

BİLİŞİM SİSTEMLERİ MÜHENDİSLİĞİ TASARIMI

Rapor 1

Konu : Python for Data Analysis

Recep Aydoğdu

B171200017

Sakarya Üniversitesi

İçindekiler

Chapter 1.....	4
1. Preliminaries.....	4
1.1 What Is This Book About?.....	4
What Kinds of Data?.....	4
1.2 Why Python for Data Analysis?.....	4
1.3 Essential Python Libraries.....	5
NumPy.....	5
Pandas.....	5
Matplotlib.....	5
IPython and Jupyter.....	6
SciPy.....	6
Scikit-learn.....	6
Statsmodels.....	6
Chapter 2.....	7
2. Python Language Basics, IPython, and Jupyter Notebooks.....	7
2.1 The Python Interpreter.....	7
2.2 IPython Basics.....	8
Running the IPython Shell.....	8
Running the Jupyter Notebook.....	9
Tab Completion.....	12
Introspection.....	14
The %run Command.....	15
Interrupting Running Code.....	16
Executing Code from the Clipboard.....	16
Terminal Keyboard Shortcuts.....	17
About Magic Commands.....	18
Matplotlib Integration.....	20
2.3 Python Language Basics.....	21
Language Semantics.....	21
Indentation, not braces.....	21
Everything is an object.....	22
Comment.....	22
Function and object method calls.....	22
Variables and argument passing.....	23
Dynamic references, strong types.....	24

Attributes and methods.....	26
Duck typing.....	27
Imports.....	28
Binary operators and comparisons.....	29
Binary Operators.....	30
Mutable and immutable objects.....	30
Scalar Types.....	31
Numeric Types.....	31
Strings.....	32
Bytes and Unicode.....	35
Booleans.....	36
Type Casting.....	36
None.....	37
Dates and times.....	38
Datetime Format Specification.....	40
Control Flow.....	40
if, elif and else.....	40
for loops.....	41
while loops.....	43
pass.....	43
range.....	43
Ternary Expressions.....	45
Chapter 3.....	46
Built-in Data Structures, Functions, and Files.....	46

Chapter 1

1. Preliminaries

1.1 What Is This Book About?

Bu kitap, Python'da verilerin manipüle edilmesi (manipulation), işlenmesi (preprocessing), temizlenmesi (cleaning) ve ezilmesiyle (crunching) ilgili ayrıntılar ile ilgilidir. Amacımız, Python programlama dilinin bölümleri ve veri odaklı kütüphane ekosistemi ve sizi etkili bir veri analisti olmamızı sağlayacak araçlar için bir rehber sunmaktır. Kitabın başlığında "veri analizi" yer alırken, veri analizi metodolojisinin aksine odak noktası özellikle Python programlama, kitaplıklar ve araçlar üzerinedir. Bu, veri analizi için ihtiyaç duyduğunuz Python programlamasıdır.

What Kinds of Data?

"Veri" dediğimizde tam olarak neyi kastediyoruz? Birincil odak noktası, aşağıdakiler gibi birçok farklı ortak veri biçimini kapsayan kasıtlı olarak belirsiz bir terim olan yapılandırılmış (**structured**) veriler üzerinedir:

- Her sütunun farklı bir türde (string, numeric, date veya başka türlü) olabileceği tablo veya elektronik tablo benzeri veriler. Bu, genellikle ilişkisel veritabanlarında veya sekme veya virgülle sınırlanmış metin dosyalarında depolanan çoğu veri türünü içerir.
- Çok boyutlu diziler (matrisler).
- Anahtar sütunlarla (bir SQL kullanıcısı için primary veya foreign key'ler olabilir) birbiriyle ilişkili birden çok veri tablosu.
- Eşit veya düzensiz aralıklı zaman serileri.

Bu kesinlikle tam bir liste değildir. Her zaman açık olmasa da, veri kümelerinin büyük bir yüzdesi, analiz ve modelleme için daha uygun olan yapılandırılmış bir forma dönüştürülebilir.

Bir veri kümesindeki özellikleri yapılandırılmış bir forma çıkarmak mümkün olabilir. Örnek olarak, bir haber makaleleri koleksiyonu, daha sonra duygu analizi yapmak için kullanılabilen bir kelime sıklığı tablosuna işlenebilir.

1.2 Why Python for Data Analysis?

Çoğu insan için Python programlama dili güçlü bir çekiciliğe sahiptir. 1991'deki ilk ortaya çıkışından bu yana, Python Perl, Ruby ve diğerleri ile birlikte en popüler yorumlanmış programlama dillerinden biri haline geldi. Python ve Ruby, 2005 yılından beri Rails (Ruby) ve Django (Python) gibi çok sayıda web framework'ünü kullanarak web siteleri oluşturmak için özellikle popüler hale geldi. Bu tür diller, küçük programları veya diğer görevleri otomatikleştirmek için komut dosyalarını hızlı bir şekilde yazmak için kullanılabildikleri için genellikle komut dosyası dilleri olarak adlandırılır. Ciddi bir yazılım oluşturmak için kullanılamayacakları çağrışımını taşıdığı için "komut dosyası dili" terimini sevmiyorum. Python, çeşitli tarihsel ve kültürel nedenlerle yorumlanan diller arasında geniş ve aktif bir bilimsel hesaplama ve veri analizi topluluğu geliştirmiştir. Son 10 yılda, Python bir kanama noktasından veya "riski size ait olmak üzere" bilimsel hesaplama dilinden, akademi ve endüstride veri bilimi, makine öğrenimi ve genel yazılım geliştirme için en önemli dillerden birine geçti.

Veri analizi ve etkileşimli hesaplama ve veri görselleştirme için Python kaçınılmaz olarak diğer açık kaynak ve ticari programlama dilleri ve R, MATLAB, SAS, Stata ve diğerleri gibi geniş kullanımda olan araçlarla karşılaştırmalar yapacaktır. Son yıllarda, Python'un kitaplıklar için geliştirilmiş desteği (pandas ve scikit-learn gibi), onu veri analizi görevleri için popüler bir seçim haline getirmiştir. Python'un genel amaçlı yazılım mühendisliği için genel gücüyle birleştiğinde, veri uygulamaları oluşturmak için birincil dil olarak mükemmel bir seçenektir.

1.3 Essential Python Libraries

NumPy

Numerical Python'un kısaltması olan NumPy, uzun zamandır Python'da sayısal hesaplamanın temel taşı olmuştur. Python'da sayısal verileri içeren çoğu bilimsel uygulama için gereken veri yapılarını ve algoritmaları sağlar. NumPy, diğer şeylerin yanı sıra şunları içerir:

- Hızlı ve verimli çok boyutlu bir dizi nesnesi *ndarray*
- Dizilerle eleman bazlı hesaplamalar veya diziler arasında matematiksel işlemler gerçekleştirmek için işlevler.
- Dizi tabanlı veri kümelerini okumak ve diske yazmak için araçlar.
- Doğrusal cebir işlemleri, Fourier dönüşümü ve rastgele sayı üretimi
- Python uzantılarının ve yerel C veya C ++ kodunun NumPy'nin veri yapılarına ve hesaplama tesislerine erişmesini sağlayan uygun bir C API

NumPy'nin Python'a eklediği hızlı dizi işleme yeteneklerinin ötesinde, veri analizindeki birincil kullanımlarından biri, algoritmalar ve kitaplıklar arasında veri aktarımı için bir container'dır. Sayısal veriler için NumPy dizileri, verileri depolamak ve işlemek için diğer yerleşik Python veri yapılarına göre daha etkilidir. Ayrıca, C veya Fortran gibi daha düşük seviyeli bir dilde yazılmış kitaplıklar, verileri başka bir bellek temsiline kopyalamadan NumPy dizisinde depolanan veriler üzerinde çalışabilir. Bu nedenle, Python için birçok sayısal hesaplama aracı, NumPy dizilerini birincil veri yapısı olarak varsayar veya NumPy ile sorunsuz birlikte çalışabilirliği hedefler.

Pandas

Pandas, yapılandırılmış veya tablo şeklindeki verilerle çalışmayı hızlı, kolay ve anlamlı hale getirmek için tasarlanmış üst düzey veri yapıları ve işlevleri sağlar. 2010'daki ortaya çıkışından bu yana, Python'un güçlü ve verimli bir veri analizi ortamı olmasına yardımcı oldu. Bu kitapta kullanılacak pandas'daki birincil nesneler, hem satır hem de sütun etiketlerine sahip tablo şeklinde, sütun yönelimli bir veri yapısı olan DataFrame ve tek boyutlu etiketli bir dizi nesnesi olan Series'dir.

Pandas, NumPy'nin yüksek performanslı, dizi hesaplama fikirlerini elektronik tabloların ve ilişkisel veritabanlarının (SQL gibi) esnek veri işleme yetenekleriyle harmanlamaktadır.

Matplotlib

Matplotlib, grafikler ve diğer iki boyutlu veri görselleştirmeleri üretmek için en popüler Python kitaplığıdır.

IPython and Jupyter

IPython projesi, 2001 yılında Fernando Pérez'in daha iyi etkileşimli bir Python yorumlayıcısı yapma yan projesi olarak başladı. Sonraki 16 yılda, modern Python veri yığınınındaki en önemli araçlardan biri haline geldi. Kendi başına herhangi bir hesaplama veya veri analitik aracı sağlamazken, IPython hem etkileşimli hesaplamada hem de yazılım geliştirmede üretkenliğinizi en üst düzeye çıkarmak için sıfırdan tasarlanmıştır. Diğer birçok programlama dilinin tipik düzenleme-derleyici çalışma iş akışı yerine yürütme-keşfetme iş akışını teşvik eder. Ayrıca işletim sisteminizin kabuğuna ve dosya sistemine kolay erişim sağlar. Veri analizi kodlamasının çoğu keşif, deneme yanılma ve yinleme içerdiğinden, IPython işi daha hızlı tamamlamanıza yardımcı olabilir.

2014'te Fernando ve IPython ekibi, dilden bağımsız etkileşimli bilgi işlem araçlarını tasarlamak için daha geniş bir girişim olan Jupyter projesini duyurdu. IPython web notebook'u, şimdi 40'tan fazla programlama dilini destekleyen Jupyter notebook oldu. IPython sistemi artık Python'u Jupyter ile kullanmak için bir kernel (bir programlama dili modu) olarak kullanılabilir.

SciPy

SciPy, bilimsel hesaplamada bir dizi farklı standart problem alanını ele alan bir paketler koleksiyonudur.

Scikit-learn

2010 yılında projenin başlangıcından bu yana scikit-learn, Python programcıları için önde gelen genel amaçlı makine öğrenimi araç seti haline geldi. Yalnızca yedi yıl içinde, dünyanın dört bir yanından 1.500'den fazla katılımcısı oldu. Bu tür modeller için alt modüller içerir:

- Classification: SVM, nearest neighbors, random forest, logistic regression, etc.
- Regression: Lasso, ridge regression, etc.
- Clustering: k-means, spectral clustering, etc.
- Dimensionality reduction: PCA, feature selection, matrix factorization, etc.
- Model selection: Grid search, cross-validation, metrics
- Preprocessing: Feature extraction, normalization

Statsmodels

statsmodels, R programlama dilinde popüler bir dizi regresyon analizi modelini uygulayan Stanford Üniversitesi istatistik profesörü Jonathan Taylor'ın çalışmasıyla tohumlanan bir istatistiksel analiz paketidir.

Chapter 2

2. Python Language Basics, IPython, and Jupyter Notebooks

2.1 The Python Interpreter

Python yorumlanmış bir dildir. Python yorumlayıcısı, her seferinde bir ifadeyi çalıştırarak bir programı çalıştırır. Standart etkileşimli Python yorumlayıcısı, python komutuyla komut satırından çağrılabilir:

```
$ python
Python 3.6.0 | packaged by conda-forge | (default, Jan 13 2017, 23:17:12)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-15)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 5
>>> print(a)
5
```

Gördüğünüz >>>, kod ifadelerini yazacağınız istemdir. Python yorumlayıcısından çıkmak ve komut istemine dönmek için **exit ()** yazabilir veya Ctrl-D tuşlarına basabilirsiniz.

Python programlarını çalıştırmak, ilk argümanı olarak bir .py dosyasıyla python'u çağırmak kadar basittir. Şu içeriklerle *hello_world.py* oluşturduğumuzu varsayalım:

```
print('Hello world')
```

Aşağıdaki komutu çalıştırarak çalıştırabilirsiniz (*hello_world.py* dosyası, geçerli çalışan terminal dizininizde olmalıdır):

```
$ python hello_world.py
Hello world
```

Bazı Python programcıları tüm Python kodlarını bu şekilde çalıştırırken, veri analizi veya bilimsel hesaplama yapanlar IPython, gelişmiş bir Python yorumlayıcısı veya orijinal olarak IPython projesi içinde oluşturulan web tabanlı kod defterleri olan Jupyter not defterlerini kullanırlar.

`%run` komutunu kullandığınızda, IPython belirtilen dosyadaki kodu aynı işlemde çalıştırarak keşfetmenizi sağlar tamamlandığında etkileşimli olarak sonuçlar:

```
$ ipython
Python 3.6.0 | packaged by conda-forge | (default, Jan 13 2017, 23:17:12)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?      -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help    -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.

In [1]: %run hello_world.py
Hello world

In [2]:
```

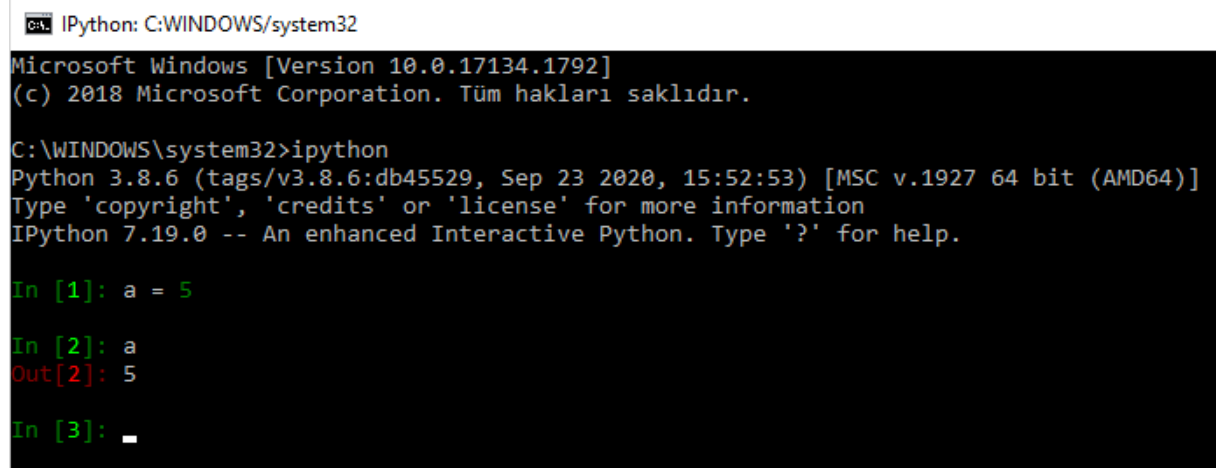
Varsayılan IPython komut istemi, numaralandırılmış [2]: stilini standart `>>>` komut istemiyle karşılaştırır.

2.2 IPython Basics

Bu bölümde, IPython shell', ve Jupyter Notebook'a hazırlanıp çalışacağız ve bazı temel kavramları size tanıyacağız.

Running the IPython Shell

IPython kabuğunu, `ipython` komutu haricinde normal Python yorumlayıcısını başlatır gibi komut satırında başlatabiliriz:



```

C:\WINDOWS\system32
Microsoft Windows [Version 10.0.17134.1792]
(c) 2018 Microsoft Corporation. Tüm hakları saklıdır.

C:\WINDOWS\system32>ipython
Python 3.8.6 (tags/v3.8.6:db45529, Sep 23 2020, 15:52:53) [MSC v.1927 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.19.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: a = 5

In [2]: a
Out[2]: 5

In [3]: _
```

İsteğe bağlı Python ifadelerini yazarak ve Return (veya Enter) tuşuna basarak çalıştırabilirsiniz. IPython'a sadece bir değişken yazdığınızda, nesnenin bir string temsilini oluşturur:


```
[4]: import numpy as np
```

```
[5]: data = {i : np.random.randn() for i in range(7)}
```

```
[6]: data
```

```
[6]: {0: -0.3982976406984385,  
      1: -0.07776196283978527,  
      2: -0.11967699048385459,  
      3: 0.5902793183256904,  
      4: 2.171831990922387,  
      5: 0.1990260547753195,  
      6: -0.278082407614929}
```

İlk iki satır Python kod ifadeleridir; ikinci ifade, yeni oluşturulan bir Python sözlüğüne başvuran `data` adlı bir değişken oluşturur. Son satır, konsoldaki verilerin değerini yazdırır.

Birçok Python nesnesi türü, normal `print` ile yazdırmaktan farklı olarak, daha okunabilir veya güzel basılmış olacak şekilde biçimlendirilmiştir. Yukarıdaki veri değişkenini standart Python yorumlayıcısında yazdırırsanız, çok daha az okunabilir olacaktır:

```
[9]: from numpy.random import randn
```

```
[10]: data = {i : randn() for i in range(7)}
```

```
[11]: print(data)
```

```
{0: 0.1089264969992243, 1: 0.031039188304209056, 2: -1.3404162410677365, 3: 0.23966690069381563,  
 4: -0.41758532623374023, 5: 0.1476185674601656, 6: -0.8879869022124973}
```

```
[12]: data
```

```
[12]: {0: 0.1089264969992243,  
      1: 0.031039188304209056,  
      2: -1.3404162410677365,  
      3: 0.23966690069381563,  
      4: -0.41758532623374023,  
      5: 0.1476185674601656,  
      6: -0.8879869022124973}
```

Running the Jupyter Notebook

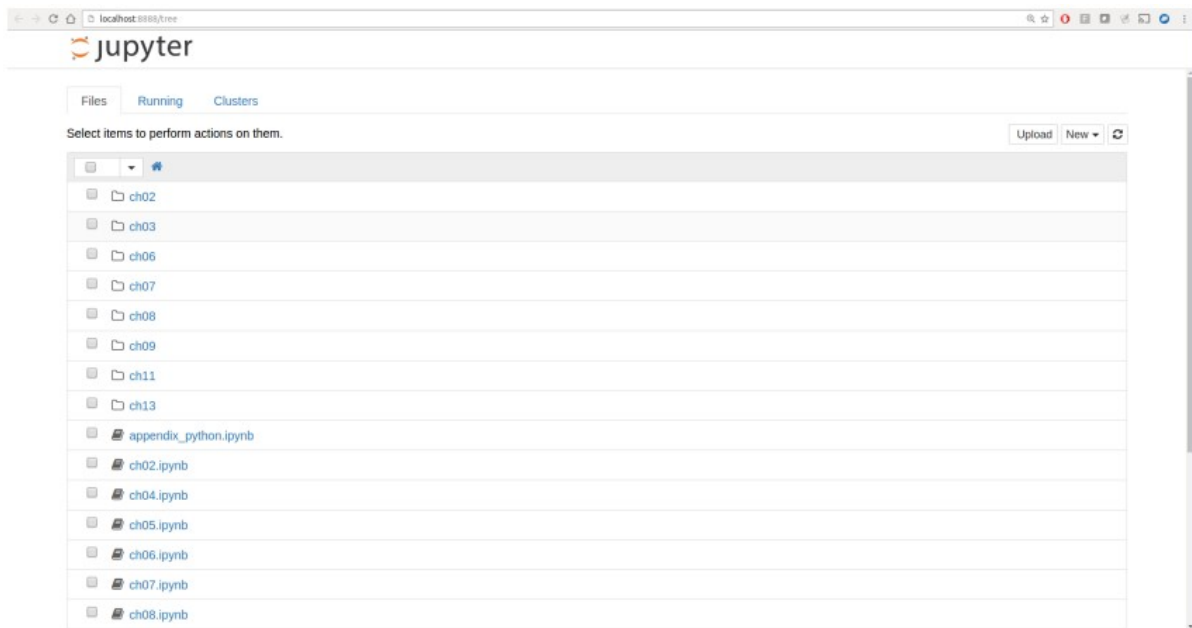
Jupyter projesinin ana bileşenlerinden biri notebook'dur, kod için bir tür etkileşimli belge, metin (işaretli veya işaretli), veri görselleştirmeleri ve diğer çıktılarıdır.

Jupyter notebook, Jupyter etkileşimli hesaplama protokolünün herhangi bir sayıda programlama dilinde uygulamaları olan çekirdeklerle etkileşim kurar. Python'un Jupyter çekirdeği, temel davranışı için IPython sistemini kullanır.

Jupyter'i başlatmak için, bir terminalde jupyter notebook komutunu çalıştırın:

```
$ jupyter notebook
[I 15:20:52.739 NotebookApp] Serving notebooks from local directory:
/home/wesm/code/pydata-book
[I 15:20:52.739 NotebookApp] 0 active kernels
[I 15:20:52.739 NotebookApp] The Jupyter Notebook is running at:
http://localhost:8888/
[I 15:20:52.740 NotebookApp] Use Control-C to stop this server and shut down
all kernels (twice to skip confirmation).
Created new window in existing browser session.
```

Pek çok platformda, Jupyter otomatik olarak varsayılan web tarayıcınızda açılır (--no-browser ile başlatmadığınız sürece). Aksi takdirde, not defterini başlattığınızda yazdırılan HTTP adresine gidebilirsiniz, burada *http://localhost:8888/*.



Örnek bir Jupyter Notebook dosyası görünümü;

```
[13]: %pwd

[13]: 'C:\\Users\\receptedek\\Desktop\\tasarım\\notebooks'

[16]: path = "datasets/bitly_usagov/example.txt"

[18]: open(path).readline()

[18]: '{ "a": "Mozilla\\5.0 (Windows NT 6.1; WOW64) AppleWebKit\\535.11 (KHTML, like Gecko) Chrome
\\17.0.963.78 Safari\\535.11", "c": "US", "nk": 1, "tz": "America\\New_York", "gr": "MA",
"g": "A6qOVH", "h": "wflQtf", "l": "orofrog", "al": "en-US,en;q=0.8", "hh": "1.usa.gov", "r": "h
ttp:\\\\www.facebook.com\\1\\7AQEFzjSi\\1.usa.gov\\wflQtf", "u": "http:\\\\www.ncbi.nlm.
nih.gov\\pubmed\\22415991", "t": 1331923247, "hc": 1331822918, "cy": "Danvers", "ll": [ 42.576
698, -70.954903 ] }\\n'

[19]: import json
path = "datasets/bitly_usagov/example.txt"
records = [json.loads(line) for line in open(path)]

[20]: records[0]

[20]: {'a': 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11 (KHTML, like Gecko) Chrome/17.0.96
3.78 Safari/535.11',
'c': 'US',
'nk': 1,
'tz': 'America/New_York',
'gr': 'MA',
'g': 'A6qOVH',
'h': 'wflQtf',
'l': 'orofrog',
'al': 'en-US,en;q=0.8',
'hh': '1.usa.gov',
'r': 'http://www.facebook.com/1/7AQEFzjSi/1.usa.gov/wflQtf',
'u': 'http://www.ncbi.nlm.nih.gov/pubmed/22415991',
't': 1331923247,
'hc': 1331822918,
'cy': 'Danvers',
'll': [42.576698, -70.954903]}

[21]: records[0]['tz']

[21]: 'America/New_York'

[22]: print(records[0]['tz'])

America/New_York
```

Tab Completion

Yüzeyde, IPython kabuğu, standart uçbirim Python yorumlayıcısının (python ile çağrılan) kozmetik olarak farklı bir versiyonu gibi görünür. Standart Python kabuğu üzerindeki en büyük iyileştirmelerden biri, birçok IDE'de veya diğer etkileşimli hesaplama analiz ortamlarında bulunan Tab tamamlamadır. Kabuğa ifadeler girerken, Tab tuşuna basmak, şimdiye kadar yazdığınız karakterlerle eşleşen tüm değişkenler (objects, functions vb.) için ad alanında arama yapacaktır:

```
In [1]: an_apple = 27
```

```
In [2]: an_example = 42
```

```
In [3]: an<Tab>
an_apple    and        an_example  any
```

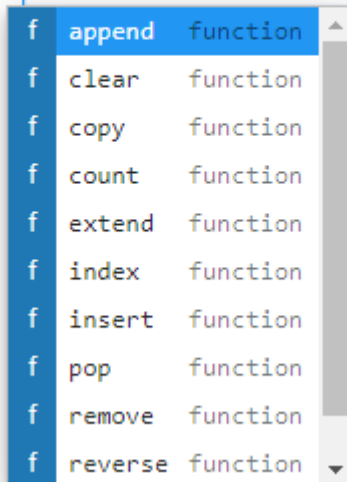
Bu örnekte, IPython'un hem tanımladığım iki değişkeni hem de Python anahtar sözcüğü *and* ve yerleşik *any* işlevini gösterdiğine dikkat edin. Doğal olarak, bir nokta yazdıktan sonra herhangi bir nesne üzerindeki yöntemleri ve nitelikleri de tamamlayabilirsiniz:

```
In [3]: b = [1, 2, 3]
```

```
In [4]: b.<Tab>
b.append  b.count  b.insert  b.reverse
b.clear   b.extend b.pop     b.sort
b.copy    b.index   b.remove
```

```
[23]: b = [1,2,3]
```

```
[ ]: b.
```



f	append	function
f	clear	function
f	copy	function
f	count	function
f	extend	function
f	index	function
f	insert	function
f	pop	function
f	remove	function
f	reverse	function

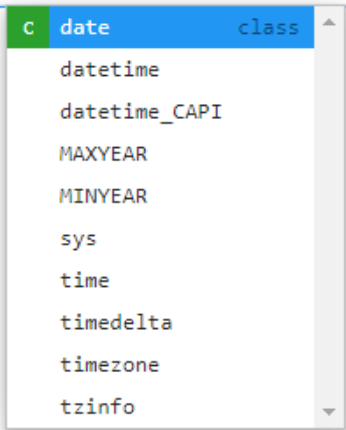
Aynı modüller için de geçerlidir:

```
In [1]: import datetime

In [2]: datetime.<Tab>
datetime.date          datetime.MAXYEAR      datetime.timedelta
datetime.datetime      datetime.MINYEAR      datetime.timezone
datetime.datetime_CAPI datetime.time          datetime.tzinfo

[24]: import datetime

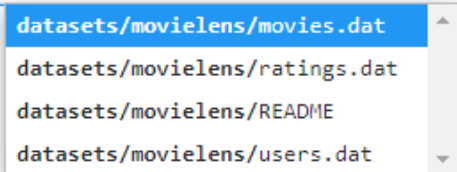
[ ]: datetime.|
```



Jupyter notebook ve IPython'un daha yeni sürümlerinde (5.0 ve üstü), otomatik tamamlamalar metin çıktısı yerine açılır bir kutuda görünür.

Sekme tamamlama, etkileşimli ad alanını arama ve nesne veya modül özniteliklerini tamamlamanın dışında birçok bağlamda çalışır. Dosya yoluna benzeyen herhangi bir şey yazarken (bir Python dizisinde bile), Sekme tuşuna basmak bilgisayarınızın dosya sistemindeki her şeyi yazdıklarınızla eşleşen şekilde tamamlayacaktır:

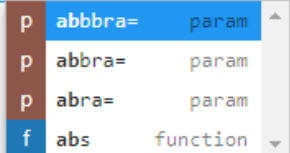
```
[ ]: path = 'datasets/movielens/'
```



Sekme tamamlamanın zaman kazandırdığı başka bir alan, anahtar kelime işlevi argümanlarının (ve = sign! dahil) tamamlanmasıdır.

```
[25]: def func_with_keywords(abra=1, abbra=2, abbbra=3):
      return abra, abbra, abbbra

[ ]: func_with_keywords(ab
```



Introspection

Bir değişkenden önce veya sonra bir soru işareti (?) Kullanmak, nesne hakkında bazı genel bilgileri görüntüler:

```
[26]: b = [1,2,3]
```

```
[27]: b?
```

```
Type:      list
String form: [1, 2, 3]
Length:    3
Docstring:
Built-in mutable sequence.
```

```
If no argument is given, the constructor creates a new empty list.
The argument must be an iterable if specified.
```

```
[28]: print?
```

```
Docstring:
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file: a file-like object (stream); defaults to the current sys.stdout.
sep:   string inserted between values, default a space.
end:   string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
Type:   builtin_function_or_method
```

Bu, nesne iç gözlemi(object introspection) olarak adlandırılır. Nesne bir function veya instance method ise, tanımlanmışsa, docstring de gösterilecektir. Aşağıdaki işlevi yazdığımızı varsayalım (IPython veya Jupyter'de çoğaltabilirsiniz):

```
[29]: def add_numbers(a, b):
      """
      Add two numbers together
      Returns
      -----
      the_sum : type of arguments
      """
      return a + b
```

```
[30]: add_numbers?
```

```
Signature: add_numbers(a, b)
Docstring:
Add two numbers together
Returns
-----
the_sum : type of arguments
File:    c:\users\recep\desktop\tasarım\notebooks\<ipython-input-29-5447cfd50127>
Type:    function
```

```
[31]: add_numbers??

Signature: add_numbers(a, b)
Source:
def add_numbers(a, b):
    """
    Add two numbers together
    Returns
    -----
    the_sum : type of arguments
    """
    return a + b
File:      c:\users\recep\appdata\local\temp\ipython-input-29-5447cfd50127>
Type:      function
```

? IPython ad alanını standart Unix veya Windows komut satırına benzer bir şekilde aramak için son bir kullanıma sahiptir. Joker karakterle (*) birleştirilen birkaç karakter, joker ifadesiyle eşleşen tüm adları gösterecektir. Örneğin, en üst düzey NumPy ad alanındaki load içeren tüm işlevlerin bir listesini alabiliriz:

```
[32]: np.*load*?

np.__loader__
np.load
np.loads
np.loadtxt
```

The %run Command

Herhangi bir dosyayı %run komutunu kullanarak IPython oturumunuzun ortamında bir Python programı olarak çalıştırabilirsiniz. *ipython_script_test.py*'de aşağıdaki basit komut dosyasının depolandığını varsayalım:

```
def f(x, y, z):
    return (x + y) / z

a = 5
b = 6
c = 7.5

result = f(a, b, c)
```

Dosya adını %run'a ileterek bunu yürütebilirsiniz.

```
[1]: %run ipython_script_test.py
```

Komut dosyası boş bir namespace'de çalıştırılır (import'lar veya diğer değişkenler tanımlanmadan), böylece davranış, programı python script.py kullanarak komut satırında çalıştırmakla aynı olmalıdır. Dosyada tanımlanan tüm değişkenler (import'lar, fonksiyon'lar ve global'ler) (varsa bir istisna oluşana kadar) IPython kabuğundan erişilebilir olacaktır:

```
[2]: c
```

```
[2]: 7.5
```

```
[3]: result
```

```
[3]: 1.4666666666666666
```

Bir Python betiği komut satırı bağımsız değişkenlerini beklerse (sys.argv'de bulunur), bunlar komut satırında çalıştırılmış gibi dosya yolundan sonra geçirilebilir.

Not: Etkileşimli IPython namespace'den önceden tanımlanmış değişkenlere bir komut dosyası erişimi vermek isterseniz, düz **%run** yerine **%run -i** kullanın.

Jupyter not defterinde, bir komut dosyasını bir kod hücresine aktaran ilgili **%load** magic işlevini de kullanabilirsiniz:

```
[4]: # %load ipython_script_test.py
def f(x, y, z):
    return (x + y) / z

a = 5
b = 6
c = 7.5
result = f(a, b, c)
```

Interrupting Running Code

Herhangi bir kod çalışırken Ctrl-C tuşlarına basmak, ister %run üzerinden bir komut dosyası isterse uzun süreli bir komut olsun, bir KeyboardInterrupt'ın yükseltilmesine neden olur. Bu, bazı olağandışı durumlar dışında neredeyse tüm Python programlarının hemen durmasına neden olur.

Not: Bir Python kodu parçası bazı derlenmiş uzantı modüllerini çağırdığında, Ctrl-C'ye basmak her zaman program yürütülmesinin hemen durmasına neden olmaz. Bu gibi durumlarda, ya kontrolün Python yorumlayıcısına geri dönmelerini beklemeniz gerekecek ya da daha kötü durumlarda Python sürecini zorla sonlandırmalısınız.

Executing Code from the Clipboard

Jupyter not defterini kullanıyorsanız, kodu herhangi bir kod hücresine kopyalayıp yapıştırabilir ve çalıştırabilirsiniz. IPython kabuğundaki panodan kod çalıştırmak da mümkündür. Başka bir uygulamada aşağıdaki koda sahip olduğunuzu varsayalım:

```
[6]: x = 5
y = 7
if x > 5:
    x += 1

y = 8
```


En kusursuz yöntemler, **%paste** ve **%cpaste** magic işlevleridir. **%paste** panodaki metni alır ve onu kabukta tek bir blok olarak yürütür:

```
In [17]: %paste
x = 5
y = 7
if x > 5:
    x += 1

    y = 8
## -- End pasted text --
```

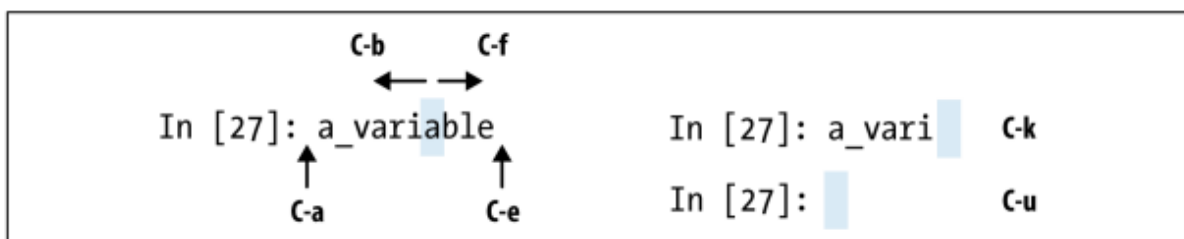
%cpaste, kodu şuraya yapıştırmanız için size özel bir uyarı vermesi dışında benzerdir:

```
In [18]: %cpaste
Pasting code; enter '--' alone on the line to stop or use Ctrl-D.
:x = 5
:y = 7
:if x > 5:
:    x += 1
:
:
:    y = 8
:--
```

Not: **%paste** magic fonksiyonu sadece IPython'un terminal sürümü için tanımlanmıştır, grafiksel ön uçları (yani notebook ve qtconsole) için değil, çünkü buna ihtiyaç duymazlar.

%cpaste bloğu ile, çalıştırmadan önce istediğiniz kadar çok kod yapıştırma özgürlüğüne sahipsiniz. Yapıştırılan koda, çalıştırmadan önce bakmak için **%cpaste** kullanmaya karar verebilirsiniz. Yanlışlıkla yanlış kodu yapıştırırsanız, Ctrl-C tuşlarına basarak **%cpaste** isteminden çıkabilirsiniz.

Terminal Keyboard Shortcuts



Keyboard shortcut	Description
Ctrl-P or up-arrow	Search backward in command history for commands starting with currently entered text
Ctrl-N or down-arrow	Search forward in command history for commands starting with currently entered text
Ctrl-R	Readline-style reverse history search (partial matching)
Ctrl-Shift-V	Paste text from clipboard
Ctrl-C	Interrupt currently executing code
Ctrl-A	Move cursor to beginning of line
Ctrl-E	Move cursor to end of line
Ctrl-K	Delete text from cursor until end of line
Ctrl-U	Discard all text on current line
Ctrl-F	Move cursor forward one character
Ctrl-B	Move cursor back one character
Ctrl-L	Clear screen

About Magic Commands

IPython'un özel komutları (Python'da yerleşik değildir) "magic" komutlar olarak bilinir. Bunlar, ortak görevleri kolaylaştırmak ve IPython sisteminin davranışını kolayca kontrol etmenizi sağlamak için tasarlanmıştır. Bir magic komut, % yüzde sembolü ile başlayan herhangi bir komuttur. Örneğin, matris çarpımı gibi herhangi bir Python ifadesinin yürütme zamanını **%timeit** magic işlevini kullanarak kontrol edebilirsiniz (daha sonra daha ayrıntılı olarak tartışılacaktır):

```
[9]: import numpy as np
a = np.random.randn(100, 100)

[10]: %timeit np.dot(a, a)

115 µs ± 1.19 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Magic komutlar, IPython sistemi içinde çalıştırılacak komut satırı programları olarak görülebilir. Birçoğunun ek "komut satırı" seçenekleri vardır, bunların tümü (beklediğiniz gibi) "?" kullanılarak görüntülenebilir:

```
[11]: %debug?

Docstring:
::

    %debug [--breakpoint FILE:LINE] [statement [statement ...]]

Activate the interactive debugger.

This magic command support two ways of activating debugger.
One is to activate debugger before executing code. This way, you
can set a break point, to step through the code from the point.
You can use this mode by giving statements to execute and optionally
a breakpoint.

The other one is to activate debugger in post-mortem mode. You can
activate this mode simply running %debug without any argument.
If an exception has just occurred, this lets you inspect its stack
frames interactively. Note that this will always work only on the last
traceback that occurred, so you must call this quickly after an
exception that you wish to inspect has fired, because if another one
occurs, it clobbers the previous one.

If you want IPython to automatically do this on every exception, see
the %pdb magic for more details.

.. versionchanged:: 7.3
    When running code, user variables are no longer expanded,
    the magic line is always left unmodified.

positional arguments:
  statement          Code to run in debugger. You can omit this in cell magic mode.

optional arguments:
  --breakpoint <FILE:LINE>, -b <FILE:LINE>
                        Set break point at LINE in FILE.
File: c:\users\recep\yede\desktop\tasarım\notebooks\venv\lib\site-packages\ipython\core\magics\execution.py
```

Söz konusu magic işlevle aynı ada sahip hiçbir değişken tanımlanmadığı sürece, magic işlevler varsayılan olarak yüzde işareti olmadan kullanılabilir. Bu özelliğe automagic denir ve %automagic ile etkinleştirilebilir veya devre dışı bırakılabilir.

Bazı magic işlevler Python işlevleri gibi davranır ve çıktıları bir değişkene atanabilir:

```
[12]: %pwd

[12]: 'D:\\PC Yedek\\Belgeler\\Çalışmalar\\Tasarım-Bitirme\\notebooks'

[13]: foo = %pwd

[14]: foo

[14]: 'D:\\PC Yedek\\Belgeler\\Çalışmalar\\Tasarım-Bitirme\\notebooks'
```

Command	Description
<code>%quickref</code>	Display the IPython Quick Reference Card
<code>%magic</code>	Display detailed documentation for all of the available magic commands
<code>%debug</code>	Enter the interactive debugger at the bottom of the last exception traceback
<code>%hist</code>	Print command input (and optionally output) history
<code>%pdb</code>	Automatically enter debugger after any exception
<code>%paste</code>	Execute preformatted Python code from clipboard
<code>%cpaste</code>	Open a special prompt for manually pasting Python code to be executed
<code>%reset</code>	Delete all variables/names defined in interactive namespace
<code>%page OBJECT</code>	Pretty-print the object and display it through a pager
<code>%run script.py</code>	Run a Python script inside IPython
<code>%prun statement</code>	Execute <i>statement</i> with <code>cProfile</code> and report the profiler output
<code>%time statement</code>	Report the execution time of a single statement
<code>%timeit statement</code>	Run a statement multiple times to compute an ensemble average execution time; useful for timing code with very short execution time
<code>%who</code> , <code>%who_ls</code> , <code>%whos</code>	Display variables defined in interactive namespace, with varying levels of information/verbosity
<code>%xdel variable</code>	Delete a variable and attempt to clear any references to the object in the IPython internals

Matplotlib Integration

IPython'un analitik hesaplamadaki popüleritesinin bir nedeni, veri görselleştirme ve matplotlib gibi diğer kullanıcı arayüzü kitaplıkları ile iyi entegre olmasıdır.

`%matplotlib` magic işlevi IPython kabuğu veya Jupyter notebook ile entegrasyonunu yapılandırır. Bu önemlidir, çünkü aksi halde oluşturduğunuz grafikler görünmez (notebook) veya kapanana kadar (kabuk) oturumun kontrolünü ele geçirmez.

IPython kabuğunda, `%matplotlib`'i çalıştırmak, entegrasyonu kurar, böylece konsol oturumuna müdahale etmeden birden çok çizim penceresi oluşturabilirsiniz:

```
In [26]: %matplotlib
Using matplotlib backend: Qt4Agg
```

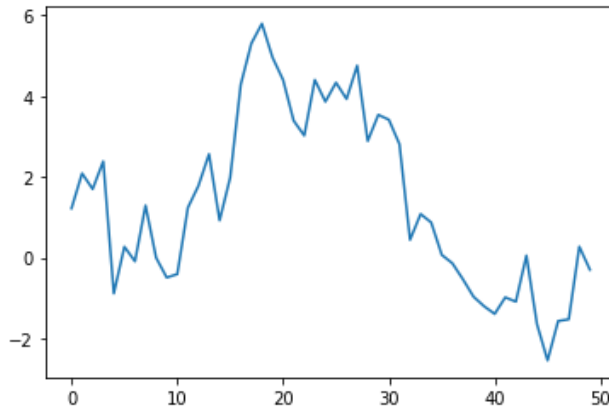
Jupyter'de komut biraz farklı:

```
[18]: %matplotlib inline
```

Matplotlib is building the font cache; this may take a moment.

```
[19]: import matplotlib.pyplot as plt  
plt.plot(np.random.randn(50).cumsum())
```

```
[19]: [<matplotlib.lines.Line2D at 0x16a40835310>]
```



2.3 Python Language Basics

Bu bölümde, temel Python programlama kavramları ve dil mekaniğine genel bir bakış sunacağız. Sonraki bölümde Python'un veri yapıları, işlevleri ve diğer yerleşik araçları hakkında daha fazla ayrıntıya gireceğiz.

Language Semantics

Python dil tasarımı, okunabilirlik, basitlik ve açıklığa vurgu yapmasıyla ayırt edilir. Bazı insanlar bunu "executable pseudocode(çalıştırılabilir sözde kod)" a benzetecek kadar ileri giderler.

Indentation, not braces

Python, R, C ++, Java ve Perl gibi diğer birçok dilde olduğu gibi parantez kullanmak yerine kodu yapılandırmak için boşluklar (tab veya boşluklar) kullanır. Bir sıralama algoritmasından bir for döngüsü düşünün:

```
[1]: for x in array:  
    if x < pivot:  
        less.append()  
    else:  
        greater.append(x)
```

İki nokta üst üste, girintili bir kod bloğunun başlangıcını belirtir, bundan sonra tüm kod bloğun sonuna kadar aynı miktarda girintilendirilmelidir.

Sevin ya da nefret edin, önemli boşluk Python programcıları için hayatın bir gerçeğidir ve benim deneyimlerime göre Python kodunu kullandığım diğer dillerden daha okunaklı hale getirebilir. İlk başta yabancı görünse de, umarım zamanla alışırsınız.

Şimdiye kadar görebileceğiniz gibi, Python ifadelerinin de noktalı virgüllerle sonlandırılmasına gerek yoktur. Bununla birlikte, tek bir satırdaki birden çok ifadeyi ayırmak için noktalı virgül kullanılabilir:

```
[ ]: a = 5; b = 6; c = 7
```

Python'da birden çok ifadeyi tek bir satıra koymak genellikle önerilmez çünkü kodu daha az okunabilir hale getirir.

Everything is an object

Python dilinin önemli bir özelliği, nesne modelinin tutarlılığıdır. Her number, string, veri yapısı, function, class, modül ve benzeri, Python yorumlayıcısında, Python nesnesi olarak adlandırılan kendi "box"ında bulunur. Her nesnenin ilişkili bir türü (örneğin, string veya function) ve dahili verileri vardır. Pratikte bu, dilin çok esnek olmasını sağlar, çünkü işlevler bile başka herhangi bir nesne gibi ele alınabilir.

Comment

Hash işareti (pound işareti) `#` ile başlayan herhangi bir metin Python yorumlayıcısı tarafından yok sayılır. Bu genellikle koda yorum eklemek için kullanılır. Bazen belirli kod bloklarını silmeden hariç tutmak isteyebilirsiniz. Kolay bir çözüm, kodu yorumlamaktır:

```
[ ]: results = []
    for line in file_handle:
        #keep the empty lines for now
        # if len(line) == 0:
        #     continue
        results.append(line.replace('foo', 'bar'))
```

Yorumlar, çalıştırılan bir kod satırından sonra da ortaya çıkabilir. Bazı programcılar yorumların belirli bir kod satırından önceki satıra yerleştirilmesini tercih ederken, bu bazen yararlı olabilir:

```
[2]: print("Reached this line") # Simple status report
      Reached this line
```

Function and object method calls

Parantez kullanarak ve sıfır veya daha fazla argüman ileterek, isteğe bağlı olarak döndürülen değeri bir değişkene atayarak işlevleri çağırırsınız:

```
[ ]: result = f(x, y, z)
      g()
```

Python'daki hemen hemen her nesnenin, nesnenin dahili içeriğine erişimi olan yöntemler olarak bilinen ekli işlevleri vardır. Onları aşağıdaki sözdizimini kullanarak arayabilirsiniz:

```
[ ]: obj.some_method(x, y, z)
```

Fonksiyonlar hem konumsal hem de anahtar kelime argümanlarını alabilir:

```
[ ]: result = f(a, b, c, d=5, e='foo')
```

Daha sonra bununla ilgili daha fazla şey göreceğiz.

Variables and argument passing

Python'da bir değişken (veya ad) atarken, eşittir işaretinin sağ tarafındaki nesneye bir referans oluşturursunuz. Pratik açıdan, tam sayıların bir listesini düşünün:

```
[ ]: a = [1, 2, 3]
```

a'yı yeni bir değişken b'ye atadığımızı varsayalım:

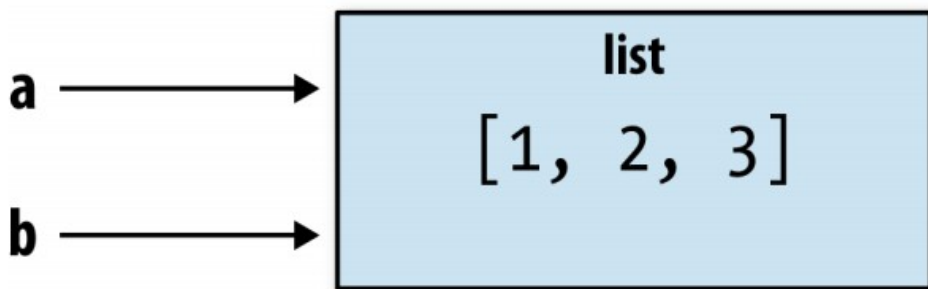
```
[ ]: b = a
```

Bazı dillerde bu atama, verilerin [1, 2, 3] kopyalanmasına neden olur. Python'da, a ve b artık aynı nesneye, orijinal listeye [1, 2, 3] atıfta bulunur. Bunu, a'ya bir öğe ekleyerek ve ardından b'yi inceleyerek kanıtlayabilirsiniz:

```
[5]: a.append(4)
```

```
[6]: b
```

```
[6]: [1, 2, 3, 4]
```



Aynı nesne için iki referans

Python'daki referansların anlamını anlamak ve verilerin ne zaman, nasıl ve neden kopyalandığını anlamak, Python'da daha büyük veri kümeleriyle çalışırken özellikle önemlidir.

Not: Bir nesneye bir isim bağladığımız için atama, bağlama olarak da adlandırılır. Atanan değişken isimleri zaman zaman bağlı değişkenler olarak adlandırılabilir.

Nesneleri bir işleve argüman olarak ilettiğinizde, herhangi bir kopyalama olmaksızın orijinal nesnelere başvuran yeni yerel değişkenler oluşturulur. Yeni bir nesneyi bir işlevin içindeki değişkene bağlarsanız, bu değişiklik üst kapsama yansıtılmayacaktır. Bu nedenle, değişebilir bir argümanın içsellerini değiştirmek mümkündür. Aşağıdaki işleve sahip olduğumuzu varsayalım:

```
[7]: def append_element(some_list, element):  
      some_list.append(element)
```

```
[8]: data = [1, 2, 3]
```

```
[9]: append_element(data, 4)
```

```
[10]: data
```

```
[10]: [1, 2, 3, 4]
```

Dynamic references, strong types

Java ve C++ gibi birçok derlenmiş dilin aksine, Python'daki nesne referanslarının kendileriyle ilişkili hiçbir türü yoktur. Aşağıdakilerle ilgili bir sorun yok:

```
[11]: a = 5
```

```
[12]: type(a)
```

```
[12]: int
```

```
[13]: a = 'foo'
```

```
[14]: type(a)
```

```
[14]: str
```

Değişkenler, belirli bir namespace içindeki nesnelerin adlarıdır; tür bilgisi nesnenin kendisinde saklanır. Bazı gözlemciler aceleyle Python'un "typed language" olmadığı sonucuna varabilir. Bu doğru değil; bu örneği düşünün:

```
[15]: '5' + 5
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-15-4dd8efb5fac1> in <module>  
----> 1 '5' + 5  
  
TypeError: can only concatenate str (not "int") to str
```


Visual Basic gibi bazı dillerde, '5' dizisi örtük olarak bir tamsayıya dönüştürülebilir (veya çevrilebilir) ve böylece 10 elde edilebilir. Yine de JavaScript gibi diğer dillerde 5 tamsayısı bir dizeye dönüştürülebilir ve sonuç olarak birleştirilmiş '55' dizesi. Bu bağlamda Python, güçlü bir şekilde yazılmış bir dil olarak kabul edilir, yani her nesnenin belirli bir türü (veya sınıfı) vardır ve örtük dönüştürmeler yalnızca aşağıdakiler gibi belirli belli durumlarda gerçekleşir:

```
[17]: a = 4.5
      b = 2

      #String formatting, daha sonra incelenecek.
      print("a is {0}, b is {1}".format(type(a), type(b)))

      a is <class 'float'>, b is <class 'int'>
```

```
[18]: a/b
```

```
[18]: 2.25
```

Bir nesnenin türünü bilmek önemlidir ve birçok farklı türde girdiyi işleyebilen işlevler yazabilmek yararlıdır. **isinstance** işlevini kullanarak bir nesnenin belirli bir türün örneği olup olmadığını kontrol edebilirsiniz:

```
[19]: a = 5
      isinstance(a, int)
```

```
[19]: True
```

```
[21]: isinstance(a, float)
```

```
[21]: False
```

isinstance, bir nesnenin type'nın dizide bulunanlar arasında olup olmadığını kontrol etmek istiyorsanız type'ları içeren bir tuple'ı kabul edebilir:

```
[25]: a = 5; b = 4.5

      print("a, int ya da float type'larından birisi mi:", isinstance(a, (int, float)))

      print("b, int ya da float type'larından birisi mi:", isinstance(b, (int, float)))

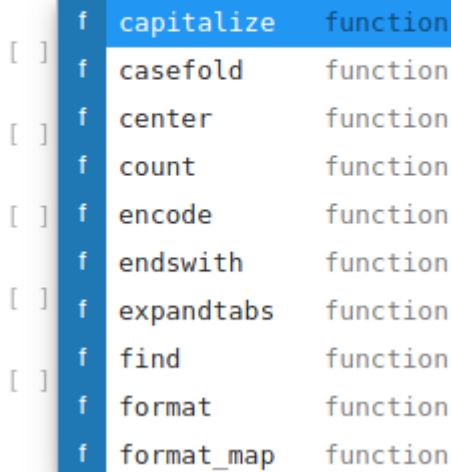
      a, int ya da float type'larından birisi mi: True
      b, int ya da float type'larından birisi mi: True
```

Attributes and methods

Python'daki nesneler tipik olarak hem niteliklere(attributes) (nesnenin "içinde" depolanan diğer Python nesnelerine) ve yöntemlere(methods) (nesnenin dahili verilerine erişebilen bir nesneyle ilişkili işlevler) sahiptir. Her ikisine de `obj.attribute_name` sözdizimi aracılığıyla erişilir:

```
[26]: a = 'foo'
```

```
[ ]: a.
```



f	capitalize	function
f	casefold	function
f	center	function
f	count	function
f	encode	function
f	endswith	function
f	expandtabs	function
f	find	function
f	format	function
f	format_map	function

```
In [2]: a.<Press Tab>
```

a.capitalize	a.format	a.isupper	a.rindex	a.strip
a.center	a.index	a.join	a.rjust	a.swapcase
a.count	a.isalnum	a.ljust	a.rpartition	a.title
a.decode	a.isalpha	a.lower	a.rsplit	a.translate
a.encode	a.isdigit	a.lstrip	a.rstrip	a.upper
a.endswith	a.islower	a.partition	a.split	a.zfill
a.expandtabs	a.isspace	a.replace	a.splitlines	
a.find	a.istitle	a.rfind	a.startswith	

Attribute(öznitelik)'lere ve method'lara `getattr` işlevi aracılığıyla adla da erişilebilir:

```
[29]: getattr(a, 'split')
```

```
[29]: <function str.split(sep=None, maxsplit=-1)>
```

Diğer dillerde, nesnelere isme göre erişim genellikle "yansıma(reflection)" olarak adlandırılır. Bu kitapta `getattr` ve ilgili fonksiyonlar `hasattr` ve `setattr` fonksiyonlarını kapsamlı bir şekilde kullanmayacak olsak da, genel, tekrar kullanılabilir kod yazmak için çok etkili bir şekilde kullanılabilirler.

Duck typing

Genellikle bir nesnenin türü ile ilgilenmeyebilirsiniz, bunun yerine yalnızca belirli metodları veya davranışları olup olmadığı ile ilgilenebilirsiniz. Buna bazen "Ördek gibi yürürse ve ördek gibi şarlatansa, o zaman bir ördektir" ifadesinden sonra "Duck Typing" olarak adlandırılır. Örneğin, yineleme protokolünü (iterator protocol) uygulayan bir nesnenin yinelenebilir olduğunu doğrulayabilirsiniz. Birçok nesne için bu, `iter` "magic metodu" olduğu anlamına gelir, ancak kontrol etmenin alternatif ve daha iyi bir yolu iter işlevini kullanmayı denemektir:

```
[30]: def isiterable(obj):  
      try:  
          iter(obj)  
          return True  
      except TypeError: #not iterable  
          return False
```

```
[31]: isiterable('a string')
```

```
[31]: True
```

```
[32]: isiterable([1, 2, 3])
```

```
[32]: True
```

```
[33]: isiterable(5)
```

```
[33]: False
```

Bu işlev string için ve çoğu Python koleksiyonu türü için True döndürür.

Bu işlevi her zaman kullandığımız bir yer, birden çok türde girdiyi kabul edebilen işlevler yazmaktır. Yaygın bir durum, her tür diziyi (liste, tuple, ndarray) veya hatta bir yineleyiciyi kabul edebilen bir işlev yazmaktır. Önce nesnenin bir liste (veya bir NumPy array) olup olmadığını kontrol edebilir ve değilse, onu bir list olacak şekilde dönüştürebilirsiniz:

```
[34]: if not isinstance(x, list) and isiterable(x):  
      x = list(x)
```

Imports

Python'da bir modül, Python kodunu içeren .py uzantılı bir dosyadır. Aşağıdaki modüle sahip olduğumuzu varsayalım:

```
1 # some_module.py
2
3 PI = 3.14159
4 def f(x):
5     return x+2
6
7 def g(a, b):
8     return a+b
9
```

some_module.py içinde tanımlanan değişkenlere ve fonksiyonlara aynı dizindeki başka bir dosyadan erişmek istersek şunları yapabiliriz:

```
[15]: import some_module
      result = some_module.f(5)
      pi = some_module.PI

      print("result:",result,
          "\npi: ",pi)

      result: 7
      pi:      3.14159
```

Veya eşdeğer olarak:

```
[17]: from some_module import f, g, PI
      result = g(5, PI)
      print(result)

      8.14159
```

as anahtar sözcüğünü kullanarak içe aktarmalara farklı değişken adları verebilirsiniz:

```
[19]: import some_module as sm
      from some_module import PI as pi, g as gf

      r1 = sm.f(pi)
      r2 = gf(6, pi)
      print(r1, "---", r2)

      5.14159 --- 9.14159
```

Binary operators and comparisons

İkili matematik işlemlerinin ve karşılaştırmalarının çoğu beklediğiniz gibidir:

```
[20]: 5-7
```

```
[20]: -2
```

```
[21]: 12+21.5
```

```
[21]: 33.5
```

```
[22]: 5 <= 2
```

```
[22]: False
```

İki referansın aynı nesneye atıfta bulunup bulunmadığını kontrol etmek için `is` anahtar sözcüğünü kullanın. `is not`, iki nesnenin aynı olmadığını kontrol etmek istiyorsanız da mükemmel bir şekilde geçerlidir:

```
[23]: a = [1, 2, 3]
```

```
[24]: b=a
```

```
[25]: c = list(a)
```

```
[26]: a is b
```

```
[26]: True
```

```
[27]: a is not c
```

```
[27]: True
```

Liste her zaman yeni bir Python listesi (yani bir kopya) oluşturduğundan, `c`'nin `a`'dan farklı olduğundan emin olabiliriz. `is` ile karşılaştırmak `==` operatörü ile aynı şey değildir, çünkü bu durumda elimizde:

```
[35]: a == c
```

```
[35]: True
```

`is` ve `is not`'ın çok yaygın bir kullanımı, bir değişkenin `None` olup olmadığını kontrol etmektir, çünkü `None`'ın yalnızca bir örneği vardır:

```
[36]: a = None
      a is None
```

```
[36]: True
```

Binary Operators

Operation	Description
<code>a + b</code>	Add a and b
<code>a - b</code>	Subtract b from a
<code>a * b</code>	Multiply a by b
<code>a / b</code>	Divide a by b
<code>a // b</code>	Floor-divide a by b, dropping any fractional remainder
<code>a ** b</code>	Raise a to the b power
<code>a & b</code>	True if both a and b are True; for integers, take the bitwise AND
<code>a b</code>	True if either a or b is True; for integers, take the bitwise OR
<code>a ^ b</code>	For booleans, True if a or b is True, but not both; for integers, take the bitwise EXCLUSIVE-OR
<code>a == b</code>	True if a equals b
<code>a != b</code>	True if a is not equal to b
<code>a <= b, a < b</code>	True if a is less than (less than or equal) to b
<code>a > b, a >= b</code>	True if a is greater than (greater than or equal) to b
<code>a is b</code>	True if a and b reference the same Python object
<code>a is not b</code>	True if a and b reference different Python objects

Mutable and immutable objects

Python'daki listeler, dicts, NumPy dizileri ve çoğu kullanıcı tanımlı türler (classlar) gibi nesnelerin çoğu değiştirilebilir. Bu, içerdikleri nesne veya değerlerin değiştirilebileceği anlamına gelir:

```
[37]: a_list = ["foo", 2, [4, 5]]
```

```
[38]: a_list[2] = (3, 4)
```

```
[39]: a_list
```

```
[39]: ['foo', 2, (3, 4)]
```

String'ler ve tuple'lar gibi diğerleri değişmezdir:

```
[40]: a_tuple = (3, 5, (4, 5))
```

```
[41]: a_tuple[1] = "four"
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-41-cd2a018a7529> in <module>
----> 1 a_tuple[1] = "four"

TypeError: 'tuple' object does not support item assignment
```

Unutma ki, bir nesneyi değiştirebilmen, her zaman yapman gerektiği anlamına gelmez. Bu tür eylemler yan etkiler olarak bilinir. Örneğin, bir işlevi yazarken, herhangi bir yan etki, işlevin dokümantasyonunda veya yorumlarında kullanıcıya açıkça bildirilmelidir. Mümkünse, değişken nesneler olsa bile, yan etkilerden kaçınmayı ve değişmezliği(*immutability*) desteklemeyi tavsiye ederim.

Scalar Types

Python, standart kütüphanesiyle birlikte sayısal verileri, dizeleri, boolean(True veya False) değerleri ve tarih ve saati işlemek için küçük bir yerleşik tür kümesine sahiptir. Bu "Single Value" türlerine bazen skaler türler denir ve bu kitapta bunlardan scalars olarak bahsediyoruz. Tarih ve saat yönetimi, standart kütüphanede *datetime* modülü tarafından sağlandığı için ayrı ayrı tartışılacaktır.

Table 2-4. Standard Python scalar types

Type	Description
None	The Python "null" value (only one instance of the None object exists)
str	String type; holds Unicode (UTF-8 encoded) strings
bytes	Raw ASCII bytes (or Unicode encoded as bytes)
float	Double-precision (64-bit) floating-point number (note there is no separate double type)
bool	A True or False value
int	Arbitrary precision signed integer

Numeric Types

Sayılar için birincil Python türleri **int** ve **float**'tır. Bir int isteğe bağlı olarak büyük sayıları depolayabilir:

```
[1]: ival = 17239871
[2]: ival ** 6
[2]: 26254519291092456596965462913230729701102721
```

Kayan noktalı sayılar, Python **float** türüyle temsil edilir. Kaputun altında her biri çift hassasiyetli (double-precision) (64 bit) bir değerdir. Ayrıca bilimsel gösterimle de ifade edilebilirler:

```
[3]: fval = 7.243
[4]: fval2 = 6.78e-5
```

Tam sayı ile sonuçlanmayan integer bölümü her zaman bir floating-number verir:

```
[5]: 3 / 2
[5]: 1.5
```

C-stili tamsayı bölme elde etmek için (sonuç tam sayı değilse kesirli bölümü bırakır), kat bölme operatörünü kullanın "//":

```
[6]: 3 // 2
```

```
[6]: 1
```

Strings

Birçok kişi Python'u güçlü ve esnek yerleşik string işleme yetenekleri için kullanır. Tek ' veya çift tırnak " işaretlerini kullanarak string değişmezleri yazabilirsiniz :

```
[7]: a = 'one way of writing a string'
     b = "another way"
```

Satır sonu içeren çok satırlı dizeler için, "" veya """" olmak üzere üçlü tırnak kullanabilirsiniz:

```
[8]: c = """
     This is a longer string that
     spans multiple lines
     """
```

Bu c dizesinin aslında dört satırlık metin içermesi sizi şaşırtabilir; """" ve sonraki satırlar dizeye dahil edilir. Yeni satır karakterlerini **count** metodu ile sayabiliriz:

```
[9]: c.count("\n")
```

```
[9]: 3
```

Python stringleri değişmezdir; bir stringi değiştiremezsiniz:

```
[10]: a = 'this is a string'
```

```
[11]: a[10] = 'f'
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-11-2151a30ed055> in <module>
----> 1 a[10] = 'f'

TypeError: 'str' object does not support item assignment
```

```
[13]: b = a.replace('string', 'longer string')
     b
```

```
[13]: 'this is a longer string'
```

Bu işlemden sonra, a değişkeni değiştirilmez:


```
[14]: a
```

```
[14]: 'this is a string'
```

Birçok Python nesnesi, **str** işlevi kullanılarak bir dizeye dönüştürülebilir:

```
[15]: a = 5.6
```

```
[17]: s = str(a)
      print(s)
      print(type(s))

      5.6
      <class 'str'>
```

String'ler, Unicode karakterlerinin bir dizisidir ve bu nedenle, listeler ve tuple'lar gibi diğer diziler gibi ele alınabilir (sonraki bölümde daha ayrıntılı olarak inceleyeceğiz):

```
[18]: s = "python"
```

```
[19]: list(s)
```

```
[19]: ['p', 'y', 't', 'h', 'o', 'n']
```

```
[20]: s[:3]
```

```
[20]: 'pyt'
```

S[:3] sözdizimi **slicing** olarak adlandırılır ve birçok Python dizisi türü için uygulanır.

Ters eğik çizgi karakteri **** bir çıkış karakteridir, yani satırsonu **\n** veya Unicode karakterleri gibi özel karakterleri belirtmek için kullanılır. Ters eğik çizgi içeren bir string yazmak için onlardan kaçmanız gerekir:

```
[23]: s = "12\\34"
      print(s)

      12\34
```

Çok fazla ters eğik çizgi içeren bir dizeniz varsa ve hiçbir özel karakteriniz yoksa, bunu biraz can sıkıcı bulabilirsiniz. Neyse ki, dizenin başındaki alıntıya **r** ile başlayabilirsiniz, bu da karakterlerin şu şekilde yorumlanması gerektiği anlamına gelir:

```
[28]: s = r'this\has\no\special\characters'
```

```
[29]: s
```

```
[29]: 'this\\has\\no\\special\\characters'
```

r, **raw** (ham) anlamına gelir.

İki dizeyi birbirine eklemek onları birleştirir ve yeni bir dize oluşturur:

```
[30]: a = 'this is the first half '  
      b = 'and this is the second half'  
      a + b
```

```
[30]: 'this is the first half and this is the second half'
```

String şablonlama(templating) veya biçimlendirme(formating) başka bir önemli konudur. Python 3'ün gelişiyile bunu yapmanın yollarının sayısı arttı ve burada ana arayüzlerden birinin mekanizmasını kısaca açıklayacağım. String nesneleri, biçimlendirilmiş bağımsız değişkenleri dizeye ikame etmek için kullanılabilen ve yeni bir dize üreten bir **format** metoduna sahiptir:

```
[31]: template = '{0:.2f} {1:s} are worth US${2:d}'
```

Bu string'de,

- **{0:.2f}**, ilk bağımsız değişkeni iki ondalık basamaklı bir **float** sayı olarak biçimlendirmek anlamına gelir.
- **{1:s}**, ikinci bağımsız değişkeni bir **string** olarak biçimlendirmek anlamına gelir.
- **{2:d}**, üçüncü bağımsız değişkeni bir **integer** olarak biçimlendirmek anlamına gelir.

Bu format parametreleri için bağımsız değişkenleri değiştirmek için, format metoduna bir dizi bağımsız değişken iletiriz:

```
[31]: template = '{0:.2f} {1:s} are worth US${2:d}'
```

```
[32]: template.format(4.5560, 'Argentine Pesos', 1)
```

```
[32]: '4.56 Argentine Pesos are worth US$1'
```

String biçimlendirme derin bir konudur; Sonuç stringinde değerlerin nasıl biçimlendirildiğini kontrol etmek için birden fazla yöntem ve çok sayıda seçenek ve ince ayar vardır.

Veri analizi ile ilgili olduğu için genel string işlemeyi Chapter 8'de daha ayrıntılı olarak tartışıyoruz.

Bytes and Unicode

Modern Python'da (yani, Python 3.0 ve üzeri), *Unicode*, ASCII ve ASCII olmayan metnin daha tutarlı işlenmesini sağlamak için birinci sınıf dize türü haline geldi. Python'un eski sürümlerinde, dizelerin tümü, herhangi bir açık Unicode kodlaması olmayan byte'lardı. Karakter kodlamasını bildiğinizi varsayarak Unicode'a dönüştürebilirsiniz. Bir örneğe bakalım:

```
[33]: val = "español"
```

```
[34]: val
```

```
[34]: 'español'
```

Encode yöntemini kullanarak bu Unicode dizesini UTF-8 bytes gösterimine dönüştürebiliriz:

```
[35]: val_utf8 = val.encode('utf-8')
```

```
[36]: val_utf8
```

```
[36]: b'espa\xc3\xblol'
```

```
[37]: type(val_utf8)
```

```
[37]: bytes
```

Herhangi bir kodlama için UTF-8 kullanılması tercih edilmekle birlikte, geçmiş nedenlerden dolayı herhangi bir sayıda farklı kodlamada verilerle karşılaşabilirsiniz:

```
[38]: val.encode('latin1')
```

```
[38]: b'espa\xflol'
```

```
[39]: val.encode('utf-16')
```

```
[39]: b'\xff\xfe\x0s\x0p\x0a\x0\xfl\x0o\x0l\x0'
```

```
[40]: val.encode('utf-16le')
```

```
[40]: b'e\x0s\x0p\x0a\x0\xfl\x0o\x0l\x0'
```

Dosyalar ile çalışma bağlamında bayt nesneleriyle karşılaşmak en yaygın olanıdır, burada tüm verilerin Unicode dizelerine örtük olarak kodunun çözülmesi istenmeyebilir.

Nadiren bunu yapmanız gerekse de, bir dizenin önüne b ile kendi byte değişmezlerinizi tanımlayabilirsiniz:

```
[41]: bytes_val = b"this is bytes"
[42]: bytes_val
[42]: b'this is bytes'
[43]: decoded = bytes_val.decode("utf8")
[44]: decoded # this is str (Unicode) now
[44]: 'this is bytes'
```

Booleans

Python'daki iki boolean değeri **True** ve **False** olarak yazılır. Karşılaştırmalar ve diğer koşullu ifadeler True veya False olarak değerlendirilir. Boolean değerleri, **and** ve **or** anahtar sözcükleriyle birleştirilir:

```
[45]: True and True
[45]: True
[46]: False or True
[46]: True
```

Type Casting

Str, bool, int ve float türleri de bu türlere değer atamak için kullanılabilen işlevlerdir:

```
[47]: s = '3.14159'
[48]: fval = float(s)
[49]: type(fval)
[49]: float
[50]: int(fval)
[50]: 3
[51]: bool(fval)
[51]: True
[55]: bool(0)
[55]: False
```

None

None, Python boş değer türüdür. Bir işlev açıkça bir değer döndürmezse, örtük olarak None döndürür:

```
[56]: a = None
```

```
[57]: a is None
```

```
[57]: True
```

```
[58]: b = 5
```

```
[59]: b is not None
```

```
[59]: True
```

None ayrıca fonksiyon bağımsız değişkenleri için ortak bir varsayılan değerdir:

```
[60]: def add_and_maybe_multiply(a, b, c=None):  
      result = a+b  
  
      if c is not None:  
          result = result*c  
  
      return result
```

```
[61]: add_and_maybe_multiply(3, 5)
```

```
[61]: 8
```

```
[62]: add_and_maybe_multiply(3, 5, 2)
```

```
[62]: 16
```

Teknik bir nokta olsa da, None'un yalnızca ayrılmış bir anahtar kelime değil, aynı zamanda benzersiz bir NoneType örneği olduğunu da unutmamak gerekir:

```
[63]: type(None)
```

```
[63]: NoneType
```

Dates and times

Yerleşik Python **datetime** modülü, datetime, date ve time türlerini sağlar. Datetime türü, tahmin edebileceğiniz gibi, date ve time'da depolanan bilgileri birleştirir ve en yaygın kullanılanıdır:

```
[64]: from datetime import datetime, date, time
```

```
[65]: dt = datetime(2011, 10, 29, 20, 30, 21)
```

```
[66]: dt.day
```

```
[66]: 29
```

```
[67]: dt.minute
```

```
[67]: 30
```

```
[68]: dt.month
```

```
[68]: 10
```

Bir datetime örneği verildiğinde, eşdeğer date ve time nesnelerini, aynı adı taşıyan datetime üzerindeki yöntemleri çağırarak ayıklayabilirsiniz:

```
[69]: dt.date()
```

```
[69]: datetime.date(2011, 10, 29)
```

```
[70]: dt.time()
```

```
[70]: datetime.time(20, 30, 21)
```

Strftime yöntemi bir datetime'ı bir dize olarak biçimlendirir:

```
[72]: dt.strftime('%d/%m/%Y %H:%M')
```

```
[72]: '29/10/2011 20:30'
```

Dizeler **strptime** işlevi ile datetime nesnelere dönüştürülebilir (ayrıştırılabilir):

```
[73]: datetime.strptime('20091031', '%Y%m%d')
```

```
[73]: datetime.datetime(2009, 10, 31, 0, 0)
```

Zaman serisi verilerini toplarken veya başka bir şekilde gruplandırırken, bazen dakika ve saniye alanlarını sıfırla değiştirmek gibi bir dizi datetime'a ait zaman alanlarını değiştirmek yararlı olacaktır:

```
[74]: dt.replace(minute=0, second=0)
```

```
[74]: datetime.datetime(2011, 10, 29, 20, 0)
```

datetime.datetime değişmez bir tür olduğundan, bunun gibi yöntemler her zaman yeni nesneler üretir.

İki datetime nesnesinin farkı bir datetime.timedelta türü oluşturur:

```
[75]: dt2 = datetime(2011, 11, 15, 22, 30)
```

```
[76]: delta = dt2 - dt
```

```
[77]: delta
```

```
[77]: datetime.timedelta(days=17, seconds=7179)
```

```
[79]: type(delta)
```

```
[79]: datetime.timedelta
```

Çıktı timedelta(17, 7179) timedelta'nın 17 gün ve 7.179 saniyelik bir ofseti kodladığını belirtir.

Bir datetime'a bir timedelta eklemek yeni bir shifted(kaydırılmış) datetime oluşturur:

```
[80]: dt
```

```
[80]: datetime.datetime(2011, 10, 29, 20, 30, 21)
```

```
[81]: dt + delta
```

```
[81]: datetime.datetime(2011, 11, 15, 22, 30)
```

Datetime Format Specification

Type	Description
%Y	Four-digit year
%y	Two-digit year
%m	Two-digit month [01, 12]
%d	Two-digit day [01, 31]
%H	Hour (24-hour clock) [00, 23]
%I	Hour (12-hour clock) [01, 12]
%M	Two-digit minute [00, 59]
%S	Second [00, 61] (seconds 60, 61 account for leap seconds)
%w	Weekday as integer [0 (Sunday), 6]
%U	Week number of the year [00, 53]; Sunday is considered the first day of the week, and days before the first Sunday of the year are "week 0"
%W	Week number of the year [00, 53]; Monday is considered the first day of the week, and days before the first Monday of the year are "week 0"
%z	UTC time zone offset as +HHMM or -HHMM; empty if time zone naive
%F	Shortcut for %Y-%m-%d (e.g., 2012-4-18)
%D	Shortcut for %m/%d/%y (e.g., 04/18/12)

Control Flow

Python, koşullu mantık, döngüler ve diğer programlama dillerinde bulunan diğer standart **Control Flow** (kontrol akışı) kavramları için birkaç yerleşik anahtar kelimeye sahiptir.

if, elif and else

if ifadesi, en iyi bilinen kontrol akışı ifadesi türlerinden biridir. True ise, aşağıdaki bloktaki kodu değerlendiren bir koşulu kontrol eder:

```
[83]: x = -1
      if x < 0:
          print("It's negative")
      It's negative
```

Bir if ifadesini isteğe bağlı olarak bir veya daha fazla elif bloğu ve tüm koşullar False ise bir tümünü yakalama, else bloğu izleyebilir:


```
[86]: x = 7

if x < 0:
    print("It's negative")
elif x == 0:
    print('Equal to zero')
elif 0 < x < 5:
    print('Positive but smaller than 5')
else:
    print('Positive and larger than or equal to 5')

Positive and larger than or equal to 5
```

Koşullardan herhangi biri True ise, başka bir elif veya başka bloğa ulaşılmayacaktır. and veya or kullanılarak bileşik bir koşulda, koşullar soldan sağa değerlendirilir ve kısa devre yapar:

```
[87]: a = 5; b = 7; c = 8; d = 4

[88]: if a < b or c > d:
        print("made it")

made it
```

Bu örnekte, $c > d$ karşılaştırması hiçbir zaman değerlendirilmez çünkü ilk karşılaştırma True idi.

Karşılaştırmaları zincirlemek de mümkündür:

```
[89]: 4 > 3 > 2 > 1

[89]: True
```

for loops

for döngüler, bir koleksiyon üzerinde (bir liste veya tuple gibi) veya bir yineleyici üzerinde yineleme yapmak içindir. Bir for döngüsü için standart sözdizimi şöyledir:

```
[90]: for value in collection:
        # do something with value
```

Continue anahtar sözcüğünü kullanarak bloğun geri kalanını atlayarak bir sonraki yinelemeye bir for döngüsü ilerletebilirsiniz. Bir listedeki tam sayıları toplayan ve None değerlerini atlayan bu kodu göz önünde bulundurun:

```
[91]: sequence = [1, 2, None, 4, None, 5]
      total = 0
      for value in sequence:
          if value is None:
              continue
          total += value
```

```
[92]: total
```

```
[92]: 12
```

Break anahtar sözcüğü ile bir for döngüsünden tamamen çıkılabilir. Bu kod, 5'e ulaşılan kadar listenin öğelerini toplar:

```
[93]: sequence = [1, 2, 0, 4, 6, 5, 2, 1]
      total_until_5 = 0
      for value in sequence:
          if value == 5:
              break #for döngüsünden çıktı
          total_until_5 += value
      print(total_until_5)
```

```
13
```

Break anahtar sözcüğü yalnızca en içteki for döngüsünü sonlandırır; herhangi bir dış for döngüsü çalışmaya devam edecek:

```
[94]: for i in range(4):
      for j in range(4):
          if j > i:
              break
          print((i, j))
```

```
(0, 0)
(1, 0)
(1, 1)
(2, 0)
(2, 1)
(2, 2)
(3, 0)
(3, 1)
(3, 2)
(3, 3)
```

Daha ayrıntılı olarak göreceğimiz gibi, koleksiyondaki veya yineleyicideki öğeler dizilerse (diyelim ki tuples veya listeler), for döngüsü deyiminde değişkenler halinde kolayca açılabilirler:

```
[ ]: for a, b, c in iterator:
      #do something
```

while loops

While döngüsü, koşul False olarak değerlendirilene veya döngü açıkça break ile sonlandırılana kadar yürütülecek bir koşul ve kod bloğunu belirtir:

```
[95]: x = 256
      total = 0
      while x > 0:
          if total > 500:
              break
          total += x
          x = x // 2
```

```
[96]: total
```

```
[96]: 504
```

pass

pass, Python'daki "işlemsiz" ifadesidir. Herhangi bir işlemin yapılmayacağı bloklarda (veya henüz uygulanmamış kod için bir yer tutucu olarak) kullanılabilir; yalnızca Python blokları sınırlamak için boşluk kullandığından gereklidir:

```
[99]: x=0
      if x < 0:
          print("negative!")
      elif x == 0:
          #TODO: put something smart here
          pass
      else:
          print("positive!")
```

range

Range işlevi, eşit aralıklı tamsayılar dizisi veren bir yineleyici döndürür:

```
[102]: range(18)
```

```
[102]: range(0, 18)
```

```
[103]: range(0, 10)
```

```
[103]: range(0, 10)
```

```
[104]: list(range(10))
```

```
[104]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Hem bir başlangıç, bitiş ve adım (negatif olabilir) verilebilir:

```
[105]: list(range(0, 20, 2))
```

```
[105]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

```
[106]: list(range(5, 0, -1))
```

```
[106]: [5, 4, 3, 2, 1]
```

Gördüğünüz gibi range, uç noktaya kadar olan ancak dahil olmayan tam sayılar üretir. range'in yaygın bir kullanımı dizine göre dizileri yinelemektir:

```
[107]: seq = [1, 2, 3, 4]
      for i in range(len(seq)):
          val = seq[i]
```

Range tarafından üretilen tüm tam sayıları başka bir veri yapısında saklamak için list gibi işlevleri kullanabilirsiniz, ancak genellikle varsayılan yineleyici formu istediğiniz şey olacaktır. Bu snippet, 0 ile 99.999 arasındaki 3 veya 5'in katları olan tüm sayıları toplar:

```
[109]: sum = 0
      for i in range(100000):
          # % is the modulo operator
          if i % 3 == 0 or i % 5 == 0:
              sum += i

      print(sum)

2333316668
```

Oluşturulan aralık keyfi olarak büyük olabilse de, herhangi bir zamanda bellek kullanımı çok küçük olabilir.

Ternary Expressions

Python'daki üçlü ifade(ternary expressions), değer üreten bir if-else bloğunu tek bir satır veya ifadede birleştirmenize izin verir. Python'da bunun sözdizimi şöyledir:

value = true-expr if condition else false-expr

Burada, true-expr ve false-expr herhangi bir Python ifadesi olabilir. Daha ayrıntılı olanla aynı etkiye sahiptir:

if condition:

value = true-expr

else:

value = false-expr

Bu daha somut bir örnek:

```
[110]: x = 5
       "Non-Negative" if x >= 0 else "Negative"
[110]: 'Non-Negative'
```

if-else bloklarında olduğu gibi, ifadelerden sadece biri çalıştırılacaktır. Bu nedenle, üçlü ifadenin "if" ve "else" tarafları maliyetli hesaplamalar içerebilir, ancak yalnızca gerçek dal değerlendirilir.

Kodunuzu yoğunlaştırmak için her zaman üçlü ifadeler kullanmak cazip gelse de, koşulun yanı sıra doğru ve yanlış ifadeler çok karmaşıksa okunabilirliği feda edebileceğinizi unutmayın.

Chapter 3

Built-in Data Structures, Functions, and Files