ', 'böğürtlen']
>>> # (sayı, karesi) şeklinde 2 öğeli demetlerden oluşan bir liste üretelim
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # demetin parantez içinde gösterilmesi gerekir, aksi halde hata alırız
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1, in ?
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # iki adet 'for' içeren bir liste üretici kullanarak listeyi düzleştirelim
>>> vek = [[1,2,3], [4,5,6], [7,8,9]]
>>> [sayı for öğe in vec for sayı in öğe]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Liste üreticiler karmaşık ifadeler ve iç içe geçmiş fonksiyonlar da içerebilir:

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.1.4 İç İçe Liste Üreticiler

Bir liste üretici içindeki ilk ifade, başka bir liste üretici de dahil olmak üzere herhangi bir ifadeden oluşuyor olabilir.

Her birinin uzunluğu 4 olan 3 adet listeden oluşacak şekilde yazılmış şu 3x4 matris örneğine bir bakalım:

```
>>> matris = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

Aşağıdaki liste üretici, matrisin satır ve sütunlarını transpoze edecektir:

```
>>> [[satır[i] for satır in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Önceki bölümde gördüğümüz gibi, iç içe geçmiş liste üreticiler, kendisinden sonra gelen `for` döngüsünün gerektirdiği şekilde işletilmektedir. Dolayısıyla bu örnek şuna eşdeğerdir:

```
>>> transpoze = []
>>> for i in range(4):
...     transpoze.append([satır[i] for satır in matris])
...
>>> transpoze
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Aynı şekilde yukarıdaki kod şununla da eşdeğerdir:

```
>>> transpoze = []
>>> for i in range(4):
...     # aşağıdaki 3 satırda iç içe liste üreticiler oluşturulmuştur
...     transpoze_satır = []
...     for satır in matris:
...         transpoze_satır.append(satır[i])
...     transpoze.append(transpoze_satır)
...
>>> transpoze
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Gerçek programlarda, karmaşık akış deyimleri yerine gömülü fonksiyonları tercih etmelisiniz. Bu tür durumlarda `zip()` fonksiyonu epey işinize yarar:

```
>>> list(zip(*matris))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

Bu satırdaki yıldıza ilişkin ayrıntılar için bkz. *Argüman Listelerinin Çözülmesi*.

5.2 `del` Deyimi

Bir listeden öğe silmek için o öğenin değerini belirtmek yerine sırasını belirtebilmenizin de bir yöntemi var: `del` deyimini kullanmak. Bu deyim, sildiği öğeyi döndüren `pop()` metodundan farklıdır. `del` deyimi ayrıca (daha önce listenin bir parçasını boş bir listeye atayarak yaptığımız gibi) listelerin bir parçasını veya tamamını silmek için de kullanılabilir. Mesela:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` deyimi, değişkenleri tamamen ortadan kaldırmak için de kullanılabilir:

```
>>> del a
```

Bu andan sonra `a` ismine atıfta bulunursanız hata mesajı alırsınız (en azından bu isme başka bir değer atanana kadar). İleride `del` deyiminin başka kullanım alanlarını da göreceğiz.

5.3 Demetler ve Diziler

Listelerin ve karakter dizilerinin, indeksleme ve dilimleme gibi pek çok ortak yönünün olduğunu görmüştük. Bunlar *dizi* adlı veri tipine iki örnektir (bkz. Dizi Tipleri - liste, demet, aralık). Python evrilmeye devam eden bir dil olduğu için, bu dile ilerde dizi nitelikli başka veri tipleri de ilave edilebilir. Bunların dışında dizi nitelikli başka bir standart veri tipi daha bulunur. Bu veri tipinin adı *demet* tir.

Demetler, birbirinden virgül ile ayrılmış birtakım değerlerden oluşur. Örneğin:

```
>>> t = 12345, 54321, 'selam!'
>>> t[0]
12345
>>> t
(12345, 54321, 'selam!')
>>> # Demetler iç içe geçirilebilir:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'selam!'), (1, 2, 3, 4, 5))
>>> # Demetler değiştirilemez:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # ama bunlar değiştirilebilen nesneler içerebilir::
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Gördüğünüz gibi, demetlerin çıktıları her zaman parantez içinde gösteriliyor. Böylece iç içe geçmiş demetler doğru bir şekilde görüntülenebiliyor. Demetler parantez içinde yazılabileceği gibi, parantezsiz olarak da yazılabilir, ancak (özellikle demet daha büyük bir ifadenin parçasıyla) çoğu durumda parantezleri kullanmanız gerekecektir. Bir demetin belirli bir öğesine değer atamak mümkün olmasa da, listeler gibi değiştirilebilen nesneler içeren demetler oluşturabilirsiniz.

Demetler listelere benzer gibi görünse de, bu ikisi genellikle farklı durumlarda ve birbirlerinden farklı amaçlarla kullanılır. Demetler değiştirilemeyen nesnelerdir. Bunlar genellikle birbirinden farklı tiplerde öğeyi dizi halinde içerir. Demetlerin öğelerine demet çözme (bu bölümün ilerleyen sayfalarına bakınız) veya indeksleme aracılığıyla (ya da `namedtuples`'ta olduğu gibi nitelikler aracılığıyla) erişilebilir. Listeler değiştirilebilen nesnelerdir. Bunların öğeleri genellikle aynı tiptedir ve bu öğelere liste üzerinde yürünerek erişilir.

Öğe içermeyen veya tek bir öğe içeren demetlerin oluşturulmasında özel bir durum söz konusudur. Bunun için söz dizimine bazı enteresan özellikler eklenmiştir. Boş demetler, bir çift boş parantez yardımıyla oluşturulabilir. Tek öğeli bir demet ise bu öğenin yanına bir virgül getirilerek oluşturulur (tek bir değeri parantez içine almak yeterli değildir). Bu söz dizimi çirkin olsa da pratiktir. Mesela:

```
>>> boş = ()
>>> tek_öğeli = 'merhaba',    # <-- sondaki virgüle dikkat
>>> len(boş)
0
>>> len(tek_öğeli)
1
>>> tek_öğeli
('merhaba',)
```

`t = 12345, 54321, 'selam!'` ifadesi *demetlemeye* bir örnektir. Yani burada `12345`, `54321` ve `selam!` değerleri bir demet içinde bir araya getirilmiş. Bu işlemin tersini yapmak da mümkündür:

```
>>> x, y, z = t
```

Buna, anlamına uygun bir şekilde, *dizi çözme* denir ve eşitliğin sağ tarafında yer alan her türlü diziye uygulanabilir. Dizi çözme işlemi, eşittir işaretinin sol tarafında, dizi içindeki öğe sayısı kadar değişken olmasını gerektirir. Esasında çoklu değer atama dediğimiz şey, demetleme ve dizi çözme işlemlerinin birleşiminden ibarettir.

5.4 Kümeler

Python'da *kümeler* için de ayrı bir veri tipi bulunur. Kümeler, tekrar eden öğeler içermeyen, sırasız bir yığındır. Kümeler temel olarak üyelik kontrolü ve tekrar eden girdilerin elenmesi için kullanılır. Küme nesneleri ayrıca, birleşim, kesişim, fark ve simetrik fark gibi matematik işlemlerini de destekler.

Küme oluşturmak için süslü parantezler veya `set()` fonksiyonu kullanılabilir. Not: Boş bir küme oluşturabilmek için `{}` işaretini değil `set()` fonksiyonunu kullanmalısınız. `{}` işareti boş bir sözlük oluşturur. Sözlük veri tipini bir sonraki bölümde ele alacağız.

İşte size kısa birkaç örnek:

```
>>> sepet = {'elma', 'portakal', 'elma', 'armut', 'portakal', 'muz'}
>>> print(sepet)                        # tekrarlayan öğelerin çıkarıldığını görüyorsunuz
{'portakal', 'muz', 'armut', 'elma'}
>>> 'portakal' in sepet                 # hızlıca üyelik kontrolü
True
>>> 'çatal otu' in sepet
False

>>> # İki farklı kelime içindeki, tekrar etmeyen harfler üzerinde küme işlemleri yapalım
...
>>> a = set('abrakadabra')
>>> b = set('alakazam')
>>> a                                   # a'daki her bir harf bir kez
{'a', 'r', 'b', 'k', 'd'}
>>> a - b                               # a'da olan, b'de olmayan harfler
{'r', 'd', 'b'}
>>> a | b                               # a'da veya b'de bulunan harfler
{'a', 'k', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                               # a'da ve b'de bulunan harfler
{'a', 'k'}
>>> a ^ b                               # a ve b'de ortak olmayan harfler
{'r', 'd', 'b', 'm', 'z', 'l'}
```

liste üreticiler ile benzer şekilde, küme üreticiler de bulunur:

```
>>> a = {x for x in 'abrakadabra' if x not in 'abk'}
>>> a
{'r', 'd'}
```

5.5 Sözlükler

Python'ın bünyesinde bulunan, kullanışlı veri tiplerinden bir tanesi de *sözlüklerdir* (bkz. Eşleşme Tipleri - sözlük). Sözlükler bazı başka dillerde 'ilişkili bellekler' ('associative memories') veya 'ilişkili diziler' ('associative arrays') adıyla da geçebilir. Öğelerine belli bir sayı aralığına göre erişilen dizilerin aksine, sözlüklerin öğelerine *anahtarları* yardımıyla erişilir. Bu anahtarlar değiştirilemeyen veri tiplerinden herhangi birisi olabilir. Karakter dizileri ve sayılar her zaman için anahtar olarak kullanılabilir. Hatta, yalnızca karakter dizileri, sayılar veya demetlerden oluşmak kaydıyla demetler de anahtar olarak kullanılabilir. Ama eğer bir demet, doğrudan veya dolaylı olarak herhangi bir değiştirilebilir nesne içeriyorsa, bu durumda bu demet bir anahtar olarak kullanılamaz. Listeler ise anahtar olarak kullanılamaz, çünkü listeler üzerinde, öğe sırası ile değer atama, dilime değer atama veya `append()` ve `extend()` gibi metotlar yoluyla doğrudan değişiklik yapılabilir.

Sözlükler *anahtar:değer* çiftlerinden oluşan sırasız bir veri tipi olarak düşünülebilir. Bu veri tipinde (aynı sözlük içindeki) anahtarlar benzersiz olmalıdır. Bir çift süslü parantez yardımıyla boş bir sözlük oluşturabiliriz: `{}`. Süslü parantezler arasına, birbirlerinden virgülle ayrılmış anahtar:değer çiftlerinden oluşan bir liste yerleştirerek, ilk anahtar:değer çiftlerini sözlüğünüze ilave edebilirsiniz. Zaten sözlükler de çıktıda böyle görünür.

Bir sözlük üzerinde yapılabilecek başlıca işlemler bir anahtar içinde değer depolamak ve anahtarı vererek bu değeri almaktır. Ayrıca `del` deyimini kullanarak bir anahtar:değer çiftini silmek de mümkündür. Halihazırda kullanımda olan bir anahtara değer atarsanız, anahtarın eski değeri kaybolacaktır. Sözlükte varolmayan bir anahtar kullanarak değer almaya çalışmak hataya yol açacaktır.

Sözlükler üzerinde `list(d.keys())` işleminin uygulanması, sözlük içinde geçen bütün anahtarları, rastgele bir sıra ile listeleyecektir (eğer anahtarların sıralı olarak gelmesini isterseniz `sorted(d.keys())` komutunu kullanabilirsiniz)[2]. Eğer tek bir anahtarın sözlükte olup olmadığını kontrol etmek istiyorsanız, `in` sözcüğünü kullanın.

Sözlükleri kullanarak şöyle ufak bir örnek verelim:

```
>>> tel = {'ahmet': 4098, 'mehmet': 4139}
>>> tel['selin'] = 4127
>>> tel
{'mehmet': 4139, 'selin': 4127, 'ahmet': 4098}
>>> tel['ahmet']
4098
>>> del tel['mehmet']
>>> tel['ayşe'] = 4127
>>> tel
{'selin': 4127, 'ayşe': 4127, 'ahmet': 4098}
>>> list(tel.keys())
['ayşe', 'selin', 'ahmet']
>>> sorted(tel.keys())
['ahmet', 'ayşe', 'selin']
>>> 'selin' in tel
True
>>> 'ahmet' not in tel
False
```

`dict()` adlı inşa fonksiyonunu kullanarak, anahtar-değer çiftlerinden oluşan dizilerden doğrudan sözlükler meydana getirebilirsiniz.

```
>>> dict([('mehmet', 4139), ('selin', 4127), ('ahmet', 4098)])
{'mehmet': 4139, 'ahmet': 4098, 'selin': 4127}
```

Ayrıca rastgele anahtar ve değer deyimlerinden sözlükler oluşturmak için sözlük üreticiler de kullanılabilir:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Eğer anahtarlar basit karakter dizileri ise, anahtar-değer çiftlerini isimli argümanlardan faydalanarak belirlemek kimi zaman daha kolay olabilir:

```
>>> dict(mehmet=4139, selin=4127, ahmet=4098)
{'mehmet': 4139, 'ahmet': 4098, 'selin': 4127}
```

5.6 Döngü Kurma Teknikleri

Sözlükler üzerinde döngü kurulurken, `items()` metodu yardımıyla aynı anda hem anahtar hem de buna karşılık gelen değer elde edilebilir.

```
>>> padişahlar = {"süleyman": "kanuni", "mehmet": "fatih"}
>>> for k, v in padişahlar.items():
...     print(v, "sultan", k)
```

[2] `d.keys()` komutu, *sözlük görünümü* [dictionary view] adlı bir nesne döndürür. Bu nesne, üyelik testi ve yürüme gibi işlemleri destekler, ancak bunun içeriği orijinal sözlükten bağımsız değildir; bu yalnızca bir *görüntüden* ibarettir.

```
...
kanuni sultan süleyman
fatih sultan mehmet
```

Bir dizi üzerinde döngü kurulurken, `enumerate()` fonksiyonu yardımıyla aynı anda hem öğe sırası hem de buna karşılık gelen değer elde edilebilir.

```
>>> for i, v in enumerate(['taş', 'kağıt', 'makas']):
...     print(i, v)
...
0 taş
1 kağıt
2 makas
```

Aynı anda iki veya daha fazla dizi üzerinde döngü kurmak için, `zip()` fonksiyonu kullanılarak girdiler birbirleri ile eşleştirilebilir.

```
>>> sorular = ['isminiz', 'isteğiniz', 'sevdiğiniz renk']
>>> cevaplar = ['Fırat', 'huzur', 'mavi']
>>> for s, c in zip(sorular, cevaplar):
...     print('{0} nedir?  Cevabım {1}.'.format(s, c))
...
isminiz nedir?  Cevabım Fırat.
İsteğiniz nedir?  Cevabım huzur.
Sevdiğiniz renk nedir?  Cevabım mavi.
```

Bir dizinin tersi üzerinde döngü kurabilmek için, bu dizi önce düz olarak belirtilir ve ardından `reversed()` fonksiyonu çağrılır.

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

Bir dizi üzerinde sıralı bir düzende döngü kurabilmek için `sorted()` fonksiyonunu kullanıyoruz. Bu fonksiyon, kaynak dizi üzerinde herhangi bir değişiklik yapmadan, yeni, sıralı bir liste döndürür.

```
>>> sepet = {'elma', 'portakal', 'elma', 'armut', 'portakal', 'muz'}
>>> for f in sorted(set(sepet)):
...     print(f)
...
armut
elma
muz
portakal
```

Bir listeyi, üzerinde döngü kurduğunuz esnada değiştirmek bazen daha cazipmiş gibi gelebilir; ancak bunun yerine yeni bir liste oluşturmak çoğu zaman hem daha basittir hem de daha güvenlidir.

```
>>> import math
>>> ham_veriler = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> süzülmüş_veriler = []
```

```
>>> for değer in ham_veriler:
...     if not math.isnan(değer):
...         süzülmüş_veriler.append(değer)
...
>>> süzülmüş_veriler
[56.2, 51.7, 55.3, 52.5, 47.8]
```

5.7 Koşullu Durumlar Hakkında Daha Fazlası

`while` ve `if` deyimlerinde kullanılan koşul yapılarında sadece karşılaştırmalar değil işleçler de yer alabilir.

`in` ve `not in` adlı karşılaştırma işleçleri bir değerin bir dizi içinde geçip geçmediğini kontrol eder. `is` ve `is not` adlı işleçler ise iki nesnenin gerçekten aynı nesne olup olmadığını kontrol eder. Bu durum yalnızca, listeler gibi değiştirilebilen nesneler için bir anlam taşır. Bütün karşılaştırma işleçleri aynı önceliğe sahiptir. Yani bunlar bütün sayısal işleçlerden düşük önceliktedir.

Karşılaştırma ifadeleri zincirleme kullanılabilir. Örneğin, `a < b == c` komutu `a`'nın `b`'den küçük olup olmadığını, `b`'nin de `c`'ye eşit olup olmadığını sorgular.

Karşılaştırma ifadeleri `and` ve `or` adlı Bool işleçleri kullanılarak birleştirilebilir. Ayrıca bir karşılaştırmanın (veya herhangi bir başka Bool ifadesinin) çıktısı `not` yardımıyla olumsuzlanabilir. Bunlar, karşılaştırma işleçlerine göre daha düşük önceliğe sahiptir. Bunlar arasında `not` işleci en yüksek önceliğe, `or` işleci ise en düşük önceliğe sahiptir. Dolayısıyla `A and not B or C` ifadesi `A (and (not B)) or C` ifadesine eştir. Her zaman olduğu gibi, istenen dizilimi elde etmek için parantezlerden yararlanılabilir.

`and` ve `or` adlı Bool işleçlerine *kısa devre* işleçleri de denir. Bunların argümanları soldan sağa doğru işlenir ve çıktı belirlenir belirlenmez işlem durur. Örneğin eğer `A` ve `C` true, ama `B` false ise, `A and B and C` ifadesinde `C` kısmı işleme alınmaz. Bir Bool olarak değil de genel bir değer olarak kullanıldığında kısa devre işleçlerinin dönüş değeri en son işleme alınan argüman olacaktır.

Bir karşılaştırmanın veya başka bir Bool ifadesinin sonucunu bir değişkene atamak da mümkündür. Örneğin:

```
>>> kardiz1, kardiz2, kardiz3 = '', 'Trakya', 'Kıvırtma Havası'
>>> boş_olmayan_değer = kardiz1 or kardiz2 or kardiz3
>>> boş_olmayan_değer
'Trakya'
```

C'nin aksine, Python'da ifade ortasında atama yapılamayacağına dikkat edin. C programcıları bu özelliğe dudak bükebilir, ama aslında bu, C programlarında sıklıkla karşılaşılan bir sorun tipinin ortaya çıkmasını engellemektedir. Yani bu sayede == kullanılmak istenirken yanlışlıkla = yazılmasının önüne geçilmiş olmaktadır.

5.8 Dizilerin Öteki Tiplerle Karşılaştırılması

Dizi nesneleri, aynı dizi tipine sahip başka nesnelerle de karşılaştırılabilir. Bu karşılaştırmada *sözlük sıralaması* kullanılır. Buna göre ilk iki öğe karşılaştırılır. Eğer bunlar birbirinden farklıysa karşılaştırmanın sonucu belli olmuş olur. Eğer bunlar birbirine eşitse sonraki iki

öğe karşılaştırılır ve dizinin sonuna gelinene kadar bu böyle devam eder. Eğer karşılaştırılan iki öğe aynı veri tipine sahip diziler ise, sözlük sırasına göre karşılaştırma özyinelemeli olarak yürütülür. Eğer iki dizinin bütün öğeleri eşitse, bu iki dizi eşit kabul edilir. Eğer bir dizi, başka bir dizinin baş taraftan alt dizisi ise, kısa dizi küçük (az) olandır. Karakter dizileri, her bir karakterin Unicode kod konumuna karşılık gelen sayıya göre sıralanır. Aynı tipteki diziler arasında karşılaştırmaya ilişkin birkaç örnek verelim:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Nesnelerin uygun karşılaştırma metotlarına sahip olması kaydıyla, farklı tipteki nesneler < veya > işleçleri ile karşılaştırılabilir. Örneğin karışık tipteki sayılar, sayısal değerlerine göre karşılaştırılır. Dolayısıyla 0 ile 0.0 birbirine eşittir. Aksi durumda, rastgele bir sıralama vermek yerine, yorumlayıcı `TypeError` istisnasını tetikleyecektir.

BÖLÜM 6

Modüller

Python'ın yorumlayıcısını kapatıp tekrar açtığınızda, daha önce yapmış olduğunuz tanımlamalar (fonksiyonlar ve değişkenler) kaybolacaktır. Bu yüzden, eğer uzunca bir program yazmak istiyorsanız, yorumlayıcıya gönderilecek girdiyi hazırlamak için bir metin düzenleyici kullanmak ve girdi olarak yorumlayıcıya bu dosyayı verdikten sonra yorumlayıcıyı çalıştırmak daha mantıklı olacaktır. Bahsettiğimiz bu işleme *betik* oluşturma denir. Kodlarınız uzadıkça, bakım işini kolaylaştırmak için kodlarınızı birkaç dosyaya bölmeyi tercih edebilirsiniz. Ayrıca, yazdığınız yardımcı bir fonksiyonu, içeriğindeki tanımları her bir programa tek tek kopyalamadan birkaç farklı programda birden kullanmak da isteyebilirsiniz.

Python bu ihtiyaçlarınızı karşılamak için, size tanımlarınızı bir dosyaya yerleştirip, daha sonra bunları bir betik içinde veya yorumlayıcının etkileşimli bir oturumunda kullanma imkanı sunar. İşte bu tür dosyalara *modül* adı verilir. Bir modül içindeki tanımlar başka modüllerin veya *ana* (main) modülün (bir betik içinden veya etkileşimli kabuktayken erişebildiğiniz en tepedeki değişkenler bütünü) *içine aktarılabilir*.

Modüller, Python tanımlarını ve deyimlerini içeren dosyalardır. Dosya adı ise, modül adının `.py` uzantısı eklenmiş halidir. Bir modül içinde, modülün adına (bir karakter dizisi olarak) `__name__` adlı global bir değişkenin değeri aracılığıyla erişebiliriz. Mesela, o anda içinde bulunduğunuz dizinde, herhangi bir metin düzenleyici kullanarak, `fibo.py` adıyla bir dosya oluşturun; içine de şunları yazın:

```
# Fibonacci sayıları modülü

def fib(n):    # n'e kadar olan Fibonacci dizilerini yazdırıyoruz
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n): # n'e kadar olan Fibonacci dizilerini döndürüyoruz
    sonuç = []
    a, b = 0, 1
    while b < n:
        sonuç.append(b)
        a, b = b, a+b
    return sonuç
```

Şimdi Python yorumlayıcısını açın ve bu modülü aşağıdaki komut yardımıyla içe aktarın:

```
>>> import fibo
```

Bu komut, `fibo` içinde tanımlanan fonksiyonların adlarını mevcut simge tablosuna doğrudan işlemez, oraya yalnızca modül adı olan `fibo`'yu yerleştirir. İşte bu modül adını kullanarak fonksiyonlara erişebilirsiniz:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Eğer modül içindeki bir fonksiyonu sık sık kullanacaksanız, bunu lokal bir isme atayabilirsiniz:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1 Modüller Hakkında Daha Fazlası

Modüller fonksiyon tanımlarının yanısıra çalıştırılabilir deyimler de içerebilir. Bu deyimlerin amacı modülü ilk kullanıma hazırlamaktır. Bunlar yalnızca, modül adı bir içe aktarma deyimi içinde *ilk* kez geçtiğinde işletilir[1]. (Bunlar ayrıca dosya bir betik olarak çalıştırıldığında da işletilir.)

Her modülün kendi özel simge tablosu bulunur. Bu tablo, modül içinde tanımlanan bütün fonksiyonlar tarafından kullanılan bir global simge tablosu olarak işlev görür. Dolayısıyla bir modülün yazarı, kullanıcıların global değişkenleriyle yanlışlıkla çakışma endişesi taşımadan modül içinde global değişkenler kullanabilir. Öte yandan, eğer yaptığınız işin sonuçlarından eminseniz, bir modülün global değişkenlerine, o modülün fonksiyonlarına erişirken kullandığınız yöntemi (`modüladı.öğeadı`) kullanarak müdahale edebilirsiniz.

Modüller başka modülleri de içe aktarabilir. Adet olduğu üzere, bütün `import` deyimleri bir modülün (dolayısıyla bir betiğin) en başına yerleştiriliyor olsa da aslında bu zorunlu değildir. İçe aktarılan modül adları, içe aktaran modülün global simge tablosuna yerleştirilir.

Bir modül içindeki isimleri, içe aktaran modülün simge tablosuna doğrudan dahil etmek için `import` deyimini farklı bir şekilde kullanabiliriz. Mesela:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Bu komut, içe aktarılan öğelerin alındığı modülün adını lokal simge tablosuna eklemez (dolayısıyla bu örnekte `fibo` adı tanımlanmamış olacaktır).

Bir modül içinde tanımlanan bütün isimleri almaya yarayan bir içe aktarma biçimi de bulunur:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Bu yöntemle, alt çizgi (_) ile başlayanlar hariç bütün isimler içe aktarılır. Çoğu durumda Python programcıları bu kolaylıktan yararlanmaz, çünkü bu yöntem yorumlayıcının içine

[1] Fonksiyon tanımları da aslında 'işletilen' birer 'deyimdir'. Modül seviyesindeki bir fonksiyon tanımının çalıştırılması fonksiyon adını modülün global simge tablosuna yerleştirir.

bilinmeyen birtakım isimleri dahil ederek, muhtemelen daha önce tanımlamış olduğunuz bazı şeyleri gizleyecektir.

Kodların okunmasını güçleştirdiği için, genellikle bir modülün veya paketin içindeki her şeyi * ile içe aktarmaya iyi gözle bakılmaz. Ancak etkileşimli oturumlarda, yazacağınız kod miktarını azaltmak için bu yöntemden yararlanabilirsiniz.

Not: Performansı artırmak için, her modül, yorumlayıcı oturumu başına yalnızca bir kez içe aktarılır. Dolayısıyla, eğer modülleriniz üzerinde değişiklik yaparsanız yorumlayıcıyı yeniden başlatmanız gerekir. Ya da, eğer etkileşimli olarak test ettiğiniz yalnızca tek bir modül varsa `imp.reload()` fonksiyonunu kullanabilirsiniz. Şöyle: `import imp; imp.reload(modüladı)`.

6.1.1 Modüllerin Betik Olarak Çalıştırılması

Python modüllerini şu komutla çalıştırdığınızda:

```
python fibo.py <argümanlar>
```

modül içindeki kod, sanki siz bunu içe aktarmışsınız gibi işletilecektir. Ancak bu durumda `__name__` değişkeninin değeri `"__main__"` olarak ayarlanacaktır. Dolayısıyla şu kodu modülünüzün en sonuna yerleştirdiğinizde:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

Dosyanızın hem bir betik olarak hem de içe aktarılabilir bir modül olarak kullanılabilmesine olanak tanımış olursunuz. Çünkü bu şekilde komut satırında girilen argümanları ayrıştıran kodlar yalnızca modülün 'ana' (main) dosya olarak işletildiği durumlarda çalıştırılacaktır:

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

Eğer modül içe aktarılırsa bu kodlar çalışmaz:

```
>>> import fibo
>>>
```

Bu yöntem genellikle bir modül için kullanışlı bir arayüz sunmak amacıyla veya test maksadıyla kullanılır (bir modülün betik olarak çalıştırılması test takımının işletilmesini sağlayacaktır).

6.1.2 Modüllerin Aranma Yolu

Mesela `falanca` adlı bir modül içe aktarıldığında Python yorumlayıcısı ilk olarak bu ada sahip bir gömülü modül arar. Eğer böyle bir modül bulamazsa, `sys.path` değişkeni içinde tanımlanan dizin listesinde `falanca.py` adlı bir dosya arar. `sys.path` değişkeninin öntanımlı değerleri şu konumlardan alınır:

- Girilen betiği içeren dizin (veya eğer herhangi bir dosya belirtilmemişse mevcut çalışma dizini).
- `PYTHONPATH` (`PATH` adlı kabuk değişkeni ile aynı söz dizimine sahip olan ve dizin adlarından oluşan bir liste).

- Kurulumda belirlenen öntanımlı dizin.

Not: Sembolik bağları destekleyen dosya sistemlerinde, girilen betiği içeren dizin, sembolik bağ takip edildikten sonra hesaplanır. Diğer bir deyişle sembolik bağı içeren dizin modülün aranma yoluna **eklenmez**.

İlk değerler atandıktan sonra `sys.path` değerini başka Python programları aracılığıyla değiştirebilirsiniz. Çalıştırılan betiği içeren dizin, standart kütüphaneyi içeren yolun önüne, arama yolunun en başına yerleştirilir. Böylece kütüphane yolunda bulunan aynı adlı modüllerin yerine bu dizindeki betikler yüklenecektir. Eğer bu değişikliği farkında olmadan yaptıysanız programınız hata verecektir. Daha fazla bilgi için bkz. *Standart Modüller*.

6.1.3 'Derlenmiş' Python Dosyaları

Modüllerin daha hızlı yüklenebilmesi için Python her bir modülün derlenmiş halini `_pycache__` dizini içinde, `modül.`*`sürüm`*`.pyc` adıyla önbelleğe alır. Burada, 'sürüm' derlenmiş dosyanın biçimini gösterir. Bu genellikle Python'ın sürüm numarasını içerir. Örneğin CPython'ın 3.3 sürümünde falanca.py'nin derlenmiş hali `__pycache__/falanca.cpython-33.pyc` olarak önbelleğe alınacaktır. Bu adlandırma şeması sayesinde Python'ın farklı dağıtım ve sürümlerine ait derlenmiş modüller bir arada bulunabilecektir.

Python, kaynak dosyanın değiştirilme tarihini derlenmiş dosyanınkiyle karşılaştırarak derlenmiş dosyanın güncel olup olmadığını, yeniden derlenmesinin gerekip gerekmediğini belirler. Tamamen otomatik bir süreçtir bu. Ayrıca derlenmiş modüller platformdan bağımsızdır. Dolayısıyla aynı kütüphane, farklı mimarilere sahip sistemler arasında paylaşılabilir.

Python şu iki durumda önbelleği kontrol etmez: İlk olarak, eğer modül doğrudan komut satırından yükleniyorsa kaynak dosyayı her defasında yeniden derler ve dosyanın derlenmiş halini saklamaz. İkincisi, eğer ortada kaynak modül yoksa önbelleği kontrol etmez. Programınızı kaynak dosyası olmadan (yalnızca dosyanın derlenmiş halini) dağıtabilmek için, derlenmiş modülü kaynak dizin içine yerleştirmeniz ve kaynak modülü de oradan çıkarmanız gerekir.

Uzmanlar için bazı ipuçları verelim:

- Derlenmiş bir modülün boyutunu küçültmek için Python komut satırında `-O` veya `-OO` seçeneklerini kullanabilirsiniz. `-O` seçeneği assert deyimlerini, `-OO` seçeneği ise hem assert deyimlerini hem de __doc__ dizilerini siler. Ama bazı programlar bunların varlığına ihtiyaç duyuyor olabileceğinden ötürü, bu seçeneği yalnızca yaptığınız işin sonuçlarının farkındaysanız kullanmalısınız. 'Optimize edilmiş' modüller `opt-` etiketi taşır ve genellikle boyut olarak daha küçüktür. Ancak gelecekteki sürümlerde optimizasyonun nitelikleri değişebilir.
- Bir programın `.pyc` dosyasından okunması, `.py` dosyasından okunmasına kıyasla onun daha hızlı çalışmasını sağlamaz. `.pyc` dosyaları yalnızca modülün daha hızlı yüklenmesini sağlar.
- `compileall` modülü, bir dizin içindeki bütün modüller için .pyc dosyaları oluşturabilir.
- Buna dair karar süreçlerine ilişkin bir akış şemasıyla birlikte, bu sürece ilişkin daha fazla ayrıntıyı **PEP 3147**'de bulabilirsiniz.

6.2 Standart Modüller

Python; Python Kütüphane Referansı (bundan böyle 'Kütüphane Referansı' olarak anılacaktır) adlı ayrı bir belge içinde tarif edilen bir standart modül kütüphanesiyle birlikte gelir. Bazı modüller yorumlayıcının içine gömülü vaziyettedir. Bu modüller, dilin çekirdeğinin bir parçası olmasalar da, işletim sistemlerine ilişkin sistem çağrıları gibi temel işlemlere erişim sağlamak veya performansı artırmak amacıyla dilin içine gömülmüş olup, bu tür işlemlere erişilebilmesini sağlar. Bu modüllerin hangileri olduğu yapılandırmaya ilişkin bir tercih olup, hangi işletim sistemini kullandığınıza da bağlıdır. Örneğin `winreg` modülü yalnızca Windows sistemlerinde bulunur. Bunlar arasında özellikle bir modül özel ilgiyi hakediyor. Bu modül, bütün Python yorumlayıcılarına gömülü vaziyette bulunan `sys` modülüdür. `sys.ps1` ve `sys.ps2` değişkenleri birincil ve ikincil komut istemi olarak kullanılacak karakter dizilerini tanımlar.

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Mıymıy!')
Mıymıy!
C>
```

Bu iki değişken, yorumlayıcı yalnızca etkileşimli kipte iken tanımlanabilir.

`sys.path` değişkeni, yorumlayıcının modül arama yolunu belirleyen, karakter dizilerinden oluşmuş bir listedir. Bu listenin öntanımlı değeri, `PYTHONPATH` adlı çevre değişkeninden veya `PYTHONPATH` tanımlı değilse gömülü bir öntanımlı değerden elde edilen yoldur. Bu yolu, standart liste işlemleri aracılığıyla değiştirebilirsiniz:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3 `dir()` Fonksiyonu

Gömülü `dir()` fonksiyonu, bir modül içinde hangi isimlerin tanımlı olduğunu bulmak için kullanılır. Bu fonksiyon karakter dizilerinden oluşan sıralı bir liste döndürür:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__loader__', '__name__',
 '__package__', '__stderr__', '__stdin__', '__stdout__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_getframe',
 '_home', '_mercurial', '_xoptions', 'abiflags', 'api_version', 'argv',
 'base_exec_prefix', 'base_prefix', 'builtin_module_names', 'byteorder',
 'call_tracing', 'callstats', 'copyright', 'displayhook',
 'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix',
 'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
 'getrefcount', 'getsizeof', 'getswitchinterval', 'gettotalrefcount',
```

```
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path',
 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
 'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit',
 'setswitchinterval', 'settrace', 'stderr', 'stdin', 'stdout',
 'thread_info', 'version', 'version_info', 'warnoptions']
```

Eğer `dir()` fonksiyonunu argümansız kullanırsanız, o anda tanımlamış olduğunuz isimlerin listesini alırsınız:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Burada bütün isim türlerinin listelendiğine dikkat edin: değişkenler, modüller, fonksiyonlar, vb.

`dir()`, gömülü fonksiyonların ve değişkenlerin isimlerini listelemez. Eğer bu isimleri de listelemek isterseniz, bunlar `builtins` adlı bir standart modül içinde tanımlanmıştır:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
 'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
 'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__',
 '__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs',
 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable',
 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',
 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',
 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview',
 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property',
 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',
 'zip']
```

6.4 Paketler

Paketler, 'noktalı modül adları' kullanarak Python'ın modül isim alanını yapılandırmanın bir yöntemidir. Örneğin, `A.B` modül adı, `A` adlı bir paket içindeki `B` adlı bir alt modüle işaret eder. Nasıl modüllerin kullanımı farklı modüllerin yazarlarını birbirlerinin global değişken adlarını dert etmekten kurtarıyorsa, noktalı modül adlarının kullanımı da, NumPy veya Python Görüntüleme Kütüphanesi gibi birden fazla modülden oluşan paketlerin yazarlarını da birbirlerinin modül adlarını dert etmekten kurtarır.

Diyelim ki, ses dosyalarının ve ses verilerinin tek elden işlenebilmesini sağlayan bir dizi modül (yani bir 'paket') tasarlamak istiyorsunuz. Etrafta pek çok farklı ses dosyası biçimi bulunur (bunlar genellikle uzantılarına bakılarak ayırt edilir. Örneğin: `.wav`, `.aiff`, `.au`). Dolayısıyla farklı dosya biçimleri arasındaki dönüştürme işlemleri için zaman içinde büyüyen bir modül yığını oluşturup bunun bakımını yapmanız gerekiyor olabilir. Üstelik ses verileri üzerinde gerçekleştirmeniz gerekebilecek pek çok farklı işlem de bulunur (karıştırma, yankı ekleme, ekolayzır fonksiyonu uygulama, yapay stereo efekti oluşturma gibi). Yani başka şeylere ek olarak, bu işlemleri gerçekleştirmek için de modül üstüne modül yazmak durumunda kalacaksınız. Söz konusu paket için muhtemel bir yapının nasıl olabileceğini gösterelim (hiyerarşik dosya sistemi olarak gösterilmiştir):

```
ses/                                Tepe paket
      __init__.py                 Ses paketinin ilk değerlerini belirler
      biçimler/                     Dosya biçimi dönüşümleri için alt paket
              __init__.py
              wavoku.py
              wavyaz.py
              aiffoku.py
              aiffyaz.py
              auoku.py
              auyaz.py
              ...
      efektler/                    Ses efektleri için alt paket
              __init__.py
              yankı.py
              surround.py
              reverse.py
              ...
      filtreler/                      Filtreler için alt paket
              __init__.py
              ekolayzır.py
              vocoder.py
              karaoke.py
              ...
```

Python bu paketi içe aktarırken `sys.path` içinde geçen dizinlerde bir arama yaparak paketin alt dizinlerini arar.

`__init__.py` dosyaları, Python'ın dizinlere paket içeriyormuş gibi davranmasını sağlamak için gereklidir. Böylece `string` gibi yaygın bir isme sahip dizinlerin, daha sonra modülün arama yolunda geçen öntanımlı modülleri farkında olmadan gizlemesi önlenmiş olur. En basit haliyle `__init__.py` boş bir dosya olabileceği gibi, paketin başlangıç kodlarını çalıştırabilir veya biraz sonra göreceğimiz `__all__` değişkenini tanımlayabilir.

Paketi kullananlar paket içinden modülleri tek tek içe aktarabilir. Örneğin:

```
import ses.efektler.yankı
```

Bu kod, `ses.efektler.yankı` adlı alt modülü yükler. Daha sonra bu modülü tam adıyla kullanmak gerekir.

```
ses.efektler.yankı.yankıfiltresi(giriş, çıkış, gecikme=0.7, azaltım=4)
```

Alt modülü içe aktarmanın bir başka yöntemi de şudur:

```
from ses.efektler import yankı
```

Bu kod da `yankı` adlı alt modülü yükler ve bunu paket ön eki olmadan kullanımınıza sunar. Dolayısıyla bu modülü şöyle kullanabiliriz:

```
yankı.yankıfiltresi(giriş, çıkış, gecikme=0.7, azaltım=4)
```

Başka bir yöntem ise, istenen fonksiyon veya değişkeni doğrudan içe aktarmaktır:

```
from ses.efektler.yankı import yankıfiltresi
```

Bu kod yine `yankı` alt modülünü yükler, ancak bu kez `yankıfiltresi()` adlı fonksiyonu doğrudan kullanımımıza sunar:

```
yankıfiltresi(giriş, çıkış, gecikme=0.7, azaltım=4)
```

`from paket import öğe` yapısı kullanılırken burada öğe paketin bir alt modülü (veya alt paketi) olabileceği gibi, paket içinde tanımlanmış başka bir isim de (mesela bir fonksiyon, sınıf veya değişken) olabilir. `import` deyimi ilk olarak ilgili öğenin paket içinde tanımlı olup olmadığını kontrol eder. Eğer değilse bu öğenin bir modül olduğunu varsayar ve bunu yüklemeye çalışır. Eğer öğeyi bulamazsa `ImportError` adlı istisnayı tetikler.

Yukarıdakinin aksine, `import öğe.altöğe.altaltöğe` gibi bir söz dizimi kullanıldığında, en sonuncu öğe hariç bütün öğeler bir paket olmalıdır. Son öğe bir modül veya paket olabilir, ama önceki öğe içinde tanımlanmış bir sınıf veya fonksiyon ya da bir değişken olamaz.

6.4.1 Bir Paket içinden * ile Aktarma

Peki kullanıcı `from ses.efektler import *` gibi bir kod yazdığında ne olur? Aslında arzu edilen, dosya sisteminin taranarak paket içinde hangi alt modüllerin bulunduğunun tespit edilmesi ve bulunan her şeyin içe aktarılmasıdır. Ancak böyle bir işlem çok uzun zaman alabilir ve ayrıca alt modüllerin içe aktarılması, alt modülün yalnızca açık bir şekilde içe aktarıldığı durumlarda ortaya çıkması gereken, istenmeyen yan etkilerin görülmesine yol açabilir.

Böyle bir durumda tek çözüm, paket yazarının paket içeriğine ilişkin açık bir dizin oluşturmasıdır. `import` deyimi aşağıdaki kaideyi takip eder: Eğer bir paketin `__init__.py` dosyası içinde `__all__` adlı bir liste tanımlanmışsa, `from paket import *` gibi bir kodla karşılaşıldığında bu liste içindeki modül adları içeri aktarılır. Paketin yeni bir sürümü çıktığında bu listeyi güncellemek paket yazarının sorumluluğundadır. Paket yazarları, eğer paketlerinden * ile içe aktarma yapılmasını gereksiz görüyorlarsa elbette bu özelliği kullanmayabilirler. Örneğin, `ses/efektler/__init__.py` dosyası şöyle bir kod içerebilir:

```
__all__ = ["yankı", "surround", "reverse"]
```

Böylece `from ses.efektler import *` komutu `ses` paketindeki belirli üç alt modülü içe aktaracaktır.

Eğer `__all__` tanımlı değilse `from ses.efektler import *` deyimi `ses.efektler` adlı paketin içindeki bütün alt modülleri mevcut isim alanına *aktarmaz*. Bu kodun yaptığı tek şey, (`__init__.py` dosyası içinde olabilecek herhangi bir başlatma kodunu da çalıştırarak) `ses.efektler` paketinin içe aktarılmasını sağlamak ve paket içinde tanımlanmış hangi isimler varsa onları içe aktarmaktır. Buna `__init__.py` tarafından tanımlanan bütün isimler (ve açıkça yüklenen alt modüller) dahildir. Bu ayrıca, paketin, önceki `import` deyimleri aracılığıyla açık bir şekilde yüklenmiş olan alt modüllerini de kapsar. Mesela şu koda bir bakalım:

```
import ses.efektler.yankı
import ses.efektler.surround
from ses.efektler import *
```

Bu örnekte `yankı` ve `surround` modülleri, `from...import` deyimi çalıştırıldığında, `ses.efektler` paketi içinde tanımlanmış oldukları için mevcut isim alanına aktarılır. (Bu ayrıca `__all__` tanımlandığında da çalışır.)

Bazı modüller, `import *` yapısı kullanıldığında yalnızca belli şablonlara uyan isimler içe aktarılacak şekilde tasarlanmış olsa da, ciddi kodlarda bu yapının kullanılması kötü bir alışkanlık olarak kabul edilir.

Ancak, `from Paket import belirli_bir_altmodül` yapısını kullanmanın hiçbir sakıncası olmadığını da unutmayın! Aslında, modülün, farklı paketlerden aynı ada sahip alt modülleri kullanması gerekmedikçe modül içe aktarmanın önerilen yöntemi budur.

6.4.2 Paket İçi Atıflar

Paketlerinizi alt paketler şeklinde yapılandırarak (örnekteki `ses` paketinde olduğu gibi), komşu paketlerin alt modüllerine erişmek için mutlak içe aktarmalardan yararlanabilirsiniz. Örneğin eğer `ses.filtreler.vocoder` modülünün, `ses.efektler` paketi içindeki `yankı` modülünü kullanması gerekirse, `from ses.efektler import yankı` kodu kullanılabilir.

Eğer isterseniz içe aktarma deyimini `from modül import isim` şeklinde kullanarak bağıl içe aktarmalar da yazabilirsiniz. Bu içe aktarma işlemlerinde, bağıl içe aktarmada yer alan mevcut ve üst paketleri göstermek için isimlerin önüne nokta işaretleri getirilir. Örneğin `surround` modülü içinde şunu kullanabilirsiniz:

```
from . import yankı
from .. import biçimler
from ..filtreler import ekolayzır
```

Göreceli içe aktarmaların mevcut modülün ismini temel aldığına dikkat edin. Ana modülün ismi her zaman `'__main__'` olduğu için, bir Python uygulamasında ana modül olarak kullanılması planlanan modüllerde daima mutlak içe aktarmadan yararlanılmalıdır.

6.4.3 Çoklu Dizinler içindeki Paketler

Paketler başka bir özel niteliğe daha sahiptir: `__path__`. Bu niteliğin ilk değeri, o dosya içindeki kod işletilmeden önce paketin `__init__.py` dosyasını tutan dizini içeren bir listedir. Bu değişkenin değeri değiştirilebilir. Yaptığınız değişiklikler, paket içinde yer alan modül ve alt paketlere ilişkin gelecekteki aramaları etkileyecektir.

Bu özelliğe genellikle pek ihtiyaç duyulmasa da, bir paket içinde yer alan modül sayısını genişletmek için bu özellikten yararlanılabilir.

BÖLÜM 7

Girdi ve Çıktı

Bir programın çıktısını sunmanın birkaç yolu bulunur: Veriler okunaklı bir şekilde ekrana basılabileceği gibi, daha sonra kullanılmak üzere bir dosyaya da yazılabilir. İşte bu bölümde, elimizdeki imkanların bazılarına değineceğiz.

7.1 Çıktıları Şık Biçimlendirme

Şimdiye kadar değerleri yazdırmanın iki yöntemini öğrendik: *ifade deyimleri* ve `print()` fonksiyonu. (Üçüncü bir yol da dosya nesnelerinin `write()` metodunu kullanmaktır. Standart çıktı dosyası `sys.stdout` olarak belirtilebilir. Buna ilişkin daha fazla bilgi için Kütüphane Referansı'na bakabilirsiniz.)

Muhtemelen, birbirlerinden boşlukla ayrılmış değerleri ekrana basmakla yetinmeyecek, programınızın çıktısını daha esnek bir şekilde biçimlendirebilmek isteyeceksiniz. Programınızın çıktısını biçimlendirmenin iki yöntemi vardır: Bunlardan ilki, karakter dizilerini tamamen elle düzenlemektir. Karakter dizisi dilimleme ve birleştirme işlemlerini kullanarak aklınıza gelebilecek her türlü dizilimi oluşturabilirsiniz. Karakter dizisi adlı veri tipinin, karakter dizilerini istediğiniz sütun genişliğine göre ayarlamanızı sağlayacak birtakım faydalı metotları bulunur. Burada bunlardan kısaca söz edeceğiz. İkinci yöntem ise `str.format()` metodunu kullanmaktır.

`string` modülü, birtakım değerleri karakter dizilerine dönüştürme imkanı sunan `Template` adlı bir sınıf içerir.

Ama şu soru hala cevapsız duruyor: Peki biz değerleri karakter dizilerine nasıl çevireceğiz? Neyse ki Python bize herhangi bir değeri karakter dizisine dönüştürmek için bazı yöntemler sunuyor: Bu amaç için, ilgili değeri `repr()` veya `str()` fonksiyonuna parametre olarak vermemiz yeterli olacaktır.

`str()` fonksiyonu değerleri insanlar tarafından okunabilecek bir şekilde temsil ederken, `repr()` fonksiyonu ise değerleri yorumlayıcı tarafından okunabilecek şekilde temsil eder (eğer eşdeğer bir söz dizimi bulamazsa da `SyntaxError` istisnasını tetikler). İnsanlar tarafından okunabilecek şekilde temsil edilemeyen nesneler için `str()` fonksiyonu `repr()` ile aynı değeri döndürecektir. Sayılar veya liste ve sözlük yapıları gibi değerler her iki fonksiyonda da aynı şekilde temsil edilir. Karakter dizileri ise iki farklı şekilde temsil edilebilir.

Birkaç örnek verelim:

```
>>> s = 'Merhaba dünya.'
>>> str(s)
'Merhaba dünya.'
```

```
>>> repr(s)
"'Merhaba dünya.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = "x'in değeri " + repr(x) + ", y'nin değeri ise " + repr(y) + "'dir..."
>>> print(s)
x'in değeri 32.5, y'nin değeri ise 40000'dir...
>>> # repr() karakter dizisi çıktılarına tırnakları ve ters bölüleri ekler:
... merhaba = 'merhaba dünya\n'
>>> merhabak = repr(merhaba)
>>> print(merhabak)
'merhaba dünya\n'
>>> # repr() fonksiyonuna argüman olarak her türlü Python nesnesi verilebilir:
... repr((x, y, ('adana', 'mersin')))
"(32.5, 40000, ('adana', 'mersin'))"
```

Aşağıda sayıların kareleri ve küplerinden meydana gelen bir tablo oluşturmanın iki farklı yöntemini görüyorsunuz:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Bir önceki satırda 'end' parametresinin kullanımına dikkat
...     print(repr(x*x*x).rjust(4))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000

>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

(İlk örnekte her bir sütun arasındaki bir adet boşluğun `print()` fonksiyonunun işleyişi gereği eklendiğine dikkat edin. Bu fonksiyon, kendisine verilen argümanlar arasına her zaman bir adet boşluk yerleştirir).

Bu örnekte, karakter dizisi nesnelerinin `str.rjust()` metodunu görüyoruz. Bu metot karakter dizilerini, belirlenen genişlikteki bir alan içinde, sol tarafa boşluk eklemek suretiyle

sağa yaslar. Buna benzer şekilde `str.ljust()` ve `str.center()` metotları da vardır. Bu metotlar karakter dizisi üzerinde herhangi bir değişiklik yapmaz; bunların yaptığı tek şey, yeni bir karakter dizisi döndürmekten ibarettir. Eğer girilen karakter dizisi çok uzunsa bu metotlar karakter dizisini kırpıp kısaltmak yerine karakter dizisini olduğu gibi döndürür. Bu durum tablodaki sütun yapısını altüst edecektir, ancak bu, değeri kırpıp sizi yanıltmaktan çok daha iyi bir alternatiftir. (Eğer istediğiniz şey karakter dizisinin kırpılması ise, bunun için `x.ljust(n)[:n]` gibi bir şey yazarak dilimleme işleminden yararlanabilirsiniz.)

Bir başka metot olan `str.zfill()` ise, sayı değerli bir karakter dizisinin sol tarafına sıfır ekleyerek bunu kaydırır. Bu metot artı ve eksi işaretlerini de tanır:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

`str.format()` metodunun temel kullanımı şöyledir:

```
>>> print('Pamuk {} ve yedi {}'.format('prenses', 'cüceler'))
Pamuk prenses ve yedi cüceler
```

Buradaki süslü parantezler ve içlerindeki karakterler (bunlara biçimlendirme alanları adı verilir), `str.format()` metoduna gönderilen nesneler ile değiştirilir. `str.format()` metoduna gönderilen nesnenin konumunu belirtmek için süslü parantezler içinde sayılar kullanılabilir.

```
>>> print('{0} ve {1}'.format('döner', 'turşu'))
döner ve turşu
>>> print('{1} ve {0}'.format('döner', 'turşu'))
turşu ve döner
```

Eğer `str.format()` metodunda isimli argümanlar kullanılırsa bunların değerlerine argümanın adı kullanılarak atıfta bulunulabilir.

```
>>> print('Bu {yemek} kesinlikle {nasıl}.'.format(
...       yemek='döner', nasıl='berbat'))
Bu döner kesinlikle berbat.
```

Konumlu ve isimli argümanları istediğiniz şekilde sıralayabilirsiniz:

```
>>> print('{0}, {1} ve {diğeri}.'.format('Atos', 'Portos', diğeri='Aramis'))

Atos, Portos ve Aramis
```

Biçimlendirmeden önce değeri dönüştürmek için `'!a'` (`ascii()` fonksiyonunu uygular), `'!s'` (`str()` fonksiyonunu uygular) ve `'!r'` (`repr()` fonksiyonunu uygular) kullanılabilir:

```
>>> import math
>>> print("PI'nin yaklaşık değeri: {}.".format(math.pi))
PI'nin yaklaşık değeri: 3.14159265359.
>>> print("PI'nin yaklaşık değeri: {!r}.".format(math.pi))
PI'nin yaklaşık değeri: 3.141592653589793.
```

Alan adının ardından, istenirse `':'` ve bir biçim düzenleyici getirilebilir. Böylelikle değerin nasıl biçimlendirileceğini daha esnek bir şekilde kontrol edebilirsiniz. Aşağıdaki örnek, Pi'nin değerini noktadan sonra üç basamak olacak şekilde yuvarlar.

```
>>> import math
>>> print("PI'nin yaklaşık değeri: {0:.3f}.".format(math.pi))
PI'nin yaklaşık değeri: 3.142.
```

':' ifadesinin ardından bir tamsayı getirilirse, o alana en az bu değer kadar karakterin sığabileceği bir boşluk ayrılacaktır. Bu yöntem özellikle tabloların görünüşünün güzelleştirilmesine yarar.

```
>>> tablo = {'Ökkeş': 4127, 'Selami': 4098, 'Kezban': 7678}
>>> for isim, tel in tablo.items():
...     print('{0:10} ==> {1:10d}'.format(isim, tel))
...
Kezban     ==>       7678
Ökkeş      ==>       4127
Selami     ==>       4098
```

Eğer elinizde bölmek istemediğiniz, çok uzun bir biçimlendirme dizisi varsa, biçimlendirilecek değişkenleri konumlarına göre değil de isimlerine göre kullanabilseniz ne iyi olurdu, değil mi? İşte bu işlem parametre olarak bir sözlük verilerek ve bu sözlüğün öğelerine erişmek için de köşeli parantezler '[]' kullanılarak yapılabilir:

```
>>> tablo = {'Ökkeş': 4127, 'Selami': 4098, 'Kezban': 8637678}
>>> print('Selami: {0[Selami]:d}; Ökkeş: {0[Ökkeş]:d}; '
...       'Kezban: {0[Kezban]:d}'.format(tablo))
Selami: 4098; Ökkeş: 4127; Kezban: 8637678
```

Aynı şey, '**' işaretleri yardımıyla tablonun isimli argüman olarak verilmesiyle de yapılabilir.

```
>>> tablo = {'Ökkeş': 4127, 'Selami': 4098, 'Kezban': 8637678}
>>> print('Selami: {Selami:d}; Ökkeş: {Ökkeş:d}; Kezban: {Kezban:d}'.format(**tablo))
Selami: 4098; Ökkeş: 4127; Kezban: 8637678
```

Bu özellik, bütün lokal değişkenleri içinde barındıran bir sözlük döndüren `vars()` gömülü fonksiyonuyla birlikte kullanıldığında epey işinize yarayacaktır.

`str.format()` ile karakter dizisi biçimlendirme hakkında en geniş bilgi için bkz. Biçimlendirme Dizilerinin Söz Dizimi

7.1.1 Eski Karakter Dizisi Biçimlendirme Yöntemi

Karakter dizisi biçimlendirme işlemleri için % işleci de kullanılabilir. Bu işleç, sol tarafındaki argümanı `sprintf()` tarzına çok benzer bir şekilde yorumlayarak sağ taraftaki argümana uygular ve bu biçimlendirme işleminin ortaya çıkardığı karakter dizisini döndürür. Mesela:

```
>>> import math
>>> print("PI'nin yaklaşık değeri: %5.3f." % math.pi)
PI'nin yaklaşık değeri: 3.142.
```

Daha fazla bilgiye printf Tarzı Karakter Dizisi Biçimlendirme bölümünden ulaşabilirsiniz.

7.2 Dosya Okuma ve Yazma

`open()` fonksiyonu bir dosya nesnesi döndürür. Bu fonksiyon yaygın olarak iki argümanla kullanılır: `open(dosyaadı, okuma_kipi)`.

```
>>> f = open('işdosyası', 'w')
```

İlk argüman, dosya adını gösteren bir karakter dizisidir. İkinci argüman ise, dosyanın ne şekilde kullanılacağını gösteren birkaç karakterden oluşur. Eğer dosya yalnızca okunacaksa *kip* `'r'` olabilir. Yalnızca yazma işlemi yapılacaksa `'w'` (aynı isimli dosya varsa silinecektir) ve dosyaya ekleme yapılacaksa da `'a'` kullanılabilir. Dosyaya yazılan her veri otomatik olarak dosya sonuna eklenecektir. `'r+'` dosyayı hem okuma hem de yazma işlemleri için açar. *kip* argümanını yazmasanız da olur. Eğer bu argüman kullanılmazsa `'r'` kullanmışsınız gibi davranılacaktır.

Normal şartlar altında dosyalar *metin kipinde* açılır. Buna göre, dosyadan okuduğunuz ve dosyaya yazdığınız veri bir karakter dizisidir. Bu veriler belirli bir dil kodlamasına göre kodlanır. Eğer dil kodlaması belirtilmemişse, öntanımlı değerin ne olacağı platforma bağlıdır (bkz. `open()`) Kipe eklenen `'b'` harfi dosyayı *ikili kipte* açar. Böylelikle veriler bayt nesneleri biçiminde okunur ve yazılır. Metin içermeyen bütün dosyalar için bu kip kullanılmalıdır.

Metin kipinde, okuma işlemi esnasındaki öntanımlı davranış platforma özgü satır sonlarının (Unix'te `\n`, Windows'ta `\r\n`) `\n`'ye dönüştürülmesidir. Metin kipinde yazma işlemi gerçekleştirilirken öntanımlı davranış, dosyada geçen `\n` işaretlerini tekrar platforma özgü satır sonlarına dönüştürmektir. Perde arkasında gerçekleşen bu veri değiştirme işlemi metin dosyalarında bir sorun çıkartmaz, ancak bu işlem `JPEG` veya `EXE` dosyaları gibi ikili veriler içeren dosyaları bozacaktır. Dolayısıyla bu tür dosyaları okuyup yazarken ikili kip kullanmaya özen göstermelisiniz.

7.2.1 Dosya Nesnelerinin Metotları

Bu bölümdeki örneklerin geri kalanında, `f` adlı bir dosya nesnesinin halihazırda oluşturulmuş olduğu varsayılacaktır.

Bir dosyanın içeriğini okumak için `f.read(boyut)` komutu çalıştırılır, böylece verinin istenen miktarı okunur ve okunan veriler bir karakter dizisi ya da bayt nesnesi olarak döndürülür. *boyut*, belirtilmesi zorunlu olmayan sayı değerli bir argümandır. Eğer *boyut* argümanı yazılmazsa veya eksi bir değer verilirse dosyanın bütün içeriği okunacak ve döndürülecektir. Bu esnada dosya boyutunun bilgisayarınızın bellek kapasitesini aşması tamamen sizin sorununuzdur. Eğer *boyut* parametresini belirtirseniz, dosyadan en fazla *boyut* değerinin gösterdiği kadar veri okunacak ve döndürülecektir. Eğer dosyanın sonuna gelinmişse `f.read()` komutu boş bir karakter dizisi döndürecektir (`''`).

```
>>> f.read()
'Dosyanın hepsi bu kadar.\n'
>>> f.read()
''
```

`f.readline()` dosyadan tek bir satır okur. Yeni satır karakterini (`\n`) karakter dizisinin sonunda bırakır. Ama eğer dosya yeni satır karakteri ile bitmiyorsa dosyanın son satırına bu karakteri yerleştirmez. Böylelikle dönüş değerinin ne anlama geldiği açıkça anlaşılmış olur. Yani, eğer `f.readline()` boş bir karakter dizisi döndürmüşse dosyanın sonuna ulaşılmış demektir. Boş satır ise `'\n'` ile (yani yalnızca satır başı karakteri içeren bir karakter dizisi ile) gösterilir.

```
>>> f.readline()
'Dosyanın ilk satırı bu.\n'
>>> f.readline()
'İkinci satır ise bu\n'
```

```
>>> f.readline()
''
```

Dosyadan satır okumak için dosya nesnesi üzerinde döngü de kurabilirsiniz. Bu yöntem belleğe yük bindirmez, hızlıdır ve kodunuzun basit olmasını sağlar:

```
>>> for satır in f:
...     print(satır, end='')
...
Dosyanın ilk satırı bu.
İkinci satır ise bu
```

Eğer bir dosyadaki bütün satırları okuyup bir listeye atmak istiyorsanız `list(f)` veya `f.readlines()` komutunu da kullanabilirsiniz.

`f.write(karakter_dizisi)` kodu *karakter dizisinin* içeriğini dosyaya yazar ve yazılan karakterlerin sayısını döndürür.

```
>>> f.write('Bu bir denemedir\n')
17
```

Eğer karakter dizisi dışında bir şey yazmak isterseniz, bunu öncelikle karakter dizisine çevirmeniz gerekir:

```
>>> değer = ('cevap', 42)
>>> s = str(değer)
>>> f.write(s)
13
```

`f.tell()` fonksiyonu, dosya nesnesinin dosya içindeki mevcut konumunu veren bir tam sayı döndürür. İkili kipte bu sayı dosyanın başından mevcut konuma kadar olan baytların sayısı iken, metin kipinde ise muğlak bir sayıdır.

Dosya nesnesinin konumunu değiştirmek için `f.seek(uzaklık, nereden_itibaren)` komutunu kullanabilirsiniz. Konum, referans noktasına *uzaklık* eklenerek hesaplanır. Referans noktası ise *nereden_itibaren* argümanının değerine göre belirlenir. *nereden_itibaren* argümanına 0 değeri verilirse, konum dosyanın başından itibaren hesaplanır. 1 değeri mevcut dosya konumunu, 2 değeri ise referans noktası olarak dosyanın en sonunu alır. *nereden_itibaren* argümanı yazılmayabilir. Bu şekilde bu argümanın öntanımlı değeri 0 olacak, böylece referans noktası dosyanın en başı olacaktır.

```
>>> f = open('işdosyası', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)     # Dosyada 6. bayt konumuna gidiyoruz
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2) # Sondan 3. bayta gidiyoruz.
13
>>> f.read(1)
b'd'
```

Metin dosyalarında (kipi gösteren karakter dizisinde `b` olmadan açılan dosyalar), yalnızca dosyanın başından itibaren arama yapılabilir (bunun istisnası `seek(0, 2)` komutuyla dosyanın en sonuna gitmektir) ve geçerli tek *uzaklık* değeri, `f.tell()` komutundan dönen değer veya sıfırdır. Bunun dışında hiçbir *uzaklık* değeri tanımlanmamıştır.

Dosyayla işiniz bittiğinde dosyayı kapatıp, açık dosya tarafından meşgul edilen bütün sistem kaynaklarını serbest bırakmak için `f.close()` fonksiyonunu çağırın. `f.close()` çağrıldıktan sonra dosya nesnesinin kullanılmaya çalışılması otomatik olarak başarısız olacaktır.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

Dosya nesneleri üzerinde çalışırken `with` sözcüğünü kullanmak iyi bir alışkanlıktır. Bu sayede, kod bloğu sona erdikten sonra, arada bir yerde bir istisna dahi tetiklense, dosya düzgün bir şekilde kapatılır. Ayrıca bu yöntem, aynı işi gören `try-finally` blokları yazmaya kıyasla çok daha kısadır.

```
>>> with open('işdosyası', 'r') as f:
...     okunan_veri = f.read()
>>> f.closed
True
```

Dosya nesnelerinin, `isatty()` ve `dosya.truncate()` gibi bazı başka metotları da bulunsa da bunlar nadiren kullanılır. Dosya nesnelerine ilişkin eksiksiz bir kılavuz için Kütüphane Referansı'na bakabilirsiniz.

7.2.2 Yapılandırılmış Verilerin `json` ile Kaydedilmesi

Karakter dizileri kolayca dosyaya yazılabilir ve dosyadan okunabilir. `read()` metodu yalnızca karakter dizisi döndürdüğü için, sayılar biraz daha fazla çaba gerektirir. Sayıların `int()` gibi bir fonksiyona gönderilmesi gerekecektir. Bu fonksiyon mesela `'123'` gibi bir karakter dizisini alıp bunun sayı değeri olan 123'ü döndürür. İç içe geçmiş listeler ve sözlükler gibi daha karmaşık veri tiplerini kaydetmek istediğinizde ayrıştırma ve diziye dönüştürme (serializing) işlemlerini elle yapmak epey dolambaçlı olacaktır.

Python, karmaşık veri tiplerini dosyaya yazmak için kullanıcıları sürekli kod yazıp hata ayıklamakla uğraştırmak yerine, gözde bir veri değişim biçimi olan JSON (JavaScript Nesne Gösterimi)'u kullanmalarına olanak tanır. `json` adlı standart bir modül aracılığıyla Python'ın veri yapılarını alıp bunları karakter dizisi olarak temsil edilecek şekilde dönüştürebilirsiniz. Bu işleme *diziye dönüştürme* adı verilir. Verilerin karakter dizisi biçimli temsilden geri dönüştürülmesi işlemine ise *diziden dönüştürme* denir. Diziye dönüştürme ve diziden dönüştürme işlemleri sırasında nesneyi temsil eden karakter dizileri bir dosya veya veri içinde saklanabileceği gibi, ağ bağlantısı üzerinden uzak bir makineye de gönderilebilir.

Not: JSON biçimi, modern uygulamalar tarafından veri dönüşümüne izin vermek için yaygın şekilde kullanılmaktadır. Pek çok programcı JSON'u halihazırda bildiği için, farklı sistemlerin birlikte çalışabilmesini temin açısından bu iyi bir seçim olmuştur.

Eğer elinizde `x` adlı bir nesne varsa, bunun JSON karakter dizisi olarak temsil edilen biçimini basit bir kod parçası yardımıyla görüntüleyebilirsiniz:

```
>>> json.dumps([1, 'basit', 'liste'])
'[1, "basit", "liste"]'
```

`dumps()` fonksiyonunun başka bir çeşidi olan `dump()` fonksiyonu, bir nesneyi diziye dönüştürüp metin dosyasına gönderir. Dolayısıyla eğer *f*, üzerine yazılmak üzere açılmış bir

metin dosyası ise şöyle bir kod yazabiliriz:

```
json.dump(x, f)
```

`f`'in, okunmak üzere açılmış bir metin dosyası olduğunu varsayarsak, nesneyi eski haline geri kodlamak için şöyle bir şey yazabiliriz:

```
x = json.load(f)
```

Bu basit diziye dönüştürme tekniği listelerin ve sözlüklerin üstesinden gelebilse de rastgele sınıf örneklerini JSON ile diziye dönüştürmek ilave çaba gerektirir. `json` modülünün kılavuzunda buna ilişkin bir açıklama var.

Ayrıca bkz.:

`pickle` - pickle modülü

JSON'un aksine *pickle*, her türden karmaşık Python nesnelerini diziye dönüştürmek için kullanılabilecek bir protokoldür. Dolayısıyla bu modül Python'a özgüdür ve başka dillerde yazılmış uygulamalarla iletişim kurmak için kullanılamaz. Bu modül ayrıca yapısı gereği güvenli değildir: Güvenilir olmayan bir kaynaktan gelen pickle verilerini diziden dönüştürmek, eğer bu veriler kabiliyetli bir saldırgan tarafından tasarlanmışsa, kodların denetlenmeden çalıştırılmasına sebep olabilir.

BÖLÜM 8

Hatalar ve İstisnalar

Şimdiye kadar hata mesajlarından yalnızca söz etmekle yetindik, ancak eğer şu ana kadar verdiğimiz örnekleri denediyseniz muhtemelen bu hata mesajlarının bazılarını kendiniz de bizzat görmüş olmalısınız. Python'da birbirinden farklı (en az) iki tür hata bulunur: *söz dizimi hataları* ve *istisnalar*.

8.1 Söz Dizimi Hataları

Henüz bu dili öğrenme aşamasındayken, Python'ın herhalde en sık, ayrıştırma hataları adıyla da bilinen söz dizimi hatalarından yakındığına şahit olmuşsunuzdur:

```
>>> while True print('Merhaba dünya')
  File "<stdin>", line 1, in ?
    while True print('Merhaba dünya')
                    ^
SyntaxError: invalid syntax
```

Burada ayrıştırıcı, sorunlu satırı ekrana basacak ve hatanın satırda ilk tespit edildiği yeri gösteren küçük bir 'ok' işareti görüntüleyecektir. Hata, oktan *önce gelen* öğeden kaynaklanıyor (veya hata hiç değilse o noktada tespit edilmiş). Bizim örneğimizde hata `print()` fonksiyonunun bulunduğu noktada tespit ediliyor. Hatanın sebebi ise bu öğeden önce getirilmesi gereken iki nokta üst üste (`':'`) işaretinin unutulmuş olması. Eğer girdiler bir betikten geliyorsa, nereye bakmanız gerektiğini anlayabilmeniz için dosya adı ve satır numarası da ekrana basılır.

8.2 İstisnalar

Deyim veya ifadeler söz dizimi bakımından doğru da olsalar, yürütülmeye çalışıldıklarında yine de hata oluşmasına yol açabilirler. Yürütme esnasında tespit edilen hatalara *istisna* adı verilir ve bunlar mutlak surette ölümcül değildirler. Biz birazdan Python'da bu tür hataları nasıl yakalayabileceğimizi öğreneceğiz. Ancak pek çok istisna, programlar tarafından yakalanamaz ve aşağıdaki gibi hata mesajlarının görüntülenmesine yol açar:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: division by zero
>>> 4 + çipetpet*3
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'çipetpet' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: Can't convert 'int' object to str implicitly
```

Neler olup bittiğini bize hata mesajının son satırı gösterir. İstisnaların pek çok farklı türü vardır. İstisnanın türü, mesajın bir parçası olarak ekrana basılır. Yukarıdaki örneklerde biz şu istisna türleri ile karşılaştık: `ZeroDivisionError`, `NameError` ve `TypeError`. İstisna türü olarak ekrana basılan karakter dizisi, aynı zamanda, tetiklenen gömülü istisnanın da adıdır. Bütün gömülü istisnalar bu yapıda olsa da, kullanıcı tarafından tanımlanan istisnalar bu biçime uymak zorunda değildir (ama tabii uysalar iyi olur). Standart istisna adları gömülü tanımlayıcılar arasında yer alır (ama bunlar tutulmuş (reserved) sözcüklerden değildir).

Mesaj satırının geri kalanında ise, istisna türüne ve buna neyin sebep olduğuna bağlı olarak birtakım ayrıntılar verilir.

Hata mesajından önce gelen kısım istisnanın meydana geldiği ortamı, bir yığın izi biçiminde gösterir. Bu kısım genel olarak, kaynak koddaki satırların listelendiği birtakım yığın izlerinden meydana gelir, ancak burada standart girdiden okunan satırlar görüntülenmeyecektir.

Gömülü İstisnalar bölümünde gömülü istisnalar ve bunların anlamları listelenmiştir.

8.3 İstisnaların Yakalanması

Python size, kendi belirlediğiniz istisnaları yakalayan programlar yazma imkanı sunar. Mesela aşağıdaki örneğe bakalım. Bu örnekte kullanıcıdan, girilen değer tam sayı olana kadar veri girişi yapması bekleniyor. Burada kullanıcının programı istediği zaman kesmesine de izin veriyoruz (`Control-C` ile veya işletim sistemi hangi klavye tuşlarını destekliyorsa onunla). Bu şekilde, `KeyboardInterrupt` adlı istisna türü tetiklenerek kullanıcı kaynaklı bir kesme sinyali üretilecektir.

```
>>> while True:
...     try:
...         x = int(input("Lütfen bir sayı girin: "))
...         break
...     except ValueError:
...         print("Olmadı!  Bu geçerli bir sayı değil.  Tekrar deneyin...")
...
```

`try` deyimi şöyle işler.

- İlk olarak *try cümleciği* (`try` ile `except` sözcükleri arasında kalan deyimler) işletilir.
- Eğer herhangi bir istisna meydana gelmezse *except cümleciği* es geçilir ve `try` deyiminin işleyişi tamamlanır.
- Eğer try cümleciğinin işleyişi sırasında bir istisna meydana gelirse cümleciğin geri kalanı es geçilir. Bu durumda, eğer bu istisnanın türü ile `except` sözcüğünün ardından gelen istisna adı birbirini tutuyorsa, except cümleciği işletilir ve `try` deyiminin ardından program işlemeye devam eder.

- Eğer except cümleciğindeki istisna adıyla birbirini tutmayan bir istisna meydana gelirse, daha dıştaki `try` deyimleri kontrol edilir. Eğer bu istisnayı yakalayan bir kod bulunamazsa, *yakalanmamış istisna* denen durum ortaya çıkmış olur ve daha önce gösterdiğimiz gibi programın işleyişi durur.

Farklı istisnaları yakalayabilmek için, `try` deyimleri içine birden fazla except cümleciği yerleştirebilirsiniz. Ancak bu durumda da yalnızca tek bir istisna yakalama bloğu işletilecektir. İstisna yakalama blokları yalnızca ilgili try cümleciği içinde meydana gelen istisnaları yakalayabilir. Aynı `try` deyimi içindeki öbür istisna yakalama blokları içinde meydana gelen istisnaları ise yakalayamaz. except cümlecikleri içinde parantezli demetler kullanarak birden fazla istisna adı da belirtebilirsiniz. Örneğin:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

except'ten sonra istisna adı belirtmezseniz, meydana gelebilecek bütün hata türlerini yakalarsınız. Ancak bu yöntem çok dikkatlice kullanılmalıdır, zira bu şekilde kodlarınızdaki programlama hatalarını gözden kaçırmanız işten bile değildir! İlgili istisnayı tetiklemeden önce ekrana bir hata mesajı basmak için de bu yöntemden yararlanabilirsiniz (bu yöntem ayrıca istisnayı bir fonksiyon aracılığıyla yakalamanıza da izin verir.) Mesela:

```
import sys

try:
    f = open('dosya.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as hata:
    print("İşletim sistemi hatası: {0}".format(hata))
except ValueError:
    print("Veri tam sayıya dönüştürülemedi.")
except:
    print("Beklenmeyen bir hata oluştu:", sys.exc_info()[0])
    raise
```

Tercihe bağlı olarak, `try` ... `except` deyimi içinde *else cümlecikleri* de kullanılabilir. Eğer böyle bir cümlecik kullanacaksanız, bunu bütün except cümleciklerinden sonra getirmelisiniz. Bu cümlecik, try cümleciği içinde herhangi bir istisna tetiklenmediğinde çalıştırılacak kodu belirtmek için kullanılabilir. Örneğin:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print('açılamayan dosya:', arg)
    else:
        print(arg, 'dosyası', len(f.readlines()), 'satırdan oluşmaktadır.')
        f.close()
```

`else` cümleciklerini kullanmak, `try` cümleciklerine ilave kod eklemekten iyidir, çünkü bu sayede `try`... `except` deyimi ile koruduğunuz kod bloğu içinde tetiklenmesini beklemediğiniz bir istisnayı yanlışlıkla yakalama riskini ortadan kaldırmış olursunuz.

Bir istisna ortaya çıktığında, bu istisna aynı zamanda belli bir değere de sahip olabilir. Buna o istisnanın *argümanı* adı verilir. Argümanın olup olmaması ve tipi istisnanın türüne bağlıdır.

except cümleciklerinde istisna adından sonra bir değişken de belirtilebilir. Bu değişken

istisnanın örneğine (instance) bağlanır ve argümanlar `örnek.args` yapısı içinde saklanır. Kullanım kolaylığı açısından istisnanın örneğinde `__str__()` metodu da tanımlıdır, böylece argümanlar, `.args` kısmını yazmaya gerek olmadan doğrudan ekrana basılabilir. Ayrıca istisna tetiklenmeden önce örneklendirilebilir ve istenirse buna çeşitli nitelikler de ilave edilebilir.

```
>>> try:
...     raise Exception('adana', 'mersin')
... except Exception as örnek:
...     print(type(örnek))     # istisnanın örneği
...     print(örnek.args)      # .args içinde saklanan argümanlar
...     print(örnek)           # __str__ sayesinde argümanlar doğrudan basılabilir,
...                           # istisnanın alt sınıfları aracılığıyla __str__'nin içeriğini değiştireb
...     x, y = örnek.args      # argümanları çözüyoruz
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('adana', 'mersin')
('adana', 'mersin')
x = adana
y = mersin
```

Eğer bir istisnanın argümanları varsa, bunlar yakalanmamış istisnalarda mesajın son kısmı ('ayrıntı') olarak ekrana basılır.

İstisna yakalama blokları, try cümleciğinin hemen altında gerçekleşen istisnaları yakalamakla kalmaz, aynı zamanda try cümleciği içinde (dolaylı olarak bile olsa) çağrılan fonksiyonlarda meydana gelen istisnaları da yakalar. Mesela:

```
>>> def bu_kod_hata_verir():
...     x = 1/0
...
>>> try:
...     bu_kod_hata_verir()
... except ZeroDivisionError as hata:
...     print('Çalışma zamanı hatası yakalanıyor:', hata)
...
Çalışma zamanı hatası yakalanıyor: int division or modulo by zero
```

8.4 İstisnaların Tetiklenmesi

`raise` deyimi, programcının belirli bir istisnayı tetikleyebilmesini sağlar. Mesela:

```
>>> raise NameError("SelamDostum')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: SelamDostum
```

`raise` ifadesinin aldığı argüman, tetiklenecek istisnanın ne olduğunu gösterir. Bu argüman ya bir istisna örneği ya da bir istisna sınıfı (`Exception` sınıfından türeyen bir sınıf) olmalıdır.

Bir istisnanın tetiklenip tetiklenmediğini tespit etmek istiyor, ama bu istisnayı yakalamak istemiyorsanız, `raise` deyimini tek başına kullanarak istisnayı tetikleyebilirsiniz:

```
>>> try:
...     raise NameError("SelamDostum')
... except NameError:
...     print('Buralardan bir istisna geçti!')
...     raise
...
Buralardan bir istisna geçti!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: SelamDostum
```

8.5 Kullanıcı Tanımlı İstisnalar

Yeni istisna sınıfları oluşturarak programlarınızda kendi istisnalarınızı yaratabilirsiniz (Python sınıfları hakkında daha fazla bilgi için bkz. *Sınıflar*). İstisnalar doğrudan veya dolaylı olarak `Exception` adlı sınıftan türetilmelidir. Örneğin:

```
>>> class ÖzelHata(Exception):
...     def __init__(self, değer):
...         self.değer = değer
...     def __str__(self):
...         return repr(self.değer)
...
>>> try:
...     raise ÖzelHata(2*2)
... except ÖzelHata as e:
...     print('Özel hata oluştu, değer:', e.değer)
...
Özel hata oluştu, değer: 4
>>> raise ÖzelHata('eyvah!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.ÖzelHata: 'eyvah!'
```

Bu örnekte `Exception` sınıfının öntanımlı `__init__()` metodunun üzerine yazılarak içeriği değiştirilmiştir. `__init__()` metodunun yeni hali sadece *değer* adlı bir nitelik oluşturuyor. Bu işlem, öntanımlı olarak oluşturulan *args* niteliğinin yerine geçer.

İstisna sınıfları, başka sınıfların yapabildiği her şeyi yapacak şekilde tanımlanabilir, ancak bunlar genellikle sade tutulur ve hata yakalama blokları tarafından işlenerek hata hakkında bilgi alınabilmesini sağlayan birtakım nitelikler sunmakla yetinilir. Birkaç farklı hatayı tetikleyebilen bir modül oluştururken, yaygın uygulama, o modül tarafından tanımlanan istisnalar için bir taban sınıf oluşturmak ve farklı hata koşullarına yönelik özel istisna sınıfları oluşturmak amacıyla bu sınıftan miras almaktır:

```
class Hata(Exception):
    """Bu modüldeki istisnalar için bir taban sınıf."""
    pass

class GirdiHatası(Hata):
    """Girdideki hatalar için üretilen istisna.

    Nitelikleri:
        ifade -- Hatanın oluştuğu girdi ifadesi
```

```
        mesaj -- hatanın açıklaması
    """

    def __init__(self, ifade, mesaj):
        self.ifade = ifade
        self.mesaj = mesaj

class AktarımHatası(Hata):
    """Bir işlem, izin verilmeyen bir durum aktarımı gerçekleştirmeye çalıştığında
    tetiklenir.

    Nitelikleri:
        önceki -- aktarımın başlangıcındaki durum
        sonraki -- gerçekleştirilmek istenen yeni durum
        mesaj -- İlgili aktarıma neden izin verilmediğine dair açıklama
    """

    def __init__(self, önceki, sonraki, mesaj):
        self.önceki = önceki
        self.sonraki = sonraki
        self.mesaj = mesaj
```

Çoğu istisna, standart istisna adlarına benzer bir şekilde, sonu "Error" (Hata) ile bitecek şekilde tanımlanır.

Pek çok standart modül, tanımladıkları fonksiyonlarda meydana gelebilecek hataları bildirmek için kendi istisnalarını tanımlar. Sınıflara ilişkin daha fazla bilgi *Sınıflar* bölümünde sunulmuştur.

8.6 Toparlama İşlemlerinin Tanımlanması

`try` deyimi, her koşulda işletilmesi gereken toparlama faaliyetlerini tanımlamak için tercihe bağlı başka bir cümleciğe daha sahiptir. Örneğin:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Güle güle dünya!')
...
Güle güle dünya!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
```

finally cümleciği, bir istisna oluşsun veya oluşmasın, her zaman `try` deyiminden çıkılmadan önce işletilir. `try` cümleciği içinde bir istisna oluştuğunda ve bu istisna `except` cümleciği ile yakalanmadığında (veya bu istisna `except` ya da `else` cümleciği içinde oluştuğunda), `finally` cümleciği işletildikten sonra yeniden tetiklenir. `finally` cümleciği, `break`, `continue` veya `return` deyimi aracılığıyla `try` deyimininin herhangi bir cümleciğinden çıkılırken 'çıkış yolunda' da işletilir. Daha karmaşık bir örnek verelim:

```
>>> def böl(x, y):
...     try:
...         sonuç = x / y
...     except ZeroDivisionError:
```

```
...         print("sayı sıfıra bölünemez!")
...     else:
...         print("sonuç:", sonuç)
...     finally:
...         print("finally cümleciği işletiliyor")
...
>>> böl(2, 1)
sonuç: 2.0
finally cümleciği işletiliyor
>>> böl(2, 0)
sayı sıfıra bölünemez!
finally cümleciği işletiliyor
>>> böl("2", "1")
finally cümleciği işletiliyor
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in böl
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Gördüğünüz gibi, `finally` cümleciği her ne olursa olsun işletiliyor. İki karakter dizisinin birbirine bölünmeye çalışılması ile tetiklenen `TypeError` istisnası, `except` cümleciği tarafından yakalanmadığı için, `finally` cümleciğinin yürütülmesinin ardından tetikleniyor.

Gerçek hayattaki uygulamalarda `finally` cümleciği, kaynağın başarılı bir şekilde kullanılıp kullanılamadığına bakılmaksızın, harici kaynakları (dosyalar veya ağ bağlantıları gibi) serbest bırakmak için kullanılır.

8.7 Önceden Tanımlanmış Toparlama İşlemleri

Bazı nesneler, nesneyi kullanan işlemin başarılı olup olmamasına bakılmaksızın, artık nesneye ihtiyaç kalmadığında yerine getirilecek bazı standart toparlama işlemleri tanımlar. Bir dosyayı açıp içeriğini ekrana basmaya çalışan aşağıdaki örneğe bakalım.

```
for satır in open("dosya.txt"):
    print(satır, end="")
```

Bu kodlardaki sorun, kodun bu kısmının yürütülmesi sona erdiğinde dosyayı belirsiz bir süre boyunca açık bırakıyor olmasıdır. Basit betiklerde bu durum bir sorun teşkil etmez, ancak büyük uygulamalarda başınız ağrıyabilir. İşte bu tür durumlarda `with` deyimi, dosya benzeri nesnelerin, kullanım sırasında derhal ve doğru bir şekilde toparlanmasını temin edecek biçimde kullanılmalarını sağlar.

```
with open('dosya.txt') as f:
    for satır in f:
        print(satır, end="")
```

Deyim işletildikten sonra *f* adlı dosya her halükarda kapatılacaktır. Hatta satırların işletilmesi sırasında bir problem ortaya çıksa dahi... Dosya benzeri nesnelerin, önceden tanımlanmış toparlama işlemlerine sahip olup olmadığı kendi belgelerinde belirtilecektir.

BÖLÜM 9

Sınıflar

Python'ın sınıf mekanizması, başka programlama dillerine kıyasla, söz dizimi ve anlam yapısı bakımından sınıflara dair çok az yenilik getirir. Python'daki sınıflar C++ ve Modula-3 dillerindeki sınıf yapılarının bir karışımıdır. Python sınıfları nesne tabanlı programlamanın bütün standart özelliklerini taşır. Örneğin miras alma mekanizması birden fazla taban sınıfın kullanılmasına izin verir, türetilmiş bir sınıf, miras aldığı taban sınıfın veya sınıfların herhangi bir metodunun içeriğini değiştirebilir ve metotlar, taban sınıftaki metotları aynı adla çağırabilir. Nesneler sınırsız sayıda ve türde veri içerebilir. Tıpkı modüller gibi, sınıflar da Python'ın dinamik yapısından payına düşeni alır. Buna göre, sınıflar çalışma zamanında oluşturulur ve oluşturulduktan sonra bunlar üzerinde değişiklik yapılabilir.

C++ terimleriyle konuşacak olursak, normalde sınıf üyeleri (veri üyeleri de buna dahil olmak üzere) 'public', yani *dışarıya açık* (bunun istisnası için bkz. *Gizli Değişkenler*), bütün üye fonksiyonlar ise 'virtual', yani *soyuttur*. Modula-3'te olduğu gibi, nesnenin üyelerine metotlar içinden erişmenin basit bir yolu yoktur. Metot fonksiyonunun ilk argümanı nesneyi temsil eder ve bu argüman, fonksiyonun tanımlanması esnasında açıkça belirtilir. Ancak fonksiyon çağrılırken bu ilk argüman gizlidir. Smalltalk'ta olduğu gibi, sınıfların kendisi de birer nesnedir. Bu özellik, içe aktarma ve yeniden adlandırma işlevlerinin anlamsal altyapısını oluşturur. C++ ve Modula-3'ün aksine gömülü tipler kullanıcı tarafından taban sınıf şeklinde miras alınıp genişletilebilir. Ayrıca, C++'taki gibi, özel bir söz dizimine sahip çoğu gömülü işleç (aritmetik işleçler, sıra belirtme işleçleri, vb.) sınıf örneklerinde kullanılmak üzere yeniden tanımlanabilir.

(Genel kabul görmüş terminoloji eksikliğinden ötürü, sınıflardan bahsederken zaman zaman Smalltalk ve C++ kavramlarını kullanacağız. Nesne tabanlı programlamaya ilişkin anlam yapısı C++'a kıyasla Python'a daha yakın olduğu için Modula-3 kavramlarını kullanabilirdik, ama muhtemelen bu dili çok az kişi duymuştur.)

9.1 İsimler ve Nesneler Hakkında Birkaç Söz

Her nesne benzersiz olup, aynı nesneye (birden fazla etki alanı içinde) birden fazla isim bağlanabilir. Başka dillerde buna 'farklı adlandırma' (aliasing) diyorlar. Python'ın bu özelliği ilk bakışta pek takdir edilmeyebilir. Zaten değiştirilemeyen temel veri tipleri (sayılar, karakter dizileri, demetler) ile çalışırken bu özelliği gönül rahatlığıyla görmezden gelebilirsiniz. Ancak farklı adlandırma; listeler, sözlükler ve diğer pek çok veri tipi gibi değiştirilebilen nesneler söz konusu olduğunda, Python kodlarının anlam yapısı üzerinde muhtemelen sizi şaşırtacak bir etki ortaya çıkaracaktır. Farklı adlandırma bazı yönlerden işaretçilere (pointer) benzediği için bu özelliği kullanmak genellikle programın yararınadır. Örneğin, bir nesneyi argüman olarak atamak, gerçekleme gereği atanan şey yalnızca bir işaretçi olduğu için, hantal bir işlem

değildir. Argüman olarak atanan bu nesne bir fonksiyon tarafından değişikliğe uğratılırsa, nesneyi çağıran fonksiyon bu değişikliği görecektir. Bu özellik sayesinde, Pascal'daki gibi iki farklı argüman atama mekanizmasına gerek duyulmamış olur.

9.2 Python'da Etki ve İsim Alanları

Sınıflara geçmeden önce Python'daki etki alanı kurallarına ilişkin birkaç söz söylesek iyi olacak. Sınıf tanımlamaları sırasında isim alanları üzerinde birtakım ince işler döner. Neler döndüğünü bütünüyle kavrayabilmek için de etki alanlarının ve isim alanlarının nasıl işlediğini bilmeniz gerekir. Bu arada, bu konuyu bilmek ileri düzey Python programcıları açısından da faydalıdır.

İşe bazı tanımlarla başlayalım.

İsim alanı isimler ile nesneler arasında bir eşleştirmedir. Çoğu isim alanı, mevcut durumda Python sözlükleri olarak gerçeklenmiştir, ancak bu durumu herhangi bir şekilde farketmeniz mümkün değildir (performans dışında). Ayrıca bu yapı ileride değişebilir de. İsim alanlarına örnek olarak şunları verebiliriz: `abs()` gibi fonksiyonları ve gömülü istisna adlarını da içine alan gömülü isimler, bir modül içinde yer alan global isimler ve bir fonksiyon çağrıldığında oluşturulan lokal isimler. Bir nesnenin sahip olduğu nitelikler de bir bakıma birer isim alanı oluşturur. İsim alanları ile ilgili olarak bilinmesi gereken önemli bir nokta da farklı isim alanlarında yer alan isimler arasında kesinlikle herhangi bir bağlantı olmamasıdır. Örneğin iki farklı modül içinde `büyüt` adlı bir fonksiyon tanımlayabilirsiniz. Bunlar birbirine karışmaz, çünkü bu modülleri kullananlar, fonksiyonun başına modül adını da ekleyeceklerdir.

Bu arada *nitelik* kavramını, noktadan sonra gelen her türlü isim için kullanıyoruz. Örneğin, `z.real` adlı ifadede `real` kelimesi, `z` nesnesinin bir niteliğidir. Doğrusunu söylemek gerekirse, modüllerin içindeki isimlere atıfta bulunduğunuzda aslında niteliklere atıfta bulunmuş olursunuz. Mesela `modüladı.fonksiyonadı` ifadesinde `modüladı` bir modül nesnesi, `fonksiyonadı` ise bunun niteliğidir. Bu durumda, modülün nitelikleri ile modülde tanımlanan global isimler arasında birebir ilişki söz konusudur. Zira bunlar birbiriyle aynı isim alanını paylaşır! [1]

Nitelikler salt okunur olabileceği gibi bunların üzerinde değişiklik de yapılabilir. Değiştirilebilme özelliğini kullanarak, niteliklere birtakım değerler atayabilirsiniz. Modül niteliklerinin değiştirilebilir özellikte olması sayesinde `modüladı.cevap = 42` gibi bir kod yazmak mümkündür. Üzerinde değişiklik yapılabilen bu nitelikleri aynı zamanda `del` deyimi ile silebilirsiniz de. Örneğin `del modüladı.cevap` komutu, `modüladı` olarak adlandırılan nesneden `cevap` niteliğini silecektir.

İsim alanları farklı anlarda oluşturulur ve bunların ömrü birbirinden farklıdır. Gömülü isimleri içeren isim alanı, Python yorumlayıcısı başlatıldığında oluşturulur ve asla silinmez. Bir modülün global isim alanı modül tanımı okunduğunda oluşturulur. Normal şartlar altında modüllere ait isim alanları da yorumlayıcıdan çıkılana kadar varolmaya devam eder. Yorumlayıcının en tepesinde işletilen deyimler, bunlar bir betik dosyasından okunmuş veya etkileşimli olarak çalıştırılmış da olsa, `__main__` adlı bir modülün parçası olarak kabul edilir. Dolayısıyla bunların da kendilerine ait bir global isim alanı vardır. (Gömülü isimler de aslında ayrı bir modül içinde yer alır. Bu modüle `builtins` adı verilir.)

[1] Bir şey hariç. Modül nesnelerinin, modülün isim alanını gerçeklemek için kullanılan sözlüğü döndüren `__dict__` adında, gizli bir salt okunur niteliği vardır. `__dict__` bir nitelik olsa da global bir isim değildir. Açıktır ki, bu niteliği kullanmak isim alanı gerçeklemesine ilişkin soyutlamanın ruhuna aykırıdır. Çökme sonrası (post-mortem) hata ayıklama gibi işlemler dışında bu niteliğin kullanımından kaçınılmalıdır.

Bir fonksiyonun lokal isim alanı fonksiyon çağrıldığında oluşturulur ve fonksiyon bir değer döndürdüğünde veya fonksiyon içinde yakalanmayan bir istisna tetiklendiğinde silinir. (Aslında burada olan şeyi 'silmek' yerine 'unutmak' diye tabir etmek daha doğru olurdu.) Elbette, özyinelemeli fonksiyonların da kendilerine ait bir lokal isim alanı vardır.

Bir Python programında, bir isim alanının doğrudan erişilebilir durumda olduğu kod bölgesine *etki alanı* adı verilir. Burada 'doğrudan erişilebilir' ifadesi ile kastedilen şey, başına modül adı gibi bir tanımlayıcı getirilmeden çağrılan bir ismin, isim alanı içinden erişilebilir durumda olmasıdır.

Etki alanları statik olarak belirlense de, dinamik olarak kullanılır. Çalışma esnasında herhangi bir zamanda, isim alanlarına doğrudan erişebileceğiniz, iç içe geçmiş en az üç etki alanı bulunur:

- En iç taraftaki etki alanı: Bu etki alanı arama sırasında ilk bakılan yerdir ve lokal isimleri içerir.
- Dış taraftaki bütün fonksiyonların etki alanları: Bu etki alanı, arama sırasında en yakın etki alanından başlanarak dışa doğru taranır ve lokal olmayan isimlerle birlikte global olmayan isimleri de içerir.
- Sondan bir önceki etki alanı: Bu etki alanı mevcut modülün global isimlerini içerir.
- En dış etki alanı: Bu etki alanı gömülü isimleri barındırır ve en son taranır.

Eğer bir isim global olarak bildirilirse, bu isme yapılan bütün atıflar ve atamalar doğrudan modülün global isimlerini içeren orta etki alanına yönlendirilir. En içteki etki alanının dışında yer alan değişkenleri yeniden bağlamak için `nonlocal` adlı deyim kullanılabilir. Bu değişkenler, eğer nonlocal olarak bildirilmezlerse salt okunur olurlar (bu değişkenin değeri değiştirilmeye çalışıldığında en iç etki alanında *yeni* bir lokal değişken oluşturulur ve dış taraftaki aynı adlı değişken değişmeden kalır).

Genellikle lokal etki alanı, (kod metnindeki) mevcut fonksiyonda yer alan lokal isimlere atıfta bulunur. Fonksiyonların dış tarafındaki lokal etki alanı, global etki alanıyla aynı isim alanına, yani modülün isim alanına atıfta bulunur. Bir sınıf tanımlandığında da, lokal etki alanına yeni bir isim alanı daha eklenmiş olur.

Bir fonksiyonun etki alanının, o fonksiyonun yazıldığı kod bölgesinde belirlendiğini anlamak önemlidir: Fonksiyon nereden veya hangi adla çağrılmış olursa olsun, eğer bir fonksiyon bir modül içinde tanımlanmışsa, o fonksiyonun global etki alanı o modülün isim alanıdır. Öte yandan, asıl isim araması, çalışma esnasında dinamik olarak gerçekleştirilir. Ancak dilin tanımı, 'derleme' sırasında statik isim çözümlemeye doğru evrilmektedir. Dolayısıyla dinamik isim çözümlemeye bel bağlamasanız iyi olur! (Aslında lokal değişkenler şimdiden statik olarak belirlenmektedir.)

Python'a özgü bir tuhaflık, eğer herhangi bir `global` deyimi etkin değilse, isimlere değer atama işleminin her zaman en içteki etki alanına yönlenmesidir. Değer atamaları verileri kopyalamaz, bunlar yalnızca isimleri nesnelere bağlar. Aynı şey silme işlemleri için de geçerlidir: `del x` deyimi, lokal etki alanının atıfta bulunduğu isim alanıyla `x`'in bağlantısını keser. Aslında yeni isimler oluşturan bütün işlemler lokal etki alanını kullanır. Özellikle de `import` deyimleri ve fonksiyon tanımları modül veya fonksiyon adını lokal etki alanına bağlar.

`global` deyimi, belirli değişkenlerin global etki alanında yer aldığını ve oraya bağlanması gerektiğini belirtmek için kullanılabilir. `nonlocal` deyimi ise belirli değişkenlerin, dış taraftaki bir etki alanında yer aldığını ve oraya bağlanması gerektiğini belirtir.

9.2.1 Etki ve İsim Alanlarına Örnek

Aşağıdaki örnek; farklı etki ve isim alanlarına nasıl atıfta bulunulacağı ile birlikte, `global` ve `nonlocal` sözcüklerinin, değişkene değer bağlama işlemlerini nasıl etkilediğini gösteriyor:

```
def etki_alanı_testi():
    def local_olarak_ata():
        değişken = "local değişken"
    def nonlocal_olarak_ata():
        nonlocal değişken
        değişken = "nonlocal değişken"
    def global_olarak_ata():
        global değişken
        değişken = "global değişken"
    değişken = "değişken testi"
    local_olarak_ata()
    print("local olarak belirlendikten sonra:", değişken)
    nonlocal_olarak_ata()
    print("nonlocal olarak belirlendikten sonra:", değişken)
    global_olarak_ata()
    print("global olarak belirlendikten sonra:", değişken)

etki_alanı_testi()
print("global etki alanındayken:", değişken)
```

Örnek kodun çıktısı şöyle olur:

```
local olarak belirlendikten sonra: değişken testi
nonlocal olarak belirlendikten sonra: nonlocal değişken
global olarak belirlendikten sonra: nonlocal değişken
Global etki alanındayken: global değişken
```

local değer atama işleminin *etki_alanı_testi* içindeki *değişken*'in değerini değiştirmediğine dikkat edin (öntanımlı davranış budur) . `nonlocal` değer atama işlemi *etki_alanı_testi* içindeki *değişken*'in değerini değiştirir, `global` değer atama işlemi ise değişkenin modül seviyesindeki değerini değiştirir.

Ayrıca `global` değer atama işleminden önce *değişken* için önceden bağlanmış bir değer olmadığını da görüyorsunuz.

9.3 Sınıflara İlk Bakış

Python'daki sınıflar; birkaç yeni söz dizimi, üç adet yeni nesne tipi ve bir miktar da yeni anlam yapısıyla karşımıza çıkar.

9.3.1 Sınıf Tanımlamanın Söz Dizimi

Sınıflar en basit haliyle şöyle tanımlanır:

```
class SınıfAdı:
    <deyim-1>
    .
    .
```

```
    .
    <deyim-N>
```

Fonksiyon tanımlarında olduğu gibi (`def` deyimleri), sınıf tanımlarının da herhangi bir etkiye sahip olabilmesi için öncelikle yürütülmesi gerekir. (Arzu ederseniz sınıf tanımlarını `if` deyimleri veya fonksiyon gövdeleri içine de yerleştirebilirsiniz.)

Uygulamada, bir sınıf tanımı içindeki deyimler genellikle fonksiyon tanımları olacaktır, ancak burada öteki deyimlerin kullanılmasına da izin verilir. Üstelik bazen bunun yararını da görürsünüz. Bu konuya daha sonra geleceğiz. Normal şartlar altında, sınıf içindeki fonksiyon tanımları, metotlara ilişkin çağırma kurallarınca belirlenen, özel bir argüman listesi biçimine sahiptir. Buna da daha sonra değineceğiz.

Bir sınıfın tanımı okunduğunda yeni bir isim alanı oluşturulur ve bu isim alanı o sınıfın lokal etki alanı olur. Dolayısıyla lokal değişkenlere yapılan bütün atamalar bu yeni isim alanına yönelir. Özellikle de sınıf içinde bir fonksiyon tanımlandığında bu yeni fonksiyonun adı bu etki alanına bağlanır.

Sınıf tanımından normal olarak çıkıldığında (sınıfın sonuna erişildiğinde), bir *sınıf nesnesi* oluşturulur. Bu nesne temel olarak, sınıfın tanımlanması ile oluşturulan isim alanının içeriğini tıpkı bir kapsül gibi sarmalayan bir yapıdır. Bir sonraki bölümde sınıf nesneleri hakkında daha fazla bilgi vereceğiz. Sınıftan çıkıldığında, sınıf tanımlanmadan hemen önce etkin durumda olan asıl lokal etki alanı yeniden etkin duruma getirilir ve sınıf nesnesi, bu etki alanı içinde, sınıf tanımı satırında belirtilen sınıf adına bağlanır (yukarıdaki örnekte `SınıfAdı`).

9.3.2 Sınıf Nesneleri

Sınıf nesneleri iki işlem türünü destekler: niteliklere atıf ve örneklendirme.

Niteliklere atıf, bütün nitelik atıfları için kullanılan standart söz dizimini, yani `nesne.ad` yapısını kullanır. Geçerli nitelik adları, sınıf nesnesi oluşturulduğu sırada sınıfın isim alanında bulunan bütün adlardır. Dolayısıyla eğer sınıf tanımı şöyle görünüyorsa:

```
class Sınıfım:
    """Basit bir örnek sınıf"""
    i = 12345
    def f(self):
        return 'merhaba dünya'
```

Bu durumda `Sınıfım.i` ve `Sınıfım.f`, sırasıyla bir tam sayı ve bir fonksiyon nesnesi döndüren, geçerli nitelik atıfları olacaktır. Sınıf niteliklerine değer de atanabilir, dolayısıyla `Sınıfım.i`'nin değerini atama yoluyla değiştirebilirsiniz. `__doc__` da, sınıfa ait belge dizisini döndüren geçerli bir niteliktir. Yukarıdaki örnekte bu nitelik `"Basit bir örnek sınıf"` değerini döndürür.

Sınıf *örneklendirme* işleminde fonksiyonlara ilişkin gösterim şekli kullanılır. Sınıf nesnesinin, sınıfın yeni bir örneğini döndüren parametresiz bir fonksiyon olduğunu varsayabilirsiniz. Mesela (yukarıdaki sınıftan hareketle):

```
x = Sınıfım()
```

Bu kod, sınıfın yeni bir *örneğini* oluşturur ve bu nesneyi `x` adlı bir lokal değişkene atar.

Örneklendirme işlemi (yani bir sınıf nesnesinin 'çağrılması') boş bir nesne oluşturur. Sınıf nesnelerini, belirli bir başlangıç değerine sahip olacak şekilde oluşturmak yaygın bir uygulamadır. Bu amaçla, sınıf içinde `__init__()` adlı özel bir metot tanımlanabilir. Şöyle:

```
def __init__(self):
    self.veriler = []
```

Bir sınıfta `__init__()` metodu tanımlandığında sınıf örneklendirme işlemi, yeni oluşturulan sınıf örneği için `__init__()` metodunu otomatik olarak çalıştırır. Dolayısıyla bir başlangıç değerine sahip yeni bir sınıf örneği şöyle elde edilebilir:

```
x = Sınıfım()
```

Tabii ki, daha fazla esneklik sağlamak için `__init__()` metoduna argüman da verilebilir. Bu durumda sınıf örneklendirme işlecine atanan argümanlar `__init__()` metoduna gönderilir. Örneğin:

```
>>> class KarmaşıkSayı:
...     def __init__(self, gerçelkısım, sanalkısım):
...         self.r = gerçelkısım
...         self.i = sanalkısım
...
>>> x = KarmaşıkSayı(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3 Örnek Nesneleri

Peki örnek nesneleri ile neler yapabiliriz? Örnek nesnelerinin tanıdığı tek işlem türü nitelik atıflarıdır. İki tür geçerli nitelik adı vardır: veri nitelikleri ve metotlar.

Veri nitelikleri Smalltalk'taki 'örnek değişkenlerine', C++'taki 'veri üyelerine' karşılık gelir. Veri niteliklerinin önceden bildirilmesine gerek yoktur. Tıpkı lokal değişkenler gibi, bunlar da bir başlangıç değeri atanır atanmaz varlık sahasına çıkar. Mesela `x`, yukarıda oluşturulan `Sınıfım` adlı sınıfın bir örneği ise, aşağıdaki kod parçası `16` değerini ekrana basacak ve ardında hiçbir iz bırakmadan kaybolacaktır:

```
x.sayaç = 1
while x.sayaç < 10:
    x.sayaç = x.sayaç * 2
print(x.sayaç)
del x.sayaç
```

Örneklere ilişkin başka bir nitelik atfı da *metottur*. Metot, bir nesneye 'ait' bir fonksiyondur. (Python'da metot kavramı sınıf örneklerine özgü değildir. Başka nesne tipleri de metotlara sahip olabilir. Örneğin liste nesnelerinin append, insert, remove, sort ve bunun gibi metotları vardır. Ancak aşağıdaki tartışmada, aksi açıkça belirtilmedikçe, metot kavramını, yalnızca sınıf örneği nesnelerinin metotları anlamına gelecek şekilde kullanacağız.)

Bir örnek nesnesinin geçerli metot adlarının neler olduğu onun sınıfına bağlıdır. Tanım gereği, bir sınıfın, fonksiyon nesnesi olan bütün nitelikleri, aynı zamanda o sınıfın örneklerinin metotlarıdır. Dolayısıyla bizim örneğimizde, `Sınıfım.f` bir fonksiyon olduğu için, `x.f` geçerli bir metot atfıdır, ama `Sınıfım.i` bir fonksiyon olmadığı için, `x.i` geçerli bir metot atfı değildir. Ancak `x.f` ile `Sınıfım.f` aynı şeyler değildir. `x.f` bir *metot nesnesidir*, fonksiyon nesnesi değil.

9.3.4 Metot Nesneleri

Genellikle bir metot, bağlandıktan hemen sonra çağrılır:

```
x.f()
```

`Sınıfım` örneğinde bu, `'merhaba dünya'` karakter dizisini döndürür. Ancak bir metodu hemen çağırmak gerekli değildir: `x.f` bir metot nesnesidir. Bu nesne daha sonraki bir zamanda çağrılmak üzere saklanabilir de. Mesela:

```
xf = x.f
while True:
    print(xf())
```

Bu kodlar sonsuza kadar `merhaba dünya` basmaya devam edecektir.

Peki bir metot çağrıldığında tam olarak neler olur? `f()` metodunun fonksiyon tanımında bir argüman tanımlanmış olsa da, yukarıda `x.f()`'nin argümansız çağrıldığını farketmişsinizdir. Peki argümana ne oldu? Elbette, bir argüman bekleyen bir fonksiyon argümansız çağrılırsa Python bir istisna tetikler. Hatta argüman aslında kullanılmıyor bile olsa...

Aslında cevabı tahmin etmişsinizdir: metotlara ilişkin özel bir durum, nesnenin kendisinin, fonksiyona ilk argüman olarak atanıyor olmasıdır. Örneğimizde `x.f()` çağrısı `Sınıfım.f(x)` ile birebir aynıdır. Genel olarak, metodun *n* sayıda argüman listesi ile çağrılması; ilgili fonksiyonun, metot nesnesinin ilk argümanın önüne yerleştirilmesi ile oluşturulan bir argüman listesi ile çağrılmasıyla aynı şeydir.

Eğer metotların nasıl işlediğini hala anlamadıysanız, gerçeklemeye bakmak işleri belki açıklığa kavuşturabilir. Veri niteliği olmayan bir örnek niteliğine atıfta bulunduğunuzda bunun ait olduğu sınıfta bir arama işlemi gerçekleştirilir. Eğer isim, geçerli bir sınıf niteliğine, yani bir fonksiyon nesnesine işaret ediyorsa, örnek nesnesi ile arama sırasında bulunan fonksiyon nesnesi (daha doğrusu bu nesnenin işaretçileri) bir soyut nesne içinde birleştirilerek bir metot nesnesi oluşturulur. Buna metot nesnesi denir. Metot nesnesi bir argüman listesi ile çağrıldığında örnek nesnesi ile argüman listesi kullanılarak yeni bir argüman listesi meydana getirilir ve fonksiyon nesnesi bu yeni argüman listesi ile çağrılır.

9.3.5 Sınıf ve Örnek Değişkenleri

Genel olarak konuşmak gerekirse, örnek değişkenleri, her bir örneğe özgü veriler içinken; sınıf değişkenleri ise sınıfın bütün örnekleri tarafından paylaşılan nitelikler ve metotlar içindir:

```
class Köpek:

    tür = 'kanin' # bütün örnekler tarafından paylaşılan sınıf değişkeni

    def __init__(self, isim):
        self.isim = isim # her bir örneğe özgü olan örnek değişkeni

>>> d = Köpek('Fino')
>>> e = Köpek('Badem')
>>> d.tür # bütün köpeklerce paylaşılır
'kanin'
>>> e.tür # bütün köpeklerce paylaşılır
'kanin'
>>> d.isim # d'ye özgüdür
'Fino'
>>> e.isim # e'ye özgüdür
'Badem'
```

İsimler ve Nesneler Hakkında Birkaç Söz bölümünde tartışıldığı gibi, listeler ve sözlükler gibi değiştirilebilen nesneler söz konusu olduğunda paylaşılan veriler şaşırtıcı etkiler gösterebilir. Örneğin, aşağıdaki kodlarda yer alan *numaralar* listesi bir sınıf değişkeni olarak kullanılmamalıdır, çünkü bu durumda tek bir liste bütün *Köpek* örnekleri tarafından paylaşılacaktır:

```
class Köpek:

    numaralar = [] # bir sınıf değişkeninin yanlış kullanımı

    def __init__(self, isim):
        self.isim = isim

    def numara_ekle(self, numara):
        self.numaralar.append(numara)

>>> d = Köpek('Fino')
>>> e = Köpek('Badem')
>>> d.numara_ekle('yerde yuvarlan')
>>> e.numara_ekle('ölü taklidi yap')
>>> d.numaralar # beklenmedik şekilde bütün köpekler tarafından paylaşılır
['yerde yuvarlan', 'ölü taklidi yap']
```

Sınıfın tasarımının doğru olabilmesi için, yukarıdaki yerine, örnek değişkeni kullanılmalıdır:

```
class Köpek:

    def __init__(self, isim):
        self.isim = isim
        self.numaralar = [] # her köpek için yeni, boş bir liste oluşturur

    def numara_ekle(self, numara):
        self.numaralar.append(numara)

>>> d = Köpek('Fino')
>>> e = Köpek('Badem')
>>> d.numara_ekle('yerde yuvarlan')
>>> e.numara_ekle('ölü taklidi yap')
>>> d.numaralar
['yerde yuvarlan']
>>> e.numaralar
['ölü taklidi yap']
```

9.4 Çeşitli Notlar

Veri nitelikleri, aynı ada sahip metot niteliklerinin üzerine yazar. Büyük programlarda tespit edilmesi güç hatalara yol açabilecek istenmeyen isim çakışmalarını engellemek için, çakışma ihtimalini en aza indirmenizi sağlayacak bir kural belirlemeniz akıllıca olacaktır. Belirleyebileceğiniz kurallara örnek olarak; metot adlarının büyük harfle yazılması, veri niteliği adlarına benzersiz bir karakter dizisi şeklinde (belki de yalnızca bir alt çizgi) önek getirilmesi veya metotlar için fiil, veri nitelikleri için ise isimlerin kullanılması verilebilir.

Veri niteliklerine, metotların yanısıra, ilgili nesneyi kullanan herkes ('istemciler') tarafından atıfta bulunulabilir. Diğer bir deyişle, sınıflar saf soyut veri tipleri gerçeklemek için kullanılamaz. Aslında Python'daki hiçbir şey verilerin gizlenebilmesine imkan tanımaz.

Verilerin gizlenmesi tamamen üzerinde uzlaşılmış birtakım kurallara dayanır. (Öte yandan, C ile yazılmış olan Python gerçeklemesi, gerektiğinde gerçekleme ayrıntılarını tamamen gizleyip bir nesneye erişimi kontrol edebilir. C ile yazılmış Python eklentileri bu özellikten yararlanabilir.)

İstemciler veri niteliklerini dikkatli kullanmalıdır. Zira istemciler, metotlara ait sabitleri ve bunların veri niteliklerini değişikliğe uğratarak bunları kullanılmaz hale getirebilir. İstemciler, isim çakışmalarından kaçındıkları müddetçe, metotların geçerliliğini etkilemeden, örnek nesnelerine kendi veri niteliklerini ekleyebilir. Burada da isimlendirmeye ilişkin birtakım kurallar belirlemek başınızın ağrımasını engelleyebilir.

Metotlar içinden veri niteliklerine (ve başka metotlara!) atıfta bulunmanın basit bir yolu yoktur. Bizce bu durum metotların okunaklılığını artırmaktadır: Böylece, bir metodu incelerken lokal değişkenlerle örnek değişkenlerini birbirine karıştırmazsınız.

Bir metodun ilk argümanı genellikle `self` olarak adlandırılır. Bu genel kabul görmüş bir adlandırmadan öte bir şey değildir. `self` adının Python açısından özel hiçbir anlamı yoktur. Ancak eğer bu yerleşik kuralı takip etmezseniz kodlarınızın başka Python programcıları tarafından okunması zorlaşabilir. Ayrıca bazı *sınıf tarama* programları da bu yerleşik kuralı temel alacak şekilde yazılmış olabilir.

Bir sınıf niteliği durumundaki herhangi bir fonksiyon nesnesi, o sınıfın örnekleri için bir metot olma vazifesi görür. Fonksiyon tanımının sınıf tanımı içinde geçmesi zorunlu değildir. İlgili fonksiyon nesnesini sınıf içinde bir lokal değişkene atamak da kabul edilebilir. Mesela:

```
# Sınıfın dışında tanımlanan fonksiyon
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'merhaba dünya'
    h = g
```

Burada `f`, `g` ve `h`'nin hepsi `C` sınıfının, fonksiyon nesnelerine atıfta bulunan nitelikleridir ve sonuç olarak bunların hepsi `C` sınıfının örnekleri için birer metot olma vazifesi görür. Burada `h` ile `g` ise birbiriyle tamamen aynıdır. Ancak bu kullanım, programı okuyanların kafasını karıştırmaktan başka bir işe yaramayacaktır.

Metotlar, `self` argümanının metot niteliklerini kullanarak başka metotları çağırabilir:

```
class Çanta:
    def __init__(self):
        self.veriler = []
    def ekle(self, x):
        self.veriler.append(x)
    def ikikezekle(self, x):
        self.ekle(x)
        self.ekle(x)
```

Metotlar, sıradan fonksiyonlar gibi global isimlere atıfta bulunabilir. Bir metodun global etki alanı, o metodun tanımını içeren modüldür. (Bir sınıf asla global etki alanı olarak kullanılmaz.) Bir metot içinde global veri kullanmanın pek anlamı olmasa da, global etki alanı kullanımının pek çok geçerli sebebi vardır. Bir defa, global etki alanına aktarılan fonksiyonlar ve modüller, metotların yanısıra, fonksiyonlar ve bunun içinde tanımlanan sınıflar tarafından da kullanılabilir. Genellikle metodu içeren sınıfın kendisi bu global etki

alanında tanımlanmıştır. Bir sonraki bölümde bir metodun neden kendi sınıfına atıfta bulunmak isteyeceğine dair iyi sebepler göstereceğiz.

Her değer bir nesnedir, dolayısıyla her değerin bir *sınıfı* vardır (*tipi* de denebilir). Bu değer, `nesne.__class__` olarak saklanır.

9.5 Miras Alma

Elbette bir dilde 'sınıf' diye adlandırılan bir özellik eğer miras almayı desteklemiyorsa, o özelliğe 'sınıf' demek pek yerinde bir adlandırma olmayacaktır. Türetilmiş bir sınıf tanımının söz dizimi şöyle görünür:

```
class TüretilmişSınıfAdı(TabanSınıfAdı):
    <deyim-1>
    .
    .
    .
    <deyim-N>
```

`TabanSınıfAdı` adı, türetilmiş sınıf tanımını içeren etki alanı içinde tanımlanmalıdır. Taban sınıf adı yerine herhangi bir başka ifade de kabul edilir. Bu özellik, örneğin, taban sınıf başka bir modül içinde tanımlanmışsa faydalı olacaktır:

```
class TüretilmişSınıfAdı(modüladı.TabanSınıfAdı):
```

Türetilmiş bir sınıfın tanımı tıpkı taban sınıfınki gibi işler. Sınıf nesnesi oluşturulduğunda taban sınıf hatırlanır. Bu özellik nitelik atıflarını çözmek için kullanılır: Eğer talep edilen bir nitelik sınıf içinde bulunamazsa, arama taban sınıf içinde ilerler. Eğer taban sınıfın kendisi başka bir sınıftan türetilmişse bu kural özyinelemeli olarak işletilir.

Türetilmiş sınıfların örneklendirilmesi konusunda özel bir durum yoktur: `TüretilmişSınıfAdı()` sınıfın yeni bir örneğini oluşturur. Metot atıfları şöyle çözülür: İlgili sınıf niteliği, gerekiyorsa taban sınıf zincirinde aşağı doğru inilerek aranır ve eğer bu işlem bir fonksiyon nesnesi verirse metot atfı geçerli olmuş olur.

Türetilmiş sınıflar taban sınıflarının metotlarının üzerine yazabilir. Çünkü metotlar; aynı sınıf içindeki başka bir metodu çağırırken herhangi bir öncelik elde etmez. Aynı taban sınıf içinde tanımlanmış başka bir metodu çağıran bir taban sınıfın bir metodu, onun üzerine yazan bir türetilmiş sınıfın metodunu çağırabilir. (C++ programcıları için; Python'daki bütün metotlar esasında `virtual`, yani soyuttur.)

Taban sınıf içindeki bir metodu değişikliğe uğratan türetilmiş sınıf içindeki bir metot aslında aynı isimli taban sınıf metodunu değiştirmek yerine onu genişletmek istiyor olabilir. Taban sınıf metodunu doğrudan çağırmanın basit bir yolu şudur: `TabanSınıfAdı.metotadı(self, argümanlar)` kodunu çalıştırmak. Bu, zaman zaman istemciler için de faydalıdır. (Ancak bu kodlar yalnızca taban sınıf `TabanSınıfAdı` olarak global etki alanından ulaşılabilir olduğunda çalışır.)

Python'ın miras alma ile birlikte çalışan iki adet gömülü fonksiyonu vardır:

- Bir örneğin tipini kontrol etmek için `isinstance()` kullanılır: `isinstance(obj, int)` kodu, `nesne.__class__` `int` veya `int`'ten türeyen başka bir sınıf olduğunda `True` olacaktır.

- Sınıfın miras alınmış olup olmadığını kontrol etmek için `issubclass()` kullanılır: `issubclass(bool, int)` kodu, `bool` `int`'in bir alt sınıfı olduğu için `True`'dur. Ancak, `issubclass(float, int)` `False`'tur, çünkü `float` `int`'in bir alt sınıfı değildir.

9.5.1 Çoklu Miras Alma

Python, bir tür çoklu miras almayı da destekler. Birden fazla taban sınıfa sahip bir sınıf şöyle tanımlanır:

```
class TüretilmişSınıfAdı(Taban1, Taban2, Taban3):
    <deyim-1>
    .
    .
    .
    <deyim-N>
```

En basit durumlarda, çoğu amaç için, ebeveyn sınıftan miras alınan niteliklerin aranması işlemini derinlik öncelikli, soldan sağa, hiyerarşide bir çakışma olduğu durumlarda aynı sınıf içinde iki kez arama yapılmayan bir yapı olarak düşünebilirsiniz. Dolayısıyla, eğer bir nitelik `TüretilmişSınıfAdı` adlı sınıfta bulunamazsa, bu nitelik `Taban1` sınıfında aranacak, daha sonra (özyinelemeli olarak) `Taban1` adlı sınıfın taban sınıflarına bakılacak ve eğer orada da bulunamazsa, `Taban2` adlı sınıfa ve diğerlerine geçilecektir.

Esasında, durum bundan biraz daha karmaşıktır. Metot çözümleme sırası, ortak `super()` fonksiyonu çağrılarını desteklemek için dinamik olarak değişecektir. Bu yaklaşım çoklu miras almayı destekleyen bazı başka dillerde 'call-next' metodu olarak bilinir ve tek miras almayı destekleyen dillerdeki `super()` çağrılarından daha güçlüdür.

Çoklu miras almaya ilişkin bütün durumlar bir veya daha fazla elmas yapısı gösterdiği için (ebeveyn sınıfların en az bir tanesinin, en dip sınıftan çoklu yollar aracılığıyla erişilebildiği durum) dinamik sıralama gereklidir. Örneğin, bütün sınıflar `object` sınıfından miras alır. Dolayısıyla çoklu mirasa ilişkin her durum, `object` sınıfına erişmek için birden fazla yol sunar. Taban sınıflara birden fazla sayıda erişilmesini engellemek için, dinamik algoritma, arama işlemini, her bir sınıfta belirlenen soldan sağa sıralamayı muhafaza edecek, ebeveyn sınıfı yalnızca bir kez çağıracak ve monotonik (yani bir sınıfın, ebeveynlerin öncelik sırasını etkilemeden alt sınıf olarak çağrılabilmesi) olacak şekilde doğrusallaştırır. Bu özellikler, birlikte ele alındıklarında, çoklu miras alma özelliğini destekleyen güvenilir ve genişletilebilir sınıflar tasarlanmasını mümkün kılar. Daha fazla ayrıntı için bkz. https://www.python.org/download/releases/2.3/mro/.

9.6 Gizli Değişkenler

Nesne dışından erişilemeyen 'gizli' örnek değişkenleri Python'da bulunmaz. Ancak pek çok Python programcısı tarafından uyulan genel kabul görmüş bir kural vardır: Buna göre, başında bir alt çizgi bulunan isimler (örn. `_falanca`) API'nin dışarıya kapalı kısmı olarak değerlendirilir (bu bir fonksiyon, metot veya veri üyesi olabilir). Bu isimler, gerçeklemeye ilişkin bir teferruat olarak kabul edilir ve her an değişebilecekleri varsayılır.

Sınıf içine gizlenmiş üyeler kullanmayı gerektiren bazı durumlar olabileceği için (alt sınıflar tarafından tanımlanan isimlerle isim çakışmasını önlemek gibi), *isim bulandırma* adı verilen bir mekanizmaya sınırlı da olsa destek verilmiştir. Buna göre, `__falanca` biçimindeki bir tanımlayıcı (ön taraftan en az iki alt çizgi, arka taraftan en fazla bir alt çizgi) kod içinde geçtiği

yerde `_sınıfadı__falanca` olarak değiştirilir. Burada `sınıfadı`, önündeki alt çizgi(ler) atılmış mevcut sınıf ismidir. Bu bulandırma işlemi, tanımlayıcının, bir sınıf tanımı içinde geçtiği sürece, söz dizimsel konumuna bakılmaksızın yapılır.

İsim bulandırma, alt sınıfların, sınıf içi metot çağrılarını bozmadan metotların üzerine yazmasını sağlamak açısından faydalıdır. Mesela:

```
class Eşleşme:
    def __init__(self, yürünebilir_nesne):
        self.öğe_listesi = []
        self.__güncelle(yürünebilir_nesne)

    def güncelle(self, yürünebilir_nesne):
        for öğe in yürünebilir_nesne:
            self.öğe_listesi.append(öğe)

    __güncelle = güncelle   # orijinal güncelle() metodunun gizli kopyası

class EşleşmeAltSınıfı(Eşleşme):

    def güncelle(self, anahtarlar, değerler):
        # güncelle() için yeni bir imza oluşturuyoruz
        # ama __init__()'i bozmuyoruz
        for öğe in zip(anahtarlar, değerler):
            self.öğe_listesi.append(öğe)
```

Bulandırma kurallarının çoğunlukla kazaları engellemek için tasarlandığını unutmayın. Gizli olması düşünülen değişkenlere erişmek veya değiştirmek yine de mümkündür. Bu durum, hata ayıklayıcı gibi özel durumlarda yararlı bile olabilir.

`exec()` veya `eval()` fonksiyonlarına verilen parametrelerin, çağıran sınıfın sınıf adını mevcut sınıf adına eklemediğine dikkat edin. Bu durum, `global` deyiminin etkisinin, bayt olarak birlikte derlenmiş kodlarla sınırlı olmasına benzer. Aynı kısıtlama `getattr()`, `setattr()` ve `delattr()` ile birlikte, doğrudan `__dict__`'e atıfta bulunulan durumlar için de geçerlidir.

9.7 Birkaç Şey

Bazen, Pascal'daki 'record' (kayıt) veya C'deki 'struct' (yapı) tiplerine benzer bir veri tipi kullanarak birkaç isimli veri öğesini bir araya getirmek faydalı olabilir. Boş bir sınıf tanımlamak bu iş için gayet uygun olacaktır:

```
class Çalışan:
    pass

mehmet = Çalışan() # Boş bir çalışan kaydı oluşturuyoruz

# Kayıttaki alanları dolduruyoruz
mehmet.isim = 'Mehmet Öz'
mehmet.bölüm = 'bilgisayar laboratuvarı'
mehmet.maaş = 1000
```

Belirli bir soyut veri tipi bekleyen bir Python kod parçasına, o veri tipinin metotlarını taklit edecek şekilde veri tipi yerine, genellikle bir sınıf verilebilir. Örneğin, bir dosya nesnesinden gelen bazı verileri biçimlendiren bir fonksiyonunuz varsa, verileri karakter dizisi

tamponundan alan `read()` ve `readline()` adlı metotlara sahip bir sınıf tanımlayabilir, bunu argüman olarak atayabilirsiniz.

Örnek metodu nesnelerinin de nitelikleri vardır: `m.__self__`, `m()` metoduna sahip bir örnek nesnesidir. `m.__func__` ise bu metoda karşılık gelen fonksiyon nesnesidir.

9.8 İstisnalar da Birer Sınıftır

Kullanıcı tarafından tanımlanan istisnalar da birer sınıf gibi işlem görür. Bu mekanizma kullanılarak, istisnalar için genişletilebilir hiyerarşik yapılar oluşturmak mümkündür.

`raise` deyimi için yeni, iki adet geçerli (semantik) biçim bulunur:

```
raise Sınıf

raise Örnek
```

İlk biçimde `Sınıf` `type` sınıfının veya bundan türeyen bir sınıfın örneği olmalıdır. İlk biçim aşağıdakinin kısaltılmış halidir:

```
raise Sınıf()
```

`except` cümleciği içinde belirtilen bir sınıf, kendisiyle aynı adı taşıyan bir sınıf ile veya o sınıfın bir taban sınıfı ile eşleşir. (Ama bunun tersi geçerli değildir. Yani türetilmiş bir sınıfın belirtildiği except cümleciği bir taban sınıf ile eşleşmez). Örneğin aşağıdaki kodlar sırasıyla B, C, D basacaktır:

```
class B(Exception):
    pass
class C(B):
    pass
class D(C):
    pass

for snf in [B, C, D]:
    try:
        raise snf()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

Eğer except cümlecikleri ters çevrilirse (`except B` ilk olacak şekilde), ekrana B, B, B basılır. Çünkü eşleşen ilk except cümleciği tetiklenecektir.

Yakalanmamış bir istisna nedeniyle bir hata mesajı basılırken, istisnanın sınıf ismi basılır, ardından iki nokta üst üste ve bir boşluk getirilir. Son olarak da gömülü `str()` fonksiyonuyla karakter dizisine dönüştürülen sınıf örneği basılır.

9.9 Yürüyücüler

Şimdiye kadar çoğu kapsayıcı nesne üzerinde `for` deyimi yardımıyla döngü kurulabileceğini farketmişsinizdir:

```
for öğe in [1, 2, 3]:
    print(öğe)
for öğe in (1, 2, 3):
    print(öğe)
for anahtar in {'bir':1, 'iki':2}:
    print(anahtar)
for karakter in "123":
    print(karakter)
for satır in open("dosya.txt"):
    print(satır, end="")
```

Bu erişim tarzı açık, öz ve kolaydır. Yürüyücülerin kullanımına Python'ın her köşesinde rastlayabilirsiniz. Perde arkasında, `for` deyimi kapsayıcı nesne üzerinde `iter()` fonksiyonunu çağırır. Fonksiyon, kapsayıcı nesne içindeki öğelere birer birer erişen `__next__()` adlı bir metot tanımlayan bir yürüyücü nesnesi döndürür. Artık erişilecek öğe kalmadığında `__next__()` metodu `StopIteration` istisnasını tetikler ve böylece `for` döngüsü sona erer. `__next__()` metodunu `next()` gömülü fonksiyonunu kullanarak çağırabilirsiniz. Aşağıdaki örnek bunun nasıl çalıştığını gösterir:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    next(it)
StopIteration
```

Yürüyücü protokolünün işleyişini öğrendiğinize göre kendi sınıflarınıza yürüyücü davranışını kolaylıkla ekleyebilirsiniz. Bunun için, `__next__()` metoduna sahip bir nesne döndüren bir `__iter__()` metodu tanımlıyoruz. Eğer sınıfta `__next__()` metodu tanımlanmışsa, `__iter__()` metodu yalnızca `self` döndürebilir:

```
class Tersi:
    """Bir dizi üzerinde ters döngü kurmaya yarayan bir yürüyücü."""
    def __init__(self, veri):
        self.veri = veri
        self.sıra = len(veri)
    def __iter__(self):
        return self
    def __next__(self):
        if self.sıra == 0:
            raise StopIteration
        self.sıra = self.sıra - 1
```

```
        return self.veri[self.sıra]
```

```
>>> tersi = Tersi('kars')
>>> iter(tersi)
<__main__.Tersi object at 0x00A1DB50>
>>> for karakter in tersi:
...     print(karakter)
...
s
r
a
k
```

9.10 Üreteçler

Üreteçler yürüyücü (iterator) oluşturmaya yarayan basit ve güçlü araçlardır. Bunlar bildiğimiz fonksiyonlar gibi yazılır, ama veri döndürmek gerektiğinde (`return` yerine) `yield` deyimi kullanılır. Üreteç üzerinde `next()` fonksiyonunun her çağrılışında üreteç kaldığı yerden devam eder (bütün veri değerlerini ve en son işletilen deyimin hangisi olduğunu hatırlar). Şu örnek, üreteçlerin ne kadar kolay oluşturulabileceğini göstermektedir:

```
def tersi(veri):
    for sıra in range(len(veri)-1, -1, -1):
        yield veri[sıra]
```

```
>>> for karakter in tersi('golf'):
...     print(karakter)
...
f
l
o
g
```

Önceki bölümde açıklandığı gibi, üreteçler ile yapılabilecek her şey sınıf tabanlı yürüyücülerle de yapılabilir. Üreteçlerin bu kadar kısa ve öz olmasını sağlayan şey `__iter__()` ve `__next__()` metotlarının otomatik olarak oluşturulmasıdır.

Üreteçlerin önemli bir özelliği de lokal değişkenlerin ve yürütme durumunun çağrılar arasında otomatik olarak kaydediliyor olmasıdır. Bu özellik fonksiyonun yazılışını kolaylaştırır ve `self.sıra` ve `self.veri` gibi örnek değişkenlerini kullanma yaklaşımına göre daha çok daha anlaşılırdır.

Otomatik metot oluşturma ve program durumunun kaydedilmesinin yanısıra, üreteçler sona erdiğinde otomatik olarak `StopIteration` istisnasını tetiklerler. Bütün bu özellikler bir araya gelerek, yürüyücülerin oluşturulmasını en az normal fonksiyonlar yazmak kadar kolay hale getirir.

9.11 Üreteç İfadeler

Liste üreticilere benzer bir söz dizimi yardımıyla, ama köşeli parantezler yerine normal parantezler kullanarak, kısa yoldan ifade biçimli basit üreteçler yazabilirsiniz. Bu tür ifadeler, üreteçlerin, içinde bulundukları fonksiyon tarafından hemen kullanılacakları durumlar için

tasarlanmıştır. Üreteç ifadeler, tam teşekküllü üreteç tanımlarına göre daha az yer tutmakla birlikte yetenekleri daha kısıtlıdır. Ama bu yapılar, eşdeğer liste üreticilere kıyasla daha bellek dostudurlar.

Örnekler:

```
>>> sum(i*i for i in range(10))                 # karelerin toplamı
285

>>> xvek = [10, 20, 30]
>>> yvek = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvek, yvek))         # iç çarpım
260

>>> from math import pi, sin
>>> sinüs_tablosu = {x: sin(x*pi/180) for x in range(0, 91)}

>>> tek_kelimeler = set(kelime  for satır in sayfa  for kelime in satır.split())

>>> sınıf_birincisi = max((öğrenci.ortalama, öğrenci.isim) for öğrenci in mezunlar)

>>> veri = 'golf'
>>> list(veri[i] for i in range(len(veri)-1, -1, -1))
['f', 'l', 'o', 'g']
```

Standart Kütüphanede Kısa Bir Gezinti

10.1 İşletim Sistemi Arayüzü

`os` modülü, işletim sistemi ile iletişim kurmamızı sağlayacak onlarca fonksiyona sahiptir:

```
>>> import os
>>> os.getcwd()  # Mevcut çalışma dizinini döndürür
'C:\\Python35'
>>> os.chdir('/sunucu/günlükler') # Mevcut çalışma dizinini değiştirir
>>> os.system('mkdir bugün') # Sistem kabuğunda mkdir komutunu çalıştırır
0
```

Burada `from os import *` yerine `import os` biçimini kullanmaya özen gösterin. Böylece `os.open()` fonksiyonunun, çok farklı bir işleyişi olan gömülü `open()` fonksiyonunun üzerini örtmesini önlemiş olursunuz.

Gömülü `dir()` ve `help()` fonksiyonları, `os` gibi kapsamlı modüller üzerinde çalışırken etkileşimli kabukta yardım alabilmenizi sağlar:

```
>>> import os
>>> dir(os)
<modüldeki bütün fonksiyonların listesini döndürür>
>>> help(os)
<modülün belge dizileri kullanılarak oluşturulan ayrıntılı bir kılavuz sayfası döndürür>
```

`shutil` modülü, günlük dosya ve dizin yönetimi işlemleri açısından kullanımı daha kolay olan yüksek seviyeli bir arayüz sunar:

```
>>> import shutil
>>> shutil.copyfile('veri.db', 'arşiv.db')
'arşiv.db'
>>> shutil.move('/inşa/çalıştırılabilirdosyalar', 'kurulumdizini')
'kurulumdizini'
```

10.2 Dosyalarda Joker Karakterler

`glob` modülü, dizinler içinde joker karakterleri kullanarak arama yapıp bu aramalardan dosya listeleri oluşturmanızı sağlayan bir fonksiyon sunar:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

10.3 Komut Satırı Argümanları

Yardımcı fonksiyonları içeren betikler genellikle komut satırı argümanlarını işlemeye ihtiyaç duyar. Komut satırı argümanları `sys` modülünün *argv* niteliği içinde bir liste halinde tutulur. Örneğin aşağıdaki çıktı, `python demo.py bir iki üç` ifadesinin komut satırında çalıştırılması sonucu elde edilmiştir:

```
>>> import sys
>>> print(sys.argv)
['demo.py', 'bir', 'iki', 'üç']
```

`getopt` modülü, Unix'teki `getopt()` fonksiyonuna benzer bir şekilde *sys.argv* listesini işler. `argparse` modülü ise size daha güçlü ve esnek bir komut satırı işleme kabiliyeti sunar.

10.4 Hata Çıktısı Yönlendirme ve Program Sonlandırma

`sys` modülü *stdin*, *stdout* ve *stderr* gibi birtakım niteliklere de sahiptir. *stderr*; *stdout* yönlendirilmiş bile olsa, uyarı ve hata mesajlarını görünür hale getirmek için kullanılabilir.

```
>>> sys.stderr.write('Dikkat, log dosyası bulunamadı. Yeni bir tane oluşturuluyor\n')
Dikkat, log dosyası bulunamadı. Yeni bir tane oluşturuluyor
```

Bir betiği sonlandırmanın en kesin yöntemi `sys.exit()` komutunu kullanmaktır.

10.5 Karakter Dizisi Desen Eşleştirme

`re` modülü, ileri düzey karakter dizisi işleme işlemleri için düzenli ifade araçları sunar. Düzenli ifadeler; karmaşık eşleştirme ve değiştirme işlemlerine yönelik hem kısa ve öz, hem de verimli çözümlere sahiptir.

```
>>> import re
>>> re.findall(r'\bg[\w]*', 'gözlerin gözlerime değince felaketim olurdu ağlardım')
['gözlerin', 'gözlerime']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'günlerden bir bir gün')
'günlerden bir gün'
```

Ancak eğer yapmak istediğiniz şey basit bir işlemse, okuması ve hatalarını ayıklaması daha kolay olduğu için karakter dizisi metotlarını tercih etmek gerekir.

```
>>> 'bilmezler yanlız yaşamayanlar'.replace('yanlız', 'yalnız')
'bilmezler yalnız yaşamayanlar'
```

10.6 Matematik

`math` modülü, kayan noktalı sayılarla aritmetik işlem yaparken, arka plandaki C kütüphane fonksiyonlarına erişebilmenizi sağlar.

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

`random` modülü rastgele seçimler yapabilmenizi sağlayan araçlar sunar:

```
>>> import random
>>> random.choice(['elma', 'armut', 'muz'])
'elma''
>>> random.sample(range(100), 10)   # öğelere dokunmadan, numune alıyoruz
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # rastgele bir kayan noktalı sayı
0.17970987693706186
>>> random.randrange(6)    # range(6) aralığında rastgele seçilen bir tamsayı
4
```

`statistics` modülü, sayısal verilerin temel istatistiksel özelliklerini (ortalama, medyam, değişkenlik, vb.) hesaplar.

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

SciPy projesi <http://scipy.org>, nümerik hesaplamalar için pek çok başka modül de barındırır.

10.7 İnternet Erişimi

İnternete erişmek ve internet protokolleri ile işlem yapabilmek için kullanabileceğiniz bazı modüller de vardır. Bunların içinde en basit iki tanesi URL'lerden veri çekmek için kullanılan `urllib.request` ve e.posta göndermek için kullanılan `smtplib` modülleridir:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl') as cevap:
...     for satır in cevap:
...         satır = satır.decode('utf-8')  #İkili verileri metin olarak kodluyoruz.
...         if 'EST' in satır or 'EDT' in satır:  # Doğu Standart Saati'ni arıyoruz
...             print(satır)

<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> sunucu = smtplib.SMTP('localhost')
```

```
>>> sunucu.sendmail('kahin@example.org', 'jsezar@example.org',
... """Kime: jsezar@example.org
... Kimden: kahin@example.org
...
... Mart'ın 15'inden sakın!
... """)
>>> sunucu.quit()
```

(İkinci örnek için localhost'ta çalışan bir mail sunucusu gerektiğine dikkat edin.)

10.8 Tarihler ve Zamanlar

`datetime` modülü tarihler ve zamanlar üzerinde hem basit hem de karmaşık şekillerde oynayabilmenizi sağlayacak sınıflar sunar. Tarih ve zaman değerleri ile aritmetik işlemler yapılabilse de, bunlara ilişkin fonksiyonların gayesi, biçimlendirme ve düzenleme işlemlerine yönelik olarak çıktıdan verimli bir şekilde öğe ayıklamaktır. Bu modül ayrıca zaman dilimlerini ayırt edebilen nesneleri de destekler.

```
>>> # tarihleri oluşturmak ve biçimlendirmek son derece kolaydır
>>> from datetime import date
>>> şuan = date.today()
>>> şuan
datetime.date(2003, 12, 2)
>>> şuan.strftime("%m-%d-%y. %d %b %Y, %B ayının %d. gününe gelen bir %A günüdür.")

'12-02-03. 02 Ara 2003, Aralık ayının 02. gününe gelen bir Salı günüdür.'

>>> # tarihlerle takvime ilişkin aritmetik işlemler yapılabilir
>>> doğumgünü = date(1964, 7, 31)
>>> yaş = şuan - doğumgünü
>>> yaş.days
14368
```

10.9 Veri Sıkıştırma

`zlib`, `gzip`, `bz2`, `lzma`, `zipfile` ve `tarfile` gibi modüller aracılığıyla yaygın veri arşivleme ve sıkıştırma biçimleri doğrudan desteklenmektedir.

```
>>> import zlib
>>> s = b'dal kalkar kartal sarkar dal sarkar kartal kalkar'
>>> len(s)
49
>>> t = zlib.compress(s)
>>> len(t)
34
>>> zlib.decompress(t)
b'dal kalkar kartal sarkar dal sarkar kartal kalkar'
>>> zlib.crc32(s)
3211303476
```

10.10 Performans Ölçme

Bazı Python kullanıcıları, aynı probleme farklı yaklaşımların karşılaştırmalı performansını öğrenmeye karşı derin bir alaka duyar. Python, buna ilişkin sorularınızı derhal cevaplamanızı sağlayacak bir ölçüm aracı sunar.

Argümanları birbiriyle değiştirmeye ilişkin geleneksel yaklaşım yerine demetleme ve demet çözme özelliğini kullanmak görünüze cazip görünebilir. `timeit` modülü, bu şekilde küçük de olsa bir performans avantajı elde edildiğini bize hemencecik gösterecektir:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

`timeit` modülünün bize sunduğu ince hassasiyet düzeyinin aksine, `profile` ve `pstats` modülleri, daha büyük kod bloklarındaki zaman açısından kritik bölümleri tespit etmeye yarayan araçlar sunar.

10.11 Kalite Kontrolü

Yüksek kaliteli yazılımlar geliştirmek için benimsenen yaklaşımlardan biri de her bir fonksiyon için, daha bunlar geliştirildiği sırada testler yazmak ve geliştirme sürecinde bu testleri sık sık çalıştırmaktır.

`doctest` modülü, bir modülü tarayıp, programın belge dizilerine gömülmüş testlerin sağlamasını yapmanıza yarayan birtakım araçlar sunar. Test yapısının oluşturulması, örnek bir komutun, çıktılarıyla birlikte belge dizisine yapıştırılmasından ibarettir. Bu şekilde hem kullanıcıya bir örnek sunularak belgelendirme iyileştirilmiş olur hem de doctest modülü sayesinde kod ile belgelendirmenin birbiriyle uyumlu kalması sağlanır:

```
def ortalama(değerler):
    """Sayılardan oluşan bir listenin aritmetik ortalamasını hesaplar.

    >>> print(ortalama([20, 30, 70]))
    40.0
    """
    return sum(değerler) / len(değerler)

import doctest
doctest.testmod()   # gömülü testlerin otomatik olarak sağlamasını yapar
```

`unittest` modülü `doctest` modülü kadar zahmetsiz değildir, ama ayrı bir dosyada daha kapsamlı testlerin tutulabilmesine olanak tanır:

```
import unittest

class İstatistikFonksiyonlarınınTesti(unittest.TestCase):

    def test_ortalama(self):
        self.assertEqual(ortalama([20, 30, 70]), 40.0)
        self.assertEqual(round(ortalama([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
```

```
            average([])
        with self.assertRaises(TypeError):
            ortalama(20, 30, 70)

unittest.main() # Komut satırından çağrıldığında bütün testler çalışır
```

10.12 Piller de Dahil

Python "piller de dahil" şeklinde tabir edilen bir felsefeyi benimser. Bunu görmek için dildeki büyük paketlerin ne kadar karmaşık ve ne kadar güçlü yeteneklere sahip olduğuna bakmak yeterli olacaktır. Örneğin:

- `xmlrpc.client` ve `xmlrpc.server` modülleri uzaktan yordam çağrılarını neredeyse çocuk oyuncağı haline getirir. Modüllerin adı sizi yanıltmasın. XML bilmeniz veya XML ile uğraşmanız gerekmiyor.
- `email` paketi, MIME ve başka RFC 2822 tabanlı mesaj belgeleri de dahil olmak üzere, e.posta mesajlarını yönetmek için kullanılan bir kütüphanedir. Yalnızca mesaj alıp göndermeye yarayan `smtplib` ve `poplib` kütüphanelerinin aksine, email paketi (ekler de dahil olmak üzere) karmaşık mesaj yapıları oluşturmak veya bu yapıları çözmek ya da internet kodlama ve başlık protokolleri gerçeklemek için eksiksiz bir alet takımı sunar.
- `json` paketi, bu gözde veri aktarım biçimini ayrıştırmak için güçlü bir destek sağlar. `csv` modülü de yaygın bir veri tabanı ve hesap çizelgesi biçimi olan csv ile doğrudan okuma ve yazma işlemleri gerçekleştirilmesine imkan tanır. XML işleme desteği ise `xml.etree.ElementTree`, `xml.dom` ve `xml.sax` paketleri ile sağlanmaktadır. Bütün bu modüller ve paketler Python uygulamaları ile başka araçlar arasındaki veri alışverişini büyük ölçüde kolaylaştırır.
- `sqlite3` **modülü, SQLite veritabanı kütüphanesi için,** standarttan biraz farklı bir SQL sözdizimi aracılığıyla güncellenebilecek

  ve erişilebilecek kalıcı bir veritabanı sunan bir arayüzdür.
- Uluslararasılaştırma, `gettext`, `locale` ve `codecs` gibi birtakım modüller aracılığıyla desteklenir.

Standart Kütüphanede Kısa Bir Gezinti – Bölüm II

Bu ikinci turda, profesyonel programlama ihtiyaçlarınıza yanıt veren daha ileri düzey modülleri ele alacağız. Bu modüllerin küçük betiklerde kullanıldığını pek görmezsiniz.

11.1 Çıktı Biçimlendirme

`reprlib` modülü; `repr()` modülünün, büyük veya çok iç içe geçmiş kapsayıcı veri tiplerini kısaltılmış bir şekilde gösterecek şekilde tasarlanmış halidir.

```
>>> import reprlib
>>> reprlib.repr(set('çekoslavakyalılaştıramadıklarımız'))
"{'a', 'c', 'd', 'e', 'f', 'g', ...}"
```

`pprint` modülü, hem gömülü hem de kullanıcı tanımlı nesnelerin yorumlayıcı tarafından okunabilecek bir biçinde ekrana basılması işlemini daha ayrıntılı bir şekilde kontrol edebilmenizi sağlar. Eğer çıktı bir satırdan uzunsa, "zarif yazdırıcı", veri yapısını daha belirgin hale getirmek için karakter dizisine satır boşlukları ve girintiler ekler:

```
>>> import pprint
>>> t = [[[['siyah', 'camgöbeği'], 'beyaz', ['yeşil', 'kırmızı']], [['eflatun',
...     'sarı'], 'mavi']]]
...
>>> pprint.pprint(t, width=30)
[[[['siyah', 'camgöbeği'],
   'beyaz',
   ['yeşil', 'kırmızı']],
  [['eflatun', 'sarı'],
   'mavi']]]
```

`textwrap` modülü, metindeki paragrafları, belirli bir ekran genişliğine sığacak şekilde biçimlendirir:

```
>>> import textwrap
>>> belge = """wrap() metodu fill() metoduna benzer, ancak farklı olarak,
... hizalanan satırları ayırmak için, yeni satırları içeren tek bir büyük karakter
... dizisi yerine karakter dizilerinden oluşan bir liste döndürür."""
...
>>> print(textwrap.fill(belge, width=40))
wrap() metodu fill() metoduna benzer, ancak farklı
olarak hizalanan satırları ayırmak için, yeni satırları
içeren tek bir büyük karakter dizisi yerine karakter
dizilerinden oluşan bir liste döndürür.
```

`locale` modülü, yerele özgü veri biçimlerini barındıran bir veritabanına erişim sağlar. Locale modülünün format fonksiyonuna ait gruplama niteliği sayesinde, sayılar grup ayraçları ile doğrudan biçimlendirilebilir.

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'Turkish_Turkey.1252')
'Turkish_Turkey.1252'
>>> yerel = locale.localeconv() # yerelleştirmeleri içeren sözlük
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1.234.567'
>>> locale.format_string("%s %.*f", (yerel['currency_symbol'],
...                      yerel['frac_digits'], x), grouping=True)
'TL 1.234.567,80'
```

11.2 Şablonlar

`string` modülü, son kullanıcılar tarafından düzenlenmeye uygun, basit bir söz dizimine sahip, esnek bir `Template` sınıfı içerir. Bu sınıf, kullanıcıların, uygulamayı değiştirmek zorunda kalmadan uygulamalarını özelleştirebilmesine izin verir.

Biçimlendirici; `$` ve geçerli Python tanımlayıcıları (alfanümerik karakterler ve alt çizgiler) ile oluşturulan yer tutucu adlarını içerir. Yer tutucuyu süslü parantez içine alarak, araya gereksiz boşluk yerleştirmek zorunda kalmadan bunun ardından başka alfanümerik harfler de getirebilirsiniz. `$` işaretini kullanabilmek için `$$` yazabilirsiniz:

```
>>> from string import Template
>>> t = Template('${kurum}İlaç $sebep yararına 100.000$$ göndermiştir.')
>>> t.substitute(kurum='Filanca', sebep='yoksullar')
'FilancaİIaç yoksullar yararına 100.000 $ göndermiştir.'
```

Eğer yer tutucu bir sözlük veya isimli argüman içinde verilmemişse, `substitute()` metodu `KeyError` istisnasını tetikleyecektir. Mektup ve adres birleştirme tarzı uygulamalarda, kullanıcı tarafından girilen veriler eksik olabilir. Bu durumda `safe_substitute()` metodunu kullanmak daha uygun olacaktır. Bu metot, eğer eksik veri varsa, o veriye karşılık gelen yer tutucuyu olduğu gibi bırakır:

```
>>> t = Template("Lütfen $kitap adlı kitabı $sahibi Bey'e iade ediniz.")
>>> d = dict(kitap='Tutunamayanlar')
>>> t.substitute(d)
Traceback (most recent call last):
  ...
KeyError: 'sahibi'
>>> t.safe_substitute(d)
"Lütfen Tutunamayanlar adlı kitabı $sahibi Bey'e iade ediniz."
```

Template sınıfını miras yoluyla kullanarak farklı bir ayraç (delimiter) da belirtebilirsiniz. Örneğin, bir fotoğraf tarayıcısı için toplu adlandırma uygulaması yazıyorsanız, güncel tarih, fotoğrafların sıra numarası veya dosya biçimi gibi yer tutucular için yüzde işaretini kullanmayı tercih edebilirsiniz:

```
>>> import time, os.path
>>> fotodosyaları = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class TopluAdlandır(Template):
...     delimiter = '%'
```

```
>>> bçm = input('Yeniden adlandırma stili: (%d-tarih %n-sıra %f-biçim):  ')
Yeniden adlandırma stili: (%d-tarih %n-sıra %f-biçim):  Ayşe_%n%f

>>> t = TopluAdlandır(bçm)
>>> tarih = time.strftime('%d%b%y')
>>> for i, dosyaadı in enumerate(fotodosyaları):
...     kök, uzantı = os.path.splitext(dosyaadı)
...     yeniad = t.substitute(d=tarih, n=i, f=uzantı)
...     print('{0} --> {1}'.format(dosyaadı, yeniad))

img_1074.jpg --> Ayşe_0.jpg
img_1076.jpg --> Ayşe_1.jpg
img_1077.jpg --> Ayşe_2.jpg
```

Şablon oluşturmaya ilişkin bir başka kullanım alanı da, programın yapısı ile farklı çıktı biçimlerine ilişkin ayrıntıları birbirinden ayırmaktır. Bu şekilde XML dosyaları, düz metin raporları ve HTML web raporları için özel şablonlar oluşturabilirsiniz.

11.3 İkili Veri Kayıtlarıyla Çalışmak

`struct` modülü, değişken uzunluktaki ikili kayıt biçimleriyle çalışabilmenizi sağlamak için `pack()` ve `unpack()` adlı iki fonksiyon sunar. Aşağıdaki örnek, `zipfile` modülünü kullanmadan, bir ZIP dosyasındaki başlık bilgileri üzerinde nasıl döngü kurulabileceğine dair bir örnek sunmaktadır. `"H"` ve `"I"` pack kodları, sırasıyla iki ve dört baytlık işaretsiz sayıları temsil eder. `"<"` işleci bunların standart boyutta olduğunu ve little-endian bayt düzenine sahip olduğunu gösterir:

```
import struct

with open('myfile.zip', 'rb') as f:
    data = f.read()

start = 0
for i in range(3): # ilk 3 dosya başlığını gösteriyoruz
    start += 14
    fields = struct.unpack('<IIIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)

    start += extra_size + comp_size # bir sonraki başlığa geçiyoruz
```

11.4 Çok Katmanlı Programlama

Çok katmanlı programlama, sıra bakımından birbirine bağlı olmayan görevleri birbirinden ayırmak için kullanılan bir tekniktir. Katmanlar, öteki görevler arkaplanda çalışmaya devam ederken kullanıcıdan girdi alabilen uygulamaların donmaması için kullanılabilir. Buna bir

örnek, başka bir katmanda devam eden hesaplamalarla paralel bir şekilde girdi/çıktı işlemleri yapmaktır.

Aşağıdaki kod, ana program çalışmaya devam ederken, `threading` adlı yüksek seviyeli modülün, arkaplandaki görevleri nasıl yürütebildiğini gösteriyor:

```
import threading, zipfile

class EşzamansızSıkıştırma(threading.Thread):
    def __init__(self, girendosya, çıkandosya):
        threading.Thread.__init__(self)
        self.girendosya = girendosya
        self.çıkandosya = çıkandosya
    def run(self):
        f = zipfile.ZipFile(self.çıkandosya, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.girendosya)
        f.close()
        print('Arkaplanda şu dosyanın sıkıştırılma işlemi tamamlandı:', self.girendosya)

arkaplan = EşzamansızSıkıştırma('veriler.txt', 'arşiv.zip')
arkaplan.start()
print('Ana program arkaplanda çalışmaya devam ediyor.')

arkaplan.join()    # Arkaplandaki görevin bitmesi bekleniyor
print('Ana program arkaplandaki görev tamamlanana kadar bekledi.')
```

Çok katmanlı uygulamaların başlıca zorluğu, verileri veya başka kaynakları paylaşan katmanları koordine etmektir. Bunu kolaylaştırmak için threading modülü, kilitler, olaylar, koşul değişkenleri ve semaforlar gibi bir dizi temel senkronizasyon araçları sunar.

Bu araçlar ne kadar güçlü olsa da, ufak tasarım hataları, tespit etmesi güç problemlere yol açabilir. Dolayısıyla, görev kooardinasyonuna ilişkin tercih edilen yaklaşım, bir kaynağa bütün erişimi tek bir katmanda toplamak ve öteki katmanlardan gelen istemleri `queue` modülü aracılığıyla bu katmana yollamaktır. Katmanlar arası iletişim ve koordinasyon için `Queue` nesnelerini kullanan uygulamalar daha kolay tasarlanmakta, daha okunaklı ve daha güvenilir olmaktadır.

11.5 Loglama

`logging` modülü, tam teşekküllü ve esnek bir loglama sistemi sunar. En basit haliyle, log mesajları bir dosyaya ya da `sys.stderr`'e gönderilir:

```
import logging
logging.debug('Hata ayıklamaya ilişkin bilgiler')
logging.info('Bilgi mesajı')
logging.warning('Uyarı:%s adlı ayar dosyası bulunamadı', 'server.conf')
logging.error('Hata meydana geldi')
logging.critical('Kritik hata -- sistem kapatılıyor')
```

Bu kodlar şu çıktıyı üretir:

```
WARNING:root:Uyarı: server.conf adlı ayar dosyası bulunamadı
ERROR:root:Hata meydana geldi
CRITICAL:root:Kritik hata -- sistem kapatılıyor
```

Öntanımlı olarak bilgi ve hata ayıklama mesajları gizlenir ve çıktı standart hata akışına yönlendirilir. Diğer çıktı seçenekleri arasında mesajların, veri iletilerinin, soketlerin e.posta aracılığıyla veya bir HTTP sunucusuna yönlendirilmesi yer alır. Yeni filtreler, mesaj önceliğine bağlı olarak farklı yönlendirme yapıları belirlemenizi sağlar: `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `CRITICAL`.

Loglama sistemi doğrudan Python içinden ayarlanabilir veya uygulamayı değiştirmek zorunda kalmadan, özelleştirilmiş loglama işlemleri için, kullanıcı tarafından düzenlenebilen bir ayar dosyasından yüklenebilir.

11.6 Zayıf Atıflar

Python, otomatik bellek yönetimi uygular (çoğu nesne için atıf sayma ve döngüleri ortadan kaldırmak için çöp toplama) Bellekteki bir nesneye son atıf da ortadan kalktıktan kısa bir süre sonra bellek serbest bırakılır.

Bu yaklaşım çoğu uygulamada başarıyla işler, ancak kimi zaman nesneleri, ancak başka bir şey tarafından kullanılmakta olduğunda takip etmek gerekebilir. Ne yazık ki, bu nesneleri takip etmek, bu nesnelere bir atıf oluşturacağı için, bu işlem bu nesneleri kalıcı hale getirir. İşte `weakref` modülü, herhangi bir atıf oluşturmadan nesneleri takip etmeyi sağlayacak araçlar sunar. Artık nesneye ihtiyaç kalmadığında, bu nesne zayıf atıf tablosundan otomatik olarak çıkarılır ve zayıf atıf nesneleri için bir fonksiyon çağrısı tetiklenir. Bu özelliği kullanan uygulamalar arasında, oluşturması zahmetli olan önbellekleme nesneleri yer alır:

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, değer):
...         self.değer = değer
...     def __repr__(self):
...         return str(self.değer)
...
>>> a = A(10)                    # bir atıf oluşturuyoruz
>>> d = weakref.WeakValueDictionary()
>>> d['birinci'] = a             # atıf oluşturmaz
>>> d['birinci']                 # Eğer hala canlıysa nesneyi getirir
10
>>> del a                        # tek atfı da siliyoruz
>>> gc.collect()                 # çöp toplama işlemini başlatıyoruz
0
>>> d['birinci']                 # öğe otomatik olarak silinir
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['birinci']                # öğe otomatik olarak silinir
  File "C:/python35/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'birinci'
```

11.7 Listelerle Çalışmaya Yönelik Araçlar

Veri yapılarına ilişkin ihtiyaçlarınızın çoğunu gömülü liste tipi ile karşılayabilirsiniz. Ancak bazen, performans konusunda farklı getirileri-götürüleri olan alternatif gerçeklemelere de ihtiyaç duyabilirsiniz.

`array` modülü, yalnızca homojen verileri, listelere kıyasla daha sıkı bir şekilde saklayan `array()` adlı bir nesne sunar. Aşağıdaki örnek, Python int nesnelerinden oluşan bildik bir listedeki her bir öğe başına 16 bayt yerine, iki baytlık işaretsiz ikili sayılar (tip kodu `"H"`) olarak tutulan bir sayı dizisi göstermektedir:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

`collections` modülü, listelere benzeyen, ama baş tarafa öğe ekleme ve baş taraftan öğe silme işlemlerini listelere kıyasla daha hızlı, ortadan öğe bakma işlemlerini ise daha yavaş yapan bir `deque()` nesnesi sunar. Bu nesneler, sıra yapılarını ve enine ağaç arama algoritmalarını gerçeklemeye gayet uygundur:

```
>>> from collections import deque
>>> d = deque(["görev1", "görev2", "görev3"])
>>> d.append("görev4")
>>> print(d.popleft(), "yürütülüyor")
görev1 yürütülüyor
```

```
aranmayan = deque([başlangıç_nodu])
def enine_arama(aranmayan):
    nod = aranmayan.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        aranmayan.append(m)
```

Alternatif liste gerçeklemelerine ilave olarak, kütüphanede ayrıca, sıralı listeleri manipüle etmeye yarayan `bisect` modülü gibi araçlar da bulunur:

```
>>> import bisect
>>> puanlar = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(puanlar, (300, 'ruby'))
>>> puanlar
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

`heapq` modülü, bildik listelere dayanarak öbek gerçeklemeyi sağlayan fonksiyonlara sahiptir. En düşük değerli öğe her zaman sıfır konumunda tutulur. Bu özellik, tekrar tekrar en küçük öğeye erişen, ama tam liste sıralaması gerçekleştirmek istemeyen uygulamalar için yararlıdır.

```
>>> from heapq import heapify, heappop, heappush
>>> veri = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(veri)                      # listeyi öbek sırasına diziyoruz
>>> heappush(veri, -5)                 # yeni bir öğe ekliyoruz
>>> [heappop(veri) for i in range(3)]  # en küçük üç öğeyi getiriyoruz
[-5, 0, 1]
```

11.8 Ondalık Kayan Nokta Aritmetiği

`decimal` modülü, ondalık kayan noktalı sayılarla aritmetik işlemleri için `Decimal` adlı bir veri tipi sunar. Gömülü `float` sınıfının ikili kayan noktalı sayı gerçeklemesine kıyasla bu sınıf

özellikle şunlar için yararlıdır:

- Mali uygulamalar ve kesin ondalık temsil gerektiren diğer kullanımlar * Hassasiyet üzerinde denetim
- Hukuki veya kanuni gereklilikleri karşılamak için yuvarlama üzerinde kontrol
- Görünen ondalık kısımların takibi veya
- Kullanıcının, sonuçların elle yapılan hesaplamalarla uyuşmasını beklediği uygulamalar.

Örneğin, 70 cent'lik bir telefon faturası üzerine %5 vergi hesaplaması, ondalık kayan noktalı sayılarla ikili kayan noktalı sayılarda birbirinden farklı sonuçlar verir. Eğer sonuçlar en yakın cent'e yuvarlanırsa bu fark daha belirgin olur:

```
>>> from decimal import *
>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
>>> round(.70 * 1.05, 2)
0.73
```

`Decimal` sınıfının çıktısında sondaki sıfırlar korunur, böylece iki adet anlamlı basamağa sahip çarpılanlardan dört adet anlamlı basamak otomatik olarak çıkarılır. Decimal modülü, matematik işlemlerini elle yapılmış gibi hesaplar ve ikili kayan noktalı sayıların ondalık miktarları tam olarak temsil edemediği durumlarda ortaya çıkabilecek sorunları önler.

Tam temsil, `Decimal` sınıfının ölçekli hesaplamaları ve ikili kaynak sayılar için uygun olmayan eşitlik testlerini hesaplayabilmesini sağlar.

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995
```

```
>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> sum([0.1]*10) == 1.0
False
```

`decimal` modülü, gerektiği kadar hassasiyetle aritmetik işlemler yapılabilmesini sağlar.

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```

BÖLÜM 12

Sanal Ortamlar ve Paketler

12.1 Giriş

Python uygulamaları, standart kütüphanede bulunmayan paketler ve modüllerden sıklıkla faydalanacaktır. Uygulamalar bazen bir kütüphanenin belirli bir sürümüne ihtiyaç duyar, çünkü bu uygulamaların çalışabilmesi için belirli bir hatanın giderilmiş olması gerekiyordur veya bu uygulamalar kütüphane arayüzünün eski bir sürümü kullanılarak yazılmıştır.

Bu durum, tek bir Python kurulumunun bütün uygulamaların gereksinimlerine cevap veremeyebileceğini göstermektedir. Eğer A uygulaması belirli bir modülün 1.0 sürümüne ihtiyaç duyuyor, ama B uygulaması aynı modülün 2.0 sürümüne ihtiyaç duyuyorsa, bu durumda gereklilikler birbiriyle çakışacak ve modülün 1.0 veya 2.0 sürümünün kurulması uygulamalardan birinin çalışamaz hale gelmesine yol açacaktır.

Bu sorunun çözümü, Python'ın belirli bir sürümü ile birlikte ilave bazı paketlerin de kurulu olduğu, kendi içinde bağımsız bir dizin yapısına sahip bir sanal ortam (İngilizcede genellikle kısaca "virtualenv" denir) oluşturmaktır.

Bu şekilde, farklı uygulamalar farklı sanal ortamları kullanabilir. Biraz önce verdiğimiz çakışan gereklilikler örneğinde bir çözüme ulaşmak için, uygulama A, ilgili modülün 1.0 sürümünün kurulu olduğu, kendine ait bir sanal ortam içinde çalışırken, uygulama B de aynı modülün 2.0 sürümünü içeren başka bir sanal ortamda çalışabilir. Eğer uygulama B, ilgili kütüphanenin 3.0 sürümüne yükseltilmesini gerektirirse, bu durum uygulama A'nın içinde bulunduğu ortamı etkilemeyecektir.

12.2 Sanal Ortamlar Oluşturmak

Sanal ortamlar oluşturmak ve bu ortamları yönetmek için kullanılan betiğin adı `pyvenv`'dir. `pyvenv` genellikle sisteminizdeki en yeni Python sürümünü kullanacaktır. Bu betik aynı zamanda sürüm numarası da içerdiği için, eğer sisteminizde birden fazla Python sürümü varsa, `pyvenv-3.4` veya istediğiniz başka bir sürümü çalıştırarak belirli bir Python sürümünü de seçebilirsiniz.

Bir sanal ortam oluşturmak için, bu ortamı hangi dizine yerleştireceğinize karar verin ve `pyvenv` komutunu o dizin yoluyla birlikte çalıştırın:

```
pyvenv kilavuz-ort
```

Bu komut, eğer yoksa `kilavuz-ort` adlı bir dizin ile birlikte, Python yorumlayıcısının, standart kütüphanenin ve çeşitli destekleyici dosyaların bir kopyasını içeren alt dizinler oluşturacaktır.

Bir sanal ortamı oluşturduktan sonra bunu etkinleştirmeniz gerekir.

Bunun için Windows'ta şu komutu kullanıyoruz:

```
kilavuz-ort/Scripts/activate
```

Unix ve MacOS üzerinde ise şunu:

```
source kilavuz-ort/bin/activate
```

(Bu betik bash kabuğu için yazılmıştır. Eğer `csh` veya `fish` kabuklarını kullanıyorsanız, yukarıdaki yerine `activate.csh` ve `activate.fish` gibi alternatifleri kullanmalısınız.)

Sanal ortam etkinleştirildikten sonra komut satırınız, hangi sanal ortamı kullandığınızı gösterecek şekilde değişikliğe uğrayacak ve `python` komutu verdiğinizde sanal ortam içine kurulan özel Python sürümü çalışmaya başlayacaktır. Mesela:

```
-> source ~/ortamlar/kilavuz-ort/bin/activate
(kilavuz-ort) -> python
Python 3.4.3+ (3.4:c7b9645a6f35+, May 22 2015, 09:31:25)
  ...
>>> import sys
>>> sys.path
['', '/usr/local/lib/python34.zip', ...,
'~/ortamlar/kilavuz-ort/lib/python3.4/site-packages']
>>>
```

12.3 Paketleri pip ile Yönetmek

Bir sanal ortamı etkinleştirdikten sonra, `pip` adlı bir program yardımıyla paketleri kurabilir, güncelleyebilir ve kaldırabilirsiniz. `pip` programı, öntanımlı olarak, <https://pypi.python.org/pypi> adresinde yer alan Python Paket Dizini'ndeki paketleri kurar. Python Paket Dizini'ne web tarayıcınız aracılığıyla göz atabilir veya `pip`'in sınırlı arama özelliğini kullanabilirsiniz:

```
(kilavuz-ort) -> pip search astronomy
skyfield               - Elegant astronomy for Python
gary                   - Galactic astronomy and gravitational dynamics.
novas                  - The United States Naval Observatory NOVAS astronomy library
astroobs               - Provides astronomy ephemeris to plan telescope observations
PyAstronomy            - A collection of astronomy related tools for Python.
...
```

`pip` birkaç alt komuta da sahiptir: "search" (arama), "install" (kurma), "uninstall" (kaldırma), "freeze" (özel listeleme), vb. ("pip" hakkında eksiksiz bir kılavuz için Python Modüllerinin Kurulması başlıklı bölüme bakınız).

Bir paketin en yeni sürümünü, o paketin ismini belirterek kurabilirsiniz:

```
-> pip install novas
Collecting novas
  Downloading novas-3.1.1.3.tar.gz (136kB)
Installing collected packages: novas
  Running setup.py install for novas
Successfully installed novas-3.1.1.3
```

Bir paket isminin ardından == işaretini ve sürüm numarasını belirterek bir paketin belirli bir sürümünü de kurabilirsiniz:

```
-> pip install requests==2.6.0
Collecting requests==2.6.0
  Using cached requests-2.6.0-py2.py3-none-any.whl
Installing collected packages: requests
Successfully installed requests-2.6.0
```

Eğer bu komutu tekrar çalıştırırsanız, `pip` talep ettiğiniz sürümün zaten kurulu olduğunu farkedecek ve herhangi bir işlem yapmayacaktır. Farklı bir sürüm numarası belirterek o sürümü kurabilir veya `pip install --upgrade` komutu ile, ilgili paketi en son sürümüne güncelleyebilirsiniz:

```
-> pip install --upgrade requests
Collecting requests
Installing collected packages: requests
  Found existing installation: requests 2.6.0
    Uninstalling requests-2.6.0:
      Successfully uninstalled requests-2.6.0
Successfully installed requests-2.7.0
```

`pip uninstall` komutu bir veya birden fazla paket adı ile birlikte kullanıldığında, belirtilen paketler sanal ortamdan kaldırılacaktır.

`pip show` komutu, belirli bir paket hakkında bilgi görüntülemenizi sağlar:

```
(kilavuz-ort) -> pip show requests
---
Metadata-Version: 2.0
Name: requests
Version: 2.7.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.com
License: Apache 2.0
Location: /Users/akuchling/ortamlar/kilavuz-ort/lib/python3.4/site-packages
Requires:
```

`pip list` komutu, sanal ortamda kurulu bütün paketleri gösterecektir:

```
(kilavuz-ort) -> pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

`pip freeze` komutu da kurulu paketlere ilişkin benzer bir liste üretecek, ancak bu komut `pip list` komutundan farklı olarak, `pip install` komutu ile birlikte kullanılmaya müsait bir biçime sahip çıktı verecektir. Bu listenin bir `gereklilikler.txt` dosyasına gönderilmesi yaygın bir uygulamadır:

```
(kilavuz-ort) -> pip freeze > gereklilikler.txt
(kilavuz-ort) -> cat gereklilikler.txt
novas==3.1.1.3
```

```
numpy==1.9.2
requests==2.7.0
```

Daha sonra bu `gereklilikler.txt` dosyasını sürüm kontrol sisteminize gönderip, uygulamanızın bir parçası olarak dağıtabilirsiniz. Böylece kullanıcılarınız gerekli bütün paketleri `install -r` komutuyla kurabilir:

```
-> pip install -r gereklilikler.txt
Collecting novas==3.1.1.3 (from -r gereklilikler.txt (line 1))
  ...
Collecting numpy==1.9.2 (from -r gereklilikler.txt (line 2))
  ...
Collecting requests==2.7.0 (from -r gereklilikler.txt (line 3))
  ...
Installing collected packages: novas, numpy, requests
  Running setup.py install for novas
Successfully installed novas-3.1.1.3 numpy-1.9.2 requests-2.7.0
```

`pip` daha pek çok seçeneğe sahiptir. `pip`'e ilişkin eksiksiz bir belgelendirme için Python Modüllerinin Kurulması başlıklı bölüme bakınız. Bir paket yazıp bu paketi Python Paket Dizini aracılığıyla sunmak isterseniz, Python Modüllerinin Dağıtılması başlıklı kılavuza başvurabilirsiniz.

Ya Şimdi?

Bu kılavuzu okuduktan sonra herhalde Python kullanmaya olan ilginiz pekişmiştir. Öyle ki, Python'ı gerçek hayatta karşılaştığınız sorunları çözmede kullanmak için sabırsızlanıyor olmalısınız. Peki Python hakkında daha fazla bilgi edinmek için acaba buradan sonraki durağınız neresi olmalı?

Bu kılavuz, Python belgelerini içeren kümenin bir parçasıdır. Bu kümedeki başka belgelerse şunlardır:

- Python'ın Standart Kütüphanesi:

  Standart kütüphane içindeki veri tipleri, fonksiyonlar ve modüller hakkında eksiksiz (ama kısa ve öz) referans materyalleri sunan bu kılavuza mutlaka göz atın. Standart Python dağıtımı içinde *pek çok* ilave kod bulunur. Unix posta kutularını okumak, HTTP ile belgeleri almak, rastgele sayılar üretmek, komut satırı seçeneklerini ayrıştırmak, CGI programları yazmak, veri sıkıştırmak ve daha başka pek çok görev için modüller bulunur. Kütüphane Referansı'na göz gezdirmek elinizdeki imkanlara dair size bir fikir verecektir.

- Python Modüllerinin Kurulması bölümünde, başka Python kullanıcıları tarafından yazılmış ilave modüllerin nasıl kurulacağına dair bilgiler bulacaksınız.

- Python'ın Dil Referansı: Bu bölümde Python'ın söz dizimi ve anlam yapısı hakkında ayrıntılı açıklamalar yer alır. Ağır bir metin olsa da, dilin kendisine dair eksiksiz bir kılavuz olması bakımından faydalıdır.

Başka Python kaynakları:

- https://www.python.org: Resmi Python Web Sitesi. Bu adreste, Python kodları, belgeleri ve internetteki Python'a ilişkin sayfalara bağlantılar bulacaksınız. Bu web sitesi, Avrupa, Japonya ve Avustralya gibi dünyanın pek çok bölgesinde yansılara sahiptir. Bulunduğunuz coğrafi konuma bağlı olarak, yansılar ana siteden daha hızlı olabilir.

- https://docs.python.org: Python belgelerine hızlı erişim.

- https://pypi.python.org: Python Paket Dizini. Önceleri Peynir Dükkanı olarak isimlendirilen bu site, kullanıcılar tarafından oluşturulmuş, indirilebilir Python modüllerini içeren bir dizindir. Kodlarınızı yayımlamak istediğinizde bunları burada kaydettirebilirsiniz. Böylece başkaları da kodlarınızı bulabilir.

- http://code.activestate.com/recipes/langs/python/: The Python Cookbook; kod örnekleri, büyük modüller ve faydalı betiklerden oluşan geniş bir koleksiyondur Dikkati çeken kod katkıları yine "Python Cookbook" adını taşıyan bir kitapta toplanmıştır. (O'Reilly & Associates, ISBN 0-596-00797-3.)

- http://www.pyvideo.org sitesi, konferanslar ve kullanıcı grubu toplantılarındaki Python'a ilişkin videolara bağlantılar içerir.
- http://scipy.org: Bilimsel Python projesi; hızlı dizi hesaplamaları ve manipülasyonu artı lineer cebir, Fourier dönüşümleri, non-lineer çözücüler, rastgele sayı dağılımları, istatistik analiz ve benzeri konularda paketler içerir.

Python'a ilişkin sorularınız ve sorun bildirimleriniz için, *comp.lang.python* adlı haber grubuna yazabilir veya bunları python-list@python.org adresindeki posta listesine gönderebilirsiniz. Haber grubu ve posta listesi ağa açıktır, dolayısıyla bir kişiye yolladığınız mesajlar otomatik olarak öbürlerine de gidecektir. Burada hergün yüzlerce mesaj gönderilmekte, sorular sorulmakta (ve cevaplar verilmekte), yeni özellikler talep edilmekte ve yeni modüller duyurulmaktadır. Posta listesi arşivlerine https://mail.python.org/pipermail/ adresinden ulaşabilirsiniz.

Buraya mesaj göndermeden önce Sık Sorulan Sorular listesine (SSS) mutlaka göz gezdirin. SSS'de tekrar tekrar sorulan sorular cevaplandırılmış olduğundan, soracağınız sorunun cevabını burada da bulabilirsiniz.

Etkileşimli Girdi Düzenleme ve Eski Kodları Getirme

Korn kabuğu ve GNU Bash kabuğundakine benzer şekilde Python yorumlayıcısının bazı sürümleri mevcut girdi satırını düzenlemeyi ve geçmiş öğeleri değiştirmeyi destekler. Bu özellik, çeşitli düzenleme tarzlarını destekleyen GNU Readline kütüphanesi kullanılarak gerçeklenmiştir. Bu kütüphanenin kendine ait bir kılavuzu var, o yüzden bu kılavuzun içeriğini burada tekrarlamayacağız.

14.1 Kod Tamamlama ve Eski Kodları Düzenleme

Değişken ve modül ismi tamamlama işlevi, bu işlev `Tab` tuşu kullanılarak yerine getirilecek şekilde, yorumlayıcı başlatıldığında otomatik olarak etkinleştirilir. Bu işlev için Python deyim isimlerine, mevcut lokal değişkenlere ve mevcut modül adlarına bakılır. `karakter_dizisi.a` gibi noktalı ifadelerde ifade en son `"."` işaretine kadar işletilir ve ardından elde edilen nesnenin niteliklerine bakılarak tamamlama önerilir. Eğer ifadenin içinde `__getattr__()` metoduna sahip bir nesne varsa, bu özellik, uygulama tarafından tanımlanan bir kodu yürütebilir. Ayrıca öntanımlı yapılandırmaya göre, kod geçmişi, kullanıcı dizini altındaki `.python_history` adlı bir dosyaya yazılır. Bu geçmiş, etkileşimli yorumlayıcının bir sonraki oturumunda da kullanılabilir durumda olacaktır.

14.2 Etkileşimli Kabuğun Alternatifleri

Bu kolaylık, yorumlayıcının eski sürümleriyle karşılaştırıldığında ileriye doğru atılmış muazzam bir adımdır. Ancak olması gereken başka şeyler de yok değil: Örneğin devam satırlarında uygun miktarda girintileme kendiliğinden yapılabilse hoş olurdu (ayrıştırıcı bir sonraki adımda girinti gerekip gerekmediğini anlayabiliyor). Kod tamamlama mekanizmasında yorumlayıcının simge tablosundan faydalanılabilir. Kapanış parantezlerini, tırnak işaretlerini, vb. kontrol eden (hatta öneren) bir komut da faydalı olabilirdi.

Uzun bir süredir ortalıkta olan gelişmiş bir alternatif etkileşimli yorumlayıcı IPython'dır. Bu araç, sekme ile kod tamamlama, nesne inceleme ve gelişmiş geçmiş yönetimi gibi özellikler barındırır. Ayrıca, IPython baştan aşağı özelleştirilebilir ve başka uygulamalara da iliştirilebilir. Bir başka benzer gelişmiş etkileşimli ortam da bpython'dır.

Kayan Noktalı Sayılarla Aritmetik İşlemler: Sorunlar ve Kısıtlamalar

Kayan noktalı sayılar bilgisayar donanımında 2 tabanlı (ikili) kesirler olarak temsil edilir. Örneğin şu onlu kesir:

```
0.125
```

1/10 + 2/100 + 5/1000 değerine, aynı şekilde şu ikili kesir ise

```
0.001
```

0/2 + 0/4 + 1/8 değerine sahiptir. Bu iki kesirin değerleri eşittir. Bunların arasındaki tek gerçek fark, ilkinin 10 tabanındaki kesirli gösterime göre, ikincisinin ise 2 tabanındakine göre yazılmış olmasıdır.

Ne yazık ki çoğu onlu kesir, ikili kesir olarak tam anlamıyla temsil edilemez. Bunun bir sonucu olarak, girdiğiniz onlu kayan noktalı sayılar, bilgisayarda depolanan asıl ikili kayan noktalı sayılar tarafından genellikle ancak yaklaşık olarak temsil edilebilir.

Bu problemi ilk olarak 10 tabanında anlamak daha kolaydır. 1/3 kesirine bakalım. Bunu 10 tabanlı bir kesir biçiminde yaklaşık olarak şöyle temsil edebilirsiniz:

```
0.3
```

ya da daha iyisi,

```
0.33
```

ya da daha iyisi,

```
0.333
```

bu böyle devam eder gider... Ne kadar hane eklerseniz ekleyin, sonuç asla tam olarak 1/3 olmayacak, ama her defasında 1/3'e biraz daha yaklaşan bir sayı elde edilecektir.

Aynı şekilde 2 tabanında kaç hane eklerseniz ekleyin, onlu 0.1 değeri 2 tabanlı bir kesir olarak tam anlamıyla temsil edilemeyecektir. 2 tabanında 1/10 sonsuza kadar tekrar eden bir kesirdir:

```
0.0001100110011001100110011001100110011001100110011...
```

Hangi bit'te durursanız durun elde ettiğiniz şey yaklaşık bir değer olacaktır. Bugünkü makinelerin çoğunda kayan noktalı sayılar, pay en anlamlı bitten başlayarak ilk 53 biti kullanacak şekilde ve payda ise ikinin kuvveti olacak şekilde ikili bir kesir kullanılmak suretiyle

yaklaşık olarak hesaplanır. 1/10 örneğinde, ikili kesir `3602879701896397 / 2 ** 55` olup, 1/10'un gerçek değerine çok yakındır, ama yine de tam olarak eşit değildir.

Pek çok kullanıcı, değerlerin gösteriliş şekli nedeniyle bu hesaplamanın yaklaşık olduğunun farkında değildir. Python, makine tarafından depolanan yaklaşık ikili değerin gerçek ondalık değerine yakın bir ondalık sayı basar. Eğer Python 0.1 için depolanan yaklaşık ikili değerin gerçek ondalık değerini basacak olsaydı, çoğu makinede şöyle bir gösterimi tercih etmesi gerekecekti:

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

Ancak bu çıktı, çoğu kimse açısından haddinden fazla basamak içeriyor. Bu nedenle Python böyle bir durumda yuvarlak bir değer göstererek basamak sayısının makul düzeyde kalmasını sağlar:

```
>>> 1 / 10
0.1
```

Ancak unutmayın, ekranda görünen sonuç 1/10'un tam değeriymiş gibi dursa da aslında depolanan değer temsil edilebilen en yakın ikili kesirdir.

İlginç bir şekilde, aynı en yakın ikili kesiri paylaşan farklı pek çok onlu sayı vardır. Örneğin, `0.1`, `0.10000000000000001` ve `0.1000000000000000055511151231257827021181583404541015625` sayılarının hepsinin yaklaşık değeri `3602879701896397 / 2 ** 55` sayısıdır. Bu onlu değerlerin hepsi aynı yaklaşık değere paylaştığı için, `eval(repr(x)) == x` sabiti muhafaza edilerek bunların herhangi biri görüntülenebilir.

Geçmişte Python komut istemcisi ve gömülü `repr()` fonksiyonu 17 anlamlı basamağa sahip olan `0.10000000000000001` değerini tercih ediyordu. Ama artık Python, 3.1 sürümünden bu yana (çoğu sistemde) bunların en kısasını seçip `0.1` değerini görüntüleyebiliyor.

Bu özellik aslında ikili kayan noktalı sayıların doğasında olan bir şeydir. Yani bu Python'dan kaynaklanan bir kusur (bug) değildir, aynı zamanda bu sizin kodunuzdan kaynaklanan bir kusur da değildir. Kayan noktalı sayılarla aritmetik işlemler yapabilen donanım parçalarını destekleyen bütün dillerde aynı şeyi görürsünüz (bazı diller farkı öntanımlı olarak veya en azından bütün çıktı kiplerinde görüntülemiyor olsa bile).

Göze daha hoş görünen bir çıktı için, sınırlı sayıda anlamlı basamak üretmek amacıyla karakter dizisi biçimlendirmeden yararlanabilirsiniz:

```
>>> format(math.pi, '.12g')  # 12 anlamlı basamak verir
'3.14159265359'

>>> format(math.pi, '.2f')   # noktadan sonra 2 basamak verir
'3.14'

>>> repr(math.pi)
'3.141592653589793'
```

Aslında bunun düpedüz bir yanılsama olduğunu kavramak önemlidir. Yaptığınız şey, gerçek makine değerinin *görüntüsünü* yuvarlamaktan başka bir şey değildir.

Bir yanılgı başka bir yanılgının kapısını aralar. Örneğin, 0.1 aslında tam olarak 1/10 olmadığı için, üç adet 0.1 değerini toplamak da tam olarak 0.3 değerini vermeyecektir:

```
>>> .1 + .1 + .1 == .3
False
```

Ayrıca 0.1, 1/10'un gerçek değerine yaklaşamadığı için, 0.3 de 3/10'un gerçek değerine yaklaşamaz. Dolayısıyla `round()` fonksiyonu yardımıyla ilgili değeri işlem öncesinde yuvarlamak fayda etmeyecektir:

```
>>> round(.1, 1) + round(.1, 1) + round(.1, 1) == round(.3, 1)
False
```

Sayılar arzu edilen kesin değerlerine yaklaştırılamasa da `round()` fonksiyonu kullanılarak işlem sonrası yuvarlama yapılabilir ve böylece kesin olmayan değerler birbirleriyle karşılaştırılabilir:

```
>>> round(.1 + .1 + .1, 10) == round(.3, 10)
True
```

İkili kayan noktalı aritmetik işlemlerin böyle daha pek çok şaşırtıcı yönü vardır. "0.1" değeri ile ilgili problem aşağıda "Temsil Hatası" bölümünde bütün ayrıntılarıyla anlatılmıştır. Sıklıkla karşılaşılan başka şaşırtıcı şeylere ilişkin daha bütünlüklü bir anlatım için The Perils of Floating Point adlı makaleyi inceleyebilirsiniz.

Makalenin sonlarına doğru ifade edildiği gibi, "bu durumun basit bir çözümü yoktur." Ama siz yine de lüzumsuz yere kayan noktalı sayılardan ürkmeyin! Python'daki kayan noktalı işlemlere dair hatalar kayan noktalı sayılara ilişkin donanımdan miras alınmıştır ve bu hatalar çoğu makinede işlem başına 2**53'te 1'den fazla değildir. Çoğu görev için bu kadar hassasiyet yeter de artar bile, ancak öyle ya da böyle, bunun onlu aritmetik olmadığını ve bütün kayan noktalı işlemlerin yeni bir yuvarlama hatasından muzdarip olabileceğini aklımızdan çıkarmamalıyız.

Sıradışı vakalar bulunsa da, kayan noktalı aritmetiğin kullanımına ilişkin çoğu gündelik durumda, nihai sonuçlarınızın görünümünü, istediğiniz onlu basamağa sahip olacak şekilde yuvarlarsanız beklediğiniz çıktıyı elde edebilirsiniz. Genellikle `str()` yetecektir. Daha ince ayar için ise Biçimlendirme Dizilerinin Söz Dizimi içinde anlatılan, `str.format()` metodunun biçim düzenleyicilerini inceleyebilirsiniz.

Kesin onlu temsil gerektiren durumlarda, onlu aritmetiği muhasebe uygulamaları ve yüksek kesinlik isteyen uygulamalara uygun olacak şekilde gerçekleyen `decimal` modülünü kullanabilirsiniz.

Kesin aritmetiğin bir başka biçimi de `fractions` modülü aracılığıyla desteklenir. Bu modülde aritmetik işlemleri, rasyonel sayılara dayanılarak gerçeklenir (dolayısıyla 1/3 gibi sayılar kesin olarak temsil edilebilir).

Eğer siz kayan noktalı sayı işlemlerini yoğun bir biçimde kullanıyorsanız, Nümerik Python paketine ve SciPy projesi ile birlikte gelen matematik ve istatistik işlemlere ilişkin diğer paketlere göz atabilirsiniz. Bkz. <http://scipy.org>.

Python, bir kayan noktalı sayının kesin değerini gerçekten bilmeniz gereken nadir durumlar için size yardımcı olabilecek araçlar sunar. `float.as_integer_ratio()` metodu bir kayan noktalı sayının değerini bir kesir olarak ifade eder:

```
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

Oran kesin olduğu için, bu değer özgün değeri kayıpsız olarak yeniden oluşturmak için kullanılabilir:

```
>>> x == 3537115888337719 / 1125899906842624
True
```

`float.hex()` metodu kayan noktalı sayıları onaltılı düzende (16 tabanı) ifade eder ve bilgisayarda depolanan kesin değeri verir.

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

Bu hassas onaltılı temsil, kayan noktalı sayıyı kesin olarak yeniden meydana getirmek için kullanılabilir:

```
>>> x == float.fromhex('0x1.921f9f01b866ep+1')
True
```

Temsil kesin olduğu için, bu yöntemler, değerleri farklı Python sürümleri arasında güvenilir bir şekilde taşımak (platform bağımsızlığı) ve aynı biçimi destekleyen başka dillerle (Java ve C99 gibi) verileri paylaşabilmek açısından faydalıdır.

Başka bir faydalı araç da `math.fsum()` fonksiyonudur. Bu fonksiyon toplama işlemi sırasında hassasiyet kaybını azaltmaya yardımcı olur. Ayrıca değerler büyüyen toplama eklendikçe "kayıp basamakları" takip eder. Bu fonksiyon, hataların son toplamı etkileyecek kadar yığılmasını engellediği için, genel doğruluk düzeyini artırabilir.

```
>>> sum([0.1] * 10) == 1.0
False
>>> math.fsum([0.1] * 10) == 1.0
True
```

15.1 Temsil Hatası

Bu bölümde, "0.1" örneği ayrıntılı olarak açıklanacak ve buna benzer durumlara ilişkin kesinlik analizlerini kendi kendinize nasıl yapacağınız gösterilecektir. İkili kayan noktalı temsil konusunu, temel olarak bildiğiniz varsayılmıştır.

Temsil hatası; bazı (aslında çoğu) onlu kesirlerin ikili (2 tabanı) kesir olarak tam temsil edilemeyeceği gerçeğini ifade eder. Python'ın (veya Perl, C, C++, Java, Fortran ve pek çok başka dilin) beklediğiniz tam onlu sayıyı çoğunlukla gösterememesinin başlıca sebebi budur.

Peki ama neden? 1/10, ikili bir kesir olarak tam temsil edilemez. Bugünkü hemen hemen bütün makineler (Kasım 2000) IEEE-754'te tarif edilen kayan noktalı aritmetiği kullanır ve hemen hemen bütün platformlar Python'ın kayan noktalı sayılarını IEEE-754'deki "çift hassasiyet"e eşleştirir. 754 çift, 53 bitlik hassasiyet içerir. Dolayısıyla bilgisayar, girilen değeri, *J*/2***N* olacak şekilde 0.1'i en yakın kesire dönüştürmeye çalıştırır (burada *J* tam olarak 53 bitten oluşan bir tam sayıdır.) Aşağıdaki kodu

```
1 / 10 ~= J / (2**N)
```

şöyle yazdığımızda:

```
J ~= 2**N / 10
```

ve *J*'nin tamı tamına 53 bit içerdiği düşünüldüğünde (`>= 2**52` ama `< 2**53`), *N* için en iyi değer 56 olacaktır:

```
>>> 2**52 <=  2**56 // 10  < 2**53
True
```

Yani, *J*'yi tam 53 bit ile bırakan tek *N* değeri 56'dır. Bu durumda *J* için en olası değer ise yuvarlanan bölüm olur:

```
>>> q, r = divmod(2**56, 10)
>>> r
6
```

Kalan, 10'un yarısından fazla olduğu için, en iyi yaklaşık değer yuvarlama ile elde edilebilir:

```
>>> q+1
7205759403792794
```

Dolayısıyla 1/10'un 754 çift hassasiyet dahilinde en olası yaklaşık değeri şudur:

```
7205759403792794 / 2 ** 56
```

Hem bölüneni hem de böleni ikiye bölmek kesiri aşağıdaki değere sadeleştirecektir:

```
3602879701896397 / 2 ** 55
```

Ancak biz burada yuvarlama işlemi yaptığımız için, bu değer 1/10'dan birazcık büyük olacaktır. Eğer yuvarlamasaydık bölüm 1/10'dan birazcık daha küçük olacaktı. Ancak hiçbir koşulda *tam olarak* 1/10 olamayacaktı!

Dolayısıyla bilgisayar 1/10'u asla "görmez": Aslında gördüğü şey, tam olarak yukarıda verilen kesirdir. Yani 754 çift hassasiyet için olabilecek en yakın değer.

```
>>> 0.1 * 2 ** 55
3602879701896397.0
```

Eğer biz kesiri 10**55 ile çarparsak, sonucu 55 onlu basamağa kadar görebiliriz:

```
>>> 3602879701896397 * 10 ** 55 // 2 ** 55
1000000000000000055511151231257827021181583404541015625
```

Bu, bilgisayarda depolanan gerçek sayının 0.1000000000000000055511151231257827021181583404541015 onlu değerine eşit olduğu anlamına gelmektedir. Pek çok dil, tam onlu değeri görüntülemek yerine (Python'ın eski sürümleri de dahil), sonucu 17 anlamlı basamağa yuvarlar.

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

`fractions` ve `decimal` modülleri bu hesaplamaları kolaylaştırır:

```
>>> from decimal import Decimal
>>> from fractions import Fraction

>>> Fraction.from_float(0.1)
Fraction(3602879701896397, 36028797018963968)

>>> (0.1).as_integer_ratio()
(3602879701896397, 36028797018963968)

>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

```
>>> format(Decimal.from_float(0.1), '.17')
'0.10000000000000001'
```

BÖLÜM 16

Ek

16.1 Etkileşimli Kip

16.1.1 Hata Yakalama

Yorumlayıcı, bir hata oluştuğunda ekrana bir hata mesajı ile birlikte bir yığın izi (stack trace) basar. Daha sonra, eğer etkileşimli kipte iseniz, birincil komut istemine döner. Eğer girdi bir dosyadan geliyorsa, yığın izini ekrana bastıktan sonra sıfır harici bir çıkış durumu vererek kapanır. (`try` deyimleri içindeki `except` cümlecikleri tarafından yakalanan istisnalar bu bağlamda birer hata olarak değerlendirilmez.) Bazı hatalar ise mutlak surette ölümcüldür ve sıfır harici bir çıkış durumu vererek programın kapanmasına yol açar. Böyle bir şey, programın işleyişinden kaynaklanan bir tutarsızlık oluşması halinde ve bellek yetersizliğinden kaynaklanan bazı durumlarda ortaya çıkar. Bütün hata mesajları standart hata akışına yazılır. Yürütülen komutlardan gelen normal çıktılar ise standart çıktıya gönderilir.

Birincil veya ikincil komut isteminde kesme karakterinin (genellikle `Control-C` veya `Delete`) verilmesi girdiyi iptal eder ve birincil komut istemine dönülmesini sağlar. Kesme karakterinin bir komutun işleyişi esnasında verilmesi `KeyboardInterrupt` adlı istisnayı tetikler. Bu istisna `try` deyimleri yardımıyla yakalanabilir.

16.1.2 Çalıştırılabilir Python Betikleri

BSD benzeri Unix sistemlerinde Python betikleri, tıpkı kabuk betikleri gibi, betiğin en başına

```
#!/usr/bin/env python3.5
```

satırının yerleştirilip dosyaya çalıştırma yetkisinin verilmesi ile doğrudan çalıştırılabilir hale getirilebilir (burada yorumlayıcının, kullanıcıya ait `PATH` değişkeni içinde yer aldığını varsayıyoruz). Dosyanın ilk iki karakteri `#!` olmalıdır. Bazı platformlarda ilk satır, Windows tarzı satır sonu (`'\r\n'`) ile değil de, Unix tarzı satır sonu ile (`'\n'`) bitmek zorundadır. Diyez veya kare adlı karakter (`'#'`) Python'da yorum satırlarının başında da kullanılır.

`chmod` komutu kullanılarak betiğe çalıştırma kipi veya izni verilebilir.

```
$ chmod +x betik_adi.py
```

Windows sistemlerinde "çalıştırma kipi" diye bir kavram bulunmaz. Python kurulum programı `.py` dosyalarını otomatik olarak `python.exe` ile ilişkilendirir, böylece Python dosyalarına çift tıkladığınızda bu dosya bir betik olarak çalıştırılır. Dosya uzantısı `.pyw` de olabilir. Bu durumda, normal şartlarda arkaplanda beliren komut penceresi görünmez.

16.1.3 Etkileşimli Başlangıç Dosyası

Python'ı etkileşimli bir şekilde kullanırken, yorumlayıcının her açılışında bazı standart komutların yürütülmesini sağlamak çoğu durumda işlerinizi kolaylaştırabilir. Bu iş için, `PYTHONSTARTUP` adlı bir çevre değişkeninin değerini, ihtiyaç duyduğunuz başlangıç komutlarını içeren bir dosya adı olacak şekilde ayarlayabilirsiniz. Unix kabuklarındaki `.profile` özelliğine benzer bir şeydir bu.

Bu dosya, Python komutlarının bir betikten okunduğu ve komut kaynağı olarak `/dev/tty`'nin açıkça belirtildiği durumlarda değil, yalnızca etkileşimli oturumlarda okunacaktır (eğer /dev/tty açıkça belirtilmezse etkileşimli oturumda olduğunuz varsayılacaktır). Dosya, etkileşimli komutlar ile aynı isim alanı içinde yürütülür. Böylece bu dosya içinde tanımlanan veya içe aktarılan nesneler, etkileşimli kabukta bu dosyanın adını belirtmeye gerek olmadan kullanılabilecektir. Bu dosya aracılığıyla ayrıca `sys.ps1` ve `sys.ps2` olarak belirtilen istemcileri de değiştirebilirsiniz.

Mevcut dizin içindeki başka bir başlangıç dosyasını da okumak istiyorsanız, bu isteğinizi global başlangıç dosyasında `if os.path.isfile('.pythonrc.py'): exec(open('.pythonrc.py').read())` gibi bir kod yardımıyla yerine getirebilirsiniz. Eğer başlangıç dosyanızı bir betik içinde kullanmak istiyorsanız, bunu betik içinde açıkça belirtmelisiniz:

```
import os
dosya_adı = os.environ.get('PYTHONSTARTUP')
if dosya_adı and os.path.isfile(dosya_adı):
    with open(dosyadı) as fobj:
        başlangıç_dosyası = fobj.read()
    exec(başlangıç_dosyası)
```

16.1.4 Özelleştirme Modülleri

Python, kendisini özelleştirebilmeniz için `sitecustomize` ve `usercustomize` adlı iki adet kanca (hook) sunar size. Bunların nasıl çalıştığını görmek için, öncelikle kullanıcıya ait site-packages dizininin yerini tespit etmeniz gerekir. Bunun için Python'ı başlatıp şu komutu verin:

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python3.5/site-packages'
```

Artık bu dizin içinde `usercustomize.py` adlı bir dosya oluşturabilir, içine istediğiniz her şeyi koyabilirsiniz. Bu dosya, yorumlayıcıyı `-s` seçeneği ile başlatarak otomatik içe aktarma özelliğini devre dışı bırakmadığınız sürece Python her çalıştığında devreye girecektir.

`sitecustomize` dosyası da aynı şekilde işler, ancak bu dosya genellikle bilgisayarın yöneticisi tarafından, sistem genelini etkileyen site-packages dizini içinde oluşturulur ve `usercustomize` dosyasından önce içe aktarılır. Daha fazla ayrıntı için `site` modülünün belgelerine bakabilirsiniz.