

Python Kılavuzu

Guido van Rossum

PythonLabs

Email: python-docs@python.org

Fred L. Drake, Jr., editor

Release 2.2.2

22. Nisan 2003

Giriş

Copyright © 2001 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

Copyright © 2001 Dinçer Aydın, tercüme. Her hakkı saklıdır.

Copyright © 2011 TD Software, güncelleme. Her hakkı saklıdır.

See the end of this document for complete license and permissions information.

Özet:

Python kolay öğrenilen güçlü bir programlama dilidir. Verimli yüksek seviyeli veri tiplerine sahiptir ve nesne tabanlı programlamaya yaklaşımı basit ve etkilidir. Python'un sık sözdizimi, dinamik veri tipleri ve yorumlanan bir dil oluşu onu çoğu alan ve platformda hızlı yazılım geliştirme için ideal yapar.

Python yorumlayıcısı ve geniş standart kütüphanesi kaynak ya da binary formunda Python Web sitesinden (<http://www.python.org/>) ücretsiz olarak alınabilir ve dağıtılabilir. Aynı sitede farklı Python dağıtımları, ücretsiz modüller, programlar ve ek dokümanlar bulunabilir. Günümüzde yaygın olarak kullanılan işletim sistemlerinin çoğu için bir Python dağıtımı mevcuttur.

Python yorumlayıcısı C veya C++ (ya da C dilinden çağırılabilen başka bir dil) ile yazılmış veri tipleri ve fonksiyonlar ile genişletilebilir. Diğer dillerde yazdığınız programlarınıza da Python yorumlayıcısını bağlayabilir ve Python ile ek özellikler eklenebilen programlar yazabilirsiniz.

Bu kılavuz okuyucuya Python dilinin temel özelliklerini, kavramlarını ve sistemini tanıtmaktadır. Örnekleri denemek için el altında bir Python yorumlayıcısı bulundurmak yararlı olur.

Bu kılavuz Pythonu'un bütün özelliklerini ya da yaygın olarak kullanılan her özelliğini açıklamak amacıyla değildir. Bunun yerine Python'un kayda değer özelliklerinin çoğu tanıtılmaktadır ve dilin tarzı ile ilgili iyi bir fikir verilmektedir. Bunu okuduktan sonra Python modülleri ve programlarını okuyup yazabileceğiniz gibi Python ile gelen geniş kütüphane ile ilgili daha çok şey öğrenmeye hazır olacaksınız.

1. İştahınızı Kabartalım

Eğer büyük bir kabuk programı yazdıysanız neler olduğunu bilirsiniz. Bir özellik daha eklemek istersiniz; ancak program yeterince büyük ve yavaş olmuştur ya da istediğiniz özelliğe sadece C aracılığıyla erişilebilir ...Genellikle program C ile baştan yazılmaya geçecek önemde değildir ya da kabukta kolay olduğu halde C'de zor elde edilen özellikleri vardır. Belki de C ile yeterince iyi değilsiniz.

Bir diğer durum düşünün: birkaç C kütüphanesi ile çalışmanız gerekiyor ve normal yaz/derle/dene/tekrar derle döngüsü çok yavaş geliyor ve daha hızlı program yazmaya ihtiyacınız var. Belki de genişletilebilir bir program yazacaksınız; ancak bunun için yeni bir dil tasarlayıp bunun için gerekli yorumlayıcıyı yazıp programınıza ekleyeceksiniz.

Bu gibi durumlarda Python tam aradığınız dil olabilir. Python kullanımı basit fakat gerçek bir dildir. Büyük programlar için kabuktan daha uygundur ve C'den çok daha fazla hata denetimi yapar. Python *çok yüksek seviyeli* bir dil olup C ile verimli şekilde yazılması günler alabilecek yüksek seviyeli veri tiplerine sahiptir (sözlükler ve listeler gibi). Daha genel veri tipleri sayesinde Python *Awk* hatta *Perl*'den çok daha geniş bir yelpazede uygulama alanı bulabilir. Ayrıca Python'da pek çok şey en az o dillerdeki kadar kolaydır.

Python ile programlarınızı daha sonra diğer Python programlarınızda tekrar kullanabileceğiniz modüllere ayırabilirsiniz. Python geniş bir standart modül koleksiyonu ile size gelmektedir. Dosya giriş/çıkışı, ses, resim, matematiksel işlemler vs. ile ilgili modüller de vardır.

Python yorumlanan bir dil olduğu için program geliştirme sırasında size önemli miktarda zaman kazandırabilir çünkü derleme ve bağlama gerekmemektedir. Yorumlayıcıyı etkileşimli olarak da kullanabilirsiniz; böylece dilin özelliklerini kolayca deneyebilir, hızlı bir şekilde küçük programlar yazabilir, ya da aşağıdan-yukarı program geliştirme sırasında fonksiyonlarınızı test edebilirsiniz. Yorumlayıcı bir hesap makinesi olarak da kullanılabilir.

Python ile son derece sıkı ve okunabilir programlar yazabilirsiniz. Birkaç nedenden Python programları eşdeğer C veya C++ programlarından çok daha kısadrlar:

- Yüksek seviyeli veri türleri ile karmaşık işlemler tek bir ifade ile yazılabilir.
- Deyimlerin gruplanması başlama/bitme deyimleri (begin, end veya {} gibi) yerine blokların girintili yazılması ile sağlanır.
- Değişken veya argüman bildiriminin yapılması gerekmez.

Python *geniřletilebilir*: Eęer C programlamayı biliyorsanız Python'a kolayca yeni modüller ekleyebilir ya da programınızın hızlı alıřması gereken kısımlarını C ile yazabilirsiniz. C programlarınıza da Python yorumlayıcısını bağlayabilir ve Python ile ek özellikler eklenebilen programlar yazabilirsiniz.

Python programlama dili adını korkun bir sürüngenden deęil, 'Monty Python's Flying Circus' adlı bir BBC komedi dizisinden almıřtır.

1.1 Öğrenmek İçin...

Artık Python konusunda heyecanlandınız ve daha ayrıntılı olarak incelemek istiyorsunuz. Bir dili öğrenmenin en iyi yolu onu kullanmak olduğundan sizi Python kullanmaya davet ediyoruz.

Bir sonraki bölümde yorumlayıcıyı kullanmayı öğreteceęiz. Bu ok basit bir řey; ancak daha sonraki bölümlerin anlaşılması için önemli.

Kılavuzun devamında basit ifadeler, deyimler ve veri tipleri ile başlayıp, fonksiyonlar ve modüllerden kullanıcı tanımlı sınıflar gibi gelişmiş konulara kadar Python'un eřitli özellikleri örnekler ile anlatılmaktadır.

2. Yorumlayıcının Kullanımı

2.1 Yorumlayıcının alıřtırılması

Python yorumlayıcısını Windows işletim sisteminde Başlat > Programlar > Python 2.1> izleyerek alıřtırabilirsiniz. Python yorumlayıcısının DOS penceresinde alıřanı olduğu gibi Windows ortamında alıřanı da vardır. Python yorumlayıcısı genellikle C:\Python21 gibi bir dizine kurulur. Bu dizinde python.exe ve pythonw.exe adlı iki alıřtırılabilir dosya bulunur. Python.exe alıřtırıldığında yorumlayıcı etkileřimli kipte (interactive mode) bir DOS penceresinde alıřır. Pythonw.exe alıřtırıldığında ise hiçbir řey olmadığı gözlenir. Yorumlayıcı alıřtırılacak olan Python programının dosya adı parametre řeklinde verilerek de kullanılabilir. Örneęin python.exe deneme.py gibi. pythonw.exe deneme.py de aynı işi görür fakat bir öncekinde olduğu gibi bir DOS penceresi açılmadan deneme.py alıřtırılır; ancak bu programın yazdırıldığı mesajlar var ise bunlar görünmez. Unix türevi işletim sistemlerinde ise yorumlayıcıyı başlatmak için konsoldan python komutunu vermek yeterlidir.

2.1.1 Etkileřimli Kip

Komutların yorumlayıcıya satır satır girildięi duruma *etkileřimli kip* denir. Bu kipte birincil komut satırı (">>> "); řeklindeyken ikincil komut satırı (". . . ")řeklinde görülür. Örneęin:

```
python
Python 1.5.2b2 (#1, Feb 28 1999, 00:02:06) [GCC 2.8.1] on sunos5
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>>
```

Birden fazla satırlık bloklar girildiğinde satırlar ikincil komut satırında girintili yazılır:

```
>>> dünya_duzdur = 1
>>> if dünya_duzdur:
...     print "Düşmemeye dikkat et!"
...
Düşmemeye dikkat et!
```

Yorumlayıcıdan çıkmak için "import sys; sys.exit()" yazabilirsiniz. Etkileşimli kipte çalıştırılan programlarda bir hata bulunması halinde yorumlayıcı bir hata mesajı basar ve birincil komut satırına döner. Ayrıca etkileşimli kipte ya da Python programlarınızı yazarken DOS komut satırı yerine daha rahat bir ortamda çalışmak isteyebilirsiniz. Bunun için Python ile birlikte gelen IDLE'ı deneyebilirsiniz. IDLE Python ile yazılmış bir program. IDLE'ı Başlat > Programlar > Python 2.1 > IDLE izleyerek çalıştırabilirsiniz. Bazı IDLE komutları şöyledir:

- *Alt-p* bir önceki komut
- *Alt-n* bir sonraki komut
- *Enter* o anki komutu çalıştırır

Ayrıca PythonWin'i de deneyebilirsiniz. IDLE dan daha üstün olduğu pek çok yön var. Parametre hatırlatma, komut tamamlama, blok katlama, otomatik girintileme vs.

PythonWin'i ayrı indirmeniz gerek. PythonWin için <http://www.activestate.com/Products/ActivePython/win32all.html> ziyaret edip uygun win32all paketini indirip kurarsanız PythonWin'e kavuşmuş olursunuz.

2.1.2 Hataların Yakalanması

Bir hata meydana geldiğinde yorumlayıcı bir hata mesajı ve hatalı noktaya gelmeden önce işletilen fonksiyon çağrılarının listesini basar.

3. Python'a Giriş

Aşağıdaki örneklerde giriş ve çıkış komut satırının varlığına veya yokluğuna (">>> " and "... ") bağlıdır. Örnekleri tekrar etmek için komut satırında görünen her şeyi yazmalısınız. (">>> " veya "... ") ile başlamayan bütün satırlar yorumlayıcı çıktısını temsil ederler. İkincil komut satırındaki boş bir satır (sadece "... ") olan yerlerde bir şey yazmanıza gerek yok. O satırlar boş olup bir deyimler öbeğinin bitirilmesi için kullanılırlar. Bu kılavuzdaki alıştırmaların çoğu, etkileşimli komut satırına yazılanlar dahil, açıklamalar içerirler. Python dilinde açıklamalar "#" ile başlarlar ve bulundukları satır sonuna kadar devam ederler. Bir

karakter dizisi içinde bulunan "#" işareti bir açıklama başlatmaz. Bu sadece bir "#" karakteridir. Örnekler:

```
# Bu bir açıklama.  
SAYI = 1 # ve bu ikinci açıklama  
# ... bu da üçüncü!  
DIZGI = "# Bu bir açıklama değil."
```

3.1 Python'u Hesap Makinesi Olarak Kullanmak

3.1.1 Sayılar

Şimdi bazı basit komutlar deneyelim. Yorumlayıcıyı çalıştırın ve birincil komut satırının gelmesini bekleyin ">>> "

Yorumlayıcı basit bir hesap makinesi olarak iş görebilir: istediğiniz herhangi bir ifadeyi yazın ve yorumlayıcı sonucu verecektir. İşleçler (+, -, *, /) çoğu programlama dillerindekine benzer çalışır (Pascal ve C de olduğu gibi mesela) . Ifadeleri gruplamak için parantezler de kullanılabilir. Örnekler:

```
2+2  
4  
>>> # Bu bir açıklama  
... 2+2  
4  
>>> 2+2 # bu da komutlarla aynı satırda bir açıklama  
4  
>>> (50-5*6)/4  
5  
>>> # Tam sayı bölme işlemlerinde ise:  
... 7/3  
2  
>>> 7/-3  
-3
```

C de olduğu gibi eşit işareti("=") bi değişkene değer atamak için kullanılır. Atamanın değeri çıktıya yazılmaz:

```
>>> genislik = 20  
>>> yukseklik = 5*9  
>>> genislik * yukseklik  
900
```

Bir değer aynı anda birden fazla değişkene atanabilir:

```
x = y = z = 0 # x, y ve z sıfır değerini alır  
>>> x  
0
```

```
>>> y
0
>>> z
0
```

Tam kayar nokta desteği vardır. Farklı türdeki değerlerin olduğu işlemlerde sonuç kayar noktalıya dönüştürülür:

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

Karmaşık sayılar da desteklenmektedir. Sayıların sanal kısımları "j" veya "J" eki ile yazılır. Gerçek kısmı sıfır olmayan karmaşık sayılar "(*gerçek* + *sanal*j)" şeklinde yazılırlar ya da "complex(*gerçek*, *sanal*)" fonksiyonu ile yaratılırlar.

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Karmaşık sayılar daima iki kayar noktalı sayı ile ifade edilirler; biri gerçek diğer sanal kısım için. Z gibi bir karmaşık sayının gerçek ya da sanal kısımlarına erişmek için `Z.real` ve `Z.imag` kullanılır.

```
a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

Tam sayı veya kayar noktalıya dönüştürme fonksiyonları (`float()`, `int()` ve `long()`) karmaşık sayılar için çalışmazlar; bir karmaşık sayıyı gerçek bir sayıya dönüştürmenin doğru bir yolu mevcut değildir. `abs(z)` ile karmaşık sayının büyüklüğünü ve `z.real` ile gerçek kısmını elde edebilirsiniz.

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use e.g. abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a) # sqrt(a.real**2 + a.imag**2)
5.0
>>>
```

Etkileşimli kipte son yazdırılan değer `_` değişkenine atanır. Yani Python'u hesap makinesi olarak kullanırken bazen işlemlere şu şekilde devam etmek daha kolaydır :

```
vergi= 17.5 / 100
>>> fiyat= 3.50
>>> fiyat * vergi
0.61249999999999993
>>> fiyat + _
4.1124999999999998
>>> round(_, 2)
4.1100000000000003
>>>
```

Bu değişken (`_`) kullanıcı tarafından salt okunur olarak kabul edilmelidir. Buna kasıtlı olarak değer atamayın. Bu aynı isimli bir yerel (local) değişken yaratır.

3.1.2 Karakter Dizileri

Sayılarla ek olarak, Python karakter dizileri üzerinde de işlemler yapabilir. Karakter dizileri farklı şekillerde ifade edilebilir. Tek veya çift tırnak işareti içine alınabilirler:

```
>>> 'karakter dizisi'
'karakter dizisi'
>>> "Python'un gücü"
"Python'un gücü"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

Karakter dizileri birkaç şekilde birden fazla satıra yayılabilirler. Yeni satırlar ters eğik çizgi ile şöyle gösterilebilirler:

```
merhaba = "Bu C de de kullandığınıza benzer\n\
birkaç satır kaplayan bir karakter dizisi.\n\
    Yeni satırın başındaki
girintinin belirgin olduğuna dikkat\
edin.\n"
print merhaba
```

ve bu şu çıktıyı verir:

```
Bu C de de kullandığınıza benzer
birkaç satır kaplayan bir karakter dizisi.
    Yeni satırın başındaki
girintinin belirgin olduğuna dikkat
edin.
```

Karakter dizisini ```ham"` yapacak olursak, `\n` yeni satıra dönüşmez ve satır sonundaki ters eğik çizgi ve yeni satır karakteri de karakter dizisine veri olarak dahil olur. Örneğin:

```
merhaba = r"Bu C de de kullandığınızın benzer\n\
birkaç satır kaplayan bir karakter dizisi."

print merhaba
```

şu çıktıyı verir:

```
Bu C de de kullandığınızın benzer\n\
birkaç satır kaplayan bir karakter dizisi.
```

Karakter dizileri bir çift üçlü tırnak içinde de gösterilebilirler: `"""` veya `'''`. Bu gösterim şeklinde satır sonlarının `\n` ile gösterilmesine gerek yoktur ve onlar olmadan da yeni satırlar doğru şekilde görünürler. Örnek:

```
print """
Kullanım şekli : seninprog [SEÇENEKLER]
    -y      Bu yardım mesajını görüntüler
    -S      bağlanılacak sunucu adı
"""
```

ifadesi şu çıktıyı verir :

```
Kullanım şekli : seninprog [SEÇENEKLER]
    -y      Bu yardım mesajını görüntüler
    -S      bağlanılacak sunucu adı
```

Yorumlayıcı karakter dizisi işlemlerinin sonucunu girişine yazıldığı şekli ile çıkışa yazar. Karakter dizileri + işleci ile birleştirilip, * ile tekrarlanabilirler:

```
kelime = 'Alo' + 'ooo'
>>> kelime
'Aloooo'
>>> '<' + word*5 + '>'
'<AlooooAlooooAlooooAlooooAloooo>'
```

Yan yana iki karakter dizisi değişkenleri otomatik olarak birleştirilirler yani yukarıdaki örnekteki ilk satır `word kelime = 'Alo' 'ooo'` şeklinde de yazılabilirdi. Bu sadece iki karakter dizisi değişkeni ile olur. Keyfi karakter dizileri arasında olamaz:

```
import string
>>> 'str' 'ing' # <- Bu doğru
'string'
>>> string.strip('str') + 'ing' # <- Bu da doğru
'string'
>>> string.strip('str') 'ing' # <- Bu geçersiz !!!
File "<stdin>", line 1, in ?
string.strip('str') 'ing'
      ^
SyntaxError: invalid syntax
```

C de olduğu gibi, Python'da da karakter dizileri indislenebilirler (index). Karakter dizisinin ilk karakterinin indisi sıfırdır. Python'da ayrı bir karakter veri tipi yoktur. Bir karakter tek karakterli bir karakter dizisidir. Icon dilinde (70'li yıllarda Ralph ve Marge Griswold'un geliştirdiği Pascal benzeri bir SNOBOL4 türevi) olduğu gibi karakter dizilerinin bölümleri *dilim gösterimi* `"[:]"` ile ifade edilebilirler.


```
>>> kelime[4]
'o'
>>> kelime[0:2]
'Al'
>>> kelime[2:4]
'oo'
```

C dilinden farklı olarak, Python'da karakter dizileri değiştirilemezler. Bir karakter dizinin indislenen bir konumuna değer atamaya çalışmak hatadır:

```
>>> kelime[0] = 'x'
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> kelime[:1] = 'Splat'
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: object doesn't support slice assignment
```

Yukarıdaki soruna elde edilmek istenen karakter dizisi için yeni bir karakter dizisi oluşturularak çözüm bulunabilir. Bu kolay ve etkilidir:

```
'x' + kelime[1:]
'xloooo'
>>> 'Splat' + kelime[4]
'Splato'
```

Dilimlerin varsayılan başlangıç ve bitiş değerleri oldukça kullanışlıdır. Başlangıç değeri yoksa sıfır kabul edilir ve eğer bitiş değeri yoksa dilimlenen karakter dizisinin boyu kadar olduğu kabul edilir. Örnekler :

```
>>> kelime[:2] # İlk iki karakter
'Al'
>>> kelime[2:] # İlk iki karakter dışındaki karakterler
'oooo'
```

`s[:i] + s[i:] = s` " olup dilimleme işlemlerinin kullanışlı bir şeklidir. Örnek:

```
>>> kelime[:2] + kelime[2:]
'Alloooo'
>>> kelime[:3] + kelime[3:]
'Alloooo'
```

Çok büyük veya küçük dilim aralıkları akıllıca ele alınır. Bitiş değeri büyük ise bunun boyu karakter dizisinin boyuna eşit olur. Başlangıç değeri bitişten büyük ise boş bir karakter dizisi elde edilir.

```
>>> kelime[1:100]
'loooo'
>>> kelime[10:]
''
>>> kelime[2:1]
''
```

İndisler negatif sayılar da olabilirler. Bu durumda saymaya sağ taraftan başlanır.

```
>>> kelime[-1] # Son karakter
'o'
>>> kelime[-2] # Sondan ikinci
'o'
>>> kelime[-2:] # Son iki karakter
'oo'
>>> kelime[:-2] # Son iki karakter dışındaki karakterler
'Aloo'
```

-0 ın 0 ile aynı olduğuna dikkat edin; yani yine sağdan sayar !

```
>>> kelime[-0] # (-0 = 0 olduğundan)
'A'
```

Sınır dışındaki negatif dilim indisleri küçültülürler; fakat bunu dilim olmayan tek bir indis ile denemeyin:

```
>>> kelime[-100:]
'Aloooo'
>>> kelime[-10] # hata !
Traceback (most recent call last):
File "<stdin>", line 1, in ?
IndexError: string index out of range
```

Dilimlerin nasıl çalıştığını hatırlamanın en iyi yolu indislerin karakterler *arasını* işaret ettiğini düşünmektir; şu şekilde bu daha iyi görülebilir:

```
+---+---+---+---+---+
| H | e | l | p | A |
+---+---+---+---+
0   1   2   3   4   5
-5  -4  -3  -2  -1
```

Negatif olmayan indisler için dilim boyu indisler arası fark kadardır. Örneğin `kelime[1:3]` diliminin boyu 2 dir.

Yerleşik (built-in) fonksiyon `len()` bir karakter dizisinin boyunu verir.

```
>>> s = 'ArrestedDevelopmentZingalamaduni'
>>> len(s)
32
```

3.1.3 Listeler

Python'da diğer veri tiplerini bir gruba almayı sağlayan birkaç *bileşik* veri tipleri vardır. Bunların en kullanışlı olanlarından biri listelerdir. Listeler kare parantez içine virgül ile birbirinden ayrılmış değerlerden (eleman) oluşurlar. Liste elemanlarının aynı tip olması gerekmez.

```
>>> a = ['salam', 'zeytin', 100, 1234]
>>> a
['salam', 'zeytin', 100, 1234]
```

Listeler de karakter dizileri gibi indislenebilir. İndisler sıfırdan başlar. Listeler dilimlenebilir, birleştirilebilir vs...

```
>>> a[0]
'salam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['salam',100]
>>> a[:2] + ['yumurta', 2*2]
['salam','zeytin', 'yumurta', 4]
>>> 3*a[:3] + ['Oley!']
['salam', 'zeytin', 100, 'salam', 'zeytin', 100, 'salam', 'zeytin', 100, 'Oley!']
```

Değiştirilemez (mutable) olan karakter dizilerinin aksine, listelerin her bir elemanı değiştirilebilir:

```
>>> a
['salam', 'zeytin', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['salam', 'zeytin', 123, 1234]
```

Liste dilimlerine de atama yapılabilir ve bu listenin boyunu da değiştirilebilir.

```
>>>
# Bazı elemanları değiştir:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Bazı elemanları sil:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Listenin içine elemanlar ekle:
... a[1:1] = ['bletch', 'xyzzzy']
>>> a
[123, 'qwerty', 'xyzzzy', 1234]
>>> a[:0] = a # Listenin kopyasını listenin başına ekle
>>> a
[123, 'qwerty', 'xyzzzy', 1234, 123, 'qwerty', 'xyzzzy', 1234]
```

Yerleşik fonksiyon `len()` listeler ile de çalışır:

```
>>> len(a)
8
```

İç içe listeler yaratılabilir. Örnek:

```
>>> q = [2, 3]
>>> p = [1, q, 4] # buradaki q üst satırda tanımlanan listedir
>>> len(p)
3
>>> p[1]
[2, 3]
```

```
>>> p[1][0]
2
>>> p[1].append('xtra')      # append daha sonra açıklanacak
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Üstteki örnekte `p[1]` ve `q` nun aynı nesne olduğuna dikkat edin !

3.2 Programlamaya Doğru İlk Adımlar

Tabi ki Python kullanarak iki ile ikiyi toplamaktan daha karmaşık işler yapabiliriz. Mesela bir *Fibonacci* serisini şöyle yazabiliriz:

```
>>> # Fibonacci serisi:
... # iki elemanın toplamı bir sonraki elemanı verir
... a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
1
2
3
5
8
```

Bu örnekte birkaç yeni özellik gösterilmektedir :

- İlk satırda bir *çoklu değer atama* var; `a` ve `b` değişkenleri bir anda 0 ve 1 değerlerini alırlar. Bu özellik son satırda da kullanılmaktadır. Son satırda dikkat çeken bir diğer olay da ifadenin sağ kısmının soldan sağa doğru atama işlemlerinden önce hesaplandığıdır.
- While döngüsü verilen koşul (burada: `b < 10`) doğru olduğu sürece tekrarlanır. Python'da, C'de olduğu gibi, sıfır dışındaki herhangi bir değer doğru ve sıfır yanlış kabul edilir. Koşul bir karakter dizisi veya liste de olabilir. Boyu sıfır olmayan her şey doğru iken, boş listeler, karakter dizileri vs yanlış kabul edilirler. Üstteki örnekte basit bir kıyaslama işlemi var. Standart kıyaslama işlemleri C'de olduğu gibi yazılır: `<` (küçük), `>` (büyük), `==` (eşit), `<=` (küçük eşit), `>=` (büyük eşit), `!=` (eşit değil).
- *Döngü öbeği* (block) *girintili* yazılmıştır. Girintili yazma Python'un ifadeleri gruplama yoludur. Etkileşimli kipte girintili bir öbek yazıldığı zaman boş bir satır ile sonlandırılmalıdır (çünkü yorumlayıcı yazmayı ne zaman bıraktığınızı bilemez). Girintili bir öbek içindeki her satırın aynı girinti miktarına sahip olması gerektiğine dikkat ediniz. Girintiler için boşluk veya tab karakterleri kullanılabilir.
- `print` deyimi kendisine verilen ifadenin veya ifadelerin değerini yazar. Birden fazla ifade verilmesi durumunda bunlar aralarında boşluk ile yazılırlar:
 - `>>>i = 256*256`
 - `>>> print 'İşlemin sonucu:', i`
 - `İşlemin sonucu: 65536`

Sona eklenen bir virgül ise yazılan satırından sonra yeni satıra geçilmesini engeller:

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

4. Daha Fazla Kontrol Akışı

Bir önceki bölümde tanıtılan `while` deyiminin yanında Python'da diğer programlama dillerinde de bulunan genel kontrol akışı deyimleri (bazı farklarla birlikte) vardır.

4.1 `if` Deyimi

Belki de en iyi bilinen deyim türü `if` deyimidir. Örnek:

```
>>> x = int(raw_input("Lütfen bir sayı girin: "))
>>> if x < 0:
...     x = 0
...     print 'Negatif sayı sıfırlandı'
... elif x == 0:
...     print 'Sıfır'
... elif x == 1:
...     print 'Bir'
... else:
...     print 'Birden büyük'
...
```

Sıfır veya daha fazla `elif` kısımları olabilir, ve `else` kısmı seçimlidir. `'elif'` deyimi `'else if'` deyiminin kısaltılmışıdır ve aşırı girintileri engellemesi açısından faydalıdır. Bir `if ... elif ... elif ...` deyimleri dizisi diğer dillerde bulunan `switch` veya `case` deyimlerinin yerine kullanılabilir.

4.2 `for` Deyimi

`for` deyimi Pascal veya C dillerinde görülenden biraz farklıdır. Python'daki `for` deyimi herhangi bir sıranın (liste, karakter dizisi vs.) elemanları üzerinde sırayla yinelenir. Örnek:

```
>>> # Bazı karakter dizilerinin boylarını ölçelim:
... a = ['kedi', 'pencere', 'kertenkele']
>>> for x in a:
...     print x, len(x)
...
```

```
kedi 4
pencere 7
kertenkele 10
```

Üzerinde yinelenilen sırada değişiklik yapmak güvenli değildir (bu sadece listelerde olabilir). Eğer böyle bir şey yapacaksanız bu iş için dilim gösterimi ile listenin bir kopyasını kullanabilirsiniz:

```
>>> for x in a[:]: # tüm listenin bir kopyasını oluştur
... if len(x) > 8: a.insert(0, x)
...
>>> a
['kertenkele', 'kedi', 'pencere', 'kertenkele']
```

4.3 `range()` Fonksiyonu

Eğer bir sayı sırası üzerinde tekrarlamalar yapmak isterseniz, belirli bir sıraya göre üretilen sayılardan oluşan bir liste yaratan `range()` yerleşik fonksiyonunu kullanabilirsiniz. Örnek:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Verilen bitiş noktası asla üretilen listenin bir parçası olmaz; `range(10)` ifadesi 10 elemanı olan bir liste oluşturur. Listenin başlayacağı sayıyı ve artış miktarını da belirlemek mümkündür. Artış miktarı negatif de olabilir.

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

`range()` ve `len()` fonksiyonlarını bir arada kullanarak da bir listenin elemanları üzerinde döngüler kurabilirsiniz:

```
>>> a = ['Python', 'programlama', 'öğrenmek', 'çok', 'kolay !']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Python
1 programlama
2 öğrenmek
3 çok
4 kolay
```

4.4 Döngülerde `break`, `continue` ve `else`

`break` deyimi, C'de olduğu gibi, içinde kaldığı en küçük `for` veya `while` döngüsünden çıkılmasına ve döngü deyiminin tamamen sona ermesine neden olur.

`continue` deyimi döngü içindeki diğer deyimlerin atlanıp bir sonraki yineleme işleminin başlamasına sebep olur.

Döngülerde `else` ifadesi de kullanılabilir; `else` öbeği döngü bittiğinde (`for` için) veya devamlılık koşulu geçersiz olduğunda (`while` için) işletilir; fakat döngü `break` deyimi ile sona erdiyse işletilmez. Bunu asal sayılar bulan aşağıdaki örnekte görebilirsiniz:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'asal sayı değil. çarpanlar:', x, '*', n/x
...             break
...         else:
...             # çarpan bulunmadan döngü biter ise
...             print n, 'asal sayıdır'
...
2 asal sayıdır
3 asal sayıdır
4 asal sayı değil. çarpanlar: 2 * 2
5 asal sayıdır
6 asal sayı değil. çarpanlar: 2 * 3
7 asal sayıdır
8 asal sayı değil. çarpanlar: 2 * 4
9 asal sayı değil. çarpanlar: 3 * 3
```

4.5 `pass` Deyimi

`pass` deyimi hiçbir şey yapmaz. Python sözdizim kurallarına göre bir ifadenin gerekli olduğu, fakat programın bir şey yapması gerekmeyişi zaman kullanılabilir:

```
>>> while True:
...     pass # klavyeden CTRL+C ile kesilene kadar sürer
...
```

4.6 Fonksiyon Tanımlama

Herhangi bir değere kadar Fibonacci serisi yazan bir fonksiyon yaratalım:

```
>>> def fib(n):      # n'e kadar Fibonacci serisini yazdır
...     "n'e kadar Fibonacci serisini yazdır"
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> # Tanımladığımız fonksiyonu çağıralım:
```

```
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

`def` anahtar kelimesi bir fonksiyon *tanımını* başlatır. Bu deyimden sonra bir fonksiyon adı ve parantez içinde parametreler yazılır. Fonksiyonun gövdesini oluşturan program satırları sonraki satırdan girintili olarak yazılırlar. Fonksiyon gövdesinin ilk satırı bir karakter dizisi de olabilir; bu karakter dizisi fonksiyonun belgelenmesinde kullanılır (*docstring*). Fonksiyonların belgelenmesinde kullanılan karakter dizilerini (*docstring*) otomatik olarak on-line ya da basılı belgeler oluşturmak için kullanan yazılımlar vardır. Ayrıca bazı geliştirme ortamları bunları program yazarken kolaylık sağlaması için etkileşimli olarak programcıya sunarlar. Yazdığınız fonksiyonlara bunları eklemeyi bir alışkanlık haline getirmeniz faydalı olur.

Fonksiyonların belgelenmesinde kullanılan karakter dizilerini (*docstring*) otomatik olarak on-line ya da basılı belgeler oluşturmak için kullanan yazılımlar vardır. Ayrıca bazı geliştirme ortamları bunları program yazarken kolaylık sağlaması için etkileşimli olarak programcıya sunarlar. Yazdığınız fonksiyonlara bunları eklemeyi bir alışkanlık haline getirmeniz faydalı olur.

Bir fonksiyonun *çağırılması* (çalıştırılması) bu fonksiyondaki yerel (local) değişkenlerin olduğu bir simge tablosu oluşturur. Fonksiyon içerisinde bütün değer atama işlemlerinde değerler yerel simge tablosuna kaydedilir. Bir değişkene değinilmesi (reference) durumunda ise önce yerel, sonra global, ve en son yerleşik (built-in) simge tablosunda arama yapılır. Bu yüzden global değişkenlere doğrudan değer atama yapılamaz (eğer `global` ifadesi içinde kullanılmamışlar ise) ; ancak bunlara değinilebilir (reference).

Fonksiyon çağırıldığında fonksiyonun parametreleri yerel simge tablosuna eklenirler; yani parametreler fonksiyona *değer ile çağrı* (*call by value*) kullanılarak iletilirler (yani parametreye yapılan değişiklikler yereldir, çağıran fonksiyondaki argümanlarda bir değişme olmaz). [4.1](#)

Bir fonksiyon başka bir fonksiyonu çağırıldığında bu çağrı için yeni bir yerel simge tablosu oluşturulur

Bir fonksiyon tanımı fonksiyon adının yürürlükte olan simge tablosuna eklenmesine sebep olur. Fonksiyonun adı yorumlayıcı tarafından *kullanıcı tanımlı fonksiyon* veri tipi olarak tanınır. Bu değer başka bir isime atanabilir ve bu da bir fonksiyon olarak kullanılabilir. Bu genel bir isim değiştirme yolu olabilir:

```
>>> fib
<function object at 10042ed0>
>>> f = fib # f de fib fonksiyonu olur
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

`fib` in bir fonksiyon olmayıp bir yordam (procedure) olduğunu düşünebilirsiniz. Python'da yordamlar çağıran fonksiyona değer geri döndürmeyen fonksiyonlardır. Aslında yordamlar da bir değer geri döndürürler, ama bu sıkıcı bir konudur. Bu değere `None` denir ve yerleşik (built-in) bir değişkendir. Yorumlayıcı yazılacak tek değer bu ise normalde `None` yazmaz. Bunu görmeyi çok istiyorsanız şunu deneyin:

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


Fibonacci serisini yazdırmak yerine, bunu bir liste şeklinde geri döndüren fonksiyon yazmak basittir:

```
>>> def fib2(n): # n e kadar fibonacci serisi geri döndürür
...     " n e kadar fibonacci serisi içeren liste geri döndürür"
...     sonuc = []
...     a, b = 0, 1
...     while b < n:
...         sonuc.append(b)      # değeri listeye ekle
...         a, b = b, a+b
...     return sonuc
...
>>> f100 = fib2(100)      # fonksiyonu çağır
>>> f100                  # sonucu yazdır
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Bu örnekte de bazı yeni Python özelliklerini görüyoruz:

- `return` deyimi bir fonksiyondan değer geri döndürür. Parametresi olmayan bir `return` deyimi `None` geri döndürür. Sona eren bir yordam (procedure) da `None` geri döndürür.
- `sonuc.append(b)` ifadesi `sonuc` liste nesnesinin bir *metodunu* çağırılmaktadır. Bir metod bir nesneye 'ait olan' ve `nesne.metodAdı` şeklinde adlandırılan bir fonksiyondur. `nesne.metodAdı` ifadesinde `nesne` herhangi bir nesne (bir ifade de olabilir) ve `metodAdı` da nesnenin türüne bağlı bir metoddur. Farklı veri tipleri farklı metodlara sahiptirler. Farklı veri tiplerinin aynı isimli metodları olabilir. Sonraki bölümlerde anlatılacağı gibi, kendi veri tiplerinizi ve metodlarınızı yaratmanız mümkündür. Yukarıdaki örnekte görülen `append()` metodu liste nesneleri için tanımlıdır ve bir listenin sonuna yeni elemanlar ekler. Bu örnekte bu "`sonuc = sonuc + [b]`" ifadesinin yaptığını yapar; ancak daha verimlidir.

4.7 Fonksiyon Tanımları Üzerine Daha Fazla Bilgi

Değişken sayıda argüman alan fonksiyonlar tanımlamak da mümkündür. Bunun için kullanılan üç yöntem olup bunlar birleştirilerek kullanılabilir.

4.7.1 Argüman Değerlerini Önceden Belirleme

Fonksiyon argümanlarına varsayılan (default) değerler atamak da mümkündür. Böylece çağırılan fonksiyon bu argümanları sağlamazsa bunlar önceden belirlenmiş varsayılan değerlerini alırlar. Örnek:

```
def onay_al(prompt, denemeler=4, sikayet='Evet veya hayır, lütfen !'):
    while True:
        ok = raw_input(prompt)
        if ok in ('e', 'evet'): return 1
        if ok in ('h', 'hayır'): return 0
```

```
denemeler = denemeler - 1
if denemeler < 0: raise IOError, 'kararsız kullanıcı'
print sikayet
```

Bu fonksiyon `onay_al('Programdan çıkmak istiyor musunuz ?')` ya da `onay_al('Dosyayı silmek istiyor musunuz ?', 2)` şeklinde çağırılabilir.

Fonksiyonun varsayılan (default) parametreleri fonksiyonun tanımlandığı anda, *o an yürürlükte olan* etki alanı (scope) içinde değerlendirilirler. Yani:

```
i = 7

def f(arg=i):
    print arg

i = 6
f()
```

7 yazacaktır.

Önemli uyarı: Fonksiyonun varsayılan (default) parametreleri sadece bir defa değerlendirilirler. Bu durum parametrenin liste gibi değiştirilebilir bir nesne olduğu durumlarda farklılık yaratır. Örneğin aşağıdaki fonksiyon ard arda çağırıldığında argümanlarını biriktirir:

```
def f(a, L = []):
    L.append(a)
    return L
print f(1)
print f(2)
print f(3)
```

Bu şu çıktıyı verir:

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

Eğer varsayılan (default) parametre değerlerinin birbirini izleyen çağrılarla paylaşılmasını istemiyorsanız yukarıdaki fonksiyonu şu şekilde yazabilirsiniz:

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

4.7.2 Anahtar Kelime Argümanlar

Fonksiyonlar "*anahtar kelime = değer*" şeklindeki anahtar kelimelerle de çağırılabilirler. Örneğin şu fonksiyon:

```
def otomobil(yakit, hareket='uçar', model='Anadol'):
```

```
print "Eğer", yakıt, "koyarsan bu", model, hareket
```

aşağıdaki gibi çağırılabilir:

```
otomobil('roket yakıtı')
otomobil(hareket = 'dans eder', yakıt = 'zeytin yağı' )
otomobil('ispirto', model = 'Kartal')
otomobil('su', 'bozulur', 'Şahin')
```

Şu çağrılar ise hatalıdır:

```
otomobil() # gerekli argüman eksik
otomobil(yakıt = 'su', 'zeytin yağı') # anahtar kelimedden sonra gelen
# anahtar kelime olmayan argüman
otomobil('mazot', yakıt = 'benzin') # aynı argüman için iki değer
otomobil(sehir = 'İzmir') # bilinmeyen anahtar kelime
```

Genel olarak, argüman listesinin başında konuma bağlı argümanlar bulunur ve anahtar kelime argümanlar onları izler; anahtar kelime adları da fonksiyonun parametrelerinden seçilir. Parametrenin varsayılan (default) değerlerinin olup olmaması önemli değildir. Bir argüman birden fazla değer alamaz; konuma bağlı parametre isimleri aynı çağrıda anahtar kelime olarak kullanılamazlar. İşte bundan dolayı hatalı olan bir örnek:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: keyword parameter redefined
```

Eğer fonksiyon tanımındaki son parametre ***isim* şeklinde ise bu parametre adları herhangi bir parametre olmayan anahtar kelime şeklindeki argümanların bulunduğu bir sözlük olur. Bu **isim* (bir sonraki kısmın konusu) şeklindeki bir parametre ile de kullanılabilir, ki bu parametre listesi içinde bulunmayan konuma bağlı argümanları içeren bir demet (daha sonra incelenecek bir veri tipidir) olur . **isim* parametresi ***isim* parametresinden önce gelmelidir. Buna örnek fonksiyon:

```
def kasapdukkani(etCinsi,*argumanlar, **anahtarKelimeler):
    print "--", etCinsi, "var mi ?"
    print "-- Maalesef", etCinsi, "kalmadı."
    for arg in argumanlar:
        print arg
    print '-'*40
    anahtarlar = anahtarKelimeler.keys()
    anahtarlar.sort()
    for ak in anahtarlar:
        print ak, ': ', anahtarKelimeler[ak]
```

Şu şekilde çağırılabilir :

```
kasapdukkani('martı eti',"Çok lezzetli.",
              "Çok satılıyor.",
              musteri = 'Martı Murat',
              kasap = 'Dev İsmail')
```

ve doğal olarak şu çıktıyı verir:

```
-- martı eti var mı ?
-- Maalesef martı eti kalmadı.
Çok lezzetli.
Çok satılıyor.
-----
kasap : Dev İsmail
musteri : Martı Murat
```

`anahtarKelimeler` isimli sözlüğün içeriği yazdırılmadan önce anahtar kelime isimleri listesinin `sort()` metodunun çağırıldığına dikkat edin; bu yapılmaz ise argümanların hangi sıra ile yazılacağı tanımlanmamıştır.

4.7.3 Keyfi Argüman Listeleri

Son olarak, en ender kullanılan seçenek de keyfi sayıdaki argümanla çağırılabilen bir fonksiyon tanımlamaktır. Bu argümanlar bir demet (değişmez liste [tuple]) içine alınırlar. Keyfi argüman listesinden önce sıfır ya da daha fazla normal argüman bulunabilir. Örnek:

```
def fprintf(file, format, *args):
    file.write(format % args)
```

4.7.4 Lambda Formundaki Fonksiyonlar

Yoğun istek üzerine fonksiyonel dillerde ve Lisp'te bulunan bazı özellikler Python'a eklenmiştir. `lambda` anahtar kelimesi ile küçük anonim fonksiyonlar yaratılabilir. İşte iki argümanının toplamını geri döndüren bir fonksiyon: "`lambda a, b: a+b`". Lambda fonksiyonları bir fonksiyon nesnesine ihtiyaç duyulan her yerde kullanılabilirler. Sözdizim (syntax) açısından bunlar tek bir ifade ile sınırlandırılmışlardır. Anlambilim (semantic) açısından ise normal fonksiyon tanımlamasına getirilen bir sözdizim güzelliğidir. İç içe tanımlanmış fonksiyonlarda olduğu gibi, lambda fonksiyonları kendilerini kapsayan etki alanındaki (scope) değişkenlere erişebilirler:

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

4.7.5 Belgeleme Karakter Dizileri

Belgelemede kullanılan karakter dizilerinin şekli ve içeriği ile ilgili şartlar yeni yeni oluşmaktadır.

İlk satır daima nesnenin amacının kısa ve öz tanımı olmalıdır. Kısa olması için, nesnenin adından ve türünden bahsedilmemeli; zira bunlar başka yollarla da öğrenilebilir. Bu satır büyük harf ile başlayıp nokta ile bitmelidir.

Eğer belgeleme karakter dizisinde birden fazla satır var ise ikinci satır boş olup özet ile açıklamamın devamını birbirinden ayırmalıdır. Diğer satırlar bir ya da daha fazla satır olabilir. Bunlarla nesnenin özellikleri, çağrı şekilleri, yan etkileri vs. açıklanabilir.

Python ayrıştırıcısı (parser) çok satırlı karakter dizilerindeki girintileri yok etmez; yani belgeleri işleyen programlar gerekirse bunları atabilirler. İlk satırdan *sonra* gelen ve boş olmayan ilk satırdaki girinti miktarı belgeleme karakter dizisinin devamındaki girinti miktarını belirler. Bu girinti miktarına ``eşdeğer" boşluk diğer satırların başından atılır. Daha az girintili satırlar olmamalı; ama olursa da bunların önündeki boşluğun tamamı atılmalı. Boşluğun eşdeğerliği tabların genişletilmesinden (1 tab 8 boşluğa) sonra sınanmalıdır.

İşte çok satırlı bir belgeleme karakter dizisi örneği:

```
>>> def benimFonksiyon():
...     """Sadece belgeler.
...
...     Başka birşey yapmaz. Gerçekten !.
...     """
...     pass
...
>>> print benimFonksiyon.__doc__
Sadece belgeler.

    Başka birşey yapmaz. Gerçekten !
```

5. Veri Yapıları

Bu bölümde öğrendiğiniz bazı şeyler daha detaylı açıklanmakta ve bazı yeni konulara da değinilmekte.

5.1 Listeler Üzerine Daha Fazla Bilgi

Liste veri tipinin birkaç metodu daha var. İşte liste nesnelerinin bütün metodları:

append(x)

Listenin sonuna bir eleman ekler; `a[len(a):] = [x]` ifadesine denktir. .

extend(L)

Listeyi verilen listedeki tüm elemanları ekleyerek genişletir; `a[len(a):] = L` ifadesine denktir. .

insert(i, x)

Verilen konuma bir eleman sokar. İlk argüman elemanın yerleştirileceği indistir.

`a.insert(0, x)` ifadesi x'i listenin başına sokar, ve `a.insert(len(a), x)` ifadesi

`a.append(x)` ifadesine denktir.

remove(x)

Liste içinde değeri x olan ilk elemanı listeden siler. Böyle bir öge yok ise bu hatadır.
`pop([i])`

Verilen konumdaki elemanı listeden siler ve bunu geri döndürür. Eğer bir indis belirtilmediyse, `a.pop()` listedeki son elemanı siler ve geri döndürür. (*i* etrafındaki kare parantezler bu parametrenin seçimlik olduğunu belirtir. Bu notasyonu Python belgelerinde sıkça görebilirsiniz.)

`index(x)`

Değeri x olan elemanın indisini geri döndürür. Böyle bir eleman yok ise bu hatadır.

`count(x)`

x 'in listede kaç adet bulunduğunu bulur ve bu değeri geri döndürür.

`sort()`

Listenin elemanlarını sıralar - yerinde.

`reverse()`

Listenin sırasını tersine çevirir - yerinde.

Liste metodlarının çoğunu kullanan bir örnek:

```
>>> a = [66.6, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.6), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.6, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.6, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.6]
>>> a.sort()
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
```

5.1.1 Listelerin Yığın Olarak Kullanılması

Liste metodları listelerin kolayca yığın olarak kullanılmasını sağlarlar. Yığına son giren eleman ilk çıkar. Yığının üzerine eleman eklemek için `append()` ve en üstteki elemanı almak için indis belirtmeden `pop()` kullanılır. Örnek:

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

```
>>> yigin
[3, 4]
```

5.1.2 Listelerin Kuyruk Olarak Kullanılması

Listeleri kuyruk olarak da kullanmak mümkün. Bir kuyrukta ilk eklenen eleman ilk alınan elemandır (ilk giren ilk çıkar). Kuyruğun sonuna bir eleman eklemek için `append()` kullanılır. Sıranın başından bir eleman almak için ise 0 indisi ile `pop()` kullanılır. Örnek:

```
>>> kuyruk = ["Ali", "Veli", "Deli"]
>>> kuyruk.append("Küpeli")          # Küpeli kuyrukta
>>> kuyruk.append("Aylin")          # Aylin kuyrukta
>>> kuyruk.pop(0)
'Ali'
>>> kuyruk.pop(0)
'Veli'
>>> kuyruk
['Deli', 'Küpeli', 'Aylin']
```

5.1.3 Fonksiyonel Programlama Araçları

Listelerle kullanıldığında çok faydalı olan yerçekik fonksiyonlar vardır: `filter()`, `map()`, ve `reduce()`.

"`filter(fonksiyon, sıra)`" sıra içerisinde `fonksiyon(eleman)` ın doğru sonuç verdiği elemanların bulunduğu (mümkünse aynı türden) bir sıra geri döndürür. Örneğin, bazı asal sayıları hesaplamak için şöyle yapılabilir:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

"`map(fonksiyon, sıra)`" sıranın her elemanı için `fonksiyon(sıra)` çağırır ve geri döndürülen değerlerin oluşturduğu listeyi geri döndürür. Örneğin bazı sayıların küplerini hesaplamak için şu yol izlenebilir:

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

`map(fonksiyon, sıra)` ifadesinde birden fazla sıra da kullanılabilir; ancak bu durumda `fonksiyon` sıra sayısı kadar argümana sahip olmalıdır. `fonksiyon` her sıranın uygun elemanını bir argüman olarak alır; ancak sıralardan biri kısa ise eksik elemanlar için `fonksiyona` `None` argümanı geçirilir. Eğer `fonksiyon` adı için de `None` kullanılırsa argümanlarını geri döndüren bir `fonksiyon` etkisi yaratılır.

Bu iki özel durumu birleştirerek "`map(None, list1, list2)`" ifadesi ile bir çift diziyi çiftlerden oluşan bir diziye çevirebiliriz. Örnek:

```
>>> sıra = range(8)
>>> def kare(x): return x*x
...
>>> map(None, sıra, map(kare, sıra))
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49)]
```

"reduce(*fonksiyon*, *sıra*)" ifadesi tek bir değer geri döndürür. Bu değer şöyle elde edilir: iki argümanlı *fonksiyona* *sıranın* ilk iki elemanı argüman olarak verilir, sonra da elde edilen sonuç ile *sıranın* sonraki elemanı argüman olarak verilir, daha sonra yine elde edilen sonuç ile bir sonraki eleman *fonksiyona* verilir ve bu işlem bütün elemanlar için tekrarlanır. Örneğin 1'den 10'a kadar olan böyle toplanabilir:

```
>>> def topla(x,y): return x+y
...
>>> reduce(topla, range(1, 11))
55
```

Sırada sadece bir eleman var ise bunun değeri geri döndürülür; sıra boş ise bir istisna oluşur (exception).

Başlangıç değerini bildirmek için üçüncü bir argüman kullanılabilir. Bu durumda *fonksiyona* ilk olarak başlangıç değeri ve *sıranın* ilk elemanına uygulanır ve diğer elemanlar ile devam eder. Örnek:

```
>>> def sonuc(sıra):
...     def topla(x,y): return x+y
...     return reduce(topla, sıra, 0)
...
>>> sonuc(range(1, 11))
55
>>> sonuc([])
0
```

5.1.4 Liste Üreteçleri

Liste üreteçleri `map()`, `filter()` ve/veya `lambda` fonksiyonlarını kullanmadan liste yaratmanın kısa bir yoludur. Bu yolla yaratılan liste tanımları genellikle daha kolay anlaşılır olur. Bir liste üretici bir ifade ve bir `for` döngüsü ile bunları izleyen sıfır ya da daha fazla `for` veya `if` ifadelerinden oluşur. Sonuç kendisini izleyen `for` ve `if` bağlamında değerlendirilen ifadeden oluşan bir listedir. Eğer ifade bir demete (değişmez liste [tuple]) dönüşecekse parantez içinde yazılmalıdır.

```
>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
>>> [weapon.strip() for weapon in freshfruit] # elemanları saran
boşlukların atıldığı yeni bir liste
['banana', 'loganberry', 'passion fruit']
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [{x: x**2} for x in vec] # sözlüklerden oluşan bir liste
[{2: 4}, {4: 16}, {6: 36}]
>>> [[x,x**2] for x in vec]
```



```

[[2, 4], [4, 16], [6, 36]]
>>> [x, x**2 for x in vec]    # hata - demet için parantez gerekir
File "<stdin>", line 1, in ?
    [x, x**2 for x in vec]
        ^
SyntaxError: invalid syntax
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]

```

Liste üreteçlerinin `for` döngülerine benzer davranması için, döngü değişkenine yapılan atamalar üreteç dışında da görünürler:

```

>>> x = 100                                # bu değişecek
>>> [x**3 for x in range(5)]
[0, 1, 8, 27, 64]
>>> x
4                                           # range(5) için son değer
>>

```

5.2 `del` Deyimi

`del` deyimi ile bir listeden indisi verilen bir eleman silinebilir. Bu deyim ile bir listeden dilimler de silinebilir (bunu daha önce dilimlere boş bir liste atayarak yapmıştık). Örnek:

```

>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.6, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.6, 1234.5]

```

`del` değişkeni tamamen silmek için de kullanılabilir:

```

>>> del a

```

Bu aşamadan sonra `a` ismine değinmek hatadır (aynı isme başka bir değer atanana kadar). Daha sonra `del` için başka kullanım alanları da göreceğiz.

5.3 Demetler ve Sıralar

Listelerin ve karakter dizilerinin indisleme ve dilimleme gibi pek çok ortak özellikleri olduğunu gördük. Bunlar *sıra* şeklindeki iki veri tipidirler. Python geliştirmekte olan bir dil; diğer sıra şeklindeki veri tipleri de Python'a eklenebilir. *demet* de başka bir sıra şekilli standart veri tipidir .

Bir demet virgül ile ayrılmış birkaç değerden oluşur.

```
>>> t = 12345, 54321, 'merhaba!'
>>> t[0]
12345
>>> t
(12345, 54321, 'merhaba!')
>>> # demetler iç içe kullanılabilirler :
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'merhaba!'), (1, 2, 3, 4, 5))
```

Gördüğümüz gibi çıktıda demetler daima parantez içinde görünürler; ki iç içe geçmiş demetler belli olsun. Demetler parantezli veya parantezsiz olarak yazılabilirler; ancak parantezler genellikle gereklidirler (özellikle de demet daha büyük bir ifadenin içinde geçiyorsa).

Demetlerin pekçok kullanım alanı var: (x,y) koordinat çifti, veri tabanındaki işçi kayıtları vb. gibi. Demetler de karakter dizileri gibi değerleri değiştirilemez veri tipleridir; bunların elemanlarına atama yapılamaz (fakat dilimleme ve birleştirme aracılığı ile bu etki sağlanabilir). Ayrıca değiştirilebilen elemanlardan oluşan demetler oluşturmak da mümkündür (örnek: listelerden oluşan bir demet).

Sıfır veya bir elemanlı demetlerin oluşturulması ile ilgili özel bir problem var: bunların ifade edilmesini sağlayan sözdizim biraz acayip. Boş demetler bir çift boş parantez ile ifade edilir. Tek elemanı olan bir demet için ise elemandan sonra bir virgül kullanılır (tek bir değeri parantez içine almak yeterli değildir). Çirkin ama etkili. Örnek:

```
>>> bos = ()
>>> tekOge = 'merhaba', # <--sattır sonundaki virgüle dikkat
>>> len(bos)
0
>>> len(tekOge)
1
>>> tekOge
('merhaba',)
```

`t = 12345, 54321, 'merhaba!'` ifadesi *demetleme* (tuple packing) işlemine bir örnektir: 12345, 54321 ve 'merhaba!' değerleri bir demet içinde toplanmışlardır. Bu işlemin tersi de mümkün:

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

Doğal olarak, buna *demet açma*(sequence unpacking) deniyor . Demet açma sol taraftaki değişken sayısının sıra içindeki öğe sayısına eşit olmasını gerektirir. Çoklu değer atama işleminin aslında demetleme ve demet açmanın bir bileşimi olduğuna dikkat edin.

Burada küçük bir asimetri var: birden fazla değeri demetleme her zaman bir demet oluşturur ve demet açma herhangi bir sıra için yapılabilir. Örnek:

```
>>> paket = 'xyz' # bir karakter dizisi
>>> a,b,c = paket
>>> a
'x'
>>> b
'y'
>>> c
'z'
```

5.4 Sözlükler

Python'da bulunan bir diğer faydalı veri tipi de *sözlüktür*. Sözlükler diğer programlama dillerinde ``çağrışımlı bellek" (associative memory) veya ``çağrışımlı dizi" (associative array) olarak bilinirler. Sayılarla indislenen sıralardan farklı olarak, sözlükler anahtarlar (*key*) ile indislenirler. Anahtar değiştirilemeyen tipdeki herhangi bir veri tipinde olabilir. Sayılar ve karakter dizileri her zaman anahtar olabilirler. Demetler de sayılar, karakter dizileri veya demetler içerdikleri sürece anahtar olabilirler. Bir demet doğrudan ya da dolaylı olarak değiştirilebilir bir nesne içeriyorsa anahtar olarak kullanılamaz. Listeler anahtar olamazlar, çünkü `append()` ile `extend()` metodları, dilimleme ve indise değer atama ile değiştirilebilirler.

Bir sözlük *anahtar* : *değer* çiftlerinden oluşur. Bir anahtar sözlükte sadece bir defa bulunabilir. Bir çift çengelli parantez boş bir sözlük yaratır : `{}`. Çengelli parantezlerin içine virgülle ayrılmış *anahtar* : *değer* çiftleri koymak anahtar ve değer çiftlerine ilk değerlerini verir. Çıktıya da sözlükler aynı şekilde yazılırlar.

Sözlüklerle ilgili ana işlemler bir değerın bir anahtar ile saklanması ve anahtar verildiğinde değerın bulunmasıdır. `del` kullanarak bir *anahtar* : *değer* çiftini silmek mümkündür. Zaten mevcut olan bir anahtar kullanarak bir değer eklerseniz bu anahtarla bağlantılı eski değer unutulur. Mevcut olmayan bir anahtar ile değer istemek hatalıdır.

Sözlük nesnesinin `keys()` metodu listedeki bütün anahtarların listesini rasgele sıralı olarak geri döndürür (sıralamak isterseniz listenin `sort()` metodundan faydalanabilirsiniz) . Bir anahtarın sözlükte olup olmadığını görmek için sözlüğün `has_key()` metodu kullanılır.

İşte sözlük kullanan küçük bir örnek:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.has_key('guido')
1
```

`dict()` fonksiyonu anahtar-değer çiftlerinden oluşan demetlerden sözlükler üretir. Çiftlerin bir kalıba uyduğu durumlarda, liste üreticileri ile anahtar-değer çiftleri kısaca ifade edilebilir.

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> dict([(x, x**2) for x in vec])      # liste üretici kullanarak
{2: 4, 4: 16, 6: 36}
```

5.5 Döngü Teknikleri

Sözlükler üzerinde döngüler kurarken o anki değer `items()` metodu ile aynı anda elde edilebilir.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print k, v
...
gallahad the pure
robin the brave
```

Bir sıra üzerinde dönerken konum indisi ve ona karşılık gelen değer de `enumerate()` fonksiyonunu kullanarak aynı anda elde edilebilir.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
...
0 tic
1 tac
2 toe
```

Aynı anda iki sıra üzerinde ilerlemek için ise `zip()` fonksiyonu ile bunlar çiftler haline getirilebilir.

```
>>> sorular = ['adın', 'görevin', 'favori rengin']
>>> cevaplar = ['Adnan', 'Uyumak', 'Mavi']
>>> for s, c in zip(sorular, cevaplar):
...     print 'Senin %s ne? %s.' % (s, c)
...
Senin adım ne? Adnan.
Senin görevin ne? Uyumak.
Senin favori rengin ne? Mavi.
```

5.6 Koşullar Üzerine Daha Fazla Bilgi

`while` ve `if` deyimlerinde kıyaslama dışında da işleçler kullanılabilir.

`in` ve `not in` kıyaslama işleçleri bir değer in bir sıra içinde olup olmadığını sınarlar. `is` ve `is not` işleçleri iki nesnenin tamamen aynı nesne olup olmadıklarını sınarlar (bu sadece liste gibi

değiştirilebilir nesnelerde önemlidir). Bütün kıyaslama işlemleri aynı önceliğe sahiptirler ve bu sayısal işlemlerinkinden düşüktür.

Kıyaslamalar zincirlenebilir: `a < b == c` gibi.

Kıyaslamalar mantıksal işlemler `and` ve `or` ile birleştirilebilirler, ve kıyaslamanın sonucu (ya da herhangi bir mantıksal ifade) `not` ile değillenebilirler. Bunların hepsi de kıyaslama işlemlerinden düşük önceliğe sahiptirler ve aralarında en yüksek öncelikli olan `not` ve en düşük öncelikli olan `or` işlecidir. Örneğin `A and not B or C` ifadesi `(A and (not B)) or C` ifadesine eşittir. İstenen bileşimi elde etmek için parantezler kullanılabilir.

`and` ve `or` mantıksal işlemlerine *kısa devre* işlem de denir. Bunların argümanları soldan sağa değerlendirilir ve sonuç belli olur olmaz değerlendirme işlemi kesilir. Örneğin `A` ve `C` doğru, fakat `B` yanlış olsun. `A and B and C` ifadesinde `C` ifadesi değerlendirilmez (çünkü `C` nin değeri sonucu değiştirmez). Genel olarak bir kısa devre işleci `Bool` değil de genel bir değer gibi kullanıldığında en son değerlendirilen argümanın değeri geri döndürülür.

Bir kıyaslamanın ya da mantıksal ifadenin sonucunu bir değişkene atamak mümkündür:

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

C dilinin tersine, Python'da ifadelerin içinde atama olamayacağına dikkat edin. C programcıları bundan şikayetçi olabilirler; ancak bu C programlarında sık karşılaşılan bazı hataları engellemektedir (`==` yerine `=` yazmak gibi).

5.7 Sıraların ve Diğer Veri Tiplerinin Kıyaslanması

Sıra nesneleri yine sıra şeklindeki diğer nesnelerle kıyaslanabilirler. Önce ilk iki eleman kıyaslanır. Bunlar farklı ise sonuç belli olmuştur; eşit olmaları halinde sonraki iki eleman kıyaslanır ve sıralardan biri tükenene kadar bu işlem tekrarlanır. Eğer kıyaslanan iki öğe de sıra ise bunlar da kendi aralarında kıyaslanırlar. İki sıranın bütün öğeleri aynı bulunursa bu sıralar eşit kabul edilir. Eğer bir sıra diğerinin başından bir kısmı ile aynı ise kısa olan sıra küçük kabul edilir. Karakterlerin kıyaslanmasında ASCII karakter sırası kullanılır. Aynı türden sıraların kıyaslanmasına bazı örnekler :

```
(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)
```

Farklı türden nesnelerin kıyaslanması yasal olduğuna dikkat edin. Türler alfabetik sırayla dizilmiştir (ingilizce isimlerine göre). Yani: liste < karakter dizisi < demet (list < string < tuple). [5.1](#)

6. Modüller

Python yorumlayıcısını kapatıp tekrar açarsanız yaptığınız tanımlar (fonksiyonlar ve değişkenler) kaybolur. Uzunca bir program yazmak isterseniz bunun için programınızı bir metin editörü ile hazırlayıp yarattığınız dosyayı yorumlayıcı girişi olarak kullanırsanız daha iyi olur. Bu işleme *betik* (script) yazmak denir. Programınız uzadıkça bunu daha kolay idare etmek için birkaç dosyaya bölmek isteyebilirsiniz. Yazdığınız bir fonksiyonu tanımını kopyalamaya ihtiyaç duymaksızın birkaç programda kullanmayı da isteyebilirsiniz.

Bu iş için Python'da *modül* denen dosyalar var. Bunlara yazılan tanımlar diğer modüllere ya da etkileşimli kipteki yorumlayıcıya `import` deyimi ile yüklenebilirler.

Modüller .py uzantılı metin dosyalarıdır ve içlerinde Python deyimleri ve tanımları bulur. Bir modül içerisinde `__name__` global değişkeninin değeri (bir karakter dizisi) o modülün adını verir. Örneğin, favori metin editörünüz ile `fib.py` adlı bir dosya yaratıp Python yorumlayıcısının bulabileceği bir dizine kaydedin. Dosyanın içeriği de şu olsun:

```
# Fibonacci sayıları modülü

def fib(n):      # n e kadar Fibonacci serisini yazdır
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # n e kadar Fibonacci serisi geri döndürür
    sonuc = []
    a, b = 0, 1
    while b < n:
        sonuc.append(b)
        a, b = b, a+b
    return sonuc
```

Yorumlayıcıyı açıp bu modülü şu komut ile yükleyin:

```
>>> import fibo
```

Bu `fibo` içindeki fonksiyon tanımlarını yürürlükte olan simge tablosuna eklemeyiz; sadece modül adı `fibo` tabloya eklenir. Fonksiyonlara modül adı kullanarak erişilebilir:

```
>>> 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'
```

Bir fonksiyonu sık sık kullanmak isterseniz bunu yerel 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 Üzerine Daha Fazla Bilgi

Fonksiyon tanımlarının yanısıra modül içinde çalıştırılabilir ifadeler de olabilir. Bu ifadeler modülün ilk kullanıma hazırlanması için kullanılabilirler ve sadece modülün *ilk* yüklenişinde çalışırlar. [6.1](#)

Her modülün o modül içindeki bütün fonksiyonlar tarafından global simge tablosu olarak kullanılan kendi simge tablosu vardır. Bu özellik sayesinde modülü yazan kişi rahatlıkla modül içinde global değişkenler kullanabilir. Modülü kullanan diğer kişilerin global değişkenleri ile isim çakışması olmaz. Modul içindeki global değişkenlere de `modulAdi.degiskenAdi` şeklinde ulaşmak ve istenirse bunları değiştirmek mümkündür.

Modüller diğer modülleri yükleyebilirler. Bütün `import` ifadelerinin modülün (ya da betiğin) başına konması gelenektendir; ancak şart değildir. Yüklenen modüller kendilerini yükleyen modülün global simge tablosuna eklenirler.

`import` deyiminin bir modüldeki isimleri doğrudan yükleyen modülün simge tablosuna ekleyen kullanım şekli var. Örnek:

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

Bu kullanım şeklinde yüklemenin yapıldığı modül adı yerel simge tablosuna eklenmez (yani örnekteki `codefibo` tanımlı değildir).

Bir modülde tanımlanmış bütün isimleri de yüklemek şu şekilde mümkündür:

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

Bu altçizgi (`_`) ile başlayanlar dışındaki bütün isimleri yükler.

6.1.1 Modül Arama Yolu

`spam` isimli bir modül yüklenmek istendiğinde yorumlayıcı önce çalıştırıldığı dizinde ve sonra `PYTHONPATH` ortam değişkenince tanımlanan dizinler içinde `spam.py` isimli bir dosya arar. `PYTHONPATH` dizin isimlerinden oluşan bir listedir (`PATH` gibi). Aranılan dosya

bulunmazsa arama kurulumu bağı başka bir yolda da aranabilir. Unix işletim sisteminde bu `./usr/local/lib/python` dizinidir.

Aslında modüller `sys.path` değişkeninde bulunan dizin listesinde aranılır. Bu değişken değerini betiğin çalıştırıldığı dizin, `PYTHONPATH` ve kurulumu bağı diğer dizinlerden alır. `sys.path` değişkeni sayesinde Python programları modül arama yolunu değiştirebilirler.

6.1.2 ``Derlenmiş'' Python Dosyaları

Derlenmiş Python dosyaları programların çalışmaya başlaması için gereken süreyi kısaltırlar. Örneğin `spam.py` adlı dosyanın bulunduğu dizinde `spam.pyc` adlı bir dosya varsa bu modüle `spam` modülünün ``byte derlenmiş'' (byte-compiled) halidir. `spam.py` dosyasının son değiştirilme tarihi `spam.pyc` dosyasının içinde de kayıtlıdır ve bu tarihler aynı değil ise `.pyc` dosyası dikkate alınmaz.

`spam.pyc` dosyasının oluşması için bir şey yapmanız gerekmez. `spam.py` her ne zaman başarılı olarak derlenirse programın derlenmiş hali `spam.pyc` dosyasına kaydedilir. Bunun yapılamaması bir hata değildir; herhangi bir nedenle `.pyc` dosyası tam olarak yazılamazsa geçersiz sayılır ve dikkate alınmaz. `.pyc` dosyalarının içeriği platformdan bağımsızdır. Bu sayede bir Python modülü dizini farklı mimarideki makineler tarafından paylaşılabilir.

Uzmanlar için birkaç ip ucu:

- Python yorumlayıcısı **-O** parametresi ile çalıştırıldığında eniyileştirilmiş (optimized) kod üretilir ve `.pyo` uzantılı dosyalarda saklanır. Eniyileştiricinin (optimizer) şu anda pek bir yararı olmuyor; sadece `assert` deyimlerini siliyor. **-O** parametresi kullanıldığında *tiim* baytkodu eniyileştirilir, `.pyc` dosyaları göz ardı edilir ve `.py` dosyaları eniyileştirilmiş baytkoduna derlenir.
- Yorumlayıcıya iki tane **-O** parametresi (**-OO**) vermek baytkodu derleyicisinin bazı ender durumlarda doğru çalışmayan programlara neden olan eniyilştirmeler yapmasına neden olur. Şu anda sadece `__doc__` karakter dizileri silinerek daha küçük `.pyo` dosyaları üretilmektedir. Bazı programların çalışması bunların varlığına bağımlı olabileceğinden bu parametreyi kullanırken dikkatli olun.
- Bir program `.pyc` ya da `.pyo` dosyasından okunduğunda `.py` dosyasından okunan halinden daha hızlı çalışmaz; sadece yüklenme süresi kısalır.
- Bir betik komut satırından ismi verilerek çalıştırıldığında bunun baytkodu asla bir `.pyc` ya da `.pyo` dosyasına yazılmaz. Bu yüzden betiğin başlama süresini kısaltmak için bunun bir kısmı bir modüle aktararak ve bu modülü yükleyen küçük bir başlatıcı betik kullanılarak kısaltılabilir. Komut satırından bir `.pyc` ya da `.pyo` dosyası da ismi verilerek doğrudan çalıştırılabilir.
- `spam.py` dosyası olmadan da `spam.pyc` (ya da **-O** kullanıldığında `spam.pyo`) dosyası kullanılabilir. Bunlar bir Python kodu kütüphanesinin tersine mühendisliği zorlaştıran şekilde dağıtılmasında kullanılabilir.
- `compileall` modülü bir dizindeki bütün dosyalar için `spam.pyc` (ya da **-O** kullanıldığında `spam.pyo`) dosyaları yaratabilir.

6.2 Standart Modüller

Python zengin bir standart modül kütüphanesine sahiptir. Bazı modüller yorumlayıcı ile bütünleşiktir. Bu modüller dilin parçası olmadıkları halde verimlerini artırmak ya da sistem çağrılarını gibi işletim sistemine ait özelliklere erişim için yorumlayıcı içine dahil edilmişlerdir. Bunlara iyi bir örnek her Python yorumlayıcısına dahil edilen `sys` modülüdür. `sys.ps1` ve `sys.ps2` değişkenleri de birincil ve ikincil komut satırı olarak kullanılan karakter dizilerini belirlerler:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'...'
>>> sys.ps1 = 'C> '
C> print 'Böö !'
Böö !
C>
```

Bu iki değişken yorumlayıcı sadece etkileşimli kipte iken tanımlıdır.

`sys.path` değişkeni de yorumlayıcının modül arama yolunu belirler. Bu değerini ortam değişkeni `PYTHONPATH` belirler. `PYTHONPATH` değişkenine değer atanmadıysa `sys.path` varsayılan (default) değerini alır. Bunun değeri listelere uygulanabilir işlemler ile değiştirilebilir:

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

6.3 `dir()` Fonksiyonu

Yerleşik fonksiyon `dir()` bir modülün hangi isimleri tanımladığını bulmak için kullanılır. Bu fonksiyon karakter dizilerinden oluşan bir liste geri döndürür:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__',
'__stderr__',
'__stdin__', '__stdout__', '_getframe', 'argv',
'builtin_module_names',
'byteorder', 'copyright', 'displayhook', 'exc_info', 'exc_type',
'excepthook', 'exec_prefix', 'executable', 'exit',
'getdefaultencoding',
'getdlopenflags', 'getrecursionlimit', 'getrefcount', 'hexversion',
'maxint', 'maxunicode', 'modules', 'path', 'platform', 'prefix',
'ps1',
'ps2', 'setcheckinterval', 'setdlopenflags', 'setprofile',
'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',
'version',
'version_info', 'warnoptions']
```

Argüman kullanmadan çağırılan `dir()` fonksiyonu o anda tanımlamış olduğunuz isimleri geri döndürür:

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

Bunun değişken, modül, fonksiyon vs. gibi her tip ismi listelediğine dikkat ediniz.

`dir()` yerleşik fonksiyon ve değişkenlerin isimlerini listelemez. Bunların bir listesini isterseniz, standart modül `__builtin__` içinde bulabilirsiniz:

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError',
 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
 'Exception', 'False', 'FloatingPointError', 'IOError',
 'ImportError',
 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'OverflowWarning',
 'PendingDeprecationWarning', 'ReferenceError',
 'RuntimeError', 'RuntimeWarning', 'StandardError', 'StopIteration',
 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit',
 'TabError',
 'True', 'TypeError', 'UnboundLocalError', 'UnicodeError',
 'UserWarning',
 'ValueError', 'Warning', 'ZeroDivisionError', '__debug__',
 '__doc__',
 '__import__', '__name__', 'abs', 'apply', 'bool', 'buffer',
 'callable', 'chr', 'classmethod', 'cmp', 'coerce', 'compile',
 'complex',
 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod',
 'enumerate', 'eval', 'execfile', 'exit', 'file', 'filter', 'float',
 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id',
 'input', 'int', 'intern', 'isinstance', 'issubclass', 'iter',
 'len', 'license', 'list', 'locals', 'long', 'map', 'max', 'min',
 'object', 'oct', 'open', 'ord', 'pow', 'property', 'quit',
 'range', 'raw_input', 'reduce', 'reload', 'repr', 'round',
 'setattr', 'slice', 'staticmethod', 'str', 'string', 'super',
 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

6.4 Paketler

Paketler ``noktalı modül isimleri" kullanarak Python'un modül isim alanının düzenlenmesinde kullanılırlar. Örneğin modül adı `A.B` adı "A" olan bir paket içindeki "B" adlı alt modülü gösterir. Nasıl modüller farklı modül yazarlarını birbirlerinin kullandığı global değişkenleri dert etmekten kurtarıyorsa, paketler de NumPy ya da PyOpenGL gibi çok sayıda modül içeren paketlerin birbirilerinin modül isimlerinin çakışması tehlikesinden kurtarır.

Ses dosyaları ve ses verisi üzerinde işlem yapacak bir modül koleksiyonu (bir ``paket") geliştirmek istediğinizi düşünelim. Farklı formatlardaki ses dosyalarını (.wav, .aiff, .au gibi dosya uzantıları olan) birbirine dönüştürmek, seslere efektler uygulamak veya sesleri filtrelemek için pek çok modüle ihtiyacınız olacak. Paketinizin muhtemel dizin yapısı şöyle olabilir:

Sound/	Paketin en üst seviyesi
__init__.py	paketi ilk kullanıma hazırlama
Formats/	Farklı dosya formatları için alt
paket	
__init__.py	
wavread.py	
wavwrite.py	
aiffread.py	
aiffwrite.py	
auread.py	
auwrite.py	
...	
Effects/	ses efektleri alt paketi
__init__.py	
echo.py	
surround.py	
reverse.py	
...	
Filters/	filtre alt paketi
__init__.py	
equalizer.py	
vocoder.py	
karaoke.py	
...	

__init__.py dosyaları Python'un bu dizinleri paket içeren dizinler olarak algılaması için gereklidirler. Bunlar aynı isimli dizinlerin modül arama yolunda bulunacak diğer geçerli modülleri istem dışı saklamasını engeller. __init__.py boş bir dosya olabileceği gibi paketi ilk çalışmaya hazırlayabilir ya da daha sonra açıklanacak olan __all__ değişkenine değer atıyor olabilir.

Paketin kullanıcısı paketten dilediği bir modülü yükleyebilir.

```
import Sound.Effects.echo
```

Bu Sound.Effects.echo modülünü yükler. Modüle tüm ismi ile atıfta bulunulmalı:

```
Sound.Effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Aynı modülü yüklemenin bir diğer yolu:

```
from Sound.Effects import echo
```

Bu da echo alt modülünü yükler; ancak bunu paket adı verilmeden erişilebilir kılar ve modül şu şekilde kullanılabilir:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Bir diğer yol da istene fonksiyon ya da değişkeni doğrudan yüklemektir:

```
from Sound.Effects.echo import echofilter
```

Bu da `echo` modülünü yükler; ancak `echofilter()` fonksiyonnu doğrudan erişilebilir kılar:

```
echofilter(input, output, delay=0.7, atten=4)
```

`from paket import isim` kullanılırken *isim* bir alt modül, alt paket ya da paket içinde tanımlı bir fonksiyon, sınıf veya değişken ifade eden herhangi bir isim olabilir. `import` deyimi önce ismin pakette tanımlı olup olmadığına bakar; tanımlı değil ise bunun bir modül olduğunu varsayar ve bunu yüklemeye teşebbüs eder. Modülü bulamaz ise `ImportError` istisnası oluşur.

`import item.subitem.subsubitem` ifadesinde ise son ismin dışındaki isimler paket olmalıdır. Son isim bir modül veya paket olabilir; ancak bir önceki ismin içinde tanımlanan bir fonksiyon ya da değişken olamaz.

6.4.1 Bir paketten * yüklemek

Kullanıcı `from Sound.Effects import *` yazdığında ne olur ? Dosya sistemine ulaşıp paketin içinde hangi alt paketlerin olduğunu bulunması ve hepsinin yüklenmesi beklenir. Ne yazık ki bu işlem küçük/ büyük harf ayrımının olmadığı Windows ve Mac işletim sistemlerinde pek iyi çalışmaz. Bu işletim sistemlerinde `ECHO.PY` gibi bir dosyanın `echo`, `Echo` veya `ECHO` isimlerinden hangisi ile yüklenmesi gerektiğini belirlemenin garantili bir yolu yoktur. Örneğin, Windows 95 dosya adlarının ilk harfini daima büyük harf ile gösterir. DOS'un 8+3 harflik dosya adı uzunluğu kısıtlaması da uzun modül isimleri için sorun olmaktadır.

Tek çözüm paket yazarının açık bir paket indeksi hazırlamasıdır. Bir paketin `__init__.py` dosyası `__all__` adlı bir liste tanımlıyorsa bu liste `from package import *` ifadesi kullanıldığında yüklenecek modül isimlerinin listesi olarak kullanılır. Pedein yeni bir sürümü hazırlandığında bu listenin uygun şekilde güncellenmesi paket yazarının sorumluluğundadır. Eğer paketten * yüklemeye ihtiyaç duyulmayacağına karar verilirse bu özellik kullanılmayabilir. Örneğin `Sounds/Effects/__init__.py` dosyasının içeriği şöyle olabilir:

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

Bu `from Sound.Effects import *` ifadesinin `Sound` paketinden isimleri `__all__` içinde geçen üç modülün yüklemesini sağlar.

`__all__` tanımlanmamış ise `from Sound.Effects import *` ifadesi `Sound.Effects` paketindeki bütün alt modülleri yürürlükte olan isim alanına *yüklemes*; sadece `Sound.Effects` paketinin ve içindeki isimlerin yüklenmesini sağlar (muhtemelen `__init__.py`) dosyasını çalıştırdıktan sonra). Bundan önceki `import` deyimlerince yüklenen alt paketler de yüklenir. Şu koda bir bakalım:

```
import Sound.Effects.echo
import Sound.Effects.surround
from Sound.Effects import *
```

Bu örnekte echo ve surround modülleri `from...import` ifadesi çalıştırıldığında `Sound.Effects` paketinde tanımlı oldukları için yürürlükte olan isim alanına yüklenirler. Bu `__all__` tanımlı olduğunda da bu çalışır.

Genel olarak bir modül ya da paketten * yüklemek hoş karşılanmaz; çünkü çoğunlukla zor okunan koda neden olur. Bunun etkileşimli kipte kullanılmasının bir sakıncası yoktur. Ayrıca bazı modüller sadece belirli bir kalıba uyan isimleri verecek şekilde tasarlanmışlardır.

`from Paket import gerekli_altmodül` ifadesini kullanmanın hiç bir kötü tarafı yoktur. Yükleyen modül farklı paketlerden aynı isimli modüller yüklemeye gereksinim duymadığı sürece tavsiye edilen kullanım şekli de budur.

6.4.2 Birbirilerini Yükleyen Modüller

Alt modüller çoğu kez birbirilerine atıfta bulunurlar. Örneğin `surround` modülü `echo` modülüne ihtiyaç duyabilir. Aslında bu türden atıflar öyle yaygındır ki `import` deyimi standart modül arama yoluna bakmadan önce çağırıldığı paketin içinde arama yapar. Bu şekilde `surround` modülü `import echo` veya `from echo import echofilter` ifadeleri ile kolayca `echo` modülüne kavuşabilir. Yüklenmek istenen modül içinde bulunan pakette (yükleme yağmaya çalışan modülün bulunduğu paket) bulunamaz ise `import` deyimi aynı isimli üst seviyeli bir modül arar.

Paketler `Sound` paketindeki gibi alt paketler şeklinde düzenlenmişler ise farklı alt paketler içindeki modüllerin birbirilerine atıfta bulunmasının kısa bir yolu yoktur; paketin tam adı kullanılmalıdır. Örneğin, `Sound.Filters.vocoder` modülünün `echo` modülünü kullanması gerekiyor ise `from Sound.Effects import echo` ifadesi ile buna erişebilir.

7. Giriş ve Çıkış

Bir programın çıktısını sunmanın birkaç yolu vardır; veri yazdırılabilir ya da gelecekte kullanılabilecek şekilde bir dosyaya kaydedilebilir. Bu bölümde giriş ve çıkış ile ilgili olanakların bazılarına değineceğiz.

7.1 Daha Güzel Çıkış Biçimi

Buraya kadar değerleri yazdırmanı iki yolunu gördük: *deyim ifadeleri* ve `print` deyimi. Üçüncü bir yol da dosya nesnelerinin `write()` metodudur. Standart çıktı dosyasına `sys.stdout` şeklinde atıfta bulunulabilir.

Çoğu zaman boşluklar ile birbirinden ayrılmış değerlerden daha iyi biçimlendirilmiş bir çıktıya ihtiyaç duyulur. Çıktınızı biçimlendirmenin iki yolu var. İlki bütün karakter dizisi işlemlerini dilimleme ve birleştirme ile yapıp istediğiniz herhangi bir biçimi elde etmek. `string` standart modülü karakter dizilerinin istenen sütun genişliğine kadar boşluklar ile

doldurulmasını sağlayan ,daha sonra değineceğimiz, bazı faydalı fonksiyonlara sahiptir. İkinci yol ise sol argümanı bir karakter dizisi olan % islecini kullanmaktır. % isleci sol argümanını sağdaki argümanına uygulanacak `sprintf()` tarzı biçim karakter dizisi olarak yorumlar ve biçimleme işleminden sonra bir karakter dizisi geri döndürür.

Sayısal değerleri karakter dizisine çevirmek için ise değer `repr()` veya `str()` fonksiyonuna geçirilebilir ya da ters tırnak işareti (```) içine alınabilir (`repr()` ile aynı etkiyi yapar).

`str()` fonksiyonu değerlerin insan tarafından okunabilir gösterimini geri döndürürken, `repr()` fonksiyonu ise yorumlayıcı tarafından okunabilir gösterimini geri döndürür (veya uygun sözdizim yok ise `SyntaxError` istisnası oluşturur). İnsan için anlam ifade edecek bir gösterimi bulunmayan nesneler için `str()` fonksiyonu `repr()` ile aynı değeri döndürür. Rakamlar, listeler ve sözlükler gibi yapılar ile daha pek çok değer için her iki fonksiyon da aynı sonucu verir. Karakter dizileri ve kayar noktalı rakamlar ise iki farklı gösterime sahiptir.

İşte birkaç örnek:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> `s`
"'Hello, world.'"
>>> str(0.1)
'0.1'
>>> `0.1`
'0.10000000000000001'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + `x` + ', and y is ' + `y` + '...'
>>> print s
The value of x is 32.5, and y is 40000...
>>> # Ters tırnaklar sayılar dışındaki tipler ile de çalışır:
... p = [x, y]
>>> ps = repr(p)
>>> ps
'[32.5, 40000]'
>>> # Karakter dizisinde ise tırnaklar ve ters bölü işareti eklenir:
... hello = 'hello, world\n'
>>> hellos = `hello`
>>> print hellos
'hello, world\n'
>>> # Ters tırnakların argümanı bir demet de olabilir:
... `x, y, ('spam', 'eggs')`
"(32.5, 40000, ('spam', 'eggs'))"
```

Sayıların kare ve küplerinden oluşan bir tablo yazdırmanın iki yolu vardır:

```
>>> import string
>>> for x in range(1, 11):
...     print string.rjust(`x`, 2), string.rjust(`x*x`, 3),
...     # Üst satırın sonundaki virgüle dikkat edin.
...     print string.rjust(`x*x*x`, 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 '%2d %3d %4d' % (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

```

Sütunların arasındaki bir karakterlik boşluk `print` tarafından eklenir; argümanların arasına daima bir boşluk karakteri eklenir.

Bu örnek karakter dizilerinin başını boşluklar ile doldurup bunları sağ tarafa dayayan `string.rjust()` fonksiyonunu kullanmaktadır. Buna benzer `string.ljust()` ve `string.center()` fonksiyonları da vardır. Bunlar bir şey yazdırmaz; sadece yeni bir karakter dizisi geri döndürürler. Verilen karakter dizisi uzun ise kırpılmaz ve aynen geri döndürülür; bu sütunlarınızın bozulmasına sebep olmasına rağmen hatalı bir değer göstermekten iyidir. Büyük bir değeri kırpmayı gerçekten istiyorsanız dilimleme ile bunu yapabilirsiniz (`"string.ljust(x, n)[0:n]"` gibi).

`string.zfill()` fonksiyonu ise rakamlar içeren karakter dizilerinin başını sıfırlar ile doldurur. Bu fonksiyon artı ve eksi işaretlerini de dikkate alır:

```

>>> import string
>>> string.zfill('12', 5)
'00012'
>>> string.zfill('-3.14', 7)
'-003.14'
>>> string.zfill('3.14159265359', 5)
'3.14159265359'

```

`%` isleci şu şekilde kullanılır:

```

>>> import math
>>> print 'PI sayısının yaklaşık değeri: %5.3f' % math.pi
PI sayısının yaklaşık değeri: 3.142

```

Karakter dizisinin içinde birden fazla biçim varsa sağ operand olarak bir demet kullanmak gerekir:

```

>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print '%-10s ==> %10d' % (name, phone)
...
Jack          ==>      4098

```

```
Dcab      ==>      7678
Sjoerd    ==>      4127
```

Çoğu biçim aynı C dilindeki gibi çalışır ve doğru veri tipinin geçirilmesi gerekir; bu yapılamaz ise bir istisna oluşur. `%s` biçiminin kullanımı daha rahattır; verilen argüman karakter dizisi değilse yerleşik fonksiyon `str()` ile karakter dizisine dönüştürülür. Genişlik ya da hassasiyeti belirtmek için `*` ile bir tamsayı argüman kullanılabilir. C dilindeki `%n` ve `%p` biçimler ise desteklenmemektedir.

Eğer bölmek istemediğiniz gerçekten uzun bir biçim karakter diziniz varsa biçimlendirmek istediğiniz argümanlara konumu yerine ismiyle atıfta bulunabilmeniz güzel olur. Bu aşağıda gösterildiği gibi `%(isim)biçim` şeklinde yapılabilir:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print 'Jack: %(Jack)d; Sjoerd: %(Sjoerd)d; Dcab: %(Dcab)d' %
table
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Bu özellik bütün yerel değişkenlerin bulunduğu bir sözlük geri döndüren yerleşik fonksiyon `vars()` ile beraber kullanıldığında faydalı olur.

7.2 Dosya Okuma ve Yazma

`open()` fonksiyonu bir dosya nesnesi geri döndürür ve genellikle iki argüman ile kullanılır: `"open(dosya_adı, mod)"`

```
>>> f=open('/tmp/workfile', 'w')
>>> print f
<open file '/tmp/workfile', mode 'w' at 80a0960>
```

İlk argüman dosya adını içeren bir karakter dizisidir. İkincisi ise dosyanın nasıl kullanılacağını belirten karakterlerden oluşur. Erişim *modu* dosyadan sadece okuma yapılacak ise `'r'`, sadece yazma için `'w'` (anı isimli bir dosya zaten var ise bu silinir) ve dosyanın sonuna eklemeler yapmak için `'a'` olur. `'r+'` modu dosyayı hem okuma hem de yazma yapmak için açar. *mod* argümanı seçimliktir; kullanılamaması halinde `'r'` olduğu varsayılır.

Windows ve Macintosh üzerinde moda eklenen `'b'` harfi dosyayı binary modunda açar; yani `'rb'`, `'wb'`, ve `'r+b'` gibi modlar da vardır. Windows metin ve binary dosyaları arasında ayırım yapmaktadır; metin dosyalarında okuma veya yazma işlemlerinde satır sonu karakterleri otomatik olarak biraz değişir. Bu görünmez değişiklik ASCII metin dosyaları için iyidir; ancak JPEG resimler veya .EXE dosyalar gibi binary verileri bozar.

7.2.1 Dosya Nesnelerinin Metodları

Bundan sonraki örneklerde `f` adlı bir dosya nesnesinin önceden yaratılmış olduğunu varsayacağız.

Dosyanın içeriğini okumak için belirli miktarda veriyi okuyup bunu karakter dizisi olarak geri döndüren `f.read(boy)` metodu kullanılabilir. *boy* okunacak bayt sayısını belirleyen seçimlik bir argümandır; kullanılmaması halinde dosyanın tamamı okunur. Dosyanın sonuna gelindiğinde `f.read()` boş bir karakter dizisi (`""`) geri döndürür.

```
>>> f.read()
'Dosyanın tamamı bu satırdan oluşuyor.\n'
>>> f.read()
''
```

`f.readline()` dosyadan tek bir satır okur. Satırın sonundaki yeni satır karakteri (`\n`) korunur; ancak dosya yeni bir satır ile bitmiyor ise son satırda bu karakter silinir. Bu özellik geri döndürülen değerin birden fazla anlama gelmesini engeller; `f.readline()` boş bir karakter dizisi geri döndürdüğünde dosyanın sonuna ulaşılrken boş bir satır tek bir `'\n'` karakteri ile ifade edilir.

```
>>> f.readline()
'Bu dosyanın ilk satırı.\n'
>>> f.readline()
'Dosyanın ikinci satırı\n'
>>> f.readline()
''
```

`f.readlines()` dosya içindeki bütün satırların bulunduğu bir liste geri döndürür. Seçimlik parametre *boy_ipucu* kullanılması durumunda ise dosyadan *boy_ipucu* kadar ve bundan bir satır tamamlamaya yetecek kadar fazla bayt okunur ve bunlar yine satırlar listesi şeklinde geri döndürülür.

```
>>> f.readlines()
['Bu dosyanın ilk satırı.\n', 'Dosyanın ikinci satırı\n']
```

`f.write(karakter_dizisi)` metodu *karakter_dizisi* içeriğini dosyaya yazar ve `None` geri döndürür.

```
>>> f.write('Bu bir deneme satırıdır.\n')
```

`f.tell()` dosya nesnesinin dosya içindeki konumunu belirten bir tamsayı geri döndürür (dosyanın başından bayt cinsinden ölçülür). "`f.seek(uzaklık, nereden)`" ile de dosyanın içinde istenen konuma gidilebilir. Konum *uzaklık* ile referans noktası *nereden* değerlerinin toplanması ile bulunur. *nereden* 0 olursa dosyanın başını, 1 o andaki konumu, 2 ise dosyanın sonunu belirtir. *nereden* kullanılmaz ise 0 olduğu varsayılır ve referans noktası olarak dosyanın başı alınır.

```
>>> f=open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)      # Dosyadaki 5'inci bayta git
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Sondan 3'üncü bayta git
>>> f.read(1)
'd'
```

Dosya ile işiniz bittiğinde metodunu `f.close()` çağırarak dosyayı kapatabilir ve dosyanın işgal ettiği sistem kaynaklarını serbest bırakabilirsiniz. `f.close()` çağırıldıktan sonra dosya üzerinde başka işlem yapmaya devam etmek mümkün değildir:

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

Dosya nesnelerinin `isatty()` ve `truncate()` gibi pek sık kullanılmayan başka metodları da vardır.

7.2.2 *pickle* Modülü

Karakter dizileri kolayca dosyalara yazılıp dosyalardan okunabilirler. Sayılar biraz zahmetlidir; çünkü `read()` metodu sadece karakter dizileri geri döndürür ve bunların `'123'` gibi bir değeri alıp sayısal değeri `123`'ü geri döndüren `string.atoi()` fonksiyonundan geçirilmeleri gerekir. Listeler, sözlükler ve sınıf fertleri (class instances) gibi daha karmaşık veri türlerini dosyalara kaydetmek isterseniz işler oldukça zorlaşır.

Programcılar karmaşık veri türlerini saklamak için kodlamak ve hata ayıklamak ile uğraştırmak yerine Python bu iş için `pickle` adlı standart modülü sağlar. Bu hayret verici modül neredeyse herhangi bir Python nesnesini (bazı Python kodu biçimlerini bile !) karakter dizisi ile ifade edilebilecek hale getirebilir ve bu halinden geri alabilir. Bu dönüşüm ve geri kazanım işlemleri arasında nesne bir dosyaya kaydedilebilir ya da ağ bağlantısı ile uzaktaki başka bir makineye gönderilebilir.

`x` gibi bir nesneniz ve yazma işlemi için açılmış `f` gibi bir dosya nesneniz varsa bu nesneyi dosyaya aktarmanız için tek satırlık kod yeterli olur:

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

Nesneyi geri almak için ise `f` okumak için açılmış bir dosya nesnesi olsun:

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

Birden fazla nesnenin dönüştürülmesi gerekiyor ya da dönüştürülmüş olan nesnelerin dosyaya yazılması istenmiyor ise `pickle` farklı şekilde kullanılır. Bunları `pickle` modülünün dokümanlarından öğrenmek mümkündür.

`pickle` modülü saklanabilen ve başka programlar tarafından ya da aynı programın farklı çalışma zamanlarında kullanılabilecek Python nesneleri yapmanın standart yoludur. `pickle` modülü çok yaygın kullanıldığından Python genişletme modülleri yazan çoğu programcı matrisler gibi yeni veri tiplerinin doğru olarak dönüştürülebilir ve geri alınabilir olmasına özen gösterirler.

8. Hatalar ve İstisnalar

Şu ana kadar hata mesajlarından pek bahsedilmedi; ancak örnekleri denediyseniz muhtemelen birkaç tane görmüşsünüzdür. Birbirinden ayırt edilebilen en az iki tür hata mevcuttur: *sözdizim hataları* ve *istisnalar*.

8.1 Sözdizim Hataları

Sözdizim hataları ayrıştırma (parsing) hataları olarak da bilinirler ve Python öğrenirken en çok bunlar ile karşılaşacaksınız:

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

Ayrıştırıcı (parser) sorun olan satırı basar ve satır içinde hatanın algılandığı ilk noktayı küçük bir `ok` ile gösterir. Hata oktan önce gelen kısımdan kaynaklanmaktadır. Örnekte hata `print` anahtar kelimesinde fark edilmektedir; çünkü ondan önce bir iki nokta üst üste (":") karakteri eksiktir. Dosya adı ve satır numarası da yazdırılmaktadır ki yorumlayıcı girişinin bir dosyadan gelmesi durumunda hatanın nereden kaynaklandığını bilesiniz.

8.2 İstisnalar

Bir deyim ya da ifade sözdizimsel olarak doğru olsa da yürütülmek istendiğinde bir hataya sebep olabilir. İcra sırasında meydana gelen hatalara *istisna* denir. İstisnaları nasıl ele alabileceğinizi yakında öğreneceksiniz. Çoğu istisnalar programlar tarafından ele alınmaz ve aşağıdakiler gibi hata mesajları ile sonuçlanırlar:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
```

Hata mesajının son satırı sorunun ne olduğunu belirtir. İstisnaların farklı türleri vardır ve istisnanın türü hata mesajının bir bölümü olarak yazdırılır. Örneklerdeki istisna türleri: `ZeroDivisionError`, `NameError` ve `TypeError`. İstisna türü olarak yazdırılan karakter dizisi

meydana gelen istisnanın yerleşik (built-in) ismidir. Bu bütün yerleşik istisnalar için geçerlidir; ancak kullanıcı tanımlı istisnalar için böyle olmayabilir. Standart istisna isimleri yerleşik belirteçlerdir (`identifier`); ayrılmış anahtar kelimeler değil.

Satırın devamı istisna türüne bağlı detaylardan oluşur ve anlamı istisna türüne bağlıdır.

Hata mesajının baş kısmında istisnanın meydana geldiği yer yığın dökümü şeklinde görülür. Bu genellikle istisnanın gerçekleştiği noktaya gelene kadar işletilen kaynak kodu şeklinde olur; ancak standart girdiden okunan satırlar gösterilmez.

Yerleşik istisnalar ve bunların anlamları için Python ile gelen dokümanlardan yararlanılabilir.

8.3 İstisnaların Ele Alınması

Belirli istisnaları ele alan programlar yazmak mümkündür. Aşağıdaki örnek kullanıcıdan geçerli bir tamsayı girilene kadar kullanıcıdan giriş yapması istenir. `Control-C` tuş kombinasyonu (ya da işletim sisteminin desteklediği başka bir kombinasyon) ile kullanıcı programdan çıkabilir. Kullanıcının sebep olduğu bu olay ise `KeyboardInterrupt` istisnasının oluşmasına neden olur.

```
>>> while True:
...     try:
...         x = int(raw_input("Lütfen bir rakam giriniz: "))
...         break
...     except ValueError:
...         print "Bu geçerli bir giriş değil. Tekrar deneyin..."
... 
```

`try` deyimi aşağıdaki gibi çalışır:

- Önce *try bloğu* (`try` ve `except` arasındaki ifade(ler)) işletilir.
- Hiçbir istisna oluşmaz ise *except bloğu* atlanır ve `try` deyimin icrası son bulur.
- Eğer `try` bloğu içinde bir istisna oluşur ise bloğun geri kalanı atlanır. İstisnanın türü `except` anahtar kelimesinden sonra kullanılan ile aynı ise `try` bloğunun kalan kısmı atlanır ve `except` bloğu yürütülür ve programın akışı `try ... except` kısmından sonra gelen ilk satırdan devam eder.
- Adı `except` bloğunda geçmeyen bir istisna oluşur ise üst seviyedeki `try` ifadelerine geçirilir; ancak bunu ele alan bir şey bulunmaz ise bu bir *ele alınmamış istisna* olur. ve yürütme işlemi yukarıda da görüldüğü gibi bir hata mesajı ile son bulur .

Bir `try` deyimi farklı istisnaları yakalayabilmek için birden fazla `except` bloğuna sahip olabilir. Bir `except` bloğu parantez içine alınmış bir liste ile birden fazla istisna adı belirtebilir. Örnek:

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

Son except bloğu istisna adı belirtilmeden de kullanılıp herhangi bir istisnayı yakalayabilir. Bunu çok dikkatli kullanın, çünkü çok ciddi bir programlama hatasını bu şekilde gözden kaçırabilirsiniz! Bu özellik bir hata mesajı bastırıp ve tekrar bir istisna oluşturarak çağırının istisnayı ele almasını da sağlamak için kullanılabilir:

```
import string, sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(string.strip(s))
except IOError, (errno, strerror):
    print "I/O error(%s): %s" % (errno, strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

try ... except ifadesinin seçimlik *else bloğu* da vardır. Bu her except bloğunun ardına yazılır ve try bloğunun istisna oluşturmadığı durumlarda icra edilmesi gereken kod bulunduğu zaman kullanılır. Örnek:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
```

else bloğu kullanmak try bloğuna ek satırlar eklemekten iyidir çünkü bu try ... except ifadesi tarafından korunan kodun oluşturmadığı bir istisnanın kazara yakalanmasını engeller.

Bir istisna meydana geldiğinde istisna *argümanı* olarak bilinen bir değer de bulunabilir. Argümanın varlığı ve tipi istisnanın tipine bağlıdır. Argümanı olan istisna tipleri için except bloğunda istisna adından (ya da listesinden) sonra argüman değerini alacak bir değişken belirtilebilir:

```
>>> try:
...     spam()
... except NameError, x:
...     print 'name', x, 'undefined'
...
name spam undefined
```

Bir istisnanın argümanı var ise ele alınmayan istisna mesajının son kısmında ('detay') basılır.

İstisna işleyiciler (exception handlers) sadece try bloğu içinde meydana gelen istisnaları değil try bloğundan çağırılan (dolaylı olarak bile olsa) fonksiyonlardaki istisnaları da ele alırlar. Örnek:

```
>>> def this_fails():
...     x = 1/0
```

```
...
>>> try:
...     this_fails()
... except ZeroDivisionError, detail:
...     print 'Handling run-time error:', detail
...
Handling run-time error: integer division or modulo
```

8.4 İstisna Oluşturma

`raise` anahtar kelimesi programcının kasıtlı olarak bir istisna oluşturmasını sağlar. Örnek:

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

`raise` için ilk argüman oluşturulacak istisnanın adıdır ve ikinci argüman ise istisnanın argümanıdır.

Eğer bir istisnanın oluşup oluşmadığını öğrenmek istiyor; fakat bunu ele almak istemiyorsanız, `raise` ifadesinin istisnayı tekrar oluşturmanıza imkan veren daha basit bir formu var:

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

8.5 Kullanıcı Tanımlı İstisnalar

Programlar yeni bir istisna sınıfı yaratarak kendi istisnalarını isimlendirebilirler. İstisnalar genellikle, doğrudan veya dolaylı olarak, `Exception` sınıfından türetilirler. Örnek:

```
>>> class bizimHata(Exception):
...     def __init__(self, deger):
...         self.deger = deger
...     def __str__(self):
...         return `self.deger`
...
>>> try:
...     raise bizimHata(2*2)
... except bizimHata, e:
...     print 'İstisnamız oluştu, deger:', e.deger
```

```
...
İstisnamız oluştu, deger: 4
>>> raise bizimHata, 'aaah!'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.bizimHata: 'aaah!'
```

İstisna sınıfları diğer sınıfların yapabildiği her şeyi yapabilecek şekilde tanımlanabilirler, fakat genellikle basit tutulurlar ve sıklıkla sadece istisnayı işleyenlerin hata hakkında bilgi almasını sağlayacak birkaç özellik sunarlar. Birkaç farklı istisna oluşturabilen bir modül yaratırken, yaygın bir uygulama da bu modül tarafından tanımlanan istisnalar için bir temel sınıf yaratıp ve farklı hata durumları için bundan başka istisna sınıfları türetmektir:

```
class Error(Exception):
    """Bu modüldeki istisnalar için temel sınıf."""
    pass

class GirisHatasi(Error):
    """Giriş hataları için oluşacak istisna.

    Özellikler:
        ifade -- hatanın oluştuğu giriş ifadesi
        mesaj -- explanation of the error
    """

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

class GecisHatasi(Error):
    """İzin verilmeyen bir durum geçişine teşebbüs edildiğinde
    oluşacak istisna.

    Özellikler:
        onceki -- geçiş başlangıcındaki durum
        sonraki -- istenen yeni durum
        mesaj -- durum geçişine izin verilmemesinin sebebi
    """

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

Çoğu standart modül kendi tanımladıkları fonksiyonlarda meydana gelen hataları rapor etmek için kendi istisnalarını tanımlarlar.

Sınıflar üzerine daha fazla bilgi sonraki bölümde sunulacaktır.

8.6 Son işlemlerinin Belirlenmesi

`try` deyiminin her durumda yürütülecek işlemleri belirten seçimlik bir bloğu da vardır. Örnek:

```
>>> try:
```

```
...     raise KeyboardInterrupt
... finally:
...     print 'Goodbye, world!'
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2
KeyboardInterrupt
```

finally bloğu try bloğu içinde bir istisna oluşsa da oluşmasa da yürütülür. Bir istisna oluşursa finally bloğu icra edildikten sonra istisna tekrar oluşturulur. Finally bloğu try deyimi break veya return ile sonlanırsa da icra edilir.

try deyiminin bir ya da daha fazla except bloğu veya bir finally bloğu olmalıdır; ancak her ikisi bir arada olamaz.

9. Sınıflar

Python'un sınıf mekanizması dile minimum yeni sözdizim ile sınıflar ekler. Bu C++ ve Modula-3 sınıf mekanizmalarının bir karışımıdır. Modüllerde de olduğu gibi Python'da sınıflar tanım ile kullanıcı arasına mutlak bir sınır koymaz. Miras mekanizması birden fazla temel sınıf kullanılabilir. Türetilmiş sınıflar temel sınıfın metodlarını geçeriz kılabilir ve kendi sınıf metodlarından temel sınıf metodlarını çağırabilirler. Nesneler herhangi miktarda özel veri içerebilirler.

C++ terminolojisi ile söylemek gerekirse, tüm sınıf üyeleri (veri üyeleri de dahil) ``public" ve tüm üye fonksiyonlar ``virtual" kabul edilir. Hiçbir özel constructor/destructor yoktur. Smalltalk dilinde olduğu gibi sınıfların kendileri de nesnedir. Python'da bütün veri tipleri nesnedir aslında. C++ veya Modula-3 dillerinden farklı olarak yerleşik (built-in) veri tipleri kullanıcı tarafından geliştirilmek üzere başka sınıflar için temel sınıf olarak kullanılamazlar. Ayrıca, çoğu yerleşik işlemler sınıf gerçeklemeleri için yeniden tanımlanabilirler.

9.1 Terminoloji üzerine

Yaygın kabul gören sınıf terminolojisini bilmediğimden, arada sırada Smalltalk ve C++ terimleri kullanacağım. Aslında, nesne yönelimi açısından Python'a daha çok benzeyen Modula-3 dili terimlerini kullanırdım; fakat bu dilden haberdar olan okuyucuların azınlıkta olduğunu sanıyorum.

Burada nesne tabanlı programlama ile ilgilenen okuyucuların dikkat etmesi gereken terminolojik bir sorun var: Python'da ``nesne" kelimesi her zaman bir sınıf gerçeklemesi anlamına gelmez. C++ ve Modula-3 dillerinde olduğu gibi, Python'da da bütün veri tipleri sınıf değildir. Örneğin tamsayılar, listeler ve hatta dosyalar gibi daha egzotik veri tipleri bile sınıf değildir. Buna rağmen, *tüm* Python veri tipleri anlambilim açısından en iyi ``nesne" kelimesi ile tanımlanabilecek benzerliklere sahiptirler.

Nesnelerin bireyselliği vardır ve birden fazla etki alanında birden fazla isim aynı nesneye bağlı olabilir. Bu diğer dillerde yansılama (aliasing) olarak bilinir. İlk bakışta Python'un bu özelliği pek beğenilmez, ve değiştirilemeyen (immutable) temel veri tipleri ile uğraşılırken bu güvenle ihmal edilebilir. Yansılama listeler ve sözlükler gibi değiştirilebilir nesneler ile program dışındaki unsurları ifade eden diğer tipler (dosyalar, pencereler vs.) içeren Python kodlarının anlamı üzerinde etkilidir. Bu genellikle programın yararına kullanılır; çünkü yansılar bazı açılardan işaretçilere (pointer) benzerler. Örneğin, bir nesne geçirildiğinde sadece bir işaretçi geçirilmiş olur; ve bir fonksiyon kendisine argüman olarak geçirilen bir nesnede değişiklikler yaparsa fonksiyonu çağıran da değişiklikleri görür. Bu durum Pascal'daki gibi iki farklı argüman geçirme mekanizmasına duyulan ihtiyacı ortadan kaldırır.

9.2 Python Etki ve İsim Alanları

Sizi sınıflar ile tanışmadan önce, biraz da Python'un etki alanı kurallarından bahsetmem gerek. Sınıf tanımlamaları tam olarak neler olduğunu anlamak için etki ve isim alanlarının nasıl çalıştığını bilmeniz gerek. Bu konudaki bilgiler ileri seviyedeki her Python programcısı için yararlıdır.

Birkaç tanım ile işe koyulalım.

Bir *isim alanı* isimler ile nesnelerin eşleşmesidir. Çoğu isim alanları şu anda Python sözlükleri olarak kodlanmışlardır, fakat bu hiçbir şekilde fark edilmez ve gelecekte değiştirilebilir. İsim alanlarına örnekler: yerleşik isimler kümesi (`abs()` gibi fonksiyonlar ve yerleşik istisna isimleri vs.), bir modül içindeki global isimler ve bir fonksiyon çağrısındaki yerel isimler. Bir nesnenin özellikleri kümesi de bir isim alanıdır. İsim alanlarına ilişkin bilinecek önemli şey farklı isim alanlarındaki isimlerin birbirileri ile hiçbir ilişkisi olmadığıdır. Örneğin, iki farklı modül karışıklık yaratmadan ``maksimize" adlı birer fonksiyon tanımlayabilir; kullanıcılar bu fonksiyonları önlerine modül adını ekleyerek kullanırlar.

Bu arada, bir noktadan sonra yazılan her herhangi bir isim için *özellik* kelimesini kullanıyorum. Örneğin, `z.real` ifadesinde `real` `z` nesnesinin bir özelliğidir. Modül içindeki isimlere atıflar da özellik atıflarıdır: `modulAdi.fonkAdi` ifadesinde `modulAdi` bir modül nesnesidir ve `fonkAdi` bunun bir özelliğidir. Bir modülün özellikleri ile içinde tanımlı global değişkenler aynı isim alanını paylaşırlar. ^{9.1}

Özellikler salt okunur veya yazılabilir olabilirler. Yazılabilir oldukları durumda özelliklere atama yapmak mümkündür. Modül özellikleri yazılabiliridir: "`modulAdi.sonuc = 42`" gibi bir ifade kullanabilirsiniz. Yazılabilir özellikleri `del` deyimini kullanarak silmek de mümkündür. Örneğin "`del modulAdi.sonuc`" ifadesi `modulAdi` nesnesinden `sonuc` isimli özelliği siler.

Farklı anlarda yaratılan isim alanlarının farklı ömürleri olur. Yerleşik isimleri içeren isim alanı Python yorumlayıcısı çalıştırıldığında yaratılır ve asla silinmez. Bir modüle ait global isim alanı modül tanımı okunduğunda yaratılır ve genellikle yorumlayıcı çalıştığı sürece silinmez. Yorumlayıcının bir dosyadan veya etkileşimli olarak çalıştığı deyimler de `__main__` isimli bir modüle ait kabul edilir, ve bunların da kendi global isim alanı vardır. Yerleşik (built-in) isimler de `__builtin__` isimli bir modülde yaşarlar.

Bir fonksiyona ait yerel isim alanı fonksiyon çağırıldığında yaratılır, ve fonksiyondan döndüğünde veya fonksiyon içinde ele alınmamış bir istisna gerçekleştiğinde silinir. Tabi ki Özyinelemeli (recursive) çağrılarının herbiri kendi yerel isim alanına sahiptir.

Bir *etki alanı* bir isim alanının doğrudan erişilebildiği bir metin bölgesidir. Burada ``doğrudan erişilebilir" ifadesini anlamı yetersiz bir isim atfının isim alanında isim bulmaya teşebbüs etmesidir.

Etki alanları statik olarak belirlenmelerine rağmen, dinamik olarak kullanılırlar. İcranın herhangi bir anında isim alanlarına doğrudan erişilebilen iç içe geçmiş en az üç etki alanı vardır: ilk aranan ve yerel isimleri içeren en iç etki alanı; en yakın olanından başlanarak aranan çevreleyen fonksiyonların isim alanları; daha sonra aranan ve o andaki modülün global değişkenlerini içeren orta etki alanı; ve yerleşik isimlerin bulunduğu isim alanı olan en dış etki alanı (en son aranır).

Eğer bir isim global olarak tanımlanmış ise tüm atıflar ve atamalar doğrudan modülün global isimlerini barındıran orta etki alanına giderler. Aksi takdirde, en iç etki alanının dışındaki tüm isimler salt okunurdur.

Genellikle yerel etki alanı o an içinde bulunan (program metninde) fonksiyonun yerel isimlerine atıfta bulunur. Fonksiyonların dışında yerel etki alanı global etki alanı ile aynı isim alanıdır: modülün isim alanı. Sınıf tanımlamaları da yerel etki alanı içerisine bir başka isim alanı daha eklerler.

Etki alanlarını metne bağlı olarak belirlendiğini anlamak önemlidir. Bir modül içinde tanımlı bir fonksiyonun global etki alanı o modülün isim alanıdır; nereden çağırıldığı ya da hangi farklı isim ile çağırıldığı bir fark yaratmaz. Diğer yandan, asıl isim araması icra anında dinamik olarak yapılır; ancak dilin tanımı ``derleme" sırasında yapılan statik isim çözümlemeye doğru değişmektedir ve dinamik isim çözümlemeye güvenmemelisiniz ! Örneğin, yerel değişkenler şu anda statik olarak belirlenmektedir.

Python'a özgü bir tuhaflık da atamaların her zaman en iç etki alanına gitmesidir. Atamalar veri kopyalamaz; sadece nesnelere isimler bağlarlar. Aynı şey silme işlemleri için de geçerlidir: `del x` ifadesi `x`'in yerel etki alanı tarafından atfedilen isim alanındaki bağımlı kaldırır. Aslında, yeni isimler yaratan tüm işlemler yerel etki alanını kullanırlar. Fonksiyon tanımları ve import deyimleri modül veya fonksiyon adını yerel etki alanına bağlarlar. Bir değişkenin global etki alanında bulunduğunu belirtmek için `global` deyimini kullanılabilir.

9.3 Sınıflara İlk Bakış

Sınıflar ile bir miktar yeni sözdizim ve anlambilim ile üç yeni nesne tipini tanıttacağız.

9.3.1 Sınıf Tanımlama

Sınıf tanımlamanın en basit şekli şöyledir:

```
class SinifAdi:
    <deyim-1>
    .
    .
    .
    <deyim-N>
```

Sınıf tanımlamalarının, fonksiyon tanımlamalarında (`def` deyimleri) olduğu gibi, etkin olmaları için önce işletilmeleri gerekir. (Yanlışlıkla sınıf tanımlarını `if` deyimleri veya fonksiyon içlerine koymamaya dikkat edin.)

Pratikte bir sınıf tanımının içindeki deyimler genellikle fonksiyon tanımları olur; fakat başka deyimler de kullanmak mümkün ve yararlıdır (buna daha sonra yine değineceğiz). Sınıf içindeki fonksiyon tanımlarının argüman listesi kendilerine özgü bir şekle sahiptir; ancak buna da daha sonra değineceğiz.

Bir sınıf tanımına girildiğinde yeni bir isim alanı (name space) oluşturulur ve bu yerel etki alanı (scope) olarak kullanılır. Yerel değişkenlere yapılan bütün atamalar bu yeni isim alanına gider. Yeni tanımlanan fonksiyonların isimleri de buraya eklenir.

Bir sınıf tanımı normal olarak tamamlandığında bir *sınıf nesnesi* yaratılmış olur. Bu, temel olarak, sınıf tanımının oluşturduğu isim alanı etrafında bir örtüdür. Sınıf nesnelerini bir sonraki bölümde daha yakından tanıyacağız. Orijinal etki alanı (sınıf tanımına girilmeden önce etkin olan) yine eski yerini alır ve sınıf nesnesi de buna sınıf tanımında kullanılan isim (örnekteki `SinifAdi`) ile dahil olur.

9.3.2 Sınıf Nesneleri

Sınıf nesneleri iki tür işlemi destekler: özelliklere değinme (attribute reference) ve sınıfın gerçekleştirilmesi (instantiation).

Özelliklere değinmek için Python'da bütün özelliklere erişmek için kullanılan standart sözdizim kullanılır: `nesne.isim`. Kullanılabilecek özellik isimleri sınıf nesnesi yaratılırken sınıfın isim alanında bulunan bütün isimlerdir. Sınıf tanımımız aşağıdaki gibi ise:

```
class benimSinif:
    "Basit bir sınıf örneği."
    i = 12345
    def f(self):
        return 'Merhaba'
```

`benimSinif.i` ve `benimSinif.f` bir tamsayı ve bir metod nesnesi geri döndüren geçerli özellik atıflarıdır. Sınıf özelliklerine atama yapmak da mümkündür. Örneğin atama yoluyla `benimSinif.i` değeri değiştirilebilir. Ayrıca, `__doc__` da geçerli bir özellik olup sınıfa ait belgeleme karakter dizisini geri döndürür: `"Basit bir sınıf örneği."`.

Sınıfın *gerçeklenmesi* fonksiyon notasyonunu kullanır. Sınıf nesnesini yeni bir sınıf gerçeklemesi geri döndüren parametresiz bir fonksiyonmuş gibi düşünebilirsiniz. Örneğin yukarıda tanımladığımız sınıf için:

```
x = benimSinif()
```

Yeni bir sınıf gerçekleştirilmesi oluşturur ve bu nesneyi `x` yerel değişkenine atar.

Gerçekleme işlemi (bir sınıf nesnesini ``çağırarak``) boş bir nesne yaratır. Pek çok sınıf nesneleri bilinen bir ilk durumda yaratmak ister. Bu yüzden bir sınıf `__init__()` adlı özel metodu şu şekilde tanımlayabilir:

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

Bir sınıfın `__init__()` metodu tanımlanmış ise gerçekleştirme işlemi yeni sınıf gerçekleştirilmesi için bu metodu otomatik olarak çağırır.

Daha fazla esneklik için `__init__()` metodunun argümanları da olabilir. Bu durumda sınıfın gerçekleştirilmesinde kullanılan argümanlar `__init__()` metoduna geçirilir. Örnek:

```
>>> class karmasikSayi:
...     def __init__(self, gercekKsm, sanalKsm):
...         self.g = gercekKsm
...         self.s = sanalKsm
...
>>> x = karmasikSayi(3.0, -4.5)
>>> x.g, x.s
(3.0, -4.5)
```

9.3.3 Nesneler (Gerçeklenen Sınıflar)

Nesneler ile ne yapabiliriz ? Bunlar ile yapabileceğimiz tek şey özellikleri ile uğraşmaktır. Nesnelerin iki tür özellikleri vardır.

Bunların ilki *veri özellikleridir*. Veri özelliklerinin tanımlanmış olması gerekmez; yerel değişkenlerde olduğu gibi bunlar da kendilerine ilk atama yapıldığında var olurlar. Örneğin `x` yukarıda tanımlanan `benimSinif` sınıfının bir gerçekleştirilmesi olduğunu düşünürsek aşağıdaki program parçası 16 değerini geride bir iz bırakmadan yazdırır:

```
x.sayac = 1
while x.sayac < 10:
    x.sayac = x.sayac * 2
print x.sayac
del x.sayac
```

Nesnelerin ikinci tür özellikleri de *metodlardır*. Metod bir sınıfa ``ait olan`` bir fonksiyondur. Python dilinde metodlar sınıf gerçeklemelerine özgü değildir; diğer nesne türlerinin de metodları vardır (listeler, sözlükler vs.). Aşağıda metod terimini, aksi belirtilmediği sürece, sadece sınıf gerçeklemelerinin metodları anlamında kullanacağız.

Bir nesneye ilişkin geçerli özellik isimleri bunun sınıfına bağlıdır. Tanıma göre fonksiyon olan tüm sınıf özellikleri o nesnenin metodları olur. Bu yüzden örnek sınıfımız için `x.f` geçerli bir metod atfıdır, çünkü `benimSinif.f` bir fonksiyondur fakat `x.i` değildir çünkü `benimSinif.i` değildir. Burada şuna dikkat edelim: `x.f` ile `benimSinif.f` aynı şey değildir.

9.3.4 Metod Nesneleri

Genellikle bir metod şu şekilde doğrudan çağırılır:

```
x.f()
```

Bizim örneğimizde bu 'Merhaba' karakter dizisini geri döndürür. Bir metodu doğrudan çağırmak şart değildir: `x.f` bir metod nesnesidir ve saklanıp daha sonra çağırılabilir. Örneğin:

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

sonsuz kadar "Merhaba" yazdırır.

Bir metod çağırıldığında tam olarak ne olur? `x.f()` çağırılırken bir argüman kullanılmadığı halde `f` fonksiyon tanımında bir argüman kullanıldığını fark etmişsinizdir. Argümana ne oldu acaba? Şüphesiz, Python argüman gerektiren bir fonksiyon argümansız çağırıldığında bir istisna oluşturur.

Cevabı belki de tahmin ettiniz: metodlar fonksiyonun ilk argümanı olarak nesneyi alırlar. Örneğimizdeki `x.f()` çağırısı aslında `benimSinif.f(x)` ile aynıdır. Genel olarak, bir metodu n elemanlı bir argüman listesi ile çağırmak, aynı fonksiyonu başına metod nesnesinin eklendiği argüman listesi kullanarak çağırmak ile aynı şeydir.

9.4 Rasgele Açıklamalar

Veri özellikleri aynı isimli metod özelliklerini bastırırlar. Büyük programlardaki zor fark edilen isim çakışması hatalarından kaçınmak için çakışmaları en aza indirecek bir isimlendirme yöntemi kullanmak akıllıca olur. Metod isimleri büyük harf ile başlatılırken, veri isimleri özel bir karakter (alt çizgi gibi) ile başlatılabilir. Metodlar için fiil ve veri yapıları için isim olan kelimeler kullanılabilir.

Veri özelliklerine o nesnenin kullanıcıları ('istemcileri') atıfta bulunabileceği gibi, metodlar da bunlara atıfta bulunabilirler. Başka bir deyişle, sınıflar tamamen soyut veri tipleri oluşturmak için kullanılamazlar. Aslında, Python'da hiçbir şey veri saklamayı zorlamayı mümkün kılmaz.

Kullanıcılar nesnelerin veri özelliklerini dikkatli kullanmalılar; çünkü kullanıcılar metodlar tarafından kullanılan önemli değişkenlere atama yaparak istenmeyen hatalara sebep olabilirler. Kullanıcıların, isim çakışmalarından kaçındıkları sürece, bir nesneye metodlarının geçerliliğini etkilemeden kendi veri özelliklerini ekleyebileceklerine dikkat edin.

Metod içlerinden veri özelliklerine (ya da diğer metodlara!) atıfta bulunmanın kestirme bir yolu yoktur. Bunun aslında metodların okunabilirliğini artırdığını düşünüyorum; bir metoda göz attığınızda yerel değişkenler ile nesne değişkenlerini birbirilerine karıştırma şansı yoktur.

Usul olarak metodların ilk argümanına `self` adı verilir. Bu tamamen usule dayanır; `self` isminin Python için kesinlikle hiç bir özel anlamı yoktur. Bu usule uymazsanız programınız diğer Python programcıları tarafından daha zor okunur ve *sınıf tarayıcısı* (`class browser`) programları da bu usule dayanıyor olabilirler.

Sınıf özelliği olan her fonksiyon o sınıfın nesneleri için bir metod tanımlar. Fonksiyon tanımının sınıf tanımı içerisinde geçmesi şart değildir; fonksiyon nesnesini sınıf içindeki yerel bir değişkene atamak da mümkündür. Örneğin:

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

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

Şimdi `f`, `g` ve `h`'in hepsi `C` sınıfının özellikleri oldular ve aynı anda `C` sınıfının nesnelerinin metodlarıdır (`h` ve `g` birbirinin tamamen aynısıdır). Bu tür uygulamanın genellikle sadece okuyucunun kafasını karıştırmaya yaradığına dikkat edin.

Metodlar `self` argümanının metod olan özelliklerini kullanarak diğer metodları çağırabilirler:

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Metodlar sıradan fonksiyonların yaptığı şekilde global değişkenlere atıfta bulunabilirler. Bir metoda ilişkin global etki alanı (scope) sınıf tanımının bulunduğu modüldür. Sınıfın kendisi asla global etki alanı olarak kullanılmaz. Bir metod içinde global veri kullanmak için ender olarak iyi bir sebep olduğu halde, global etki alanı kullanımının pek çok mantıklı sebebi vardır. Örneğin, global etki alanına yüklenmiş fonksiyon ve modülleri, metodlar ve bunun içinde tanımlanmış diğer fonksiyon ve sınıflar kullanılabilir. Genellikle metodu içeren sınıfın kendisi bu global etki alanı içinde tanımlanmıştır ve bir sonraki kısımda bir metodun kendi sınıfına atıfta bulunmak istemesi için birkaç iyi sebep bulacağız!

9.5 Miras

Tabi ki, miras alma desteği olmayan bir ``sınıf'' özelliği adına layık olmaz. Türetilmiş sınıf tanımının söz dizimi (syntax) aşağıdaki gibidir:

```
class turemisSinifAdi(TemelSinifAdi):
    <deyim-1>
    .
```

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

TemelSinifAdi ismi türetilmiş sınıfın tanımının bulunduğu etki alanında tanımlı olmalıdır. Temel sınıf adı yerine bir ifade kullanmak da mümkündür. Bu temel sınıf adı başka bir modül içinde tanımlı olduğunda yararlıdır:

```
class turemisSinifAdi(modulAdi.TemelSinifAdi):
```

Bir türetilmiş sınıf tanımının işletilmesi bir temel sınıf ile aynıdır. Bir sınıf nesnesi yaratıldığında temel sınıf hatırlanır. Bu özellik atıflarını çözümlemeye kullanılır; atıfta bulunulan özellik o sınıfta yok ise temel sınıfta aranır. Bu kural temel sınıfın kendisi de başka bir sınıftan türetiliyse özyinelemeli (recursive) olarak çağırılır.

Türetilmiş sınıftan nesne oluşturma özel bir tarafı yoktur: turetilmisSinifAdi() o sınıfın yeni bir nesnesini yaratır. Metod atıfları şu şekilde çözülürler: ilgili sınıf özelliği, gerekirse temel sınıflar zinciri taranarak, aranır ve metod atfı geçerli ise bu bir fonksiyon nesnesi verir.

Türetilmiş sınıflar temel sınıflarının metodlarını bastırabilirler. Metodlar aynı nesnenin diğer metodlarının çağırırken özel önceliklere sahip olmadıkları için aynı temel sınıfta tanımlı bir metodu çağırarak temel sınıf metodu bunu bastıran bir türetilmiş sınıf metodunu çağırması olabilir. C++ programcıları için not: Tüm Python metodları sanaldır (virtual).

Türetilmiş sınıftaki bir bastıran metod aslında temel sınıftaki metodun yerini almak yerine onu geliştirmek isteyebilir. Temel sınıf metodunu doğrudan çağırmanın basit bir yolu vardır: "temelSinifAdi.metodAdi(self, argumanlar)". Bu bazen kullanıcılar için de faydalıdır. Bunun sadece temel sınıf global etki alanı içine doğrudan yüklendiyse çalıştığına dikkat edin.

9.5.1 Çoklu Miras

Python çoklu miras almanın kısıtlı bir şeklini destekler. Birçok temel sınıfı olan bir sınıf tanımı aşağıdaki gibidir:

```
class turemisSinifAdi(temel1, temel2, temel3):  
    <ifade-1>  
    .  
    .  
    .  
    <ifade-N>
```

Burada sınıf özelliği atıflarını çözümlemeye kullanılan kuralı açıklamamız gerekiyor. Bir özellik turemisSinifAdi içinde bulunamazsa temel1 içinde sonra temel1'in temel sınıfları içerisinde ve burada da bulunamazsa temel2 içinde aranır vs.

Bazı kişilere temel1'den önce temel2 ve temel3 içinde arama yapmak daha doğal gelir. Bu temel1'in herhangi bir özelliğinin temel1 içinde veya bunun temel sınıflarında tanımlanmış olup olmadığını bilmenizi gerektir ki temel2 içindeki isimler ile çakışmalardan kaçınabilesiniz.

Python kazara oluşan isim çakışmalarına karşı usule güvenir ve bu yüzden çoklu kalıtımın rasgele kullanımının program bakımı yapan kişi için bir kabus olduğu açıktır. Çoklu kalıtımın iyi bilinen bir problemi aynı temel sınıfa sahip iki sınıftan türeme yapmaktır. Bu durumda ne olduğunu anlamak kolaydır; ancak bunun ne işe yarayacağı pek açık değildir.

9.6 Özel Değişkenler

Sınıfa özel tanımlayıcılar (identifier) için sınırlı destek vardır. `__spam` formundaki (en az iki ön alt çizgi ve en fazla bir alt çizgi son eki) belirteç `_sinifadi__spam` şeklini alır. Burada `sinifadi` o anki sınıf adının alt çizgileri atılmış olan halidir. Bu değişiklik tanımlayıcının sözdizimsel konumuna bakılmaksızın yapılır ki bu sınıfa özel değişkenler yaratılabilsin. Değiştirilen tanımlayıcı 255 karakteri aşarsa kırılabilir. Sınıflar dışında veya sınıf adı sadece alt çizgilerden oluşuyorsa kırılma olmaz.

İsim değiştirmenin amacı sınıflara, türemiş sınıflarca tanımlanan nesne değişkenlerini dert etmeden veya sınıf dışındaki nesne değişkenleri ile uğraşmadan, kolayca özel nesne değişkenleri ve metodlar tanımlama yolu sağlamaktır. Değiştirme kurallarının genelde kazaları önlemeye yönelik olduğuna dikkat edin; ancak yine de buna niyet eden kişi özel değişkenlere ulaşip bunları değiştirebilir. Bu bazı özel durumlarda kullanışlı da olabilir.

`exec`, `eval()` veya `evalfile()` fonksiyonlarına geçirilecek kod çağıran sınıf adının o anki sınıf adı olduğunu düşünmez; bu da ``bayta derlenmiş" kod ile sınırlı `global` deyiminin etkisine benzer. Aynı kısıtlama `getattr()`, `setattr()` ve `delattr()` fonksiyonları için ve doğrudan atfedildiğinde `__dict__` için de mevcuttur.

9.7 Öteberi

Bazen Pascal'daki ``record" veya C'deki ``struct" benzeri bir veri tipi birkaç veriyi bir arada toplamak için yararlı olabilir. Boş bir sınıf tanımı bu iş için yeterlidir:

```
class Eleman:
    pass

ali = Eleman() # Boş bir eleman kaydı yarat

# Kaydın alanlarını doldur
ali.isim = 'Ali Veli'
ali.bolum = 'Muhasebe'
ali.maas = 1000000
```

Soyut bir veri tipi bekleyen Python koduna o veri tipinin metodlarını emüle eden bir sınıf geçirilebilir. Örneğin bir dosya nesnesinden bir miktar veriyi formatlayan bir fonksiyonunuz varsa, `read()` ve `readline()` metodları olan ve veriyi bir karakter dizisinden alan bir sınıfı o fonksiyona argüman olarak geçirebilirsiniz.

Gerçekleme metodu nesnelerinin de kendi özellikleri var: metodun gerçekleştirilmesi olduğu nesne `metod.im_self` ve o metoda karşılık gelen fonksiyon `metod.im_func`.

9.7.1 İstisnalar Sınıf Olabilir

Kullanıcı tanımlı istisnalar artık karakter dizisi olmakla sınırlı değiller; sınıf da olabilirler. Bu mekanizmayı kullanarak genişletilebilir istisna hiyerarşileri yaratılabilir.

`raise` deyimi için iki yeni form mevcut:

```
raise Sınıf, gerçekleştirme

raise gerçekleştirme
```

İlk formda `gerçekleme Sınıf`a ait bir gerçekleştirme olmalıdır. İkinci form ise şunun kısaltmasıdır:

```
raise gerçekleştirme.__class__, gerçekleştirme
```

Bir `except` bloğu hem sınıflar hem de karakter dizileri içerebilir. Bir `except` bloğu içindeki sınıf eğer aynı sınıf veya bir temel sınıf ise istisna ile uyumludur. Türetilmiş sınıf içeren bir `except` bloğu temel sınıf ile uyumlu değildir. Örneğin aşağıdaki program B, C, D çıktısını o sırayla verir:

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

Eğer `except` blokları ters sırayla yazılmış olsalardı ("`except B`" başta olacak şekilde) çıktı B, B, B olacaktı; çünkü uyan ilk `except` bloğu tetiklenecekti.

Ele alınmamış sınıf istisnası için bir mesaj yazılacağı zaman, önce sınıf adı yazılır, ardından iki nokta üst üste ve bir boşluk ve son olarak da gerçekleştirilmenin yerleşik `str()` fonksiyonun geri döndürülen karakter dizisi karşılığı yazılır.

10. Ya bundan sonra ?

Bu kılavuzu okumak muhtemelen Python kullanmaya olan ilginizi artırmıştır. Peki şimdi ne yapmalısınız ?

Standart Python dağıtımı çok zengin bir modül kitaplığı ile gelmektedir. Python programları yazarken zamandan büyük tasarruf sağlayacak modüllerin nasıl kullanıldığını öğrenmek için bunlar ile ilgili dokümanlara başvurun.

Resmi Python web sitesi <http://www.python.org/> olup; programlar, dokümanlar ve netteki Python ile ilgili diğer sitelere bağlantılar içerir. Daha az resmi bir site ise <http://starship.python.net/> olup; burada, çoğunda indirilebilir programlar bulunan, Python ile ilgili kişisel web siteleri mevcuttur.

Python ile ilgili sorular ve problem raporları için comp.lang.python haber grubunu veya python-list@python.org e-posta listesini kullanabilirsiniz. Haber grubu ile e-posta listesi birbirine bağlıdır; yani birine göndereceğiniz bir mesaj diğerine de iletilir. Mesaj atmadan önce <http://www.python.org/doc/FAQ.html> adresindeki sıkça sorulan sorular listesini kontrol etmeyi unutmayın. Bu listeyi Python kaynak dağıtımının Misc/ dizininde de bulabilirsiniz. E-posta listesi arşivleri ise <http://www.python.org/pipermail/> adresinde bulunabilir.