Contents

1 Flow Control, Loops and Exception Handling

1.1 Whitespace and Flow Control

Python uses white space changes to indicate the start and end of flow control blocks, and so indention matters. For example, when using `if . . . elif . . . else` blocks, all of the control blocks must have the same indentation level and all of the statements inside the control blocks should have the same level of indentation.

Returning to the previous indentation level instructs Python that the block is complete. Best practice is to only use spaces (and not tabs), and to use 4 spaces when starting a indented level, which is a good balance between readability and wasted space.

1.2 `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:
```

A few simple examples

```
>>> x = 5
>>> if x<5:
... x += 1
... else:
... x =
1
>>> x
4
```

```
>>> x = 5;
>>> if x<5:
... x = x + 1
... elif x>5:
... x = x 1
... else:
... x = x * 2
>>> x
10
```

1.3 `for`

`for` loops begin with for item in iterable:, and the generic structure of a for loop is

```
for item in iterable:
Code to run
```

item is an element fromiterable, and iterable can be anything that is iterable in Python. The mostcommon examples are xrange or range, lists, tuples, arrays or matrices.

The `for` loop will iterate across all items in iterable, beginning with item 0 and continuing until the final item. When using multidimensional arrays, only the outside dimension is directly iterable.

For example, if x is a 2-dimensional array, then the iterable elements are x[0], x[1] and so on.

Indentation Errors in Type Setting

```
count = 0
for i in xrange(100):
count += i

count = 0
x = linspace(0,500,50)
for i in x:
count += i

count = 0
x = list(arange(20,21))
for i in x:
count += i
```

Loops can also be nested

```
count = 0
for i in xrange(10):
for j in xrange(10):
count += j
```

or can contain flow control variables

```
returns = randn(100)
count = 0
for ret in returns:
if ret<0:
count += 1
```

This for expression can be equivalently expressed using xrange as the iterator and len to get the number of items in the iterable.

```
returns = randn(100)
count = 0
for i in xrange(len(returns)):
if returns[i]<0:
count += 1
```

Finally, these ideas can be combined to produce nested loops with flow control.

```
x = zeros((10,10))
for i in xrange(size(x,0)):
for j in xrange(size(x,1)):
if i<j:
x[i,j]=i+j;
else:
x[i,j]=ij
```

or loops containing nested loops that are executed based on a flow control statement.

```
x = zeros((10,10))
for i in xrange(size(x,0)):
if (i % 2) == 1:
for j in xrange(size(x,1)):
x[i,j] = i+j
else:
for j in xrange(int(i/2)):
x[i,j] = ij
```

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. **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.

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.

1.4 while

`while` loops are useful when the number of iterations needed depends on the outcome of the loop contents. `while` loops are commonly used when a loop should only stop if a certain condition is met, such as when the change in some parameter is small. The generic structure of a while loop is

```
while logical:
Code to run
Update logical
```

Two things are crucial when using a while loop: first, the logical expression should evaluate to true when the loop begins (or the loop will be ignored) and second, the inputs to the logical expression must be updated inside the loop. If they are not, the loop will continue indefinitely (hit CTRL+C to break an interminable loop in IPython). The simplest while loops are (wordy) drop-in replacements for for loops:

```
count = 0
i = 1
while i<10:
count += i
i += 1
```

which produces the same results as

```
count=0;
for i in xrange(0,10):
count += i
```

while loops should generally be avoidedwhenfor loops are sufficient. However, there are situations where no for loop equivalent exists.

```
# randn generates a standard normal random number
mu = abs(100*randn(1))
index = 1
```

```
while abs(mu) > .0001:
mu = (mu+randn(1))/index
index=index+1
```

In the block above, the number of iterations required is not knownin advance and since randn is a standard normal pseudo-random number, it may take many iterations until this criteria is met. Any finite for loop cannot be guaranteed to meet the criteria.

`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
```

1.4.1 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.

2 Importing and Exporting Data

2.1 Importing Data using pandas

All of the data reading functions in pandas load data into a pandas DataFrame, 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 main formats, 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.

2.2 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]
```

When the entire file is numeric, the data will be stored as a homogeneous array using one of the numeric data types, typically float64. In this example, the first column contains Excel dates as numbers, which are the number of days past January 1, 1900.

```
>>> csv_data = read_csv('FTSE_1984_2012_numeric.csv')
>>> csv_data = csv_data.values
>>> csv_data[:4,:2]
array([[ 40954. , 5899.9],
[ 40953. , 5905.7],
[ 40952. , 5852.4],
[ 40949. , 5895.5]])
```

2.2.1 Excel files

Excel files, both 97/2003 (xls) and 2007/10/13 (xlsx), can be imported using `read_excel`. Two inputs are required to use `read_excel`, the filename and the sheet name containing the data. In this example, pandas makes use of the information in the Excel workbook that the first column contains dates and converts these to datetimes. Like the mixed CSV data, the array returned has object data type.

```
>>> from pandas import read_excel
>>> excel_data = read_excel('FTSE_1984_2012.xls','FTSE_1984_2012')
>>> excel_data = excel_data.values
>>> excel_data[:4,:2]
array([[datetime.datetime(2012, 2, 15, 0, 0), 5899.9],
[datetime.datetime(2012, 2, 14, 0, 0), 5905.7],
[datetime.datetime(2012, 2, 13, 0, 0), 5852.4],
[datetime.datetime(2012, 2, 10, 0, 0), 5895.5]], dtype=object)
>>> open = excel_data[:,1]
```

3 File System Operations

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

```
import os
import shutil
```

have been included.

3.1 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`.Alternatively, path can can be .. to more 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:\\'
```

3.2 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 is 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
```

3.3 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)
```

3.4 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 streams) is important, it is better use system commands (copy or move on Windows, cp or mv on Linux) as an external program.

```
os.chdir('c:\\temp\\python')
# Make an empty file
f = file('file.ext','w')
f.close()
# Copies file.ext to 'c:\temp\'
shutil.copy('file.ext','c:\\temp\\')
# Copies file.ext to 'c:\temp\\python\file2.ext'
shutil.copy('file.ext','file2.ext')
# Copies file.ext to 'c:\\temp\\file3.ext', plus metadata
shutil.copy2('file.ext','file3.ext')
shutil.copytree('c:\\temp\\python\\','c:\\temp\\newdir\\')
shutil.move('c:\\temp\\newdir\\','c:\\temp\\newdir2\\')
```

4 Data Wrangling

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

4.1 Numerical Indexing

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

Numerical indexing, also called list-of-location indexing, is an alternative to slice notation. The fundamental idea underlying numerical indexing is to use coordinates to select elements, which is similar to the underlying idea behind slicing.

A numerical index can be either a list or a NumPy array and must contain integer data.

```
>>> 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]])
>>> # 2 by 1 and 1 by 2 - difference shapes, broadcasted as 2 by 2
>>> x[sel_row,sel_col]
array([[ 0., 1.],
[ 5., 6.]])
```

Mixing Numerical Indexing with Scalar Selection

NumPy permits using difference 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))
```

Mixing Numerical Indexing with Slicing

Mixing numerical indexing and slicing allow for entire rows or columns to be selected.

```
>>> x[:,[1]]
array([[ 2.],
[ 7.]])
>>> x[[1],:]
array([[ 6., 7., 8., 9., 10.]])
```

Note that the mixed numerical indexing and slicing uses a list ([1]) so that it is not a scalar. This is important since using a scalar will result in dimension reduction.

```
>>> x[:,1] # 1dimensional
array([ 2., 7.])
```

Numerical indexing and slicing can be mixed in more than 2-dimensions, although some care is required. In the simplest case where only one numerical index is used which is 1-dimensional, then the selection is equivalent to calling `ix_` where the slice a:b:s is replaced with arange(a,b,s).

```
>>> x = reshape(arange(3**3), (3,3,3)) # 3d
array
>>> sel1 = x[::2,[1,0],:1]
>>> sel2 = x[ix_(arange(0,3,2),[1,0],arange(0,1))]
>>> sel1.shape
(2L, 2L, 1L)
>>> sel2.shape
(2L, 2L, 1L)
>>> amax(abs(sel1sel2))
0
```

Linear Numerical Indexing using flat

Like slicing, numerical indexing can be combined with flat to select elements from an array using the row-major ordering of the array. The behavior of numerical indexing with flat is identical to that of using numerical indexing on a flattened version of the underlying array.

```
>>> x.flat[[3,4,9]]
array([ 4., 5., 10.])
>>> x.flat[[[3,4,9],[1,5,3]]]
array([[ 4., 5., 10.],
[ 2., 6., 4.]])
```

4.2 Logical Indexing

Logical indexing differs fromslicing and numeric indexing by using logical indices to select elements, rows or columns. Logical indices act as light switches and are either "on" (True) or "off" (False). Pure logical indexing uses a logical indexing array with the same size as the array being used for selection and always returns a 1-dimensional array.

```
>>> x = arange(3,3)
>>> x < 0
array([ True, True, True, False, False, False], dtype=bool)
>>> x[x < 0]
array([3,
2,
1])
>>> x[abs(x) >= 2]
array([3,
2,
2])
>>> x = reshape(arange(8,
8), (4,4))
>>> x[x < 0]
array([8,
7,
6,
5,
4,
3,
2,
1])
```

It is tempting to use two 1-dimensional logical arrays to act as row and column masks on a 2-dimensional array. This does not work, and it is necessary to use `ix_` if interested in this type of indexing.

```
>>> x = reshape(arange(8,8),(
4,4))
>>> cols = any(x < 6,
0)
>>> rows = any(x < 0, 1)
>>> cols
array([ True, True, False, False], dtype=bool
>>> rows
array([ True, True, False, False], dtype=bool)
>>> x[cols,rows] # Not upper 2 by 2
array([8,
3])
>>> x[ix_(cols,rows)] # Upper 2 by 2
array([[8,
7],
[4,
3]])
```

The difference between the final 2 commands is due to how logical indexing operates when more than logical array is used. When using 2 or more logical indices, they are first transformed to numerical indices using nonzero which returns the locations of the non-zero elements (which correspond to the True elements of a Boolean array).

```
>>> cols.nonzero()
(array([0, 1], dtype=int64),)
>>> rows.nonzero()
(array([0, 1], dtype=int64),)
```

The corresponding numerical index arrays have compatible sizes – both are 2-element, 1-dimensional arrays – and so numeric selection is possible. Attempting to use two logical index arrays which have non-broadcastable dimensions produces the same error as using two numerical index arrays with nonbroadcastable sizes.

```
>>> cols = any(x < 6,
0)
>>> rows = any(x < 4, 1)
>>> rows
array([ True, True, True, False], dtype=bool)
>>> x[cols,rows] # Error
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

`argwhere`

`argwhere` returns an array containing the locations of elements where a logical condition is True. It is the same as `transpose(nonzero(x))`

```
>>> 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)
```

5 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.

5.1 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 – aswell 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.

5.1.1 Keyword Arguments

All input variables in functions are automatically keyword arguments, so that the function can be accessed either by placing the inputs in the order they appear in the function (positional arguments), or by calling the input by their name using keyword=value.

```
from __future__ import print_function, division
import numpy as np
def lp_norm(x,y,p):
d = x y
return sum(abs(d)**p)**(1/p)
# Call the function
x = np.random.randn(10)
y = np.random.randn(10)
z1 = lp_norm(x,y,2)
z2 = lp_norm(p=2,x=x,y=y)
print("The Lp distances are ",z1,z2)
```

Because variable names are automatically keywords, it is important to use meaningful variable names when possible, rather than generic variables such as a, b, c or x, y and z. In some cases, x may be a reasonable default, but in the previous example which computed the Lp norm, calling the third input z would be bad idea.

5.1.2 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
def bad_function(x = zeros(1)):
print(x)
x[0] = np.random.randn(1)
# Call the function
bad_function()
bad_function()
bad_function()
```

Each call to `bad_function` shows that x has a different value – despite the default being 0. The solution to this problem is to initialize mutable objects to None, and then the use an if to check and initialize only if the value is None. Note that tests for None use the is keyword rather the testing for equality using ==.

```
from __future__ import print_function, division
import numpy as np
def good_function(x = None):
if x is None:
x = zeros(1)
print(x)
x[0] = np.random.randn(1)
# Call the function
good_function()
good_function()
```

5.2 Variable Number of Inputs

Most function written as an "end user" have an known (ex ante) number of inputs. However, functions which evaluate other functions often must accept variable numbers of input. Variable inputs can be handled using the *args (arguments) or **kwargs (keyword arguments) syntax. The *args syntax will generate tuple a containing all inputs past the required input list. For example, consider extending the Lp function so that it can accept a set of p values as extra inputs (Note: in practice it would make more sense to accept an array for p).

```
from __future__ import print_function, division
import numpy as np
def lp_norm(x,y,p = 2, *args):
d = x-y
print('The L' + str(p) + ' distance is :', sum(abs(d)**p)**(1/p))
out = [sum(abs(d)**p)**(1/p)]
print('Number of *args:', len(args))
for p in args:
print('The L' + str(p) + ' distance is :', sum(abs(d)**p)**(1/p))
out.append(sum(abs(d)**p)**(1/p))
return tuple(out)
# Call the function
x = np.random.randn(10)
y = np.random.randn(10)
# x & y are required inputs and so are not in *args
lp = lp_norm(x,y)
# Function takes 3 inputs, so no *args
lp = lp_norm(x,y,1)
```

```
# Inputs with default values can be ignored
lp = lp_norm(x,y,1,2,3,4,1.5,2.5,0.5)
```

The alternative syntax, **kwargs, generates a dictionary with all keyword inputs which are not in the function signature. One reason for using **kwargs is to allow a long list of optional inputs without having to have an excessively long function definition. This is how this input mechanism operates in many matplotlib functions such as plot.

```
from __future__ import print_function, division
import numpy as np
def lp_norm(x,y,p = 2, **kwargs):
d = x y
print('Number of *kwargs:', len(kwargs))
for key in kwargs:
print('Key :', key, ' Value:', kwargs[key])
return sum(abs(d)**p)
# Call the function
x = np.random.randn(10)
y = np.random.randn(10)
# Inputs with default values can be ignored
lp = lp_norm(x,y,kword1=1,kword2=3.2)
# The p keyword is in the function def, so not in **kwargs
lp = lp_norm(x,y,kword1=1,kword2=3.2,p=0)
```

5.3 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 formlambda 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 returnthe 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')]
```

5.4 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 219

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.

6 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.

6.1 `matlibplot`

- 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.

6.2 `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(stype='whitegrid') (light grid) or sns.set(stype='nogrid') (no grid, most similar to matplotlib).

6.3 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.

6.4 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')
```

6.5 Plotting

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