Python Kılavuzu

Çeviren:
Dinçer Aydın
<dinceraydin (at) gmx.net>

Yazan:
Guido van Rossum
<python–docs (at) python.org>

Yazan:
Fred L. Drake, Jr.
<python–docs (at) python.org>

22 Nisan 2003

Özet

Bu belge, Python Belgeleri arasında yeralan "Python Tutorial"'ın bir çevirisidir.

Konu Başlıkları

Geçmiş

2.2.2	14 Ekim 2002	Özgün belge: PythonLabs, Çeviri: DA

Çevirinin özgün sürümünü http://www.geocities.com/dinceraydin/ adresinde bulabilirsiniz.

2.1.1	11 Kasım 2001	Özgün belge: PythonLabs, Çeviri: DA

http://www.geocities.com/dinceraydin/python/indextr.html adresindeki Windows sürümünden çevrilmiş Python Kılavuzu'nun Linux için uyarlamasıdır.

1. Giriş

Python kolay öğrenilen güçlü bir programlama dilidir. Verimli yüksek seviyeli veri türlerine sahiptir ve nesne tabanlı programlamaya yaklaşımı basit ve etkilidir. Python'un şık sözdizimi, dinamik veri türleri 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 çalıştırılabilir paket olarak Python Web sitesi[B4]nden alınabilir ve dağıtılabilir. Aynı sitede farklı Python dağıtımları, modüller, programlar ve belgeler 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 türleri ve işlevler 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 Python'un bütün özeliklerini ya da yaygın olarak kullanılan her özelliğini açıklamak amacında 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.

2. İştahınızı Kabartalım

Eğer büyük bir kabuk betiği 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 değecek 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 türlerine sahiptir (sözlükler ve listeler gibi). Daha genel veri türleri 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 kolleksiyonu 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 ilintileme gerekmemektedir. Yorumlayıcıyı etkileşimli olarak da kullanabilirsiniz; böylece dilin özelliklerini kolayca deneyebilir, hızlı bir şekilde küçük programlar yazabilir, ya da aşağıdan–yukarı program geliştirme sırasında işlevlerinizi 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 kısadırlar:

- 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 bildirimlerinin yapılması gerekmez.

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

Python programlama dili adını korkunç bir sürüngenden değil, 'Monty Python's Flying Circus" adlı bir BBC komedi dizisinden almıştır.

2.1. 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 öğreteceğiz. Bu çok basit bir şey; ancak daha sonraki bölümlerin anlaşılması için önemli.

Kılavuzun devamında basit ifadeler, deyimler ve veri türleri ile başlayıp, işlevler ve modüllerden kullanıcı tanımlı sınıflar gibi gelişmiş konulara kadar Python'un çeşitli özellikleri örnekler ile anlatılmaktadır.

3. Yorumlayıcının Kullanımı

3.1. Yorumlayıcının Çalıştırılması

Python yorumlayıcısını Linux Konsolunda ya da xterm'de `python` komutunu girerek çalıştırabilirsiniz. Python yorumlayıcısının konsolda çalışanı olduğu gibi X Pencere Sisteminde çalışanı da vardır. X pencere sisteminde çalışan yorumlayıcı `idle` komutu ile çalıştırılabilir. Menüleriyle kolay kullanılabilen bir Python düzenleyicisidir ve konsolda çalışan `python` gibi "Python Kabuğu" penceresinde doğrudan komut icra edilebilir.

`python` çalıştırıldığında yorumlayıcı konsolda etkileşimli kipte çalışır. Yorumlayıcı çalıştırılacak olan Python programının dosya adı parametre şeklinde verilerek de kullanılabilir. Örneğin `python deneme.py` gibi.

3.2. Etkileşimli Kip

Komutların yorumlayıcıya satır satır girildiği duruma etkileşimli kip denir. Bu kipte birincil komut satırı `>>>` şeklindeyken ikincil komut satırı `...` şeklinde görülür. Örneğin:

```
bash-2.05$ python
Python 2.2.2 (#1, Feb 24 2003, 19:13:11)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

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

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

Yorumlayıcıdan çıkmak için `ctrl+D` tuşlayın. Etkileşimli kipte çalıştırılan programlarda bir hata bulunması halinde yorumlayıcı bir hata mesajı basar ve birincil komut satırına döner:

```
>>> if dunya_duzdur:
... print "Düşmemeye dikkat et!"
  File "<stdin>", line 2
    print "Düşmemeye dikkat et!"
        ^
IndentationError: expected an indented block
>>>
```

Burada 2. satırın girintili yazılması gerektiği hatırlatılıyor.

Ayrıca etkileşimli kipte ya da Python programlarınızı yazarken komut satırı yerine daha rahat bir ortamda calışmak isteyebilirsiniz. Bunun için Python ile birlikte gelen `idle`'ı deneyebilirsiniz. `idle` Python ile yazılmış bir uygulama. Bazı `idle` komutları şöyledir:

- **`Alt+p`** Yazdığınız bir önceki komutu getirir.
- **`Alt+n`** Yazdığınız bir sonraki komutu getirir.
- **`enter`** Yazdığınız komutları çalıştırır.

3.3. Python Betikleri

Kabuk betiklerine benzer şekilde Python betikleri de, dosyanın ilk satırına

```
#! /usr/bin/env python
```

yazılarak ve dosyaya çalıştırma izni verilerek (`chmod +x dosya.py`) doğrudan kabuktan çalıştırılabilirler.

3.4. Hataların Yakalanması

Bir hata oluştuğunda yorumlayıcı bir hata mesajı ve hatalı noktaya gelmeden önce işletilen işlev çağrılarının listesini basar.

4. Python'a Giriş

Aşağıdaki örneklerde giriş ve çıkış, komut satırının varlığına veya yokluğuna (`>>>` veya `...`) 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 dizge içinde bulunan "#" işareti bir açıklama başlatmaz. Bu sadece bir "#" karakteridir. Örnekler:

```
# Bu bir açıklama.
```

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

4.1. Python'u Hesap Makinesi Olarak Kullanmak

4.1.1. Sayılar

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

Yorumlayıcı basit bir hesap makinesi olarak iş görürebilir: istediğiniz herhangi bir ifadeyi yazın ve yorumlayıcı sonucu verecektir. İşleçler (+,–,*,/) çoğu programlama dillerindekine benzer çalışır (Pascal ve C de olduğu gibi mesela). İfadeleri 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 (=) bir değişkene değer atamak için kullanılır. Atamanın değeri çıkışta yazılmaz:

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

Bir değer aynı anda birden fazla değişkene atanabilir:

```
>>> x = y = z = 5 # x, y ve z beş değerini alır
>>> x
5
>>> y
5
>>> z
5
```

Tam gerçel sayı desteği vardır. Farklı türdeki değerlerin olduğu işlemlerde sonuç gerçel sayıya dönüştürülür:

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

Karmaşık sayılar da desteklenmektedir. Sayıların sanal kısımları `j` veya `J` soneki ile yazılır. Gerçek kısmı sıfır olmayan karmaşık sayılar `(gerçek + sanalj)` şeklinde yazılırlar ya da `complex(gerçek, sanal)` işlevi ile kullanılırlar.

```
>>> 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 gerçel 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
```

Tamsayı veya gerçel sayıya dönüştürme işlevleri (`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 değişken yaratır.

4.1.2. Dizgeler

Sayılara ek olarak, Python dizgeler üzerinde de işlemler yapabilir. Dizgeler farklı şekillerde ifade edilebilir. Tek veya çift tırnak işareti içine alınabilirler:

```
 >>> 'dizge'
'dizge'
>>> "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.'
```

Dizgeler 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 dizge.\n\
...     Bu 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 dizge.
        Bu satırın başındaki girintinin belirgin olduğuna dikkat edin

>>>
```

Karakter dizisini `r` ile imleyerek *ham dizge* yapacak olursak, `\n\` karakterleri yorumlanmaz, dizgenin bir parçası haline gelirler. Örneğin:

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

print merhaba
```

şu çıktıyı verir:

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

Karakter dizileri bir çift üçlü tırnak içinde de gösterilebilirler: Dizgeler 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 dizge işlemlerinin sonucunu girişine yazıldığı şekli ile çıkışa yazar. Dizgeler + işleçi ile birleştirilip, * ile tekrarlanabilirler:

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

Yan yana iki dizge değişkeni otomatik olarak birleştirilir yani yukarıdaki örnekteki ilk satır `kelime = 'Alo' 'ooo'` şeklinde de yazılabilirdi. Bu sadece iki dizge değişkeni ile olur. Keyfî dizgeler 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 dizgeler indislenebilirler. Dizgenin ilk karakterinin indisi sıfırdır. Python'da ayrı bir karakter veri türü yoktur. Bir karakter tek karakterli bir dizgedir. Icon dilinde (70'li yıllarda Ralph ve Marge Griswold'ün geliştirdiği Pascal benzeri bir SNOBOL4 türevi) olduğu gibi dizgelerin bölümleri dilim gösterimi `[ : ]` ile ifade edilebilirler.

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

C dilinden farklı olarak, Python'da dizgeler değiştirilemezler. Bir dizgenin 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 dizge 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 dizgenin 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:]
'Aloooo'
>>> kelime[:3] + kelime[3:]
'Aloooo'
```

Çok büyük veya küçük dilim aralıkları akıllıca ele alınır. Bitiş değeri büyük ise bunun boyu dizgenin boyuna eşit olur. Başlangıç değeri bitişten büyük ise boş bir dizge 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 soldan 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
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 işlev `len()` bir dizgenin boyunu verir.

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

4.1.3. Listeler

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

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

Listeler de dizgeler 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 dizgelerin 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', 'xyzzy']
>>> a
[123, 'qwerty', 'xyzzy', 1234]
>>> a[:0] = a # Listenin kopyasını listenin başına ekle
>>> a
[123, 'qwerty', 'xyzzy', 1234, 123, 'qwerty', 'xyzzy', 1234]
```

Yerleşik işlev `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!

4.2. Programlamaya Doğru İlk Adımlar

Tabii 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 dizge veya liste de olabilir. Boyu sıfır olmayan her şey doğru iken, boş listeler, dizgeler, 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 bloğu girinitli 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). Grintili bir öbek içindeki her satırın aynı girinti miktarına sahip olması gerektiğine dikkat ediniz. Girintiler için boşluk veya sekme 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 çıktı 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
```

5. Akış Denetimi

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

5.1. `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` deyimi olabilir, ve `else` deyimi seçimliktir. `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.

5.2. `for` Deyimi

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

```
>>> # Bazı dizgelerin 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']
```

5.3. `range()` işlevi

Eğer bir sayı sırası üzerinde tekrarlamalar yapmak isterseniz, belirli bir sıraya göre üretilen sayılardan oluşan bir liste oluşturan `range()` yerleşik işlevini 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()` işlevlerini 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
```

5.4. Döngülerde `break`, `continue` ve `else` Deyimleri

`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` bloğu 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
```

5.5. `pass` Deyimi

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

```
>>> while 1:
...       pass # klavyeden CTRL+C ile kesilene kadar sürer
...
```

5.6. İşlev Tanımlama

Herhangi bir değere kadar Fibonacci serisi yazan bir işlev yazalım:

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

`def` anahtar kelimesi bir işlev tanımını başlatır. Bu deyimden sonra bir işlev adı ve parantez içinde parametreler yazılır. İşlevin gövdesini oluşturan program satırları sonraki satırdan itibaren girintili olarak yazılırlar. İşlev gövdesinin ilk satırı bir dizge de olabilir; bu dizge işlevin belgelenmesinde kullanılır (docstring).

İşlevlerin belgelenmesinde kullanılan dizgeleri (docstring) otomatik olarak çevrim içi ya da basılı belgeler oluşturmak için kullanan yazılımlar vardır. Ayrıca bazı geliştirme ortamları bunları program yazarken kolaylık

sağlaması için etkileşimli olarak programcıya sunarlar. Yazdığınız işlevlere bunları eklemeyi bir alışkanlık haline getirmeniz faydalı olur.

Bir işlevin çağrılması (çalıştırılması) bu işlevdeki yerel değişkenlerin olduğu bir simge tablosu oluşturur. İşlev içerisinde bütün değer atama işlemlerinde değerler yerel simge tablosuna kaydedilir. Bir değişkene başvuru durumunda ise önce yerel (local), sonra genel (global) ve en son yerleşik (built–in) simge tablosunda arama yapılır. Bu yüzden genel değişkenlere doğrudan değer atama yapılamaz (eğer `global` ifadesi içinde kullanılmamışlar ise); ancak bunlara başvuru yapılabilir (reference).

İşlev çağırıldığında işlevin parametreleri yerel simge tablosuna eklenirler; yani parametreler işleve değeri ile çağrı (call by value) kullanılarak iletilirler (yani parametreye yapılan değişiklikler yereldir, çağıran işlevdeki argümanlarda bir değişme olmaz). [1]

Bir işlev başka bir işlevi çağırdığında bu çağrı için yeni bir yerel simge tablosu oluşturulur.

Bir işlev tanımı işlev adının yürürlükte olan simge tablosuna eklenmesine sebep olur. İşlevin adı yorurumlayıcı tarafından kullanıcı tanımlı işlev veri türü olarak tanınır. Bu değer başka bir isime atanabilir ve bu da bir işlev olarak kullanılabilir. Bu genel bir isim değiştirme yolu olabilir:

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

`fib`'in bir işlev olmayıp bir yordam (procedure) olduğunu düşünebilirsiniz. Python'da yordamlar, çağıran işleve değer geri döndürmeyen işlevlerdir. Aslında yordamlar da bir değer geri döndürürler, ama bu sıkıcı bir konudur. Bu değere `None` denir ve yerleşik 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 işlev 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)    # işlevi 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 işlevden değer geri döndürür. Parametresi olmayan bir `return` deyimi `None` geri döndürür. Sona eren bir yordam (procedure) da `None` geri döndürür.
- `sonuc.append(b)` ifadesi sonuç liste nesnesinin bir yöntemini çağırmaktadır. Bir yöntem bir nesneye 'ait olan' ve `nesne.yöntemAdı` şeklinde adlandırılan bir işlevdir. `nesne.yöntemAdı` ifadesinde nesne herhangi bir nesne (bir ifade de olabilir) ve `yöntemAdı` da nesnenin türüne bağlı bir yöntemdir. Farklı veri türleri farklı yöntemlere sahiptirler. Farklı veri türlerinin ayni isimli yöntemleri olabilir. Sonraki

bölümlerde anlatılacağı gibi, kendi veri türlerinizi ve yöntemlerinizi oluşturmanız mümkündür. Yukarıdaki örnekte görülen `append()` yöntemi liste nesneleri için tanımlıdır ve bir listenin sonuna yeni elemanlar ekler. Bu örnekte bu `sonuc = sonuc + [b]` ifadesinin yaptığını yapar; ancak daha verimlidir.

5.7. İşlev Tanımları Üzerine Daha Fazla Bilgi

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

5.7.1. Argüman Değerlerini Önceden Belirleme

İşlev argümanlarına öntanımlı değerler atamak da mümkündür. Böylece çağıran işlev bu argümanları sağlamazsa bunlar önceden belirlenmiş öntanımlı 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 işlev `onay_al('Programdan çıkmak istiyor musunuz?')` ya da `onay_al('Dosyayı silmek istiyor musunuz?', 2)` şeklinde çağırılabilir.

İşlevin öntanımlı parametreleri işlevin tanımlandığı anda, o an yürürlükte olan etki alanı (scope) içinde değerlendirilirler. Yani:

```
i = 7
def f(arg = i):
    print arg

i = 6
f()
7
```

Uyarı

İşlevin öntanımlı parametreleri sadece bir defa değerlendirilirler. Bu durum parametrenin liste gibi değiştirilebilir bir nesne olduğu durumlarda farklılık yaratır. Örneğin aşağıdaki işlev 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 öntanımlı parametre değerlerinin birbirini izleyen çağrılarla paylaşılmasını istemiyorsanız yukarıdaki işlevi şu şekilde yazabilirsiniz:

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

5.7.2. Anahtar Kelime Argümanlar

İşlevler `anahtar kelime = değer` şeklindeki anahtar kelimelerle de çağırılabililer. Örneğin şu işlev:

```
def otomobil(yakit,  hareket=' uçar', model='Anadol'):
    print "Eğer", yakit, "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 kelimeden sonra gelen
                                    # anahtar kelime olmayan argüman
otomobil('mazot', yakit = '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 işlevin parametrelerinden seçilir. Parametrenin öntanımlı 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 işlev tanımındaki son parametre `**isim` şeklinde ise bu parametre adları herhangi bir parametre olmayan anahtar kelime şeklindeki argümanların bulunduğu bir sözlük olur. Bu `*isim` (bu konu *Sözlükler (Çağrışımlı Listeler)* (sayfa: 26) bölümünde anlatılacaktır.) şeklindeki bir parametre ile de kullanılabilir, ki bu parametre listesi içinde bulunmayan konuma bağlı argümanları içeren bir demet (daha sonra *Demetler (tuples)* (sayfa: 25) bölümünde incelenecek bir veri türüdür) olur. `*isim` parametresi `**isim` parametresinden önce gelmelidir. Buna örnek işlev:

```
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 çağrılabilir:

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

ve doğal olarak şu çıktıyı verir:

```
-- martı eti var mi ?
-- 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()` yönteminin çağırıldığına dikkat edin; bu yapılmaz ise argümanların hangi sıra ile yazılacağı tanımlanmamış olur.

5.7.3. Keyfî Argüman Listeleri

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

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

5.7.4. Lambda Biçemli İşlevler

Yoğun istek üzerine işlevsel dillerde ve Lisp'te bulunan bazı özellikler Python'a eklenmiştir. `lambda` anahtar kelimesi ile küçük anonim işlevler yazılabilir. İşte iki argümanının toplamını geri döndüren bir işlev: `lambda a, b: a+b`.

Lambda işlevleri bir işlev nesnesine ihtiyaç duyulan her yerde kullanılabilirler. Sözdizim (syntax) açısından bunlar tek bir ifade ile sınırlandırılmışlardır. Anlambilim (semantics) açısından ise normal işlev tanımlamasına getirilen bir sözdizim güzelliğidir. İç içe tanımlanmış işlevlerde olduğu gibi lambda işlevleri de kendilerini kapsayan etki alanindaki değişkenlere erişebilirler:

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

5.7.5. Belgelendirme Dizgeleri

Belgelendirmede kullanılan dizgelerin 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 belgelendirme dizgesinde 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 çözümleyicisi (parser) çok satırlı dizgelerdeki girintileri yok etmez; yani belgeleri işleyen programlar gerekirse bunları atabilirler. İlk satırdan sonra gelen ve boş olmayan ilk satırdaki girinti miktarı belgelendirme dizgesinin 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 sekmelerin genişletilmesinden (1 sekme= 8 boşluk) sonra sınanmalıdır.

İşte çok satırlı bir belgelendirme dizgesi örneği:

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

   Başka birşey yapmaz. Gerçekten!
```

6. Veri Yapıları

Bu bölümde öğrendiğiniz bazı şeyler daha ayrıntılı açıklanmakta ve bazı yeni konulara da değinilmektedir.

6.1. Listeler Üzerine Daha Fazla Bilgi

Liste veri türünün birkaç yöntemi daha var. İşte liste nesnelerinin bütün yöntemleri:

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

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

insert(i, x)
Belirtilen konuma bir eleman yerleştirir. İlk argüman elemanın yerleştirileceği indistir. `a.insert(0, x)` ifadesi `x`'i listenin başına yerleştirir, 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 öğe yok ise bu hatadır.

pop([i])

Belirtilen konumdaki elemanı listeden siler ve bunu geri döndürür. Eğer bir indis belirtilmediyse, `a.pop()` listedeki son elemanı siler ve geri döndürür. (`i` etrafındaki köşeli ayraçlar bu parametreanin seçimlik olduğunu belirtir. Bu yazım biçimini 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 yöntemlerinin 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]
```

6.1.1. Listelerin Yığın Olarak Kullanılması

Liste yöntemleri 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]
```

6.1.2. 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']
```

6.1.3. İşlevsel Yazılım Geliştirme Araçları

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

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

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

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

`reduce(işlev, sıra)` ifadesi tek bir değer geri döndürür. Bu değer şöyle elde edilir: iki argümanlı işleve 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 işleve verilir ve bu işlem bütün elemanlar için tekrarlanır. Örneğin 1'den 10'a kadar olanlar 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 işleve ilk olarak başlangıç değeri ve sıranın ilk elemanına uygulanır ve diğer elemanlar ile devam eder. Örnek:

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

6.1.4. Liste Üreteçleri

Liste üreteçleri `map()`, `filter()` ve/veya `lambda` işlevlerini kullanmadan liste yaratmanın kısa bir yoludur. Bu yolla yaratılan liste tanımı genellikle daha kolay anlaşılır olur. Bir liste üreteci 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
>>
```

6.2. `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` deyimi tamamen silmek için de kullanılabilir:

```
>>> del a
```

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

6.3. Demetler (tuples)

Listelerin ve dizgelerin indisleme ve dilimleme gibi pek çok ortak özellikleri olduğunu grdük. Bunlar sıra şeklindeki iki veri türüdürler. Python gelişmekte olan bir dil; diğer sıra şeklindeki veri türleri de Python'a eklenebilir. Demet de başka bir sıra şekilli standart veri türüdür.

Bir demet virgül ile ayrılmış bir kaç 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))
```

Gördüğünüz gibi çıktıda demetler daima parantez içinde görünürler; ki iç içe geçmiş demetler belli olsun. Demetler parantezli veya parantezsiz olarak yazılabilirler; ancak parantezler genelikle 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 dizgeler gibi değerleri değiştirilemez veri türleridir; 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', # <--satır 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 dizge
>>> a,b,c = paket
>>> a
'x'
>>> b
'y'
>>> c
'z'
```

6.4. Sözlükler (Çağrışımlı Listeler)

Python'da bulunan bir diğer faydalı veri türü 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 türdeki herhangi bir veri türünde olabilir. Sayılar ve dizgeler her zaman anahtar olabilirler. Demetler de sayılar, dizgeler 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()` yöntemleri, 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 değerin bir anahtar ile saklanması ve anahtar verildiğinde değerin 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()` yöntemi listedeki bütün anahtarların listesini rasgele sıralı olarak geri döndürür (sıralamak isterseniz listenin `sort()` yönteminden faydalanabilirsiniz). Bir anahtarın sözlükte olup olmadığını görmek için sözlüğün `has_key()` yöntemi 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()` işlevi anahtar–değer çiftlerinden oluşan demetlerden sözlükler üretir. Çiftlerin bir kalıba uyduğu durumlarda, liste üreteçleri 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 üreteci kullanarak
{2: 4, 4: 16, 6: 36}
```

6.5. Döngü Teknikleri

Sözlükler üzerinde döngüler kurarken o anki değer `items()` yöntemi 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()` işlevini 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()` işlevi 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 adın ne? Adnan.
Senin görevin ne? Uyumak.
Senin favori rengin ne? Mavi.
```

6.6. Koşullu İfadeler Üzerine Daha Fazla Bilgi

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

`in` ve `not` kıyaslama işleçleri bir değerin 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 işleçleri aynı önceliğe sahiptirler ve bu sayısal işleçlerinkinden düşüktür.

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

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

`and` ve `or` mantıksal işleçlerine kısa devre işleç 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 işleçi `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).

6.7. Listeler, Dizgeler ve Demetler Arasında Kıyaslama

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 kıyaslanmasının yasal olduğuna dikkat edin. Türler alfabetik sırayla dizilmiştir (ingilizce isimlerine göre). Yani: liste < dizge < demet (list < string < tuple). [2]

7. Modüller

Python yorumlayıcısını kapatıp tekrar açarsanız yaptığınız tanımlar (işlevler ve değişkenler) kaybolur. Uzunca bir program yazmak isterseniz bunun için programınızı bir metin düzenleyici ile hazırlayıp yazdığınız dosyayı yorumlayıcı girişi olarak kullanırsanız daha iyi olur. Bu işleme betik yazmak denir. Programınız uzadıkça bunu daha kolay idare etmek için birkaç dosyaya bölmek isteyebilirsiniz. Yazdığınız bir işlev 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 dizge) o modülün adını verir. Örneğin, favori metin düzenleyiciniz ile `fibo.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 işlev tanımlarını yürürlükte olan simge tablosuna eklemez; sadece modül adı fibo tabloya eklenir. İşlevlere 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 işlevi 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
```

7.1. Modüller Üzerine Daha Fazla Bilgi

İşlev tanımlarının yanısıra modül içinde çalıştırılabilir ifadeler de olabilir. Bu ifadeler modülün ilk kullanıma hazırlanması için kullanılabilirler ve sadece modülün ilk yüklenişinde çalışırlar. [3]

Her modülün o modül içindeki bütün işlevler tarafından global simge tablosu olarak kullanılan kendi simge tablosu vardır. Bu özellik sayesinde modülü yazan kişi rahatlıkla modül içnde global değişkenler kullanabilir. Modülü kullanan diğer kişilerin global değişkenleri ile isim çakışması olmaz. Modül 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 ekleniriler.

`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 fibo 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.

7.1.1. 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). Aranan dosya bulunmazsa arama, kuruluma bağlı başka bir yolda da aranabilir. Genelde bu `/usr/local/lib/python` dizinidir.

Aslında modüller `sys.path` değişkeninde bulunan dizin listesinde aranırlar. Bu değişken değerini betiğin alıştırıldığı dizin, `PYTHONPATH` ve kuruluma bağlı diğer dizinlerden alır. `sys.path` değişkeni sayesinde Python programları modül arama yolunu değiştirebilirler.

7.1.2. "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 modul, `spam` modülünün ikilik derlenmiş

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. Eniyileştircinin (optimizer) şu anda pek bir yararı olmuyor; sadece assert deyimlerini siliyor. -O parametresi kullanıldığında tüm ikilik kod eniyileştirilir, .pyc dosyaları göz ardı edilir ve .py dosyaları eniyileştirilmiş ikilik kod olarak derlenir.
- Yorumlayıcıya iki tane -O parametresi (-OO) vermek derleyicinin bazı ender durumlarda doğru çalışmayan programlara neden olan eniyileştirmeler yapmasına neden olur. Şu anda sadece __doc__ dizgeleri 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 ikilik kodu asla bir .pyc ya da .pyo dosyasına yazılmaz. Bu yüzden betiğin başlama süresini kısaltmak için bunun bir kısmı bir modüle aktarılarak 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.

7.2. 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 çağrıları 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 dizgeleri belirlerler:

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

Bu iki değişken yorumlayıcı sadece etkileşimli kipte iken tanımlıdırlar.

`sys.path` değişkeni de yorumlayıcının modül arama yolunu belirler. Bu değerini ortam değişkeni `PYTHONPATH` belirler. `PYTHONPATH` değişkenine değer atanmadıysa `sys.path` öntanımlı değerini alır. Bunun değeri listelere uygulana işlemler ile değiştirilebilir:

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

7.3. `dir()` İşlevi

Yerleşik işlev `dir()` bir modülün hangi isimleri tanımladığını bulmak içik kullanılır. Bu işlev dizgelerden 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()` işlevi 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, işlev vs. gibi her tür ismini listelediğine dikkat ediniz.

`dir()` yerleşik işlev 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']
```

7.4. 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 birbirlerinin modül isimlerinin çakışması tehlikesinden kurtarır.

Ses dosyaları ve ses verisi üzerinde işlem yapacak bir modül kolleksiyonu (bir "paket") geliştirmek istediğinizi düşünelim. Farklı biçemlerdeki 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 biçemleri 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 istemdışı 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 istenen işlev ya da değişkeni doğrudan yüklemektir:

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

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

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

`from PAKET import İSİM` kullanılırken İSİM bir alt modül, alt paket ya da paket içinde tanımlı bir işlev, 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 ÖĞE.ALTÖĞE.ALTALTÖĞE` ifadesinde ise son ismin dışındaki isimler paket olmalıdır. Son isim bir modül veya paket olabilir; ancak bir önceki ismin içinde tanımlanan bir işlev ya da değişken olamaz.

7.4.1. Bir paketten * yüklemek

Kullanıcı `from Sound.Effects import *` yazdığında ne olur? Dosya sistemine ulaşılıp paketin içinde hangi alt paketlerin olduğunun 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 PAKET import *` ifadesi kullanıldığında yüklenecek modül isimlerinin listesi olarak kullanılır. Paketin yeni bir sürümü hazırlandığında bu listenin uygun şekilde güncellenmesi paket yazarının sorumluğundadır. 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 yüklemez; 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 GEREKLİ_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.

7.4.2. Birbirlerini Yükleyen Modüller

Alt modüller çoğu kez birbirlerine atıfta bulunurlar. Örneğin surround modülü echo modülüne ihtiyaç duyabilir. Aslında bu türden atıflar öyle yaygındır ki `import` deyimi standart modül arama yoluna bakmadan önce çağrıldığı 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 yapmaya 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.

8. 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.

8.1. Daha Güzel Çıkış Biçemi

Buraya kadar değerleri yazdırmanın iki yolunu gördük: deyim ifadeleri ve print deyimi. Üçüncü bir yol da dosya nesnelerinin `write()` yöntemidir. Standart çıktıya `sys.stdout` şeklinde atıfta bulunulabilir.

Çoğu zaman boşluklar ile birbirinden ayrılmış değerlerden daha iyi biçimlendirilimiş bir çıktıya ihtiyaç duyulur. Çıktınızı biçimlendirmenin iki yolu var. İlki bütün dizge işlemlerini dilimleme ve birleştirme ile yapıp istediğiniz herhangi bir biçimi elde etmek. `string` standart modülü dizgelerin istenen sütun genişliğine kadar boşluklar ile doldurulmasını sağlayan, daha sonra değineceğimiz, bazı faydalı işlevlere sahiptir. İkinci yol ise sol argümanı bir dizge olan `%` işlecini kullanmaktır. `%` işleci sol argümanını sağdaki argümanına uygulanacak `sprintf()` tarzı biçim dizgesi olarak yorumlar ve biçimleme işleminden sonra bir dizge geri döndürür.

Sayısal değerleri dizgeye çevirmek için ise değer `repr()` veya `str()` işlevine geçirilebilir ya da ters tırnak işareti (**`**) içine alınabilir (`repr()` ile aynı etkiye sahiptir).

`str()` işlevi değerlerin insan tarafından okunabilir gösterimini geri döndürürken, `repr()` işlevi 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()` işlevi `repr()` ile aynı değeri döndürür. Rakamlar, listeler ve sözlükler gibi yapılar ile daha pek çok değer için her iki işlev de aynı sonucu verir. Dizgeler ve gerçel sayılar 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 dizgelerin başını boşluklar ile doldurup bunları sağ tarafa dayayan `string.rjust()` işlevini kullanmaktadır. Buna benzer `string.ljust()` ve `string.center()` işlevleri de vardır. Bunlar bir şey yazdırmaz; sadece yeni bir dizge geri döndürürler. Verilen dizge 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()` işlevi ise rakamlar içeren dizgelerin başını sıfırlar ile doldurur. Bu işlev 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'
```

`%` işleçi şu şekilde kullanılır:

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

Dizgenin içinde birden fazla biçem varsa sağ terim 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 türünün geçirilmesi gerekir; bu yapılamaz ise bir istisna oluşur. `%s` biçiminin kullanımı daha rahattır; verilen argüman dizge değilse yerleşik işlev `str()` ile dizgeye 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 dizgeniz 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 işlev `vars()` ile beraber kullanıldığında faydalı olur.

8.2. Dosya Okuma ve Yazma

`open()` işlevi bir dosya nesnesi geri döndürür ve genellikle iki argüman ile kullanılır: `open(dosya_adı, kip)`

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

İlk argüman dosya adını içeren bir dizgedir. İkincisi ise dosyanın nasıl kullanılacağını belirten karakterlerden oluşur. Erişim kipi dosyadan sadece okuma yapılacak ise `'r'`, sadece yazma için `'w'` (aynı isimli bir dosya zaten var ise üzerine yazılır) ve dosyanın sonuna eklemeler yapmak için `'a'` olur. `'r+'` kipi dosyayı hem okuma hem de yazma yapmak için açar. `kip` argümanı seçimliktir; kullanılamaması halinde `'r'` olduğu varsayılır.

Windows ve Macintosh üzerinde kipe eklenen `'b'` harfi dosyayı ikilik kipte açar; yani `'rb'`, `'wb'` ve `'r+b'` gibi kipler de vardır. Windows metin ve ikilik dosyaları arasında ayrım yapmaktadır; metin dosyalarında okuma veya yazma işlemlerinde satır sonu karakterleri otomatik olarak biraz değişir. Bu görünmez değişiklik ASCII metin dosyaları için iyidir; anacak JPEG resimler veya .EXE dosyalar gibi iklik verileri bozar.

8.2.1. Dosya Nesnelerinin Yöntemleri

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

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

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

`f.readline()` dosyadan tek bir satır okur. Satırın sonundaki satırsonu karakteri (`\n`) korunur; ancak dosya bir satırsonu karakteri ile bitmiyor ise son satırda bu karakter silinir. Bu özellik geri döndürülen değerin birden fazla anlama gelmesini engeller; `f.readline()` boş bir dizge geri döndürdüğünde dosyanın sonuna ulaşılırken 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(dizge)` yöntemi `dizge` 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 başvuru 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 başvuru 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 `f.close()` yöntemini çağırarak dosyayı kapatabilir ve dosyanın işgal ettiği sistem kaynaklarını serbest bırakabilirsiziz. `f.close()` çağrıldıktan 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 yöntemleri de vardır.

8.2.2. `pickle` Modülü

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

Programcıları 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!) dizge 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 belgelerinden 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 türlerinin doğru olarak dönüştürülebilir ve geri alınabilir olmasına özen gösterirler.

9. 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.

9.1. Sözdizim Hataları

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

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

Ayrıştırıcı 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.

9.2. 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: spam
>>> '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 dizge meydana gelen istisnanın yerleşik ismidir. Bu bütün yerleşik istisnalar için geçerlidir; ancak kullanıcı tanımlı istisnalar için böyle olmayabilir. Standart istisna isimleri yerleşik belirteçlerdir; ayrılmış anahtar kelimeler değil.

Satırın devamı istisna türüne bağlı ayrıntılardan 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 belgelerden yararlanılabilir.

9.3. 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. Kullanıcın 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. 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 çağıranın 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 türü istisnanın türüne bağlıdır. Argümanı olan istisna türleri 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 ('ayrıntı') basılır.

İstisna işleyiciler (exception handlers) sadece `try` bloğu içinde meydana gelen istisnaları değil `try` bloğundan çağırılan (dolaylı olarak bile olsa) işlevlerdeki 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
```

9.4. İstisna Oluşturma

`raise` deyimi 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 biçimi 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
```

9.5. 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 işlevlerde meydana gelen hataları rapor etmek için kendi istisnalarını tanımlarlar.

Sınıflar üzerine daha fazla bilgi sonraki bölümünde sunulacaktır.

9.6. Son İşlemlerin 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.

10. Sınıflar

Python'un sınıf mekanizması sınıfları çok az bir sözdizimi ve kavramsallıkla dile ekler. Sınıf mekanizması C++ ile Modula–3 karışımıdır denebilir. Python'da sınıflar modüllerdeki gibi kullanıcı ile tanımlar arasına somut engeller koymaz. Python'da sınıflara gücünü veren başlıca özelliklerini şöyle sıralayabiliriz: sınıflar çok sayıda sınıfı miras alabilir. Bu türetilmiş sınıflar, atalarının yöntemlerini değiştirerek kullanabilirler; ataları ile aynı isme sahip yöntemlerle, atasındaki yöntemleri çağırabilirler. Nesneler özel verilere sahip olabilir.

C++ terminolojisinde, tüm sınıf üyeleri (veri üyeleri dahil) `public` ve tüm üye işlevler `virtual`'dir. Özel bir kurucu ya da yıkıcı yoktur. Modula–3'deki gibi, bir nesnenin yöntemlerinden üyelerine başvuru için bir kestirme yol yoktur: bir üye yöntem, dolaylı olarak çağrı ile sağlanan ve doğrudan nesneyi ifade eden bir ilk argüman ile bildirilir. Smalltalk'daki gibi, sınıfların kendileri nesnelerdir (sınıfların ve nesnelerin kavramsal özelliklerine uymasa da böyledir). Python'da tüm veri türleri birer nesnedir. Bu onların yeniden isimlendirilebilmesi ve başka veri türlerine dahil edilmesi için bir yol sağlar. Ancak kullanıcı bunları genişletmek için temel sınıf olarak kullanamaz. Ayrıca Modula–3'de olmayan ama C++'da olan, özel yazımlı bir çok yerleşik işleç (aritmetik işleçler, alt indisleme, vs), sınıf gerçeklemeleri için yeniden tanımlanabilir.

10.1. Terminoloji hakkında...

Sınıflar hakkında konuşurken, evrensel olarak kabul edilmiş bir terminolojinin olmayışından dolayı arasıra Smalltalk ve C++ terimlerini kullanacağım. (Aslında, nesne yönelimi açısından Python'a daha çok benzeyen Modula–3 dili terimlerini kullanırdım; fakat bu dilden haberdar olan okuyucuların azınlıkta olduğunu sanıyorum.)

Ayrıca nesne yönelim kavramını bilenleri bir terminoloji tuzağına karşı uyarmalıyım: "Nesne" sözcüğü, Python'da "bir sınıfın gerçeklenmesi" anlamına gelmez. Smalltalk'da olmayan ancak C++ ve Modula–3'dekine benzer şekilde Python'daki veri türlerinin hepsi birer sınıf değildir: tamsayılar ve listeler gibi yerleşik veri türleri ile dosyalar sınıf değildirler. Yine de tüm Python veri türleri için ortak bir kavramı paylaşmaları adına nesnelerdir demek yanlış olmaz.

Nesneler tek tek ve çoklu isimlerle (çoklu etki alanları içinde), aynı nesneye bağlı olabilir. Bu diğer dillerde kod isimlendirme (aliasing) olarak bilinir. Bu genellikle Python'da ilk bakışta anlaşılmaz ve değişmez temel türler (sayılar, dizgeler, demetler) ile çalışırken yoksayılabilir. Yine de kod isimlendirme, listeler sözlükler gibi değiştirilebilen nesneler ve program dışındaki öğeler (dosyalar, pencereler, vs) için kullanılan bazı türlerinde katılımıyla Python kodunun kavramsallaştırılmasında (kasıtlı!) bir etkiye sahiptir. Bazı yönleriyle kod isimlendirme göstergelere benzer bir davranış sergilediğinden genellikle, program yararına kullanılmıştır. Örneğin, sadece göstergesi aktarıldığından bir nesnenin aktarılması kolaydır; eğer bir işlev, bir argüman olarak aktarılan bir nesneyi değiştirirse, işlevi çağıran değişikliği görecektir – bu, Pascal'daki gibi iki farklı argüman aktarma gereksinimini ortadan kaldırır.

10.2. Python Etki ve İsim Alanları

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

Birkaç tanım ile işe koyulalım.

Bir *isim alanı* isimler ile nesnelerin eşleşmesidir. Çoğu isim alanı şu anda Python sözlükleri olarak kodlanmışlardır, fakat bu hiçbir şekilde fark edilmez ve gelecekte değiştirilebilir. İsim alanlarına örnekler: yerleşik isimler kümesi (`abs()` gibi işlevler ve yerleşik istisna isimleri vs.), bir modül içindeki global isimler ve bir işlev çağrısındaki yerel isimler. Bir nesnenin özellikler 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 işlev tanımlayabilir; kullanıcılar bu işlevleri önlerine modül adını ekleyerek kullanırlar.

Bu arada, bir noktadan sonra yazılan her herhangi bir isim için *öznitelik* sözcüğünü kullanıyorum. Örneğin, `z.real` ifadesinde `real`, `z` nesnesinin bir özniteliğidir. Modül içindeki isimlere atıflar da öznitelik atıflarıdır: `modulAdi.fonkAdi` ifadesinde `modulAdi` bir modül nesnesidir ve `fonkAdi` bunun bir özniteliğidir. Bir modülün öznitelikleri ile içinde tanımlı global değişkenler aynı isim alanını paylaşırlar.[4]

öznitelikler salt okunur veya yazılabilir olabilirler. Yazılabilir oldukları durumda özniteliklere atama yapmak mümkündür. Modül öznitelikleri yazılabilirdir: `modulAdi.sonuc = 42` gibi bir ifade kullanabilirsiniz. Yazılabilir öznitelikleri `del` deyimini kullanarak silmek de mümkündür. Örneğin `del modulAdi.sonuc` ifadesi `modulAdi` nesnesinden sonuc isimli özniteliği 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 çalıştırdığı deyimler de `__main__` isimli bir modüle ait kabul edilir ve bunların da kendi global isim alanı vardır. Yerleşik isimler de `__builtin__` isimli bir modülde bulunurlar.

Bir işleve ait yerel isim alanı işlev çağırıldığında yaratılır ve işlevden dönüldüğünde veya işlev içinde ele alınmamış bir istisna gerçekleştiğinde silinir. Tabii ki özyinelemeli çağrıların herbiri kendi yerel isim alanına sahiptir.

Bir *etki alanı* bir isim alanının doğrudan erişilebildiği bir metin bölgesidir. Burada "doğrudan erişilebilir" ifadesinin 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 işlevlerin 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. Zaten, en iç etki alanının dışındaki tüm isimler salt okunurdur.

Genellikle yerel etki alanı o an içinde bulunulan (program metninde) işlevin yerel isimlerine atıfta bulunur. İşlevlerin dışında yerel etki alanı global etki alanı ile aynı isim alanıdır: modülün isim alanı. Sınıf tanımlamaları ayrıca yerel etki alanı içerisine bir başka isim alanı daha eklerler.

Etki alanlarının metne bağlı olarak belirlendiğini anlamak önemlidir. Bir modül içinde tanımlı bir işlevin 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 bağını kaldırır. Aslında, yeni isimler yaratan tüm işlemler yerel etki alanını kullanırlar. İşlev tanımları ve `import` deyimleri modül veya işlev adını yerel etki alanına bağlarlar. Bir değişkenin global etki alanında bulunduğunu belirtmek için `global` deyimi kullanılabilir.

10.3. Sınıflara İlk Bakış

Sınıflar ile bir miktar yeni sözdizim ve kavram ile üç yeni nesne türü tanıtacağız.

10.3.1. Sınıf Tanımlama

Sınıf tanımlamanın en basit şekli şöyledir:

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

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

Pratikte bir sınıf tanımının içindeki deyimler genellikle işlev tanımları olur; fakat başka deyimler de kullanmak mümkün ve yararlıdır (buna daha sonra yine değineceğiz). Sınıf içindeki işlev 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 işlevlerin 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. Orjinal 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.

10.3.2. Sınıf Nesneleri

Sınıf nesneleri iki tür işlemi destekler: özniteliklere başvuru (attribute reference) ve sınıfın gerçeklenmesi (instantiation).

Özniteliklere başvuru için Python'da bütün özniteliklere erişmek için kullanılan standart sözdizim kullanılır: `nesne.isim`. Kullanılabilecek öznitelik 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 yöntem nesnesi geri döndüren geçerli öznitelik başvurularıdır. Sınıf özniteliklerine atama yapmak da mümkündür. Örneğin atama yoluyla `benimSinif.i` değeri değiştirilebilir. Ayrıca, `__doc__` da geçerli bir öznitelik olup sınıfa ait belgeleme dizgesini geri döndürür: "Basit bir sınıf örneği.".

Sınıfın gerçeklenmesi, işlev sözdizimini kullanır. Sınıf nesnesini yeni bir sınıf gerçeklemesi geri döndüren parametresiz bir işlevmiş gibi düşünebilirsiniz. Örneğin yukarıda tanımladığımız sınıf için:

```
x = benimSinif()
```

Yeni bir sınıf gerçeklemesidir ve sınıf `x` yerel değişkenine atanarak bir nesne oluşur.

Gerçeklenme işlemi (bir sınıf nesnesini "çağırmak") boş bir nesne yaratır. Pek çok sınıf nesnesinin bilinen bir ilk durumda oluşturulması istenir. Bu yüzden bir sınıfta `__init__()` adlı özel yöntem şu şekilde tanımlanabilir:

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

Bir sınıfın `__init__()` yöntemi tanımlanmış ise sınıf gerçeklenmesi işlemi, yeni sınıf gerçeklemesi sırasında bu yöntemi otomatik olarak çağırır.

Daha fazla esneklik için `__init__()` yönteminin argümanları da olabilir. Bu durumda sınıfın gerçeklenmesinde kullanılan argümanlar `__init__()` yöntemine aktarılır. [5] 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)
```

10.3.3. Nesneler (Gerçeklenen Sınıflar)

Nesnelerle ne yapabiliriz? Bunlar ile yapabileceğimiz tek şey öznitelikleri ile uğraşmaktır. Nesnelerin iki tür özniteliği vardır. Bunların ilki veri özniteliklerdir. Veri özniteliklerinin tanımlanmış olması gerekmez; yerel değişkenlerde olduğu gibi bunlar da kendilerine ilk atama yapıldığında var olurlar. Örneğin `x`'in yukarıda tanımlanan `benimSinif` sınıfının bir gerçeklemesi 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 öznitelikleri de yöntemlerdir. Yöntem bir sınıfa "ait olan" bir işlevdir. Python dilinde yöntemler sınıf gerçeklemelerine özgü değildir; diğer nesne türlerinin de yöntemleri vardır (listeler, sözlükler vs.). Aşağıda yöntem terimini, aksi belirtilmediği sürece, sadece nesnelerin yöntemleri anlamında kullanacağız.

Bir nesneye ilişkin geçerli öznitelik isimleri bunun sınıfına bağlıdır. Tanıma göre işlev olan tüm sınıf öznitelikleri o nesnenin yöntemleri olur. Bu yüzden örnek sınıfımız için `x.f` geçerli bir yöntem başvurusudur, çünkü `benimSinif.f` bir işlevdir, fakat `x.i` bir yöntem değildir, çünkü `benimSinif.i` bir işlev değildir. Burada şuna dikkat edelim: `x.f` ile `benimSinif.f` aynı şey değildir.

10.3.4. Yöntem Nesneleri

Genellikle bir yöntem şu şekilde doğrudan çağırılır:

```
x.f()
```

Bizim örneğimizde bu 'Merhaba' dizgesini geri döndürür. Bir yöntemi doğrudan çağırmak şart değildir: `x.f` bir yöntemdir ve bir değişkene saklanıp daha sonra çağırılabilir. Örneğin:

```
xf = x.f
while 1:
    print xf()
```

Sonsuza kadar "'Merhaba'" yazdırır.

Bir yöntem çağrıldığında tam olarak ne olur? `x.f()` çağrılırken bir argüman kullanılmadığı halde `f` işlev tanımında bir argüman kullanıldığını (`self`)fark etmişsinizdir. Argümana ne oldu acaba? Şüphesiz Python, argüman gerektiren bir işlev argümansız çağırıldığında bir istisna oluşturur. Cevabı belki de tahmin ettiniz: yöntemler, işlevin ilk argümanı olarak nesneyi alırlar. Başka bir deyişle Python'da yöntemler, kendi içinde, tanımlı olduğu nesneyi barındıran nesnelerdir. Örneğimizdeki `x.f()` çağrısı aslında `benimSinif.f(x)` ile aynıdır. Genel olarak, bir yöntemi `n` elemanlı bir argüman listesi ile çağırmak, aynı işlevi başına nesnenin de eklendiği bir argüman listesi kullanarak çağırmak ile aynı şeydir.

10.4. Bazı Açıklamalar

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

Veri özniteliklerine o nesnenin kullanıcıları ("istemcileri") başvuru yapabileceği gibi, yöntemler de bunlara başvuruda bulunabilirler. Başka bir deyişle, sınıflar tamamen soyut veri türleri oluşturmak için kullanılamazlar. Aslında, Python'da hiçbir şey veri saklamayı zorlamayı mümkün kılmaz.

Kullanıcılar nesnelerin veri özniteliklerini dikkatli kullanmalılar; çünkü istemciler yöntemler tarafından kullanılan önemli değişkenlere atama yaparak istenmeyen hatalara sebep olabilirler. İstemcilerin, isim çakışmalarından kaçındıkları sürece, bir nesneye yöntemlerinin geçerliliğini etkilemeden kendi veri özniteliklerini ekleyebileceklerine dikkat edin.

Yöntem içinden veri özniteliklerine (ya da diğer yöntemlere!) başvuruda bulunmanın kestirme bir yolu yoktur. Bunun aslında yöntemlerin okunabilirliğini artırdığını düşünüyorum; bir yönteme göz attığınızda yerel değişkenler ile nesne değişkenlerini birbirilerine karıştırma şansı yoktur.

Usul olarak yöntemlerin 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 özniteliği olan her işlev, o sınıfın nesneleri için bir yöntem tanımlar. İşlev tanımının sınıf tanımı içerisinde olması şart değildir; işlevi, sınıf içindeki yerel bir değişkene atamak da mümkündür. Örneğin:

```
# Sınıf dışında tanımlanmış işlev
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`'ın hepsi `C` sınıfının özellikleri oldular ve aynı anda `C` sınıfının nesnelerinin de yöntemleridirler (`g` ve `h` birbirinin tamamen aynısıdır). Bu tür uygulamanın genellikle sadece okuyucunun kafasını karıştırmaya yaradığına dikkat edin. Yöntemler `self` argümanının yöntem olan özelliğini kullanarak diğer yöntemleri 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)
```

Yöntemler sıradan işlevlerin yaptığı şekilde global değişkenlere başvuru yapabilirler. Bir yönteme ilişkin global etki alanı sınıf tanımının bulunduğu modüldür. Sınıfın kendisi asla global etki alanı olarak kullanılmaz. Bir yöntem içinde global veri kullanmak için ender olarak iyi bir sebep olduğu halde, global etki alanı kullanımın pek çok mantıklı sebebi vardır. Örneğin, global etki alanına yüklenmiş işlev ve modülleri, yöntemler ve bunun içinde tanımlanmış diğer işlev ve sınıflar kullanılabilir. Genellikle yöntemi içeren sınıfın kendisi bu global etki alanı içinde tanımlanmıştır ve bir sonraki kısımda bir yöntemin kendi sınıfına başvurmak istemesi için birkaç iyi sebep bulacağız!

10.5. Miras

Tabii ki, miras alma desteği olmayan bir "sınıf" adına layık olmaz. Türetilmiş sınıf tanımının sözdizimi 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 başvurularını çözümlemede kullanılır; başvuruda bulunulan öznitelik o sınıfta yok ise temel sınıfta aranır. Bu kural temel sınıfın kendisi de başka bir sınıftan türetildiyse ardışık olarak çağırılır.

Türetilmiş sınıftan nesne oluşturmanın özel bir tarafı yoktur: `turetilmisSinifAdi()` o sınıfın yeni bir nesnesini yaratır. Yöntem başvuruları şu şekilde çözümlenirler: ilgili sınıf özelliği, gerekirse temel sınıflar zinciri taranarak, aranır ve yöntem başvurusu geçerli ise bu bir işlev verir.

Türetilmiş sınıflar temel sınıflarının yöntemlerini bastırabilirler. Yöntemler aynı nesnenin diğer yöntemlerini çağırırken özel önceliklere sahip olmadıkları için aynı temel sınıfta tanımlı bir yöntemi çağıran temel sınıf yöntemi bunu bastıran bir türetilmiş sınıf yöntemini çağırmış olabilir. C++ programcıları için not: Tüm Python yöntemleri sanaldır (virtual).

Türetilmiş sınıftaki bir bastıran yöntem aslında temel sınıftaki yöntemin yerini almak yerine onu geliştirmek isteyebilir. Temel sınıf yöntemini doğrudan çağırmanın basit bir yolu vardır: `temelSinifadi.yontemAdi(self, argumanlar)`. Bu bazen istemciler için de faydalıdır. Bunun sadece, temel sınıf, global etki alanı içine doğrudan yüklendiyse çalıştığına dikkat edin.

10.5.1. Çoklu Miras

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

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

Burada sınıf özniteliği başvurularını çözümlemede kullanılan kuralı açıklamamız gerekiyor. Bir öznitelik `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 özniteliğinin `temel1` içinde veya bunun temel sınıflarında tanımlanmış olup olmadığını bilmenizi gerektir ki `temel2` içindeki isimler ile çakışmalardan kaçınabilesiniz.

Python'un kazara oluşan isim çakışmalarına karşı usule dayanması çoklu kalıtımın rasgele kullanımı programın bakımını yapan için bir kabus olduğu açıktır. Çoklu kalıtımın iyi bilinen bir problemi aynı temel sınıfa sahip iki sınıftan türetme yapmaktır. Bu durumda ne olduğunu anlamak kolaydır; ancak bunun ne işe yarayacağı pek açık değildir.

10.6. Özel Değişkenler

Sınıfa özel belirteçler (identifier) için sınırlı destek vardır. `__spam` formundaki (en az iki alt çizgilik bir önek ve en fazla bir alt çizgilik sonek) bir belirteç `_sinifadi__spam` şeklini alır. Burada `sinifadi` o anki sınıf adının sonek alt çizgileri atılmış olan halidir. Bu değişiklik belirtecin sözdizimsel konumuna bakılmaksızın yapılır ki bu sınıf üyesine özel değişkenler yaratılabilsin. Değiştirilen belirteç 255 karakteri aşarsa kırpılabilir. Sınıflar dışında veya sınıf adı sadece alt çizgilerden oluşuyorsa kırpma olmaz.

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

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

10.7. Sona Kalanlar

Bazan isimli veri öğelerini bir arada paketlemek, Pascal kayıtları ya da C veri yapılarına benzer bir veri türü oluşturmak kullanışlı olabilir. Bir boş sınıf ile bu yapılabilir:

```
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 türü bekleyen Python koduna o veri türünün yöntemlerini taklit eden bir sınıf geçirilebilir. Örneğin bir dosya nesnesinden bir miktar veriyi biçimleyen bir işleviniz varsa, `read()` ve `readline()` yöntemleri olan ve veriyi bir dizgeden alan bir sınıfı o işleve argüman olarak aktarabilirsiniz.

Yöntem nesnelerinin de öznitelikleri vardır: `m.im_self` kendinin tanımlı olduğu nesneyi çağıran bir yöntem nesnesidir ve `m.im_func` ise kendini oluşturan işlevi çağıran bir yöntem nesnesidir.

10.8. İstisnalar Sınıf Olabilir

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

`raise` deyimi için iki yeni biçem mevcut:

```
raise Sinif, gercekleme

raise gercekleme
```

İlk biçemde `gercekleme` `Sinif`a ait bir gerçekleme olmalıdır. İkinci biçem ise şunun kısaltmasıdır:

```
raise gercekleme.__class__, gercekleme
```

Bir `except` bloğu hem sınıflar hem de dizgeleri 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 B` bloğu tetiklenecekti.

Ele alınmamış sınıf istisnası için bir ileti yazılacağı zaman, önce sınıf adı yazılır, ardından iki nokta üst üste ve bir boşluk ve son olarak da gerçeklemenin yerleşik `str()` işlevinden geri döndürülen dizgenin karşılığı yazılır.

11. 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 belgelere başvurun.

Resmi Python sanalyöresi `http://www.python.org/` olup; programlar, belgeler ve internetteki Python ile ilgili diğer sanalyörelere bağlantılar içerir. Daha az resmi bir sanalyöre ise `http://starship.python.net/` olup; burada, çoğunda indirilebilir programlar bulunan, Python ile ilgili kişisel sanalyöreler mevcuttur.

Python ile ilgili sorular ve problem raporları için `news:comp.lang.python` haber grubunu veya <python–list (at) python.org> e–posta listesini kullanabilirsiniz. Haber grubu ile e–posta listesi birbirine bağlıdır; yani birine göndereceğiniz bir ileti diğerine de iletilir. Eposta atmadan önce `http://www.python.org/doc/FAQ.html` adresindeki SSS 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.

Notlar

Belge içinde dipnotlar ve dış bağlantılar varsa, bunlarla ilgili bilgiler bulundukları sayfanın sonunda dipnot olarak verilmeyip, hepsi toplu olarak burada listelenmiş olacaktır.

(B4) `http://www.python.org/`

(1) Aslında nesne başvurusu ile çağrı daha iyi bir tanım olur, çünkü işleve değiştirilebilir bir nesne aktarılırsa çağıran çağrılanın o nesneye uyguladığı tüm değişiklikleri görür (listeye eklenen elemanlar gibi).

(2) Değişik türlerin kıyaslanmasına ilişkin kurallara güvenilmemeli; gelecek Python sürümlerinde bu kurallar değişebilir!

(3) Aslında işlev tanımları da 'çalıştırılan' ifadelerdir; işlev adını modülün global simge tablosuna eklerler.

(4) Buna bir istisna: modül nesnelerinin, modülün isim alanını oluşturmada kullanılan sözlüğü oluşturan `__dict__`isimli salt okunur ve gizli bir özniteliği vardır. `__dict__` bir özniteliktir fakat global bir isim değildir.

(5) [Ç.N.]: Diğer nesne yönelimli programlama dillerinde bu işleme "nesnenin ilklendirilmesi" denir.

Bu dosya (python-tutorial.pdf), belgenin XML biçiminin TEXLive ve belgeler-xsl paketlerindeki araçlar kullanılarak PDF biçimine dönüştürülmesiyle elde edilmiştir.

20 Ocak 2007