



Python for R Users

Workshop 4 at the EARL Conference

Date: 11 September 2018

Time: 2pm – 5pm

Room: Bridge 2

@ earl-team@mango-solutions.com

☎ +44 (0)1249 705 450

💻 mango-solutions.com

Copyright © Mango Business Solutions Ltd

All rights reserved

These notes remain the property of Mango Business Solutions Ltd and are not to be photocopied, sold, duplicated or reproduced without the express permission of Mango Business Solutions Ltd

Chapter 1

Introduction to Python for R Users

1.1 Introduction to the Training

1.1.1 Course Aims

This course has been designed to introduce Python in a way that means you can quickly get started with data analysis, building on your R knowledge. This includes functions for importing, manipulating and visualising data as well as some basic statistical analysis. With the emphasis on using Python for these tasks there will be a number of features of Python that would be taught in a typical programming course that we will not cover, including some of Python's in-built objects and their methods, working with matrices and arrays in NumPy and conditional flow and function writing.

1.1.2 Course Materials

Items appearing in this material are sometimes given a special appearance to set them apart from regular text. Here's how they look:

```
>>> This is a section of code           # This is a comment
```

The format of these code blocks emulates how the code would be run in an interactive Python console, with lines starting with the prompt >>> showing input lines of code, and those without the prompt displaying the outputs you would expect to be returned on screen.



A warning, typically describing non-intuitive aspects of the Python language



A tip: additional features of Python or “shortcuts” based on user experience



Exercises to be performed during (or after) the training course

1.1.3 Course Script and Exercise Answers

A great deal of code will be executed within Jupyter notebooks during the delivery of this training. This includes the answers to each exercise, as well as other code written to answer questions that arise. Following the course, you will be sent a notebook containing all the code that was executed.

1.3 What is Python

Python is a powerful general-purpose programming language with widespread use in many applications domains. Python is open source and free to use (even for commercial products), and available for all major operating systems.

The principal author of Python Guido van Rossum, who began work on it in the 1980's, played a central role in its development as president of the Python Software Foundation (PSF) until the summer of 2018. In his absence, the non-profit organization will continue to be devoted to the Python programming language and its administration. Python has many contributors from all over the world, and the PSF is supported by many international sponsors.

1.3.1 Key Features

The main differentiating factors of the Python language can be described as follows:

- At its core Python was designed for readability and clarity, and therefore has minimised the necessary syntax, making it easier to pick up and understand code you did not write.
- Python comes with an extensive standard library, but also allows easy addition of custom libraries.

Many ways exist to interface Python with other languages meaning that Python works well as 'glue' in many development applications. A few other advantages with working in Python are:

- The multiple-purpose language allows for easier integration between your data science, data engineering and software development teams.
- Python is multi-threaded making parallelisation easier.

1.3.2 The Python Website

There are many online Python resources, almost all of which can be reached via the main Python site: <http://www.python.org/>. From this site you can do many things, including:

- Download the latest copy of the core Python language.
- Find links, source code and documentation to many additional Python libraries.
- Find help on the use of Python in the online wiki.
- Join the "Python-Help" mailing list.
- Look for Python books and events.

1.3.3 Python Versions

The most current version of Python is 3.6 and is the focus of this course. The rest of this section elaborates more on the history of Python versions and why this distinction is important.

In 2010 Python 3.0 was released and introduced a range of improvements to clean up the base language and improve its efficiency. However, some of these changes required fundamental changes under the hood, and the decision was made to focus on new features and future enhancements to the Python language, rather than patching them onto legacy code. For this reason, Python 3 is intentionally not backward-compatible with previous versions.

This obviously has some knock-on effects, and even though the changes to syntax and usability of the language are minimal, it has resulted in a protracted effort to port over the vast amount of 3rd party libraries that have previously been written.

Python 2.7.x is still the most widely used version mostly because of the concern over library compatibility mentioned above. However, Python 2 is in its end of life stage (sunset in 2020), meaning that there will never be a Python 2.8 release as all future development of the language is now focused on Python 3.

This course is taught using Python 3 as it has more consistent syntax for beginners to become familiar with and will future proof any code that trainees end up writing after the course.



A list of the top 360 Python packages and their compatibility with Python 3 is available from <http://py3readiness.org/>. Also, a detailed discussion of the differences between Python 2 and Python 3 can be found at <https://wiki.python.org/moin/Python2orPython3>

1.3.4 The Python Package Index

Similar to the R community, the Python community has actively produced many libraries of new classes and functions for Python (called packages), which extend the capabilities of Python in many directions. The Python Package Index (PyPI) is the main repository for these packages, and at the time of writing there are over 90,000 packages available to download.

Chapter 2

The Python Environment for Data Analysis

2.1 Introduction

One of the major hurdles in getting up and running with Python is the complexity of the installation process, as there are not only multiple versions of Python to consider, but packages must be downloaded and installed to the right location, and any dependencies between them must also be managed.

Fortunately, several Python distributions have been put together that work as a one stop installation of both Python and many of the extended libraries.

2.2 Installing the Anaconda Distribution

The Anaconda distribution is cross platform and comes with Python 2.7 or 3.6, as well as other tools and libraries such as Jupyter, Spyder, IPython, NumPy, SciPy, pandas, seaborn and matplotlib already configured.

The fastest way to get up and running with Python is to download the Anaconda Python distribution from <https://www.anaconda.com/download>, and follow the installation instructions.

The default installation path for the Python 3 Anaconda distribution on Windows is:

C:\Users\<username>\AppData\Local\Continuum\Anaconda3

2.3 Jupyter Notebooks

There are several ways of using Python for data analysis. Spyder is a full interactive development environment (IDE), similar to RStudio. Another tool is Jupyter Notebooks which we will be using here. Like R Notebooks and R Markdown, they are designed for an easy integration of text and programming. They are aimed at providing a more interactive workflow for Python programming, analysis and reporting. While an R Notebook is a script that gets rendered into an output file in one step, Jupyter Notebooks contain cells which can all be executed and rendered interactively.

2.3.1 Starting Jupyter

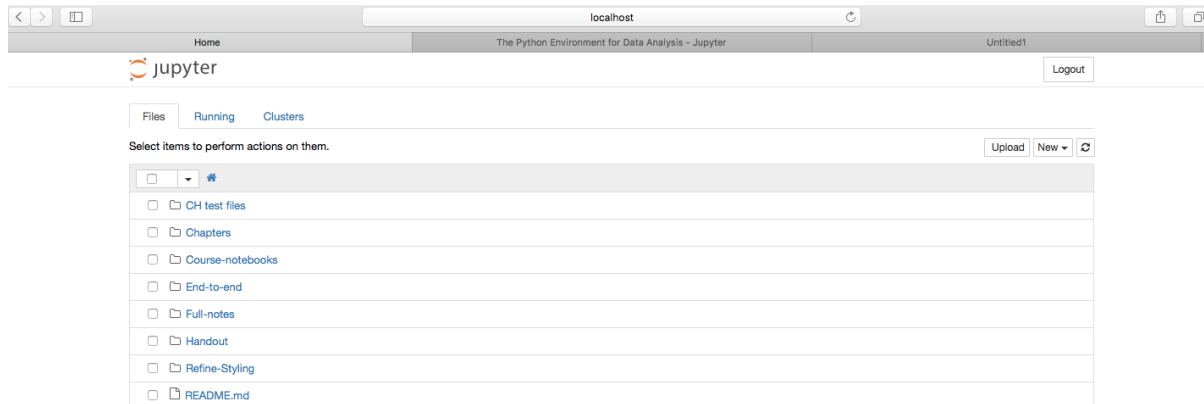
Once installed look for and run an application called Jupyter Notebook, which on Windows is located in:

Start menu > All programs > Anaconda3 > Jupyter Notebook

Alternatively, open up a command line console and type:

```
jupyter notebook
```

After about a minute, you should see the Jupyter application appear. The figure shows an example of what Jupyter looks like on Mac. The appearance on Windows and Linux operating systems will be slightly different.



You should see three main tabs in Jupyter on start-up:

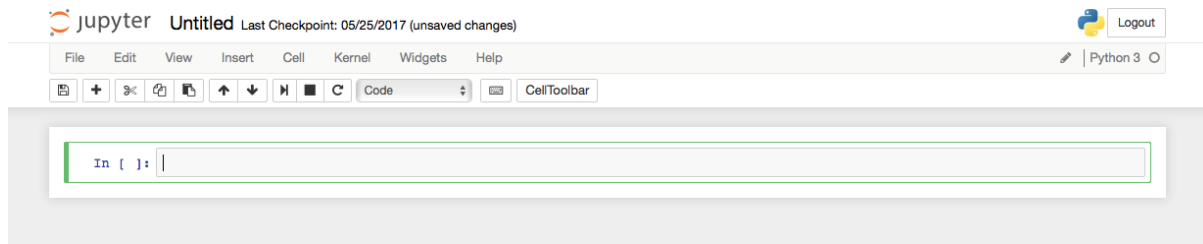
- Files: Your file directory
- Running: Lists all of the notebooks currently running
- Clusters: For using IPython in parallel with your cluster (beyond the scope of this course)

2.3.2 Opening a New Jupyter Notebook

To open a Jupyter Notebook, click the New drop down menu on the File tab and select "Python 3" under the Notebooks heading.



This will open a blank Notebook with an IPython console (using Python 3) running underneath it.



The IPython console is used to input and execute Python code interactively. Outputs, errors and warning messages are directly shown in the same window. A command which has been input into the console is executed by simply pressing Shift+Enter.

2.3.3 The IPython Kernel

The IPython kernel is the processing back-end that Jupyter notebooks use to execute Python code. The IPython kernel is a python interpreter designed for interactive use, and adds many helpful features such as:

- Coloured input and output lines, as well as error messages for added clarity.
- Aliases to useful operating system calls for directory navigation e.g. `pwd`, `cd`, `ls` and `mkdir`.
- 'Magic' functions provide many other helpful shortcuts. Type `%quickref` for a full list. A few are covered later in this course.
- Tab completion of variable names, imports, class attributes and functions as well as file names in the current working directory.
- Object exploration through the use of `?` and `??` (see below).
- Input/Output history referencing and searching.

A detailed introduction and overview of these features can also be found by entering a single question mark `?` within IPython. Key features will be introduced gradually throughout this course with examples.

2.3.4 Cell Based Execution

It is common when working interactively to only want to run a section of the code, rather than the entire script. To aid in this, Jupyter uses a concept of 'code cells', which can be executed individually in the IPython console. Each block represents a code cell and can be executed by:

- Pressing Ctrl+Enter to execute all the code in the current active cell.
- Pressing Shift+Enter to execute all the code in the current active cell and advance the cursor to the next cell.

Using Shift+Enter multiple times to incrementally step through the code and investigate the results in the IPython console is a common workflow when prototyping and doing exploratory work.

```
In [1]: x = 1
        y = 2
        y + x

Out[1]: 3

In [ ]:
```

2.3.5 Command and Edit Modes

Jupyter notebook is a modal editor which means that the keyboard does different things depending on which mode the notebook is in. There are two modes: edit mode and command mode. To go from command mode to edit mode, press Enter. To return to command mode from edit mode, press Esc.

Edit mode is indicated by a green cell border and left sidebar, and a prompt showing in the editor area:


```
In [ ]: a = 10
```

When a cell is in edit mode, you can type into the cell, like a normal text editor. Enter edit mode by pressing Enter or using the mouse to click on a cell's editor area.

Command mode is indicated by a grey cell border and a blue sidebar:

```
In [ ]: a = 10
```

When you are in command mode, you are able to edit the notebook as a whole, but not type into individual cells. Most importantly, in command mode, the keyboard is mapped to a set of shortcuts that let you perform notebook and cell actions efficiently. For example, if you are in command mode and you press c, you will copy the current cell - no modifier is needed.

 Don't try to type into a cell in command mode; unexpected things will happen!

A summary of useful shortcuts is included at the end of this section. However, a full list is always available by going to Help > Keyboard Shortcuts.



You can also access the Keyboard Shortcuts list by pressing h in command mode.

2.3.6 *Markdown Text Cells*

The default cells are IPython console cells though a cell can be changed to include Markdown text. To change the type of cell from code to Markdown go to the top menu bar Cell > Cell Type > Markdown. Or press m while in Command Mode and highlighting the cell.

Markdown text cells support plain text, Markdown and HTML. For information about Markdown and HTML:

- Markdown - <https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet>
- HTML - <https://web.stanford.edu/group/csp/cs21/htmlcheatsheet.pdf>

2.3.7 *Useful Keyboard Shortcuts*

All of the cell manipulation commands also have keyboard shortcuts to execute. A list of the commonly used ones is below; however, to view a full list of shortcuts either go to Help > Keyboard Shortcuts, or in Command Mode, press H.

Command Mode:

- Enter: enter edit mode
- H: show keyboard shortcuts
- Shift + Enter: run cell, select below
- Ctrl + Enter: run cell
- Alt + Enter: run cell, insert below
- Y: to code
- M: to markdown
- A/B: insert cell above or below
- X: cut selected cell
- C: copy selected cell
- V: paste cell
- D, D (i.e. D then D again): delete cell

Edit Mode:

- Tab: code completion or indent
- Ctrl + Z: undo

2.4 Modules, Packages and Libraries

2.4.1 Modules

A module in Python is simply a ".py" file that contains additional code to define functions, classes or variables. Modules provide a way to logically group related code, making it easier to understand and use.

In Python terms, a module is just another Python object with attributes that you can bind and reference, though in this case the attributes are the various functions, classes or variables defined in the .py file. To use them within our namespace we use the `import` statement in one of three ways:

```
import module_name
import module_name as alias
from module_name import object1, object2, object3
```

The method used to import will affect how the objects imported are used. Some examples are shown below.

```
>>> # method 1
>>> import math
>>> math.sin(math.pi/3)
>>>
>>> # method 2
>>> import math as m
>>> m.sin(m.pi/3)
>>>
>>> # method 3
>>> from math import sin, pi
>>> sin(pi/3)
0.8660254037844386
```

There is also a fourth way of importing things with a `*` instead of specifying each object individually. This imports all the functions, classes and variables contained within the module.

```
>>> from math import *
>>> log(sin(pi/3))
-0.14384103622589053
```

However, this method of importing objects is frowned upon as it is less clear what is imported and will also override variables already defined if their names clash.

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur in a single script.

2.4.2 Packages and Libraries

A package is a collection of modules, similar to packages in R. A collection of packages is usually referred to as a library. There is no formal distinction between a package and a library so sometimes package and library are used interchangeably.

Packages and libraries can be made available in the same manner as above, simply using the package or library name instead of the `module_name` in the `import` statements.

It is also possible to selectively import only certain modules in a package or library. To access a module (or sub-package), we must use the `.` operator. For example, to extract a module named `foo` from a package named `mypackage`, as the alias `mpf`, we use the following code:

```
import mypackage.foo as mpf
```

Then to use any functions from this module, we simply call the module alias, the `.`, and then the function name, i.e., for function `func1` we have the following:

```
x = mpf.func1()
```

2.4.3 The Python Standard Library

Python comes with many useful modules and packages as standard. Details on using the Python Standard Library can be found in the documentation at

<https://docs.python.org/3.6/library/>

with additional examples available at this fantastic site

<http://pymotw.com/2/contents.html>



1. Open up a new Jupyter Notebook and print “Hello Python world!” on the console.
2. Add a markdown cell and include a note that this is the notebook for today’s workshop.
3. Import the numpy library using the alias np.

Extension:

4. Draw a (pseudo) random number from a uniform distribution on the interval [2, 5].

2.5 Installing Additional Packages

If you have the Anaconda Python distribution, this comes with many third party python packages already installed. However you may come across additional packages that you wish to install.

Unlike R’s `install.packages`, Python itself has no built in system to manage packages, but there is a central repository for Python packages known as the Python Package Index or PyPI. Several command line utilities have been developed to look up and install Python packages from PyPI.

The recommended tool to use to install additional Python packages is pip. Python ≥ 3.4 now comes with pip by default. You can search for packages on PyPI using,

```
pip search <package name>
```

Packages are then installed with

```
pip install <package name>
```

Other helpful pip commands are `pip list`, which lists all the packages installed by pip, and `pip uninstall <package name>` to remove an installed package.

2.6 The Help System

In Python, to find out information about a function we can use the online documentation for the package that it is in. However, we can also find out more about functions by using Jupyter's inbuilt help system.

The `?` can be used to display help on a function, Jupyter allows the use of the `?` before or after the function in question. For example, to find out information about the `np.floor` function we can use the following:

```
>>> import numpy as np
>>> ?np.floor
```

```
Call signature: np.floor(*args, **kwargs)
Type:          ufunc
String form:   <ufunc 'floor'>
File:          c:\users\kmarks\appdata\local\continuum\anaconda3\lib\site-packages\numpy\__init__.py
Docstring:
floor(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])

Return the floor of the input, element-wise.

The floor of the scalar `x` is the largest integer `i`, such that
`i <= x`. It is often denoted as :math:\lfloor x \rfloor.

Parameters
-----
x : array_like
    Input data.
```


Chapter 3

Data Manipulation

3.1 Reading Data

The **pandas** package provides lots of simple functions for reading data from and writing data to various different file formats, such as CSV, Excel, SQL, JSON, and HTML. All the read functions in the package (`pd.read_*` functions) create a pandas `DataFrame` object. This object, like a `data.frame` in R, is meant for storing tabular data where columns can potentially contain different data types. The dimensions of a data frame are typically labelled with row names and column names in R. Here, the labels are called index and columns, respectively.

```
>>> import pandas as pd
>>> tips = pd.read_csv("Data/tips.csv")
```



In Python, only the equals sign `=` is used to assign values to variables. Variable names can start with an underscore `_` (unlike in R) and are case sensitive (like in R).



Documentation for each function can be accessed via the websites for the package in question, e.g., http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html, or can be viewed directly from Jupyter notebooks via `?`. It can be placed in front of the function call as in R, e.g., `?pd.read_csv`, or after it.

3.1.1 Functions, Methods and Attributes

R has been characterised well by “Everything that exists is an object. Everything that happens is a function call.” The Python equivalent is “Everything is an object, and all objects can have attributes.”

When working in R, in most cases we use a function on an object, i.e., `function(object, ...)`. This is the case for functions from a package, e.g., to read in data, generic functions like `summary` which do different things depending on the object class, or functions to access attributes of the object such as the dimension (via the `dim` function).

In Python, functions from a package such as `pd.read_csv` are utilised like in R, i.e., `function(arguments)`. Beyond that, objects typically have attributes which are references to other Python objects. Those can be data types such as a list to store the

dimension of a data frame but also can be functions specific to that object class. Such functions are referred to as object methods, or simply methods.

Attributes, including methods, are accessed via the dot operator `.`,
`<object_name>.<attribute>`.

For example, attributes of a pandas DataFrame include the dimension (`.shape`) and the dimension labels (`.index` and `.columns`).

```
>>> tips.columns
Index(['total_bill', 'tip', 'sex', 'smoker', 'day', 'time', 'size'],
      dtype='object')
```

Methods require parentheses after the method name, where any argument can be specified, `<object_name>.<method>(arg=*)`.


Methods for a DataFrame include the `.head` and `.tail` functions for preview the first or last few rows of the data frame as well the `.describe` function which provides a summary of the data frame, similar to R's `summary` function.

```
>>> tips.describe(include="all")
```

	total_bill	tip	sex	smoker	day	time	size
count	244.000000	244.000000	244	244	244	244	244.000000
unique	NaN	NaN	2	2	4	2	NaN
top	NaN	NaN	Male	No	Sat	Dinner	NaN
freq	NaN	NaN	157	151	87	176	NaN
mean	19.785943	2.998279	NaN	NaN	NaN	NaN	2.569672
std	8.902412	1.383638	NaN	NaN	NaN	NaN	0.951100
min	3.070000	1.000000	NaN	NaN	NaN	NaN	1.000000
25%	13.347500	2.000000	NaN	NaN	NaN	NaN	2.000000
50%	17.795000	2.900000	NaN	NaN	NaN	NaN	2.000000
75%	24.127500	3.562500	NaN	NaN	NaN	NaN	3.000000
max	50.810000	10.000000	NaN	NaN	NaN	NaN	6.000000



In Jupyter, we can view all the attributes and methods associated with an object by using tab completion. Once an object has been created (i.e., `tips` for above), typing `tips.` into a cell and pressing the tab key shows all the attributes and methods that can be used on that object.

- 
1. Import the mtcars data (mtcars.csv), ensuring that the car names are used for the row index.
 2. What are the dimensions of the imported data?
 3. Print the top 5 rows of the data.

Extension

4. Print the last 10 rows of the data.
5. Import only the miles per gallon (mpg) and weight (wt) columns from the mtcars data

3.2 Data Manipulation

The **pandas** package can be used for more than importing and exporting data. Written originally by Wes McKinney, the **pandas** package has grown to also include data manipulation, statistics and visualisation. Here, we will focus on the functionality for data manipulation.

Common data manipulation tasks mostly have corresponding `DataFrame` methods:

Description	Method
Filter rows based on values	<code>.query</code>
Sort rows	<code>.sort_values</code>
Rename columns	<code>.rename</code>
Summarise/Aggregate columns	<code>.agg</code>
Group the data	<code>.groupby</code>

However, some tasks such as selecting columns, changing or adding new columns, or selecting rows based on position are not done via a corresponding method. We will show how this is done in Python and introduce the concept of dictionaries which are needed for setting arguments to the `.rename` and `.agg` methods.

3.2.1 Selecting Columns

We can access columns in a data frame by indexing with `[]` and supplying the corresponding column labels. To access a single column, a string with the column label is sufficient.

```
>>> tips['time'].head()
0    Dinner
1    Dinner
2    Dinner
3    Dinner
4    Dinner
Name: time, dtype: object
```

To access multiple columns, we provide the labels in form of a list. Lists are created using square brackets `[]`, with individual items separated by commas.

```
>>> list_of_names = ['total_bill', 'size', 'day']
>>> tips[list_of_names].head()
   total_bill  size  day
0        16.99    2  Sun
1        10.34    3  Sun
2        21.01    3  Sun
3        23.68    2  Sun
4        24.59    4  Sun
```



Lists in Python, like lists in R, are very versatile and can hold any collection of objects with mixing of multiple types allowed. They are often used to specify a collection of elements for use in function argument. In R, we typically would do that via vectors or lists.

3.2.2 Series Objects

Selecting multiple columns returns another data frame but selecting a single column returns a `Series` object. This object consists of two parts, `values` and `index`, which can be accessed through the corresponding attributes. A `Series` has similar methods as a `DataFrame`, such as `.head` and `.describe`. Other helpful methods include

- `.isnull()/notnull()` to perform a boolean test for missing/non-missing values
- `.isfinite()` to perform a boolean test for finite values
- `.value_counts()` to return counts of unique values
- `.idxmax()` to return the index of the row with the largest value



Operations such as `+` on two `Series` objects are done based on the index, i.e., matching up elements with the same index, regardless of their position in the series. For comparison, addition of two vectors in R (which do not have an index), is carried out element-wise where the matching up is done by position, i.e., the two first elements are added up, the two second elements are added up, etc.

3.2.3 Adding and Editing Columns

To add a new column to a `DataFrame`, we select a new column via the indexing method (`[]` as above), using the new column name for which column is to be selected, and assign the new values to it. To edit a column, we can overwrite it and assign new values to the column in the same way.

For example, we can add the tip share to the tips dataset by dividing the tip amount by the total amount of the bill and assign the result to a new column called `tip_share`.

```
>>> tips['tip_share'] = tips['tip'] / tips['total_bill']
>>> tips.head()
```

	total_bill	tip	sex	smoker	day	time	size	tip_share
0	16.99	1.01	Female	No	Sun	Dinner	2	0.059447
1	10.34	1.66	Male	No	Sun	Dinner	3	0.160542
2	21.01	3.50	Male	No	Sun	Dinner	3	0.166587
3	23.68	3.31	Male	No	Sun	Dinner	2	0.139780
4	24.59	3.61	Female	No	Sun	Dinner	4	0.146808

3.2.4 Selecting Rows

To select rows based on values in specific columns, we can use the `.query` method which takes logical statements to describe the rows which are to be selected. Unlike for the `subset` and `filter` functions in R (from base R and the **dplyr** package, respectively), the logical statement is here provided in one string.

```
>>> tips_2 = tips.query("(time == 'Dinner' & day != 'Sun') | size > 4")
>>> tips_2.head()
```

	total_bill	tip	sex	smoker	day	time	size	tip_share
19	20.65	3.35	Male	No	Sat	Dinner	3	0.162228
20	17.92	4.08	Male	No	Sat	Dinner	2	0.227679
21	20.29	2.75	Female	No	Sat	Dinner	2	0.135535
22	15.77	2.23	Female	No	Sat	Dinner	2	0.141408
23	39.42	7.58	Male	No	Sat	Dinner	4	0.192288

3.2.5 Slicing Notation

To select rows based on their position, we can also use indexing via `[]` but instead of a string or list to select columns, we use numeric arguments to select rows. The basic concept of this so-called slicing notation is `object[start:stop:step]`. If we wanted to view the first 10 variables at even locations we would use `object[0:20:2]`. This works for selecting sections of lists, strings and rows within data frames. The value of `start` is inclusive, the value for `stop` is not, i.e., everything up until but not including `stop` will be selected. If the value of `start`, `stop` or `step` is left empty, then the default values will be taken, which are that `start` and `stop` are the first and last values in the object, and `step` is by default 1.

```
>>> # Slicing a list
>>> my_list = [12, 45, 74, 12, 97, 62, 53, 78]
>>> my_list[1::2]
[45, 12, 62, 78]

>>> # Slicing a DataFrame for the 1st row
>>> tips[:1:]
   total_bill  tip  sex smoker  day  time  size  tip_share
0      16.99  1.01 Female    No  Sun  Dinner     2    0.059447

>>> # Slicing a DataFrame to view 1 every 50 rows
>>> tips[::50]
   total_bill  tip  sex smoker  day  time  size  tip_share
0      16.99  1.01 Female    No  Sun  Dinner     2    0.059447
50      12.54  2.50  Male    No  Sun  Dinner     2    0.199362
100     11.35  2.50 Female   Yes  Fri  Dinner     2    0.220264
150     14.07  2.50  Male    No  Sun  Dinner     2    0.177683
200     18.71  4.00  Male   Yes  Thur  Lunch     3    0.213789
```



While indexing in R starts at 1, it starts at 0 in Python. For example, use index 2 to select the third element.

1. Create a subset of the mtcars data which includes cars that have a weight larger than 3, or four cylinders. How many cars are in that dataset?
2. Slice the mtcars dataset to show the first 10 even rows, i.e., the second, fourth, etc row.
3. Add a new variable which contains the ratio of horse power to weight.



Extension

4. Read in the tips dataset (tips.csv).
5. Calculate the mean tip for for each day.
6. Add a new variable which contains the party size as a binary factor (choose your own cutoff). Hint: The pd.cut function works similarly to R's cut function.
7. Ensure that the two categories are "small/medium" (four or fewer people) and "large" (more than four people) and labeled appropriately.

3.2.6 Dictionaries

Dictionaries are objects which hold an unordered collection of key-value pairs. They are defined with curly braces using the following notation: {key : value}, with different key-value pairs being separated with commas.

This can be used to define pairs of old and new names for the `.rename` method.

```
>>> tips_2 = tips.rename(columns = {"time" : "time_of_day", "size" :  
"party_size"})  
>>> tips_2.head()  
   total_bill  tip  sex smoker  day time_of_day  party_size  
tip_share  
0      16.99  1.01 Female    No  Sun      Dinner           2  
0.059447  
1      10.34  1.66  Male    No  Sun      Dinner           3  
0.160542  
2      21.01  3.50  Male    No  Sun      Dinner           3  
0.166587  
3      23.68  3.31  Male    No  Sun      Dinner           2  
0.139780  
4      24.59  3.61 Female    No  Sun      Dinner           4  
0.146808
```

Dictionaries are also used to specify which columns should be summarised or aggregated in which way, using the `.agg` method. Dictionaries can also hold lists as values, allowing us to specify several summary functions for a column.

Some fundamental summary functions such as `min` and `max` are provided by base Python. More statistical summary functions such as `mean` and `median` are provided by the **NumPy** package.

```
>>> import numpy as np  
>>> tips.agg({"size" : min, "tip" : [min, max, np.mean]})  
   size      tip  
max   NaN  10.000000  
mean   NaN   2.998279  
min    1.0   1.000000
```



In R, the `%>%` operator from the **magrittr** package allows us to make pipelines of a series of steps on an object, e.g., a data frame. In Python, the `.` can be used to chain multiple methods on an object together into one line of code, without having to reassign each time.

3.2.7 Working with Different Data Types

A `Series` in a `DataFrame` can hold different types of data such as numeric data, categorical data, datetimes, etc. For most common data types other than numeric, we need so-called accessors to access the type-specific attributes and methods. For example, the accessor for categorical data is `.cat` and the methods include `.cat.rename_categories()`, `.cat.remove_unused_categories()`, and `.cat.reorder_categories()`.



It is, of course, possible to access the categorical data via `.values` and manipulate this directly. However, the resulting object is an array, not a `Series`.

Chapter 4

Visualisation

4.1 Visualisation

The main plotting library in Python is **Matplotlib**. While **Matplotlib** is very powerful, until very recently, it took quite a lot of tweaking to make the plots look good.

Seaborn essentially treats **Matplotlib** as a core library and aims to make visualisation a central part of exploring and understanding data. It also has the goal of simplifying the process of creating more complicated plots, similar to the **ggplot2** package in R. While it does not implement a grammar of graphics, it does:

- create aesthetically pleasing plots by default
- create statistically meaningful plots
- understand the **pandas** `DataFrame` so the two work well together

The **seaborn** library contains functions to create a large number of different types of graphs. The most common graphs are listed below. For a full list of the seaborn plotting functions see seaborn.pydata.org/api.html.

Plot Category	Plot Type	Function
Categorical	Categorical plots on a grid	<code>factorplot</code>
	Box and whisker plot	<code>boxplot</code>
	Point estimates and confidence intervals	<code>pointplot</code>
	Bar plot	<code>barplot</code>
	Violin plot combining a box plot with a kernel density estimate	<code>violinplot</code>
Distribution	Histogram	<code>distplot</code>
	Plot pairwise relationships	<code>pairplot</code>
	Joint of two different plots	<code>jointplot</code>
Regression	Scatter plots with linear regression fit across a grid	<code>lmplot</code>
	Scatter plot with linear regression fit	<code>regplot</code>
	Plot the residuals of a linear regression	<code>residplot</code>
Matrix	Plot rectangular data as a color-encoded matrix	<code>heatmap</code>
	Plot a matrix dataset as a hierarchically-clustered heatmap	<code>clustermap</code>

The function `factorplot` is a wrapper function for the different individual plotting functions and takes a `kind` argument such as `box`, `point`, `bar` and `violin`.



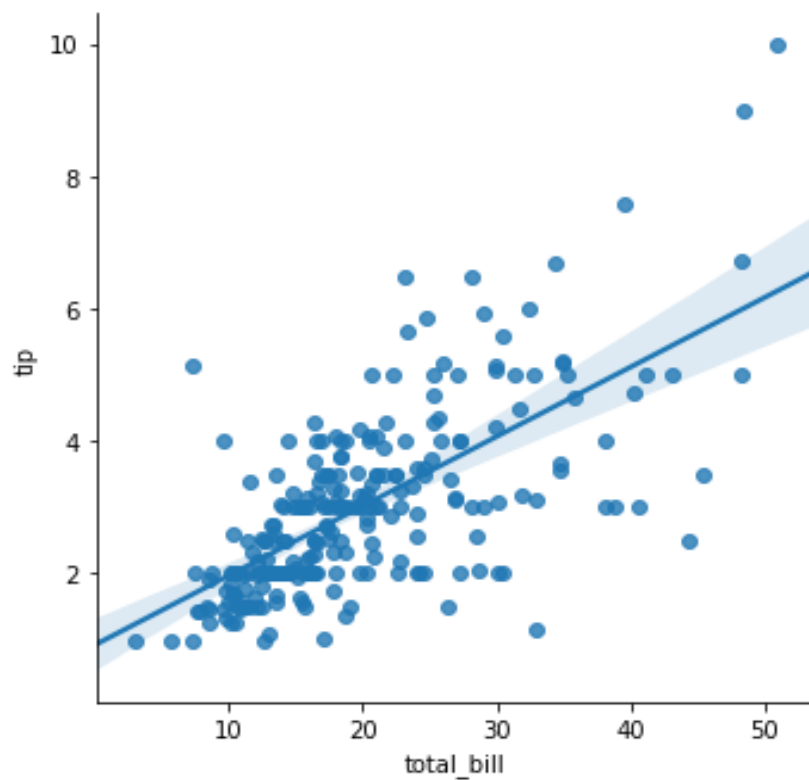
For an online gallery showing examples of graphics built using seaborn see seaborn.pydata.org/examples/.

For the plots to show in line with the cells of our Jupyter notebook, instead of an external graphics viewer, we can change the Jupyter plot settings as follows:

```
>>> import seaborn as sns
>>> %matplotlib inline
```

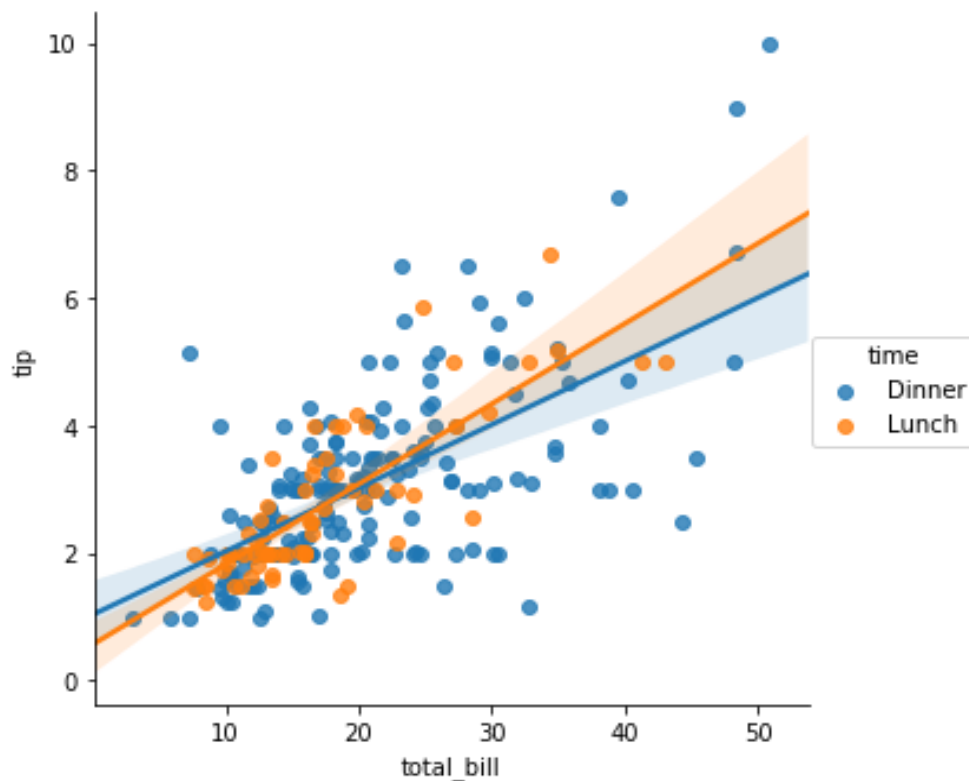
To take a look at the relationship between the tip and the total amount of the bill in the tips dataset, we can use the `lmplot` function.

```
>>> sns.lmplot(x="total_bill", y="tip", data=tips)
```



Similarly to **ggplot2**, we can colour the points according to a variable very easily. Instead of setting a colour aesthetic, here we set the argument `hue` to the corresponding column name.

```
>>> sns.lmplot(x="total_bill", y="tip", hue="time", data=tips)
```



4.1.1 Panelling (Faceting)

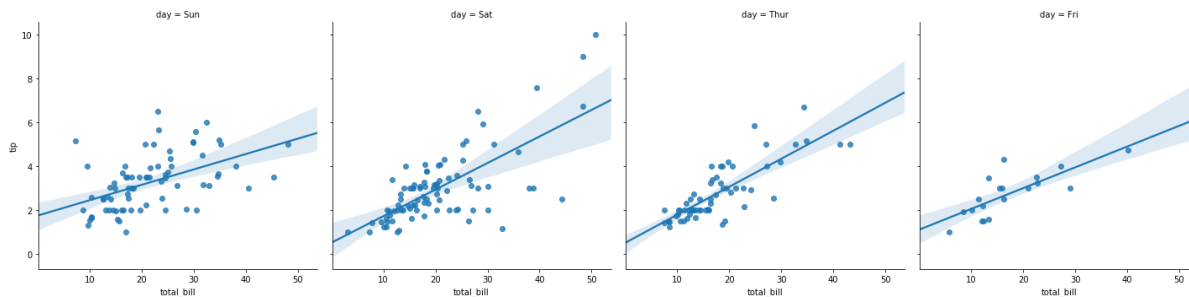
If we want to explore a relationship like that between tip and total amount of the bill for more groups, using colour to distinguish between groups can result in plots that are hard to read. An alternative in those situations can be to plot the data in separate panels. In **ggplot2** this is also called faceting.

While in **ggplot2** we add a facet layer to our plot, in **seaborn** we first create a grid object and then map a plotting function onto the grid. The grids available in **seaborn** are facet grids, pair grids and joint grids. Here we focus on facet grids which allow us to create plots on a grid defined by rows and columns. For details on the other types of grids, see seaborn.pydata.org/api.html.

For categorical and regression plots, the functions `catplot` and `lmplot` provide a convenience interface which takes care of the paneling on a facet grid for us. All we have to specify, via the `row` and `col` arguments, is which variables should be used for rows and columns of the grid.

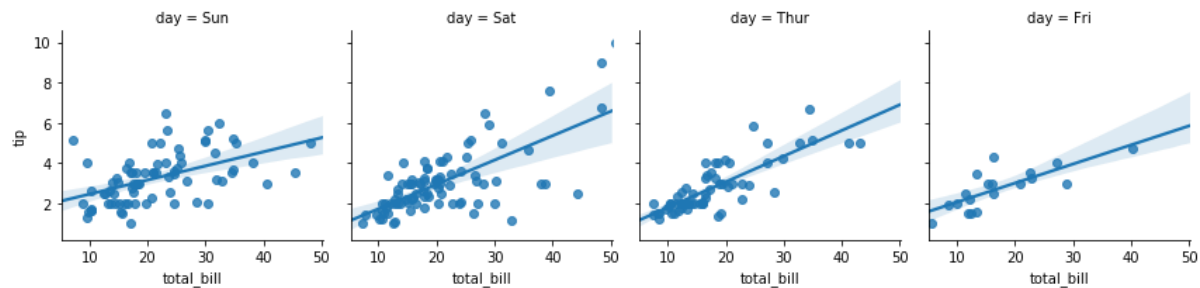
For example, if we wish to plot a `lmplot` for the tips data set, with `tip` against `total_bill` for each day, we get the following graph:

```
>>> sns.lmplot(x="total_bill", y="tip", col="day", data=tips)
```



The general approach to paneling with **seaborn** is illustrated below.

```
>>> g = sns.FacetGrid(tips, col="day")
>>> g.map(sns.regplot, "total_bill", "tip")
```



3.1.1 Controlling the Appearance

To customise our plots, we can

- change the context of the plot
- pick a style
- use a specific colour palette

These can be set directly via the `sns.set_context`, `sns.set_style` and `sns.set_palette` functions or together via the `sns.set` function.

The context has no direct corresponding element in **ggplot2**. This defines the scaling of the plot (font, axes, and content) as appropriate for a paper, notebook, talk or poster.

The style corresponds to **ggplot2**'s themes. Available styles are `white`, `dark`, `whitegrid`, `darkgrid`, and `ticks`.

The following pre-defined colour palettes are available: `deep`, `muted`, `bright`, `pastel`, `dark`, and `colorblind`. Further palettes can be defined with the `sns.color_palette` function.



1. Make a scatterplot with a regression line for `tip` against `total_bill` and differentiate between smokers and non-smokers using colour.
2. Make a scatterplot with a regression line for `tip` against `total_bill` and use a separate panel for each party size.

Extension:

3. Set the context to `notebook`, use a dark style, and a bright colour palette and redo the plot from exercise 1 - how do the two plots compare?
4. Plot the pairwise relationships in the `tips` dataset.

Chapter 5

Modelling

5.1. Statistical Modelling

There are a number of packages that contain functions for statistics and modelling in the Python standard library. We will use the **statsmodel** package which contains a large number of models and statistical tests, organised in several modules. Some of the more common model fitting functions can be found in the following table:

Package/Module	Function	Model
<code>statsmodels.formula.api</code>	<code>ols</code> /OLS	Linear Regression by OLS (Ordinary Least Squares)
<code>statsmodels.formula.api</code>	<code>gl</code> s/GLS	Linear Regression by GLS (Generalised Least Squares)
<code>statsmodels.api</code>	GLM	GLM (Generalised Linear Model)
<code>statsmodels.api</code>	RLM	RLM (Robust Linear Model)
<code>statsmodels.tsa.arima_model</code>	ARIMA	Time Series Analysis - ARIMA Model
<code>statsmodels.formula.api</code>	<code>phreg</code>	Survival Analysis - Cox Models
<code>statsmodels.api</code>	<code>stats.anova_lm</code>	ANOVA (Analysis of Variance)

For a full list of functionality, see the package's site www.statsmodels.org/dev/index.html.

Similar to modelling in R, the model is specified through a formula with a tilde (~) which typically has the response on the left-hand side and the independent variables on the right-hand side. The standard formula notation is the same as in R, e.g.,

Python Formula	Model	Actual Model Formula
<code>y ~ x</code>	Y against X + an intercept	$y = a + bx + \epsilon$
<code>y ~ x-1</code>	Y against X (no intercept)	$y = bx + \epsilon$
<code>y ~ x+z</code>	Y against X and Z + an intercept	$y = a + bx + cz + \epsilon$
<code>y ~ x:z</code>	Y against the X/Z interaction term	$y = a + bxz + \epsilon$

Where in the actual model formulae above, ϵ is the error term.

In R, we typically provide a modelling function with the formula and the data and it returns a fitted model. Here, model specification and model fitting are two separate steps. Using the `ols` function, we specify a linear model for the tip as explained by the total amount of the bill.

```
>>> import statsmodels.formula.api as smf
>>>
>>> m1 = smf.ols(formula="tip ~ total_bill", data=tips)
```

The resulting model object has a `.fit` method which is then used to estimate the parameters of the model. For the fitted model object, further methods and attributes are available, e.g., a summary method (`.summary`) and the estimated parameters (`.params`).

```
>>> m1 = m1.fit()
>>> print(m1.summary())
```

```

                                OLS Regression Results
=====
Dep. Variable:                  tip    R-squared:
0.457
Model:                          OLS    Adj. R-squared:
0.454
Method:                        Least Squares    F-statistic:
203.4
Date:                          Mon, 06 Aug 2018    Prob (F-statistic):
6.69e-34
Time:                          19:15:05    Log-Likelihood:
-350.54
No. Observations:              244    AIC:
705.1
Df Residuals:                  242    BIC:
712.1
Df Model:                      1
Covariance Type:               nonrobust
=====
=====
                                coef    std err          t      P>|t|      [0.025
0.975]
-----
-----
Intercept                    0.9203    0.160      5.761    0.000    0.606
1.235
total_bill                   0.1050    0.007     14.260    0.000    0.091
0.120
...

```



1. Fit a linear model for `tip` as explained through `total_bill` and `smoker`

Alternatively, we can fit a model with an interaction between `total_bill` and `smoker`, and compare this to the simpler model via an ANOVA (analysis of variance).

```
>>> import statsmodels.api as sm
>>>
>>> mf2 = smf.ols(formula="tip ~ total_bill + smoker +
total_bill:smoker", data=tips).fit() # or total_bill*smoker
>>>
>>> sm.stats.anova_lm(mf1, mf2)
```

	df_resid	ssr	df_diff	ss_diff	F	Pr(>F)
0	242.0	252.788744	0.0	NaN	NaN	NaN
1	240.0	229.809386	2.0	22.979358	11.999175	0.000011



You may notice a warning appear when running the above function. This occurs due to the `NaN` values in our output; however, this function will always produce `NaN` values there so this is nothing to worry about.

Warnings appear red in Jupyter, but do not stop the computation of the function, so can in some cases be ignored.

5.2. Machine Learning

While R has its strength in statistical modelling, Python shines in machine learning, in particular with the **scikit-learn** package.

5.1.1 Getting Data into the Right Format

So far, we stored all our data in one `DataFrame`. The functions in **scikit-learn** for models and algorithms expect the features, or explanatory variables, and the target, or response, in two separate objects: The features are in a 2-dimensional array with one row per sample or observation and one column per feature or variable. The target is in an one-dimensional array.

We can convert a `DataFrame` using the `values` attribute. Here we will use the tips data to create an array of features and the one-dimensional array of the tips as our target.

```
>>> import pandas as pd
>>> tips = pd.read_csv("Data/tips.csv")
>>> tips_data = tips.drop("tip", axis=1).values
>>> tips_target = tips["tip"].values
```

The example datasets in **scikit-learn** come in the form of dictionaries which hold the two arrays for features and target along with the corresponding names. Here we will use the iris data which contains three different species of iris flowers as the target and four different measurements (length and width of the sepal and petal, respectively) as the features.

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> iris.keys()
dict_keys(['data', 'target', 'target_names', 'DESCR',
'feature_names'])
```

We will use the generic names `x` and `y` for the features and target, respectively. Python allows us to assign multiple objects at once as follows:

```
>>> X, y = iris.data, iris.target
```

5.1.2 Classification

Classification is common machine learning task so we will use this as example to illustrate a common workflow with **scikit-learn**.

Similar to working with **statsmodel**, specifying a model, also called instantiating a model, is a separate step from fitting a model. Here we will first instantiate a K-nearest neighbor classifier and then fit it on the iris data.

```
>>> from sklearn import neighbors
>>>
>>> # create the model
>>> knn = neighbors.KNeighborsClassifier(n_neighbors=5)
>>>
>>> # fit the model
>>> knn.fit(X, y)
KNeighborsClassifier(algorithm='auto', leaf_size=30,
                    metric='minkowski',
                    metric_params=None, n_jobs=1, n_neighbors=5, p=2,
                    weights='uniform')
```

Predictions can be accessed with the `.predict` method for the model object.

```
>>> knn.predict(X)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0,
      0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1,
      1, 1, 1, 1, 2, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1,
1,
      1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2,
2,
      2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2,
      2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

5.1.3 Model Validation

These predictions can be used to calculate the accuracy of the predictions. However, choosing a model based on its accuracy on the same data that was used to fit the model often leads to choosing a model too closely adapted to the data.

To avoid this over-fitting, the data is typically split into train and test data. We can easily do this with the `train_test_split` function and then train our classifier only on the training set.

```
>>> from sklearn import model_selection
>>>
>>> X_train, X_test, y_train, y_test =
model_selection.train_test_split(X, y)
>>>
>>> knn = neighbors.KNeighborsClassifier(n_neighbors=5)
>>> knn.fit(X_train, y_train)
KNeighborsClassifier(algorithm='auto', leaf_size=30,
                    metric='minkowski',
                        metric_params=None, n_jobs=1, n_neighbors=5, p=2,
                        weights='uniform')
```

Now we can calculate the accuracy on the test set.

```
>>> from sklearn import metrics
>>>
>>> y_pred = knn.predict(X_test)
>>> metrics.accuracy_score(y_test, y_pred)
0.9473684210526315
```

This is also available as a method, `.score`, for the model object.

```
>>> knn.score(X_test, y_test)
0.9473684210526315
```

Cross-validation takes this idea a step further: The data is split into k folds and the model is fit k times, always leaving one of the folds as the test set.

```
>>> from sklearn import model_selection
>>>
>>> cv =
model_selection.cross_val_score(neighbors.KNeighborsClassifier(1), X,
y, cv=10)
>>> cv.mean()
0.96
```

The modules in **scikit-learn** are designed such that we can easily swap out one model for another.



1. Use the SVC function from sklearn.svm to solve the iris classification problem with a support vector classifier. What is the accuracy on the training set?

Extension:

2. What is the average accuracy when using a 10-fold cross-validation?

Chapter 6

More Packages

6.1. Taking Python further

Like R, Python's strength comes from the range of packages that are being developed by the open source community.

So far, we have introduced the popular packages for data science. All of these are part of the standard distribution installed with Anaconda. Below are examples of more great packages for visualisation, modelling and general data science.

6.1.1 More Visualisation

Python has a lot of packages created for data visualisation. **Seaborn** creates flexible graphics for plotting data but it doesn't follow *The Grammar of Graphics* that R users are familiar with from **ggplot2**. Seaborn also has a limited number of diagnostic plots for visualising our models. To overcome these, we could use:

- **ggplot** – a plotting system for Python based on **ggplot2**
- **bokeh** – a package native to Python and using The Grammar of Graphics. You can also use it to create interactive plots in either JSON or HTML for web applications.
- **Yellowbrick** – a package extending on **Scikit-Learn** for creating diagnostic plots to help during model selection.

6.1.1.1 Bokeh example

Below is an example showing how to create an interactive plot in a notebook using **bokeh**:

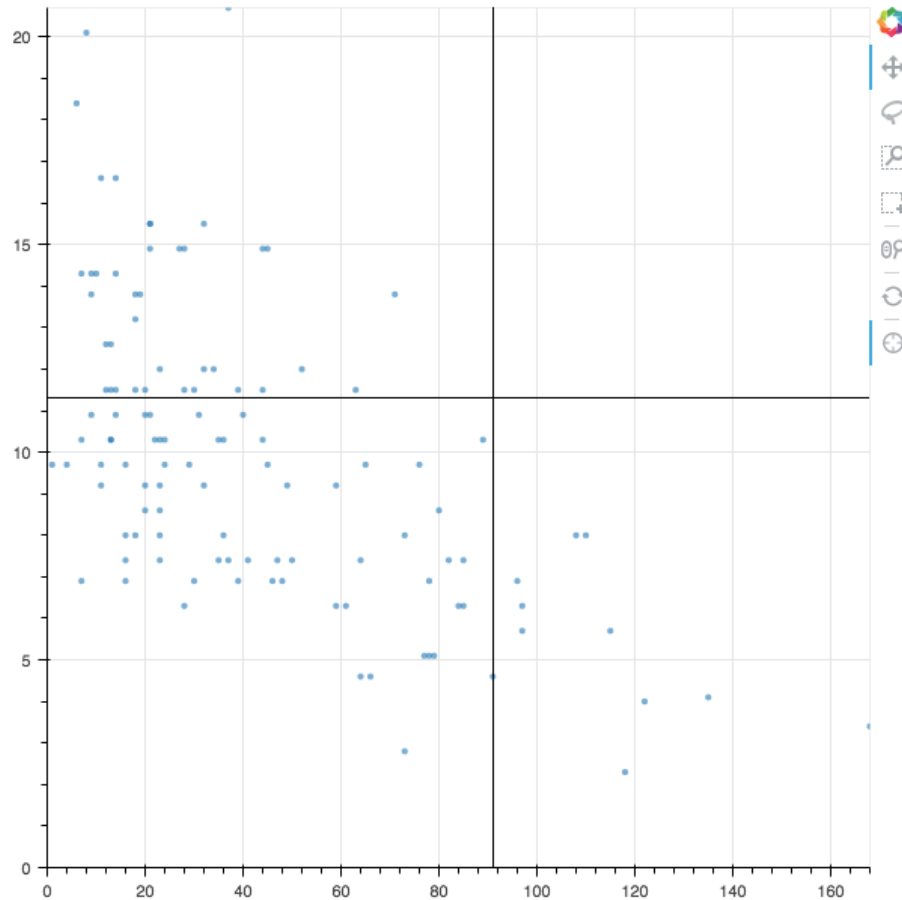
```
>>> import pandas as pd
>>> from bokeh.plotting import figure, output_file, show
>>> from bokeh.io import output_notebook
>>> output_notebook()

# Load some data
>>> airq = pd.read_csv('Data/airquality.csv')
# Select tools
>>> TOOLS="crosshair,wheel_zoom,box_zoom,reset,box_select,lasso_select"

# create a new plot with the tools above, and explicit ranges
>>> p = figure(tools=TOOLS, x_range=(0,airq.Ozone.max()),
y_range=(0,airq.Wind.max()))

# add a circle renderer with vecorized colors and sizes
>>> p.circle(airq.Ozone,airq.Wind , fill_alpha=0.6, line_color=None)

# show the results
>>> show(p)
```



Plotting in **pandas** currently uses the matplotlib library. There is current development to integrate **bokeh** plots into the **pandas** libra

6.1.1.2 Yellowbrick example

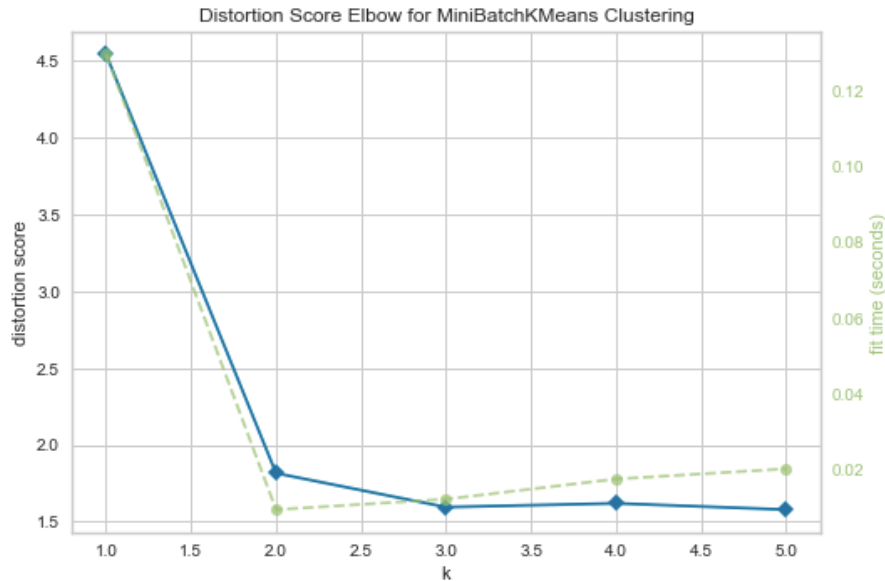
We can use **yellowbrick** with the iris data that we modelled. If we did not know there were three flowers in our dataset (Setosa, Virginica and Versicolor), the Elbow method helps show how many clusters we need to model with. Here, we test using 1, 2, 3, 4, 5 and 6 clusters:

```
>>> %matplotlib inline
>>> from sklearn import datasets

>>> iris = datasets.load_iris()
>>> X, y = iris.data, iris.target

>>> from sklearn.cluster import MiniBatchKMeans
>>> from yellowbrick.cluster import KElbowVisualizer

# Instantiate the clustering model and visualizer
>>> visualizer = KElbowVisualizer(MiniBatchKMeans(), k=(1,6))
>>> visualizer.fit(X) # Fit the training data to the visualizer
>>> visualizer.poof() # Draw/show/poof the data
```



6.1.2 More Modelling

6.1.2.1 Gensim

Gensim is a Python library for topic modelling, document indexing and similarity retrieval with large corpora. Target audience is the natural language processing (NLP) and information retrieval (IR) community.

6.1.2.2 PyBrain

PyBrain is a popular package for working in Reinforcement Learning, Artificial Intelligence and Neural Networks. PyBrain contains algorithms for neural networks, for reinforcement learning (and the combination of the two), for unsupervised learning, and evolution.

6.1.3 A few more packages

There are many more packages, e.g.,

- Beautiful Soup (**beautifulsoup4**)– a package for working with website text (HTML, XML). Beautiful Soup is similar to the tidyverse's rvest
- **Pipenv** – a software development package aimed at streamlining development and making managing requirements easy.
- **NLTK** – the Natural Language Toolkit is a package created to contain a rich set of natural language processing tools and make prototyping quick.