

## Contents

1	Importing and Exporting Data	2
1.1	9.1 Importing Data using pandas	2
1.2	9.1.1 CSV and other formatted text files	2
1.3	Reading Data	3
1.4	CSV	3
1.5	Excel	5
1.6	Databases	7
2	File System Operations	8
2.1	22.1 Changing the Working Directory	8
2.2	22.2 Creating and Deleting Directories	8
2.3	22.3 Listing the Contents of a Directory	9
2.4	22.4 Copying, Moving and Deleting Files	9
2.5	Mixing Numerical Indexing with Scalar Selection	11
3	Custom Function and Modules	12
3.1	Functions	12
3.1.1	Default Values	14
3.2	Anonymous Functions	15
3.3	Python Coding Conventions	15
4	Flow Control, Loops and Exception Handling	16
4.1	13.2 if . . . elif . . . else	16
4.2	break	18
4.2.1	13.4.2 continue	18
5	Graphics	19
5.1	matplotlib	19
5.2	seaborn	19
5.3	Histograms	19
5.4	Adding a Title and Legend	20
5.5	Plotting	21
5.5.1	isnull and notnull	21
5.5.2	ix	21
5.5.3	plot	21

# 1 Importing and Exporting Data

## 1.1 9.1 Importing Data using pandas

Pandas is an increasingly important component of the Python scientific stack, and a complete discussion of its main features is included in Chapter 17.

All of the data readers in pandas load data into a pandas DataFrame (see Section 17.1.2), and so these examples all make use of the `values` property to extract a NumPy array.

In practice, the DataFrame is much more useful since it includes useful information such as column names read from the data source. In addition to the three formats presented here, pandas can also read json, SQL, html tables or from the clipboard, which is particularly useful for interactive work since virtually any source that can be copied to the clipboard can be imported.

### 1.2 9.1.1 CSV and other formatted text files

Comma-separated value (CSV) files can be read using `read_csv`. When the CSV file contains mixed data, the default behavior will read the file into an array with an object data type, and so further processing is usually required to extract the individual series.

```
>>> from pandas import read_csv
>>> csv_data = read_csv('FTSE_1984_2012.csv')
>>> csv_data = csv_data.values
>>> csv_data[:4]
array([[ '2012-02-15', 5899.9, 5923.8, 5880.6, 5892.2, 801550000L, 5892.2],
       [ '2012-02-14', 5905.7, 5920.6, 5877.2, 5899.9, 832567200L, 5899.9],
       [ '2012-02-13', 5852.4, 5920.1, 5852.4, 5905.7, 643543000L, 5905.7],
       [ '2012-02-10', 5895.5, 5895.5, 5839.9, 5852.4, 948790200L, 5852.4]], dtype=object)
>>> open = csv_data[:,1]
```

### 1.3 Reading Data

To create a DataFrame out of common Python data structures, we can pass a dictionary of lists to the DataFrame constructor.

Using the `columns` parameter allows us to tell the constructor how we'd like the columns ordered. By default, the DataFrame constructor will order the columns alphabetically (though this isn't the case when reading from a file - more on that next).

```
In [17]:
data = {'year': [2010, 2011, 2012, 2011, 2012, 2010, 2011, 2012],
        'team': ['Bears', 'Bears', 'Bears', 'Packers', 'Packers', 'Lions', 'Lions', 'Lions'],
        'wins': [11, 8, 10, 15, 11, 6, 10, 4],
        'losses': [5, 8, 6, 1, 5, 10, 6, 12]}
football = pd.DataFrame(data, columns=['year', 'team', 'wins', 'losses'])
print football
```

	year	team	wins	losses
0	2010	Bears	11	5
1	2011	Bears	8	8
2	2012	Bears	10	6
3	2011	Packers	15	1
4	2012	Packers	11	5
5	2010	Lions	6	10
6	2011	Lions	10	6
7	2012	Lions	4	12

Much more often, you'll have a dataset you want to read into a DataFrame. Let's go through several common ways of doing so.

### 1.4 CSV

Reading a CSV is as simple as calling the `read_csv` function. By default, the `read_csv` function expects the column separator to be a comma, but you can change that using the `sep` parameter.

```
%cd ~/Dropbox/tutorials/pandas/
/Users/greda/Dropbox/tutorials/pandas

# Source: baseball-reference.com/players/r/riverma01.shtml
```

## Data Analysis with Python

```
!head -n 5 mariano-rivera.csv
Year, Age, Tm, Lg, W, L, W-L%, ERA, G, GS, GF, CG, SHO, SV, IP, H, R, ER, HR, BB, IBB, SO, HBP, BK, WP, BF, ERA+,
1995, 25, NYY, AL, 5, 3, .625, 5.51, 19, 10, 2, 0, 0, 0, 67.0, 71, 43, 41, 11, 30, 0, 51, 2, 1, 0, 301, 84, 1.507,
1996, 26, NYY, AL, 8, 3, .727, 2.09, 61, 0, 14, 0, 0, 5, 107.2, 73, 25, 25, 1, 34, 3, 130, 2, 0, 1, 425, 240, 0.99,
1997, 27, NYY, AL, 6, 4, .600, 1.88, 66, 0, 56, 0, 0, 43, 71.2, 65, 17, 15, 5, 20, 6, 68, 0, 0, 2, 301, 239, 1.186,
1998, 28, NYY, AL, 3, 0, 1.000, 1.91, 54, 0, 49, 0, 0, 36, 61.1, 48, 13, 13, 3, 17, 1, 36, 1, 0, 0, 246, 233, 1.06

In [20]:
from_csv = pd.read_csv('mariano-rivera.csv')
from_csv.head()
```

```
Year Age Tm Lg W L W-L% ERA G GS GF CG SHO SV IP H R ER HR BB IBB SO
HBP BK WP BF ERA+ WHIP H/9 HR/9 BB/9 SO/9 SO/BB Awards
0 1995 25 NYY AL 5 3 0.625 5.51 19 10 2 0 0 67.0 71
43 41 11 30 0 51 2 1 0 301 84 1.507 9.5 1.5 4.0 6.9
1.70 NaN
1 1996 26 NYY AL 8 3 0.727 2.09 61 0 14 0 0 5 107.2
73 25 25 1 34 3 130 2 0 1 425 240 0.994 6.1 0.1 2.8
10.9 3.82 CYA-3MVP-12
2 1997 27 NYY AL 6 4 0.600 1.88 66 0 56 0 0 43 71.2
65 17 15 5 20 6 68 0 0 2 301 239 1.186 8.2 0.6 2.5
8.5 3.40 ASMVP-25
3 1998 28 NYY AL 3 0 1.000 1.91 54 0 49 0 0 36 61.1
48 13 13 3 17 1 36 1 0 0 246 233 1.060 7.0 0.4 2.5
5.3 2.12 NaN
4 1999 29 NYY AL 4 3 0.571 1.83 66 0 63 0 0 45 69.0
43 15 14 2 18 3 52 3 1 2 268 257 0.884 5.6 0.3 2.3
6.8 2.89 ASCYA-3MVP-14
```

Our file had headers, which the function inferred upon reading in the file. Had we wanted to be more explicit, we could have passed `header=None` to the function along with a list of column names to use:

```
# Source: pro-football-reference.com/players/M/MannPe00/touchdowns/passing/2012/
!head -n 5 peyton-passing-TDs-2012.csv
1,1,2012-09-09,DEN,,PIT,W 31-19,3,71,Demaryius Thomas,Trail 7-13,Lead 14-13*
2,1,2012-09-09,DEN,,PIT,W 31-19,4,1,Jacob Tamme,Trail 14-19,Lead 22-19*
```

```

3,2,2012-09-17,DEN,@,ATL,L 21-27,2,17,Demaryius Thomas,Trail 0-20,Trail 7-20
4,3,2012-09-23,DEN,,HOU,L 25-31,4,38,Brandon Stokley,Trail 11-31,Trail 18-31
5,3,2012-09-23,DEN,,HOU,L 25-31,4,6,Joel Dreessen,Trail 18-31,Trail 25-31

In [22]:
cols = ['num', 'game', 'date', 'team', 'home_away', 'opponent',
        'result', 'quarter', 'distance', 'receiver', 'score_before',
        'score_after']
no_headers = pd.read_csv('peyton-passing-TDs-2012.csv', sep=',', header=None,
                        names=cols)
no_headers.head()
Out[22]:
num game date team home_away opponent result quarter distance
receiver score_before score_after
0  1  1  2012-09-09  DEN  NaN  PIT  W 31-19  3  71  Demaryius Thomas
Trail 7-13  Lead 14-13*
1  2  1  2012-09-09  DEN  NaN  PIT  W 31-19  4  1  Jacob Tamme
Trail 14-19  Lead 22-19*
2  3  2  2012-09-17  DEN  @  ATL  L 21-27  2  17  Demaryius Thomas
Trail 0-20  Trail 7-20
3  4  3  2012-09-23  DEN  NaN  HOU  L 25-31  4  38  Brandon Stokley
Trail 11-31  Trail 18-31
4  5  3  2012-09-23  DEN  NaN  HOU  L 25-31  4  6  Joel Dreessen
Trail 18-31  Trail 25-31

```

pandas various reader functions have many parameters allowing you to do things like skipping lines of the file, parsing dates, or specifying how to handle NA/NULL datapoints.

There's also a set of writer functions for writing to a variety of formats (CSVs, HTML tables, JSON). They function exactly as you'd expect and are typically called `to_format`:

```
my_dataframe.to_csv('path_to_file.csv')
```

Take a look at the IO documentation to familiarize yourself with file reading/writing functionality.

## 1.5 Excel

Know who hates VBA? Me. I bet you do, too. Thankfully, pandas allows you to read and write Excel files, so you can easily read from Excel, write your code in Python, and

## Data Analysis with Python

then write back out to Excel - no need for VBA.

Reading Excel files requires the xlrd library. You can install it via pip (`pip install xlrd`).

Let's first write a DataFrame to Excel.

```
# this is the DataFrame we created from a dictionary earlier
print football.head()
```

```
year team wins losses
0 2010 Bears 11 5
1 2011 Bears 8 8
2 2012 Bears 10 6
3 2011 Packers 15 1
4 2012 Packers 11 5
```

```
# since our index on the football DataFrame is meaningless, let's not write it
football.to_excel('football.xlsx', index=False)

!ls -l *.xlsx
-rw-r--r--  1 greda  staff  5618 Oct 26 00:44 football.xlsx

# delete the DataFrame
del football

# read from Excel
football = pd.read_excel('football.xlsx', 'sheet1')
print football
```

```
year team wins losses
0 2010 Bears 11 5
1 2011 Bears 8 8
2 2012 Bears 10 6
3 2011 Packers 15 1
4 2012 Packers 11 5
5 2010 Lions 6 10
6 2011 Lions 10 6
7 2012 Lions 4 12
```

## 1.6 Databases

pandas also has some support for reading/writing DataFrames directly from/to a database [docs]. You'll typically just need to pass a connection object to the `read_frame` or `write_frame` functions within the `pandas.io` module.

Note that `write_frame` executes as a series of INSERT INTO statements and thus trades speed for simplicity. If you're writing a large DataFrame to a database, it might be quicker to write the DataFrame to CSV and load that directly using the database's file import arguments.

```
from pandas.io import sql
import sqlite3

conn = sqlite3.connect('/Users/greda/Dropbox/gregreda.com/_code/towed')
query = "SELECT * FROM towed WHERE make = 'FORD';"

results = sql.read_frame(query, con=conn)
print results.head()
```

	tow_date	make	style	model	color	plate	state	towed_address \
0	01/19/2013	FORD	LL		RED	N786361	IL	400 E. Lower Wacker
1	01/19/2013	FORD	4D		GRN	L307211	IL	701 N. Sacramento
2	01/19/2013	FORD	4D		GRY	P576738	IL	701 N. Sacramento
3	01/19/2013	FORD	LL		BLK	N155890	IL	10300 S. Doty
4	01/19/2013	FORD	LL		TAN	H953638	IL	10300 S. Doty

	phone	inventory
0	(312) 744-7550	877040
1	(773) 265-7605	6738005
2	(773) 265-7605	6738001
3	(773) 568-8495	2699210
4	(773) 568-8495	2699209

## 2 File System Operations

Manipulating files and directories is surprisingly useful when undertaking complex projects. The most important file system commands are located in the modules `os` and `shutil`. This chapter assumes that

```
import os
import shutil
```

have been included.

### 2.1 22.1 Changing the Working Directory

The working directory is where files can be created and accessed without any path information. `os.getcwd()` can be used to determine the current working directory, and `os.chdir(path)` can be used to change the working directory, where `path` is a directory, such as `/temp` or `c:`:

temp.1 Alternatively, `path` can be `..` to move up the directory tree.

```
pwd = os.getcwd()
os.chdir('c:\\temp')
os.chdir(r'c:\temp') # Raw string, no need to escape \
os.chdir('c:/temp') # Identical
os.chdir('..') # Walk up the directory tree
os.getcwd() # Now in 'c:\\'
```

### 2.2 22.2 Creating and Deleting Directories

Directories can be created using `os.mkdir(dirname)`, although it must be the case that the higher level directories exist (e.g. to create `/home/username/Python/temp`, it `/home/username/Python` already exists). `os.makedirs(dirname)` works similar to `os.mkdir(dirname)`, except that it will create any higher level directories needed to create the target directory. Empty directories can be deleted using `os.rmdir(dirname)` – if the directory is not empty, an error occurs. `shutil.rmtree(dirname)` works similarly to `os.rmdir(dirname)`, except that it will delete the directory, and any files or other directories contained in the directory.



```
os.mkdir('c:\\temp\\test')
os.makedirs('c:/temp/test/level2/level3') # mkdir will fail
os.rmdir('c:\\temp\\test\\level2\\level3')
shutil.rmtree('c:\\temp\\test') # rmdir fails, since not empty
```

### 2.3 22.3 Listing the Contents of a Directory

The contents of a directory can be retrieved in a list using `os.listdir(dirname)`, or simply `os.listdir('.')` to list the current working directory. The list returned contains all files and directories. `os.path.isdir( name )` can be used to determine whether a value in the list is a directory, and `os.path.isfile(name)` can be used to determine if it is a file. `os.path` contains other useful functions for working with directory listings and file attributes.

```
os.chdir('c:\\temp')
files = os.listdir('.')
for f in files:
    if os.path.isdir(f):
        print(f, ' is a directory.')
    elif os.path.isfile(f):
        print(f, ' is a file.')
    else:
        print(f, ' is a something else.')
```

A more sophisticated listing which accepts wildcards and is similar to `dir` (Windows) and `ls` (Linux) can be constructed using the `glob` module.

```
import glob
files = glob.glob('c:\\temp\\*.txt')
for file in files:
    print(file)
```

### 2.4 22.4 Copying, Moving and Deleting Files

File contents can be copied using `shutil.copy( src , dest )`, `shutil.copy2( src , dest )` or `shutil.copyfile( src , dest )`. These functions are all similar, and the differences are:

- `shutil.copy` will accept either a filename or a directory as `dest`. If a directory is given, the a file is created in the directory with the same name as the original file
- `shutil.copyfile` requires a filename for `dest`.
- `shutil.copy2` is identical to `shutil.copy` except that metadata, such as last access times, is also copied.

Finally, `shutil.copytree( src , dest )` will copy an entire directory tree, starting from the directory `src` to the directory `dest`, which must not exist. `shutil.move( src,dest)` is similar to `shutil.copytree`, except that it moves a file or directory tree to a new location. If preserving file metadata (such as permissions or file

Elements from NumPy arrays can be selected using four methods: scalar selection, slicing, numerical (or list-of-locations) indexing and logical (or Boolean) indexing.

Numerical indexing uses lists or arrays of locations to select elements while logical indexing uses arrays containing Boolean values to select elements.

```
>>> x = 10 * arange(5.0)
>>> x[[0]] # List with 1 element
array([ 0.])
>>> x[[0,2,1]] # List
array([ 0., 20., 10.])
>>> sel = array([4,2,3,1,4,4]) # Array with repetition
>>> x[sel]
array([ 40., 20., 30., 10., 40., 40.])
>>> sel = array([[4,2],[3,1]]) # 2 by 2 array
>>> x[sel] # Selection has same size as sel
array([[ 40., 20.],
       [ 30., 10.]])
>>> sel = array([0.0,1]) # Floating point data
>>> x[sel] # Error
IndexError: arrays used as indices must be of integer (or boolean) type
>>> x[sel.astype(int)] # No error
array([ 10., 20.])
>>> x[0] # Scalar selection, not numerical indexing
1.0
```

```
>>> x = reshape(arange(10.0), (2,5))
>>> x
```

```

array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.]])
>>> sel = array([0,1])
>>> x[sel,sel] # 1-dim arrays, no broadcasting
array([ 0.,  6.])
>>> x[sel, sel+1]
array([ 1.,  7.])
>>> sel_row = array([[0,0],[1,1]])
>>> sel_col = array([[0,1],[0,1]])
>>> x[sel_row,sel_col] # 2 by 2, no broadcasting
array([[ 0.,  1.],
       [ 5.,  6.]])
>>> sel_row = array([[0],[1]])
>>> sel_col = array([[0,1]])
>>> x[sel_row,sel_col] # 2 by 1 and 1 by 2 - difference shapes, broadcasted as 2 by 2
array([[ 0.,  1.],
       [ 5.,  6.]])

```

## 2.5 Mixing Numerical Indexing with Scalar Selection

NumPy permits using different types of indexing in the same expression. Mixing numerical indexing with scalar selection is trivial since any scalar can be broadcast to any array shape.

```

>>> x = array([[1,2],[3,4]])
>>> sel = x <= 3
>>> indices = nonzero(sel)
>>> indices
(array([0, 0, 1], dtype=int64), array([0, 1, 0], dtype=int64))

```

```

>>> x = randn(3)
>>> x
array([-0.5910316 ,  0.51475905,  0.68231135])
>>> argwhere(x<0.6)
array([[0],
       [1]], dtype=int64)

```

```

>>> argwhere(x<-10.0) # Empty array
array([], shape=(0L, 1L), dtype=int64)
>>> x = randn(3,2)
>>> x
array([[ 0.72945913,  1.2135989 ],
       [ 0.74005449, -1.60231553],
       [ 0.16862077,  1.0589899 ]])
>>> argwhere(x<0)
array([[1, 1]], dtype=int64)
>>> argwhere(x<1)
array([[0, 0],
       [1, 0],
       [1, 1],
       [2, 0]], dtype=int64)

```

### 3 Custom Function and Modules

Python supports a wide range of programming styles including procedural (imperative), object oriented and functional. While object oriented programming and functional programming are powerful programming paradigms, especially in large, complex software, procedural is often both easier to understand and a direct representation of a mathematical formula. The basic idea of procedural programming is to produce a function or set of function (generically) of the form:

$$y = f(x).$$

That is, the functions take one or more inputs and produce one or more outputs.

#### 3.1 Functions

Python functions are very simple to declare and can occur in the same file as the main program or a standalone file. Functions are declared using the `def` keyword, and the value produced is returned using the `return` keyword. Consider a simple function which returns the square of the input,  $y = x^2$ .

```

from __future__ import print_function, division

def square(x):
    return x**2
# Call the function
x = 2

```

```
y = square(x)
print(x,y)
```

In this example, the same Python file contains the main program– the final 3 lines – as well as the function. More complex function can be crafted with multiple inputs.

```
from __future__ import print_function, division
def l2distance(x,y):
    return (xy)**
    2
# Call the function
x = 3
y = 10
z = l2distance(x,y)
print(x,y,z)
```

Function can also be defined using NumPy arrays and matrices.

```
from __future__ import print_function, division
import numpy as np
def l2_norm(x,y):
    d = x y
    return np.sqrt(np.dot(d,d))
# Call the function
x = np.random.randn(10)
y = np.random.randn(10)
z = l2_norm(x,y)
print(xy)
print("The L2 distance is ",z)
```

When multiple outputs are returned but only a single variable is available for assignment, all outputs are returned in a tuple. Alternatively, the outputs can be directly assigned when the function is called with the same number of variables as outputs.

```
from __future__ import print_function, division
import numpy as np
def l1_l2_norm(x,y):
    d = x y
    return sum(np.abs(d)),np.sqrt(np.dot(d,d))
# Call the function
x = np.random.randn(10)
y = np.random.randn(10)
# Using 1 output returns a tuple
z = l1_l2_norm(x,y)
print(xy)
print("The L1 distance is ",z[0])
```

```

print("The L2 distance is ",z[1])
# Using 2 output returns the values
l1,l2 = l1_l2_norm(x,y)
print("The L1 distance is ",l1)
print("The L2 distance is ",l2)

```

All of these functions have been placed in the same file as the main program. Placing functions in modules allows for reuse in multiple programs, and will be discussed later in this chapter.

### 3.1.1 Default Values

Default values are set in the function declaration using the syntax `input=default`.

```

from __future__ import print_function, division
import numpy as np
def lp_norm(x,y,p = 2):
    d = x y
    return sum(abs(d)**p)**(1/p)
# Call the function
x = np.random.randn(10)
y = np.random.randn(10)
# Inputs with default values can be ignored
l2 = lp_norm(x,y)
l1 = lp_norm(x,y,1)
print("The l1 and l2 distances are ",l1,l2)
print("Is the default value overridden?", sum(abs(xy))==
l1)

```

Default values should not normally be mutable (e.g. lists or arrays) since they are only initialized the first time the function is called. Subsequent calls will use the same value, which means that the default value could change every time the function is called.

```

from __future__ import print_function, division
import numpy as np

```

### 3.2 Anonymous Functions

Python support anonymous functions using the keyword `lambda`. Anonymous functions are usually encountered when another function expects a function as an input and a simple function will suffice. Anonymous function take the generic form `lambda a,b,c, . . . :code using a,b,c`. The key elements are the keyword `lambda`, a list of comma separated inputs, a colon between the inputs and the actual function code. For example `lambda x,y:x+y` would return the sum of the variables `x` and `y`.

Anonymous functions are simple but useful. For example, when lists containing other lists it isn't directly possible to sort on an arbitrary element of the nested list. Anonymous functions allow sorting through the keyword argument `key` by returning the element Python should use to sort. In this example, a direct call to `sort()` will sort on the first element (first name). Using the anonymous function `lambda x:x[1]` to return the second element of the tuple allows for sorting on the lastname. `lambda x:x[2]` would allow for sorting on the University.

```
>>> nested = [('John','Doe','Oxford'),\
... ('Jane','Dearing','Cambridge'),\
... ('Jerry','Dawn','Harvard')]
>>> nested.sort()
>>> nested
[('Jane', 'Dearing', 'Cambridge'),
 ('Jerry', 'Dawn', 'Harvard'),
 ('John', 'Doe', 'Oxford')]
>>> nested.sort(key=lambda x:x[1])
>>> nested
[('Jerry', 'Dawn', 'Harvard'),
 ('Jane', 'Dearing', 'Cambridge'),
 ('John', 'Doe', 'Oxford')]
```

### 3.3 Python Coding Conventions

There are a number of common practices which can be adopted to produce Python code which looks more like code found in other modules:

1. Use 4 spaces to indent blocks – avoid using tab, except when an editor automatically converts tabs to 4 spaces
2. Avoid more than 4 levels of nesting, if possible
3. Limit lines to 79 characters. The `\` symbol can be used to break long lines

4. Use two blank lines to separate functions, and one to separate logical sections in a function.
5. Use ASCII mode in text editors, not UTF-8
6. One module per import line
7. Avoid from module import \* (for any module). Use either from module import func1, func2 or import module as shortname.
8. Follow the NumPy guidelines for documenting functions

More suggestions can be found in PEP8.

## 4 Flow Control, Loops and Exception Handling

### 4.1 13.2 if . . . elif . . . else

if . . . elif . . . else blocks always begin with an if statement immediately followed by a scalar logical expression. elif and else are optional and can always be replicated using nested if statements at the expense of more complex logic and deeper nesting. The generic form of an if . . . elif . . . else block is

```
if logical_1:  
Code to run if logical_1  
elif logical_2:  
Code to run if logical_2 and not logical_1  
elif logical_3:  
Code to run if logical_3 and not logical_1 or logical_2  
...  
...  
else:
```

Code to run if all previous logicals are false



**13.3.1 Whitespace** Like if . . . elif . . . else flowcontrol blocks, for loops are whitespace sensitive. The indentation of the line immediately below the for statement determines the indentation that all statements in the block must have. **13.3.2 break** A loop can be terminated early using break. break is usually used after an if statement to terminate the loop prematurely if some condition has been met.

```
x = randn(1000)
for i in x:
    print(i)
    if i > 2:
        break
```

Since for loops iterate over an iterable with a fixed size, break is generally more useful in while loops.

**13.3.3 continue** continue can be used to skip an iteration of a loop, immediately returning to the top of the loop using the next item in iterable. continue is commonly used to avoid a level of nesting, such as in the following two examples.

```
x = randn(10)
for i in x:
    if i < 0:
        print(i)
    for i in x:
        if i >= 0:
            continue
        print(i)
```

Avoiding excessive levels of indentation is essential in Python programming – 4 is usually considered the maximum reasonable level. continue is particularly useful since it can be used to in a for loop to avoid one level of indentation.

## 4.2 break

`break` can be used in a `while` loop to immediately terminate execution. Normally, `break` should not be used in a `while` loop – instead the logical condition should be set to `False` to terminate the loop. However, `break` can be used to avoid running code below the `break` statement even if the logical condition is `False`.

```
condition = True
i = 0
x = randn(1000000)
while condition:
    if x[i] > 3.0:
        break # No printing if x[i] > 3
    print(x[i])
    i += 1
```

It is better to update the logical statement which determines whether the `while` loop should execute

```
i = 0
while x[i] <= 3:
    print(x[i])
    i += 1
```

### 4.2.1 13.4.2 continue

`continue` can be used in a `while` loop to skip any remaining code in the loop, immediately returning to the top of the loop, which then checks the `while` condition, and executes the loop if it still true. Using `continue` when the logical condition in the `while` loop is `False` is the same as using `break`.

## 5 Graphics

Matplotlib is a complete plotting library capable of high-quality graphics. Matplotlib contains both high level functions which produce specific types of figures, for example a simple line plot or a bar chart, as well as a low level API for creating highly customized charts. This chapter covers the basics of producing plots and only scratches the surface of the capabilities of matplotlib. Further information is available on the matplotlib website or in books dedicated to producing print quality graphics using matplotlib. Throughout this chapter, the following modules have been imported.

### 5.1 matplotlib

- Matplotlib is a complete plotting library capable of high-quality graphics. Matplotlib contains both high level functions which produce specific types of figures, for example a simple line plot or a bar chart, as well as a low level API for creating highly customized charts.
- This chapter covers the basics of producing plots and only scratches the surface of the capabilities of matplotlib.
- Further information is available on the matplotlib website or in books dedicated to producing print quality graphics using matplotlib.

### 5.2 seaborn

seaborn is a Python package which provides a number of advanced data visualized plots. It also provides a general improvement in the default appearance of matplotlib-produced plots, and so I recommend using it by default.

```
import seaborn as sns
```

All figure in this chapter were produced with seaborn loaded, using the default options. The dark grid background can be swapped to a light grid or no grid using `sns.set(style='whitegrid')` (light grid) or `sns.set(style='nogrid')` (no grid, most similar to matplotlib).

### 5.3 Histograms

Histograms can be produced using `hist`. A basic histogram produced using the code below is presented in Figure 15.5, panel (a). This example sets the number of bins used in producing the histogram using the keyword argument `bins`.

## 5.4 Adding a Title and Legend

Titles are added with `title` and legends are added with `legend`. `legend` requires that lines have labels, which is why 3 calls are made to `plot` – each series has its own label. Executing the next code block produces a the image in figure 15.8, panel (a).

```
>>> x = cumsum(randn(100,3), axis = 0)
>>> plot(x[:,0], 'b',
label = 'Series 1')
>>> hold(True)
>>> plot(x[:,1], 'g.',
label = 'Series 2')
>>> plot(x[:,2], 'r:', label = 'Series 3')
>>> legend()
>>> title('Basic Legend')
```

`legend` takes keyword arguments which can be used to change its location (`loc` and an integer, see the docstring), remove the frame (`frameon`) and add a title to the legend box (`title`). The output of a simple example using these options is presented in panel (b).

```
>>> plot(x[:,0], 'b',
label = 'Series 1')
>>> hold(True)
>>> plot(x[:,1], 'g.',
label = 'Series 2')
>>> plot(x[:,2], 'r:', label = 'Series 3')
>>> legend(loc = 0, frameon = False, title = 'The Legend')
>>> title('Improved Legend')
```

## 5.5 Plotting

```
close_px.plot(label='AAPL')
mavg.plot(label='mavg')
plt.legend()
```

pandas is a high-performance module that provides a comprehensive set of structures for working with data. pandas excels at handling structured data, such as data sets containing many variables, working with missing values and merging across multiple data sets.

While extremely useful, pandas is not an essential component of the Python scientific stack unlike NumPy, SciPy or matplotlib, and so while pandas doesn't make data analysis possible in Python, it makes it much easier. pandas also provides high-performance, robust methods for importing from and exporting to a wide range of formats.

pandas provides a set of data structures which include Series, DataFrames and Panels. Series are 1-dimensional arrays. DataFrames are collections of Series and so are 2-dimensional, and Panels are collections of DataFrames, and so are 3-dimensional. Note that the Panel type is not covered in this chapter.

head() shows the first 5 rows of a series, and tail() shows the last 5 rows. An optional argument can be used to return a different number of entries, as in head(10).

### 5.5.1 isnull and notnull

isnull() returns a Series with the same indices containing Boolean values indicating True for null values which include NaN and None, among others.

notnull() returns the negation of isnull() – that is, True for non-null values, and False otherwise.

### 5.5.2 ix

ix is the indexing function and s.ix[0:2] is the same as s[0:2]. ix is more useful for DataFrames.

### 5.5.3 plot

plot is the main plotting method, and by default will produce a line graph of the data in a DataFrame. Calling plot on a DataFrame will plot all series using different colors and generate a legend. A number of keyword arguments are available to affect the contents and appearance of the plot.

- `style`, a list of matplotlib styles, one for each series plotted. A dictionary using column names as keys and the line styles as values allows for further customization.
- `title`, a string containing the figure title.
- `subplots`, a Boolean indicating whether to plot using one subplot per series (True). The default is False.
- `legend`, a Boolean indicating whether to show a legend
- `secondary_y`, a Boolean indicating whether to plot a series on a secondary set of axis values. See the example below.
- `ax`, a matplotlib axis object to use for the plot. If no axis is provided, then a new axis is created.
- `kind`, a string, one of: