WMG
THE UNIVERSITY OF WARWICK

Introduction to Python

Pre-Module Work: *Computer Programming* and *Computational Statistics with Python*

Dr Michael Mortenson, WMG

Version 2.0: December 2019

Table of Contents

1. Introduction 5

1.1. What is this Book for? 5

1.2. Why Python? 5

1.3. Working with Github 6

2. Installing Python and the Jupyter Notebooks 7

2.1. Installing Anaconda and Python 7

2.2. Setting a Working Directory 11

2.3. The Jupyter Notebook Console 16

2.4. Additional Resources and Documentation 28

3. Creating a Python Program 29

3.1. A First (and Second) Program 29

3.2. Data Types 34

3.3. Python Lists 44

3.4. Other Data Structures: Dictionaries, tuples and sets 51

3.5. If Statements and Loops 58

3.6. Functions 68

3.7. Programming Best Practice 74

3.8. Python Operators 78

3.9. Exercises 79

4. Packages and Modules 79

4.1. Installing with Pip 83

4.2. Creating Modules 86

4.3. Exercises 89

5. Summary and Further Reading 89

List of Figures

Figure 1: Anaconda Download 8
Figure 2: Anaconda Version (Windows and Mac) 8
Figure 3: Jupyter Notebook: Kernel 9
Figure 4: Jupyter Notebook: Browser 9
Figure 5: Jupyter Notebook: New notebook 10
Figure 6: Jupyter Notebook: New program 10
Figure 7: Jupyter Notebook: Running code 11
Figure 8: Jupyter Notebook: Home page view 12
Figure 9: Renaming the Root Directory 13
Figure 10: Creating a New Directory Name 13
Figure 11: Renamed Folder Name 13
Figure 12: Updated URL and Folder Display 14
Figure 13: New Notebook in URL 14
Figure 14: Creating a Local Folder 15
Figure 15: Verifying a Local Folder Assignment 16
Figure 16: Rerunning Code 17
Figure 17: The 'Cell' Menu 17
Figure 18: The 'File' Menu 18
Figure 19: Renaming a Notebook 19
Figure 20: Verifying a Notebook Name 19
Figure 21: Running the "print" Command 20
Figure 22: Last Checkpoint 20
Figure 23: Error Messages 21
Figure 24: Reverting Notebook to the Last Checkpoint 22
Figure 25: Verifying Checkpoint 22
Figure 26: Opening a Saved Notebook 23
Figure 27: Accessing a Saved Notebook 24
Figure 28: ZeroDivisonError 25
Figure 29: Deleting Cells 25
Figure 30: The "Kernel" Menu 26
Figure 31: Error Message from Interrupting a Kernel 27
Figure 32: Interrupt Button in the GUI 27
Figure 33: The "Hello, World" Program 29
Figure 34: The "Hello & Goodbye" Program 30

Figure 35: Retrieving a Value Stored as a Variable 31
Figure 36: Combining Variables in a Print Function 32
Figure 37: Our Second Program 33
Figure 38: Our Second Program (changed value) 34
Figure 39: Addition in Python 35
Figure 40: Addition in Python (string plus integer) 36
Figure 41: Addition in Python (integer plus string) 36
Figure 42: Process Strings with Quotation Marks 37
Figure 43: Using Backslash to Ignore Escape Characters 38
Figure 44: Integer and Float Addition 39
Figure 45: Integer and Float Multiplication 39
Figure 46: Type Conversion using str( ) 41
Figure 47: Type Conversion using int( ) 42
Figure 48: Float to Integer Conversion 42
Figure 49: Converting Floats using round( ) 43
Figure 50: A Python List 45
Figure 51: Indexing the Third Item in a List 45
Figure 52: Slicing a Python List 46
Figure 53: Open Ended Slices 46
Figure 54: Slicing using Negative Indices 46
Figure 55: Indexing a List inside a List 47
Figure 56: Slicing a String Object 47
Figure 57: Changing a Value in a List 48
Figure 58: Appending an Item to a List 48
Figure 59: Inserting an Item into a Specific Position in a List 48
Figure 60: Removing the Last Item from a List 49
Figure 61: Removing an Item from a List by its Position 49
Figure 62: Removing the First Occurrence of a Value in a List 49
Figure 63: Sorting a Python List 50
Figure 64: Reverse Sorting and Indexing a List 50
Figure 65: Counting the Occurrences of a Value in a List 50
Figure 66: A Python Dictionary 52
Figure 67: Adding New Items to a Dictionary 52
Figure 68: Updating a Dictionary Item 52
Figure 69: Adding a List as a Value in a Dictionary 53
Figure 70: Calling Items in a Nested Dictionary 53

Figure 71: Deleting an Item from a Dictionary 54
Figure 72: Sorting a Dictionary 54
Figure 73: A Python Tuple 55
Figure 74: Tuples are Immutable 55
Figure 75: Changing Mutable Items inside a Tuple 56
Figure 76: Changing a Tuple 56
Figure 77: TypeError when Indexing a Set 56
Figure 78: Adding and Updating a Set 57
Figure 79: Removing Item from a Set using discard( ) 57
Figure 80: Testing Set Membership 58
Figure 81: Customer Dictionary Output 59
Figure 82: A Python If Statement for a Customer with Open Orders 60
Figure 83: A Python If Statement for a Customer without Open Orders 61
Figure 84: Counting Items in a List using len( ) 62
Figure 85: The Logic Flow of our Nested If Statement 63
Figure 86: The Logic Flow of our Second Nested If Statement 64
Figure 87: Output of our Second Nested If Statement 64
Figure 88: A For Loop 66
Figure 89: A For Loop using the range( ) Function 66
Figure 90: A While Loop 68
Figure 91: Printing all Customer Orders using a For Loop 71
Figure 92: The order_printer Custom Function 73
Figure 93: Adding Error Handing to the order_printer Customer Function 73
Figure 94: Example Output from our Play Your Cards Right Simulator 82
Figure 95: Installing flashtext in the Notebook Interface 84
Figure 96: Upgrading a Package using pip 84
Figure 97: Converting Text to Lower Case using lower() 86
Figure 98: Find Keywords using flashtext 86
Figure 99: Using our Custom Module 89

1. Introduction

The *Computational Statistics with Python* module (CSP) focuses on computational approaches to statistics; approaches to statistics and data analysis made available through modern computing. The *Computer Programming* module (CPRO) focuses on developing applications, particularly web-based application, using modern frameworks and tools. For both modules, however, the initial focus is on providing familiarity with a programming environment in which to work. For these modules the chosen environments is Python.

1.1. What is this Book for?

The principal goal of the book is to provide participants with the fundamentals of Python programming required to complete the exercises in the module, and the coursework (assessed, in part, by a timed test which starts the module). For the test, participants will be asked to create programming scripts based on the content of this book. Participants will be permitted access to this book during the test, as well as internet access. Any participant who completes the core exercises in this book (chapters two to five), should be adequately equipped to successfully complete this test.

Additionally, the book aims to introduce participants to many of the common programming patterns and techniques that are used in almost all languages and environments. Finally, the book aims to provide participants with tutorials and scripts that can be used for common eBusiness and programming tasks potentially relevant to projects and/or future work.

1.2. Why Python?

There are many languages used for computational statistics, and the choice of Python is not to say that this is necessarily the "best" language for such tasks. There are, however, several reasons why Python can be considered a good choice:

1) Python is one of the most widely used programming languages. According to the TIOBE Index for December 2018, Python is the third most "popular" language, behind only Java and C (https://www.tiobe.com/tiobe-index/);
2) In the data science and analytics communities, Python is often considered *the* most popular. In a KDNuggets survey for 2018, Python was reported to be the most widely used tool for such tasks, above RapidMiner and R (Piatetsky G, 2018. *Top Analytics, Data Science, Machine Learning Tools*. https://www.kdnuggets.com/2018/05/poll-tools-analytics-data-science-machine-learning-results.html).

3) Python is open-source and free to download and use.
4) Python is considered an interpreted language, meaning that often scripts are shorter and easier to develop. For instance, in comparison to a compiled language such as C or Java, in Python there is no requirement to declare data types and the language will assume the type based on the input, potentially saving 000's of line of code on large projects.
5) Python has a rich array of packages (libraries) to support statistical and data science work, many of which are introduced in this book and in the module.

However, as discussed it is certainly not the only option. Other languages and software commonly used in computational statistics include R, Matlab, SPSS, SAS, RapidMiner, Java, Julia, Scala and SQL.

1.3. Working with Github

Modern data science development, and indeed modern development as whole, uses some form of "distributed version control", a system where multiple developers can work on the same code simultaneously by marking each development with a version number or label, almost all of which is based on Git (https://git.kernel.org/pub/scm/git/git.git/). Of the various implementations of Git, the best known (purchased by Microsoft in June 2018) is Github (https://github.com/).

Github will be an important tool in this module, for both "pulling" and "pushing" code. In terms of the former, this means that various scripts/solutions used in this book can be "pulled" (downloaded or copy and pasted) from the Github repository setup for this course (https://github.com/MJMortensonWarwick/msc_programming_python). To the latter, during the module you will be "pushing" (publishing online) the code that you create using Github.

As a first step, you will need to setup a Github account. To do so, please go to https://github.com/ and setup an account with your preferred username, email address and password. You will need to verify for your email in the usual fashion. Another useful tool is Github Desktop (https://desktop.github.com/) which makes it easier to pull and push code from the site (both Mac and Windows versions are available). You will need your username and password to complete this. A short tutorial on how to use Github Desktop (and Github in general) is available here: https://www.youtube.com/watch?v=77W2JSL7-r8.

2. Installing Python and the Jupyter Notebooks

There are multiple versions of Python and multiple implementations. At the time of writing, and the version recommended here is Python 3.7. However, any version of Python that is 3.X (i.e. is at a minimum version 3.0) will work for these exercises, and most will work in Python 2 (versions 2.0 to 2.7). The move from version 2.X to 3.X has been quite a slow process, with lots of developments and packages only maintained in Python 2. However, in the vast majority of cases Python 3 is now preferred and is therefore the version recommended here.

For this book, and indeed recommended in general, we are using a specific implementation of Python based on the Anaconda distribution. This gives us access to Python 3.7, but also a range of useful tools such as package management (via Conda - https://docs.conda.io) and also, importantly for this book, a variety of interfaces to Python beyond the standard command line (programming straight into a shell with no support). This includes:

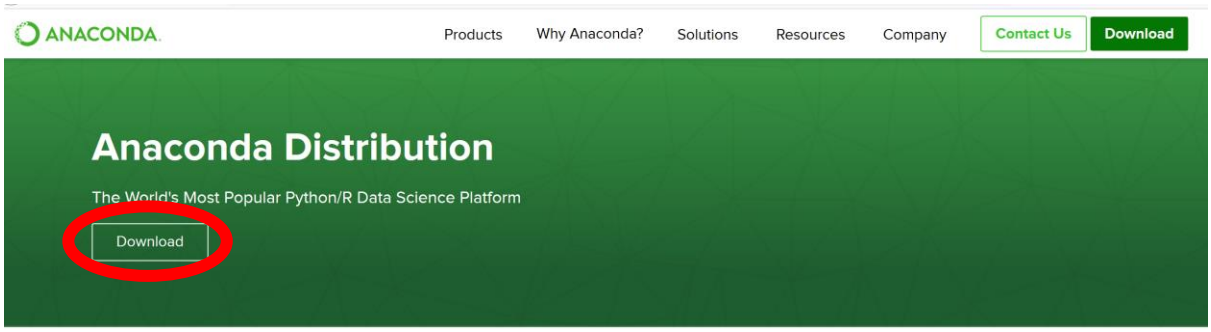
1) The IPython Shell: A more interactive version of the command line shell allowing for additional support (https://ipython.org/);
2) Spyder: An interactive development environment (IDE) designed specifically for scientific computing and engineering work flows (https://www.spyder-ide.org/);
3) Qt Console: A more interactive version of IPython essentially with graphical user interface (GUI) functionality such as multi-line editing and inline graphics (https://qtconsole.readthedocs.io/en/stable/);
4) The Jupyter Notebook: An interactive web-based console/GUI for developing, and more importantly, sharing Python code, text and visualisations.

For this workbook the recommended environment to use will be Jupyter Notebooks (4). The main reason behind this, is the increased interactivity and the ease with which Notebooks can be shared. The remainder of this chapter will describe installing the software, and getting started in the environment.

2.1. Installing Anaconda and Python

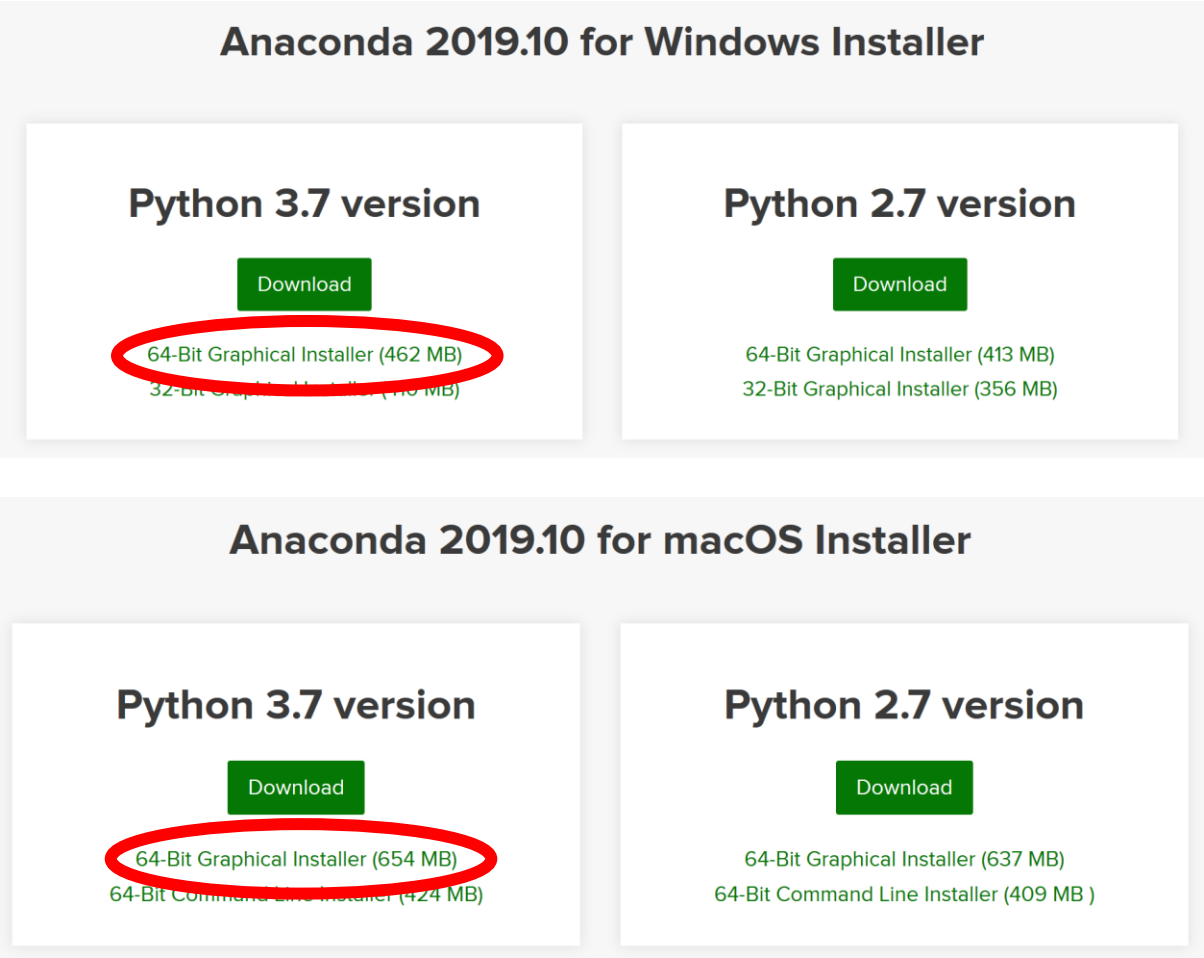
Fortunately, installing Python and other add-ons (such as Anaconda/Jupyter Notebooks) is considerably easier than a few years ago. Firstly, visit https://www.anaconda.com/distribution/ from your browser of choice. The correct version for your operating system (i.e. Windows or Mac) can be selected (if this does not happen automatically) by clicking on the download button (circled in red in **Figure 1**).

Figure 1: Anaconda Download



You then need to select the version to download. Irrespective of your operating system, you need to select the Python 3.7 version. For Windows, select the appropriate bit-type for your machine (usually 64-bit on modern machines), and for Mac select the "Graphical Installer" (as opposed to the "Command Line Installer"). These are shown in **Figure 2**.

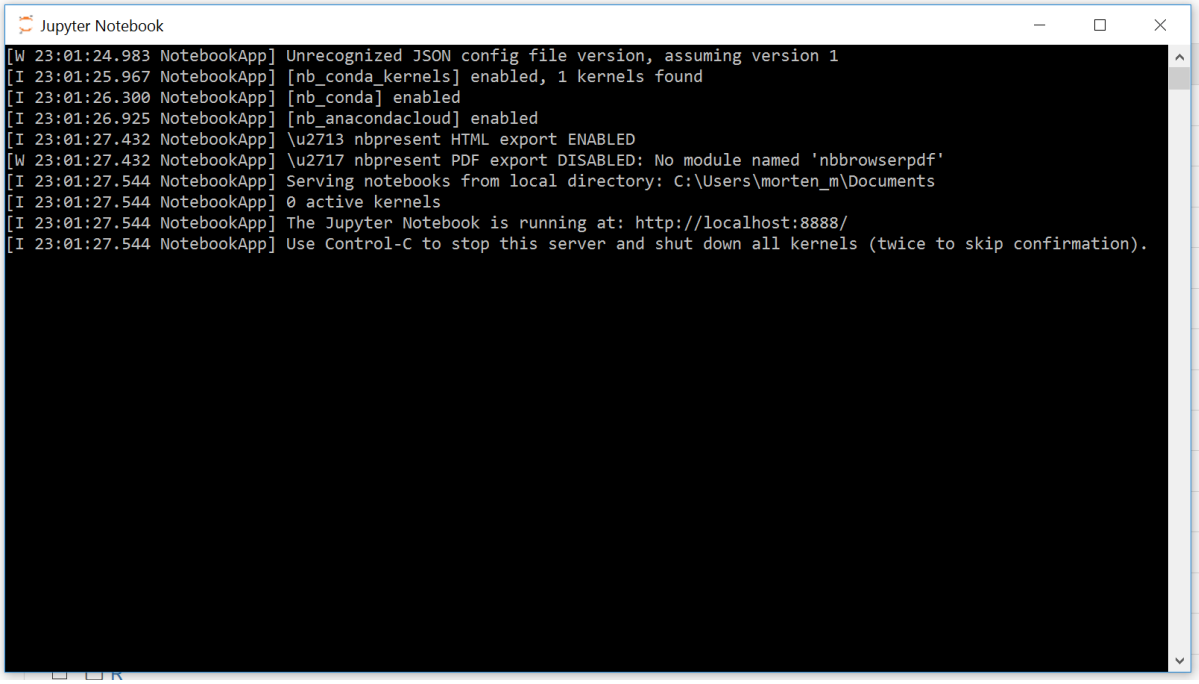
Figure 2: Anaconda Version (Windows and Mac)



An installer will download, which will click-to-run in the usual fashion, and launch a GUI that you'll need to agree the terms of service and specify users, as normal. Unless you have particular reason not to, tick to make Anaconda your default Python 3.7 application and select an appropriate folder to extract the application to. Once everything has installed you should see an "Anaconda" folder with various apps and folders within it.

To start the application search for "jupyter notebook" (in the usual way for your operating system) and click to start. The application is essentially two parts, a kernel window and a browser window. The kernel window will start first and look something like **Figure 3**:

Figure 3: Jupyter Notebook: Kernel



After the start-up code has run (will take ~1 minute and should display similar to the above) in the kernel window, a browser window will automatically open via your default browser (e.g. Firefox, Chrome, Safari, etc.) and look like **Figure 4**.

Figure 4: Jupyter Notebook: Browser

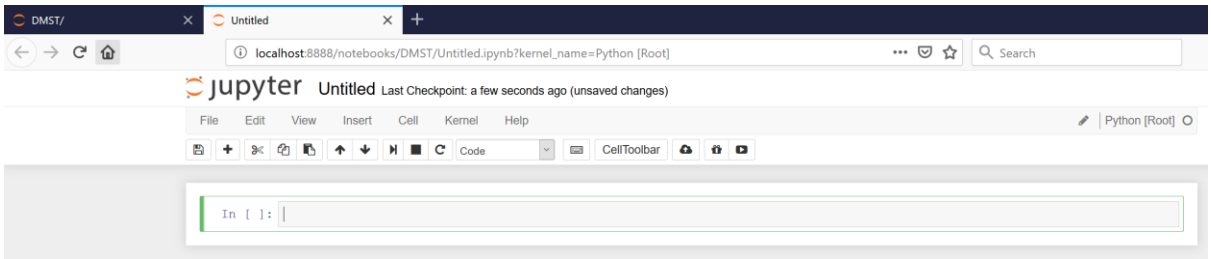
You can note that the URL is slightly different to a standard web URL. Rather than starting "http://", "https://", or "www.", the URL begins "localhost:8888/". What this means is that the de-facto web server is not online but on the machine itself (localhost) and the Notebook is running on port 8888 (for reference, normal http traffic is port 80 and https 443). For this reason, the URL is only valid on the machine which the kernel window is working on, so cannot be copied and pasted on to another machine (for instance).

The initial window will be linked to the current working directory (more on this in section 2.2), and offer the ability to open existing Notebooks you may have previous saved (discussed in section 2.3). You can also start a new Notebook by clicking on "New" in the top right corner of the console (shown in **Figure 5**):

Figure 5: Jupyter Notebook: New notebook

There are various options here (which may vary depending on your machine), such as creating a text file (e.g. web page) or folders. The part we are interested in, for all of the exercises in the book, will be listed under the heading "Notebooks" and will probably say "Python [Root]" (as above) or "Python 3" (or similar). This creates a new Notebook file (with the extension ".ipynb"). If everything has worked a new tab should open with your Notebook file (named "Untiltled1.ipynb") which looks something like **Figure 6**:

Figure 6: Jupyter Notebook: New program



The ipynb extension contains the contents of your code and outputs as JSON (JavaScript Object Notation) data. This JSON data is then rendered in the web page.

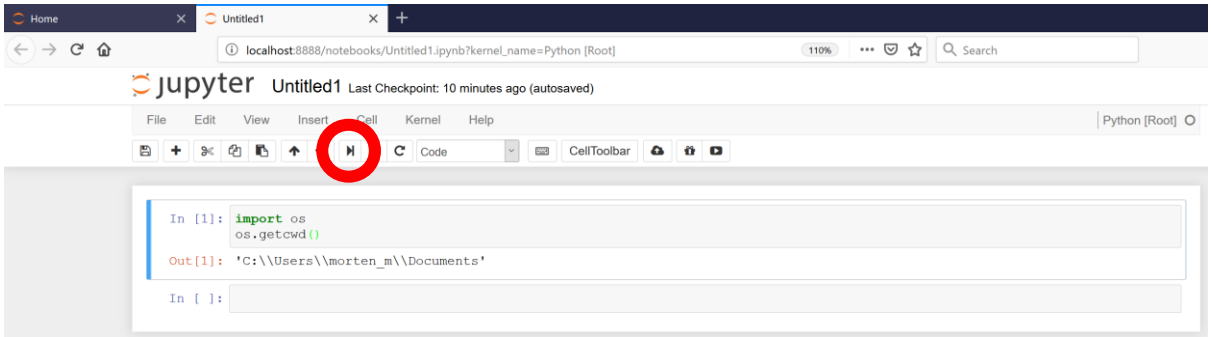
2.2. Setting a Working Directory

When programming, as with most things in life, it is sensible to save your work and have an organised space (i.e. a folder) where you keep everything. When you first start up Jupyter Notebooks the software will select a directory to work in (in my case, for instance, it defaults to `'C:\\Users\\morten_m\\Documents'`). To find the folder you have defaulted to type the following into your new Notebook started in section 2.1:

```
import os
os.getcwd()
```

To run this code in your Notebook you need to click on the run symbol, , located in the header bar (circled in the figure below). This will run your code and produce the output in **Figure 7** (if written correctly!):

Figure 7: Jupyter Notebook: Running code



You will notice there are two parts, the "In [1]" section which shows the code you have entered, and the "Out [1]" section which shows the output of your code. Note: if the code has an error it will also be shown in this "Out" section. Also worth noting is that some code will not have any output.

It is probably worth a brief diversion to give some intuition into how the code we have typed in works. The first line is "`import os`". This makes a call to one of Python's in-built modules "os" (short for operating system). Modules will be discussed in more detail in chapter four, but the brief summary is that these are libraries that have pre-built functionality included. In this case, the "os" library gives Python functions which enable access to information stored in the operating system. The second line is "`os.getcwd()`". The first part, ("os."), tells Python that we are using a function from the "os" module. The second part is the function itself, "getcwd". This is essentially a short-hand for "get current working directory". The final part is empty parentheses, "()". This is common to all Python functions, and allows us to pass certain parameters or values to the function (explained in

chapter four). Because we have no parameters to pass the parentheses are empty. In other words, we are using an external module called "os", from which we are calling the "getcwd" (get current working directory) function.

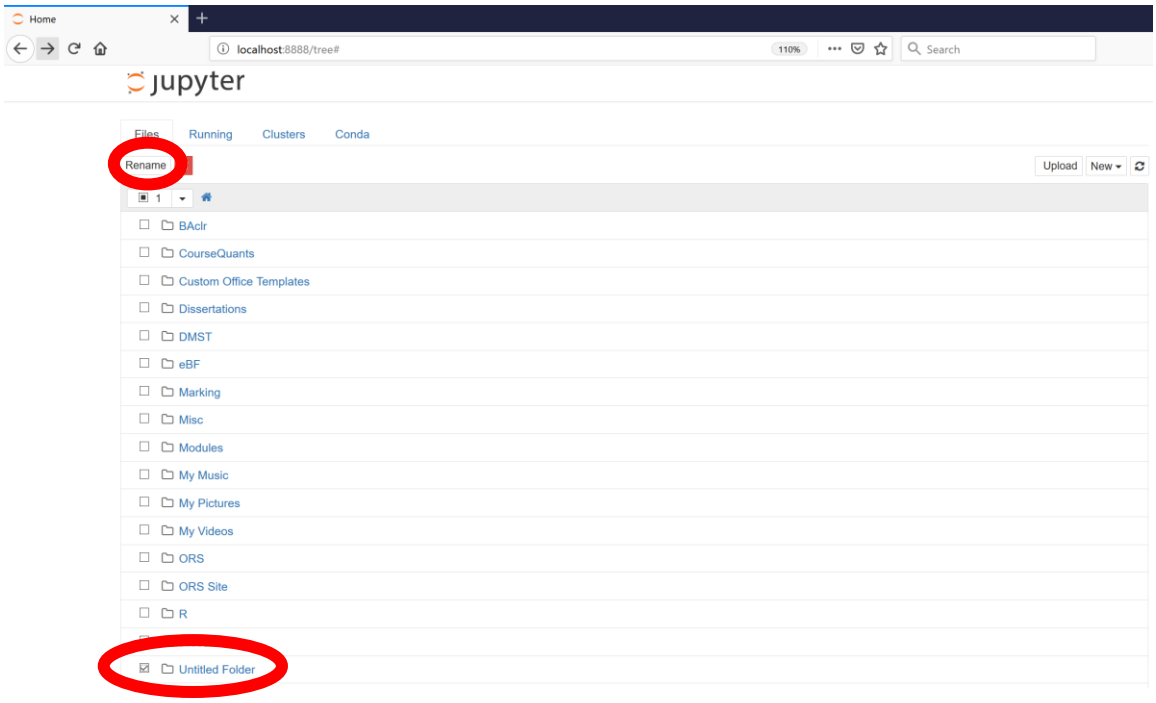
Now that we know our current directory we have two options. Either we are happy working in this directory and just want to build a new folder within it, or we wish to change the directory to another one on our machine (e.g. the Desktop).

The first case is the easiest, and would be strongly recommended for this project. Firstly close down the new Notebook tab ("Untitled1"), but leave open the first page of the console ("Home"). This will return you to the following UI (**Figure 8**):

Figure 8: Jupyter Notebook: Home page view

We can actually create a new folder from this interface. Again click on "New" in the top right corner but this time select "Folder". This will create a new folder in the directory named "Untitled Folder". We can just stop there, but this is not very specific nor easy to remember. It would make sense to rename it to something more sensible. To do this we need to check the box next to our new folder and then the "Rename" button in the top left corner (**Figure 9**):

Figure 9: Renaming the Working Directory



Let's give it a more sensible name such as "CSP" by typing this in the textbox and clicking "OK" (**Figure 10**):

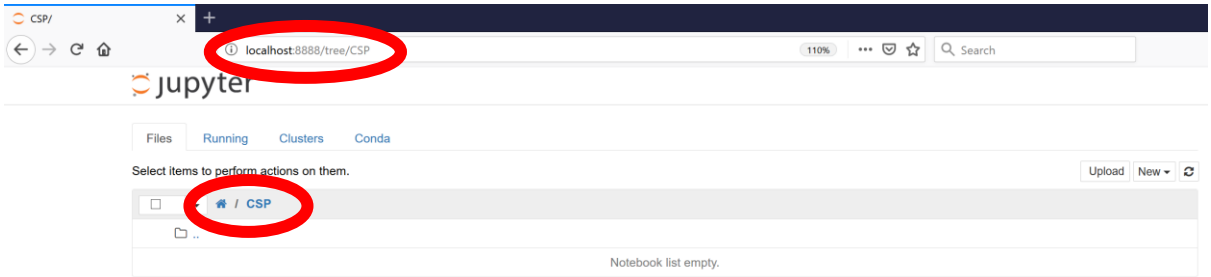
Figure 10: Creating a New Directory Name

Now our folder should be renamed and visible on the main screen (**Figure 11**):

Figure 11: Renamed Folder Name

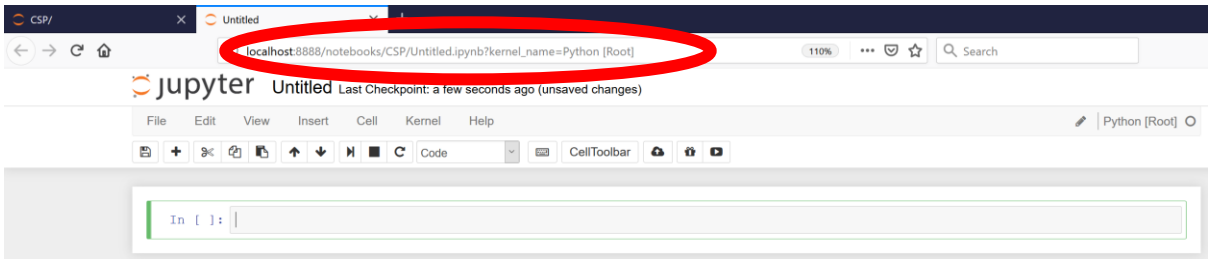
Click on the folder name (e.g. "CSP"), and you will enter into this directory. As this is a new folder it will be empty. You can confirm that you are in the folder because the URL has changed (it will now be something like "localhost:8888/tree/CSP") and also the path is changed in the UI to show the home symbol and the path to CSP (shown in **Figure 12**).

Figure 12: Updated URL and Folder Display



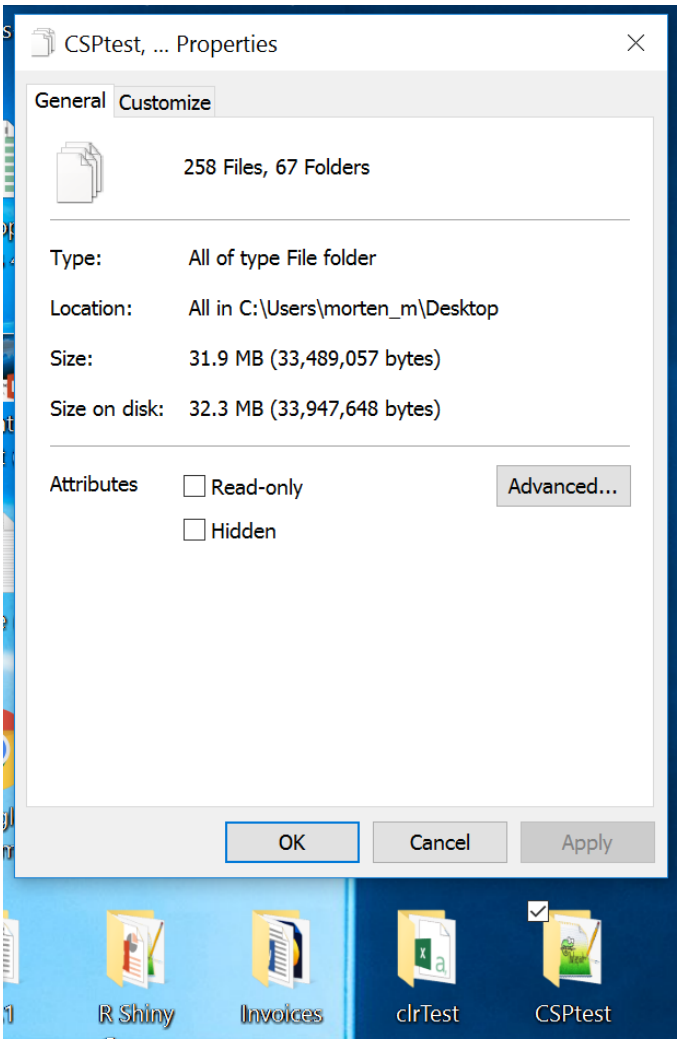
If you now create a new Notebook (as described at the top of page 10) it will be within this folder. This can again be confirmed by looking at the URL of the page (**Figure 13**):

Figure 13: New Notebook in URL



If you are not happy with the current folder, for instance if you want the folder to be on your desktop rather than in documents, the process is slightly more complicated (both at this early stage and also when we want to open files). For this reason I am not recommending doing this, but the instructions are here for reference or if you think it is necessary for the way your machine is set up. Firstly, we need to create the folder in our desired destination, as shown in **Figure 14** (Windows output):

Figure 14: Creating a Local Folder



We then need to return to a Notebook (either use the one we started earlier or by following the steps set out on page 10). In this Notebook we type the following code:

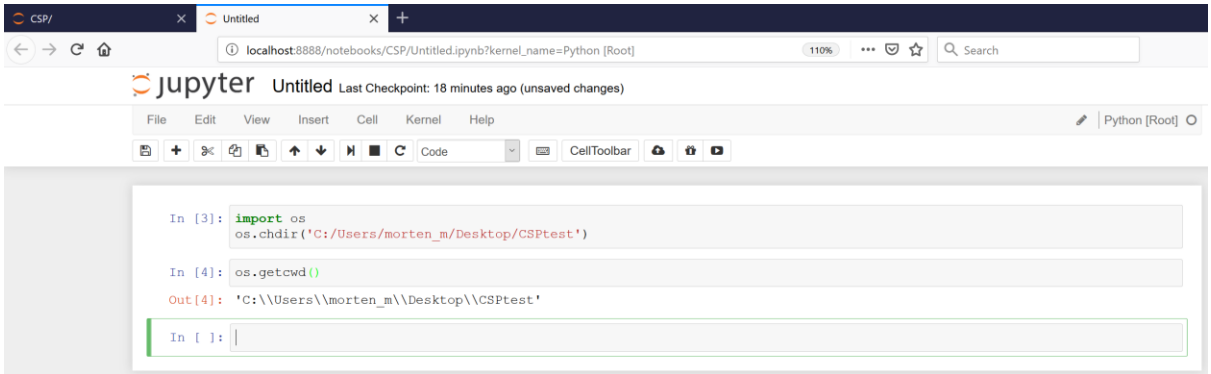
```
import os
os.chdir('C:/Users/morten_m/Desktop/CSPtest')
```

The second line will need to include the full path of the folder we created wrapped in apostrophes. For instance, if your folder is in "D:/Desktop" and titled "CSP", the path would be "D:/Desktop/CSP". Also note, for Windows users, the path will be displayed with the slash symbol ("/") the wrong way round. E.g. the above screenshot has the path as "C:\Users\morten_m\Desktop" rather than "C:/Users/morten_m/Desktop".

Again it is worth briefly discussing the code we just ran. As before we are using (and importing) the "os" module. This time our command is "chdir", short for "change directory". This time, however, there is a parameter/value passed between the parentheses, the full pathway of our folder.

We have now changed the working directory to our new folder. This can be confirmed by running the code presented at the bottom of page 10 again, which should produce the following output (**Figure 15**):

Figure 15: Verifying a Local Folder Assignment



As can be seen, the first command ("In [3]") produces no output. The second ("In [4]"), produces the new pathway that will be used which *should* be the path to our new folder.

Note that because I am using a Windows machine, the output shows double-backward slashes ("\\") rather than a forward slash ("/"). Python regards these as the same, and this is just another way of dealing with the fact that Window's paths use the wrong direction of slash. Mac users would see a single forward slash in the output.

We have now set up a folder to work in so that we can save and return to our work (discussed further in section 2.3). Remember to ensure you move the Jupyter Notebook interface to your preferred folder each time you start the application (i.e. repeat the relevant steps in this section – dependent on the approach you have used).

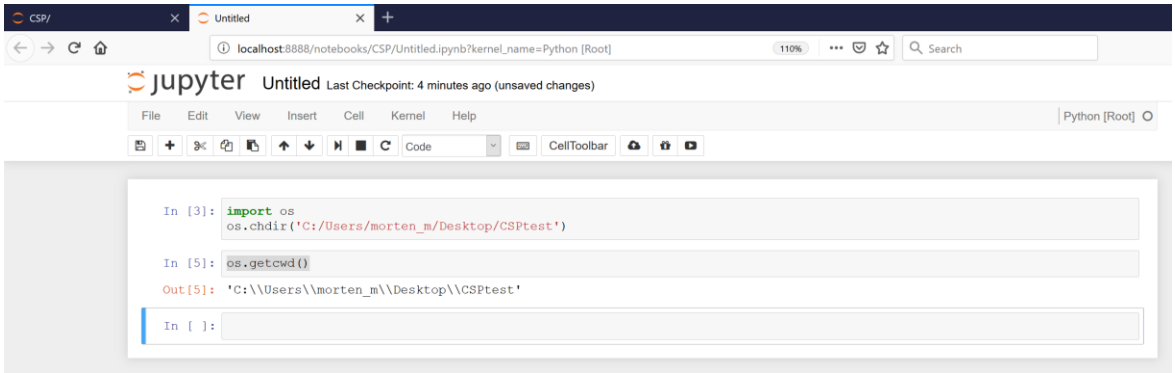
As a note, there are various ways to change the default settings of Jupyter so that the above steps are not necessary. However, they are slightly more technical than would be desirable for the level this book is pitched at. However, if you do wish to do this, the following link would likely be of use: https://stackoverflow.com/questions/15680463/change-ipython-jupyter-notebook-working-directory.

2.3. The Jupyter Notebook Console

Now that we have installed all the software and pointed the application to a suitable folder (and even ran our first lines of code!), we can explore the rest of the interface.

In practice, there are very few elements that will be commonly used. Probably the most used is the "run" button, , introduced on page 10. This will run the whole code of your notebook, or can be used to run a specific line(s) of code if you highlight these via dragging the cursor over them first as shown in **Figure 16**:

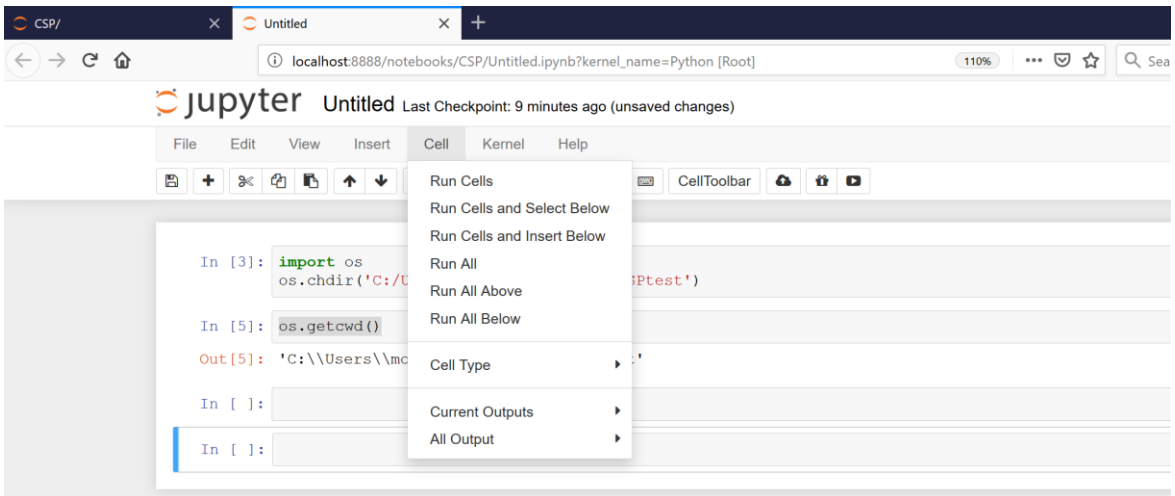
Figure 16: Rerunning Code



In this example, I have rerun the "os.getcwd()" command by first highlighting the text. Click at the start of the text, keep the click button depressed until you have moved the mouse pointer to the end of the line and then release. The text will now be highlighted in grey shadow. If we now press the run button only the highlighted code runs (i.e. the commands in "In [3]" are ignored).

We can achieve the same result, albeit in a slightly less intuitive way, by using the "Cell" heading in menu bar at the top of the page (**Figure 17**):

Figure 17: The 'Cell' Menu



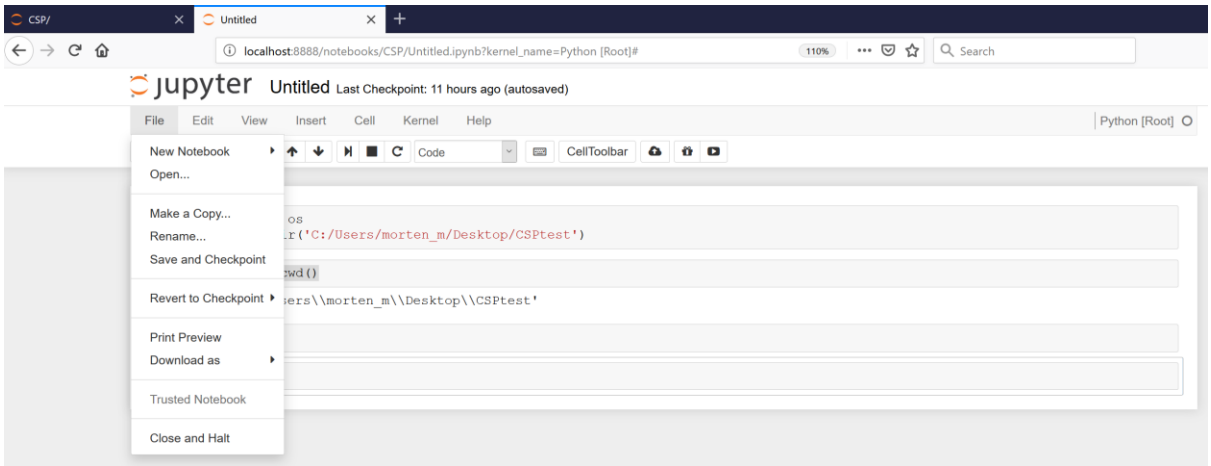
The key options here are "Run Cells" (meaning run the highlighted parts of the code); "Run All" (meaning run every line of code in the Notebook); "Run All Above" (meaning run the

highlighted code, but after running all lines of code above the highlighted code); and finally "Run All Below" (run the highlighted code and every line beneath it). The latter two become useful when we start working with longer sets of code and/or for debugging (discussed in section 3.7).

File Menu

Another key part of the menu bar is "File", shown in **Figure 18**. Many of the options here are self-explanatory. "New Notebook" will open a new tab with an additional Notebook (useful if you want to test a small amount of code before committing to your main code). "Open" allows you to open an existing Notebook from your working directory (again into a new tab). Some others are worth a little more discussion.

Figure 18: The 'File' Menu



Alongside creating a dedicated folder, it will significantly help our organisation if you save your Notebooks with appropriate names. In certain projects, such as building a web application, the file names are relatively prescriptive (see section the exercise of 7.3 for an example of this), however, typically you will have free choice as to how you name your files. It is strongly recommended that you follow some naming conventions that make sense to you, to allow you to easily remember what each Notebook contains.

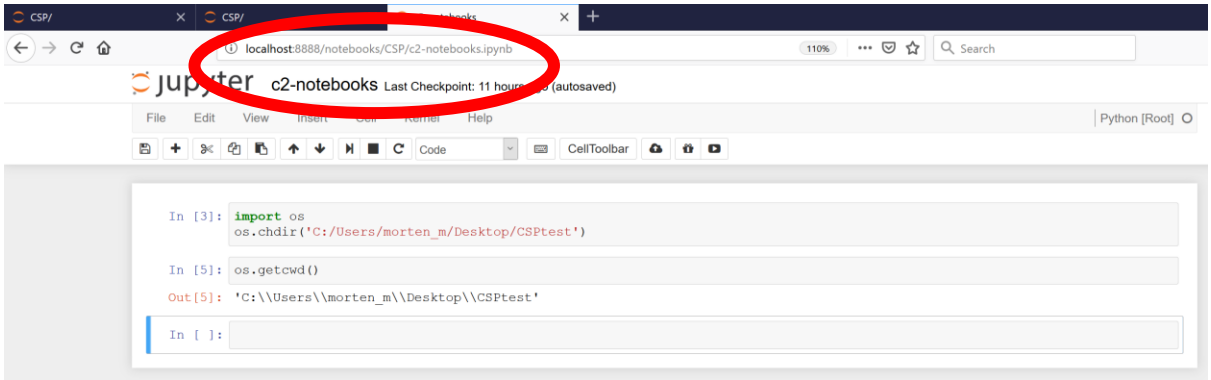
For our project (i.e. this book), a sensible structure maybe something like "[chapter]-[name]". So for this particular Notebook we can save as "c2-notebooks" (and the next section would be "c3-programs"). This means we can quickly find the appropriate Notebook by either chapter number or by title. While this may seem like overkill, over time you will quickly build a lot of Notebooks (and other similar files) and it will become confusing.

To change the name of the Notebook simply click on "Rename" from the "File" menu. This opens a self-explanatory modal (pop-up) that we can use to change the name (**Figure 19**):

Figure 19: Renaming a Notebook

We can verify the change from both the URL and the file name at the top of the page:

Figure 20: Verifying a Notebook Name



Another option worthy of discussion is "Save and Checkpoint". Effectively Jupyter Notebooks has two methods of saving that work at the same time. Overall the Notebook will save to file (our "c2-notebooks.ipynb" file we created), and will do so automatically. This can be useful in case we lose power or connection but should not replace regularly saving to avoid loss of work. However, we also have a useful facility of "Checkpoints". This allows us to create particular versions of our work that we can revert back to in case things go wrong.

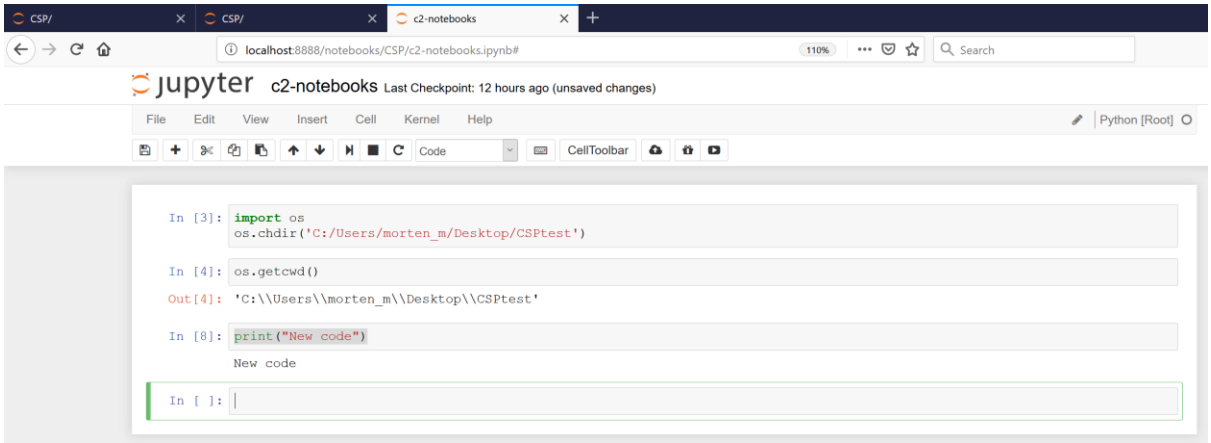
Let's add something new to our Notebook, verify it works, and then save as a checkpoint. In the new code line at the bottom of the Notebook add the following line of code:

```
print("New code")
```

Without any real understanding of Python (yet!) it should be pretty clear what this should do. Again we have a function name ("print") which is followed by parentheses containing a

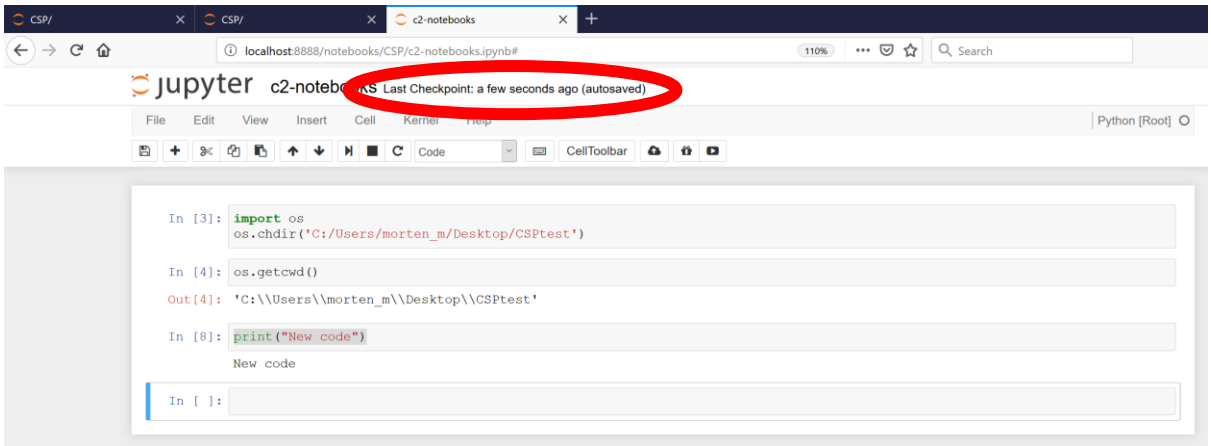
parameter ("New code"). Effectively we are asking Python to apply the function to the parameter … i.e. to print the phrase "New code". Run the code using the run button , highlighting just this new line ideally and you should get the following output:

Figure 21: Running the "print" Command



Below our input ("In [8]" in my example) we can see the phrase "New code" which is exactly how we would expect the function to work. Therefore our very simple program is working and is verified, so it is a good idea to save to "Checkpoint" (see below for more on this). Simply click on "File" in the menu and select "Save and Checkpoint". If all has worked then the text by our file name should say that our "Last Checkpoint" was a few seconds ago (shown in **Figure 22**).

Figure 22: Last Checkpoint

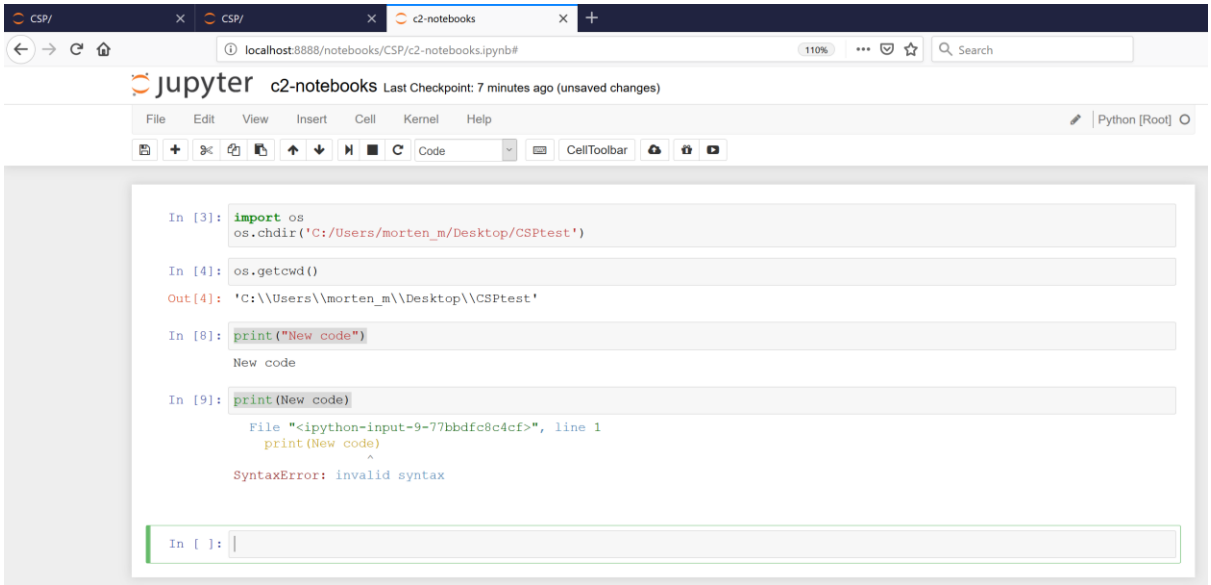


Now we are going to write some that doesn't work. We'll again call the print function but this time we'll introduce an error:

```
print(New code)
```

Try running this code and you should get the following error message (**Figure 23**):

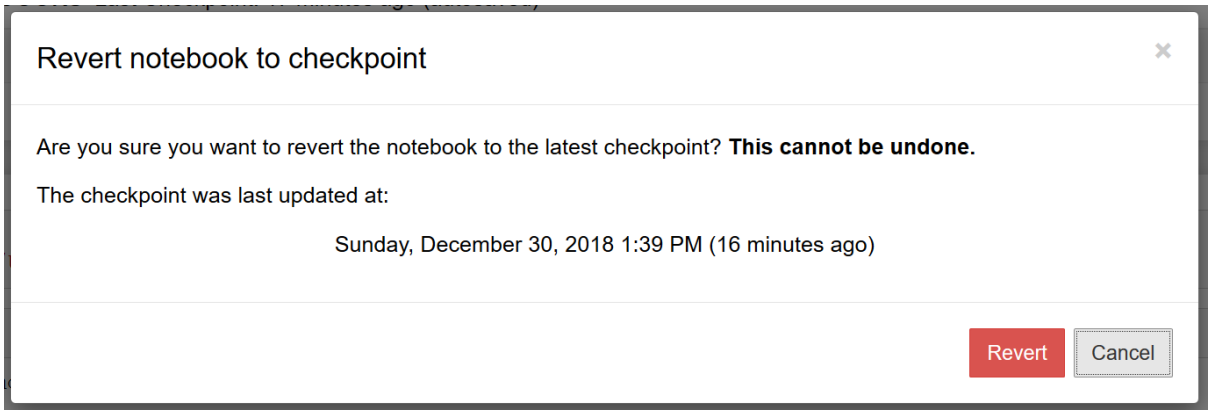
Figure 23: Error Messages



Without digging too much into the error message (this will be one of the subjects discussed in chapter three) we can see there is a syntax error. My online dictionary tells me that syntax is defined as "the arrangement of words and phrases to create well-formed sentences in a language" (via Google). In other words, our code is not correctly written for the Python language. In comparing the two pieces of code (the one that worked and the one that didn't) the difference is that in "In [8]" the phrase was wrapped in quotation marks and in "In [9]" there are none. This issue will be discussed in section 3.2, but basically the quotation marks tells Python that "New code" is a text phrase (known as a string). Without these quotation marks Python cannot interpret the parameter we are passing to the function (hence the error message).

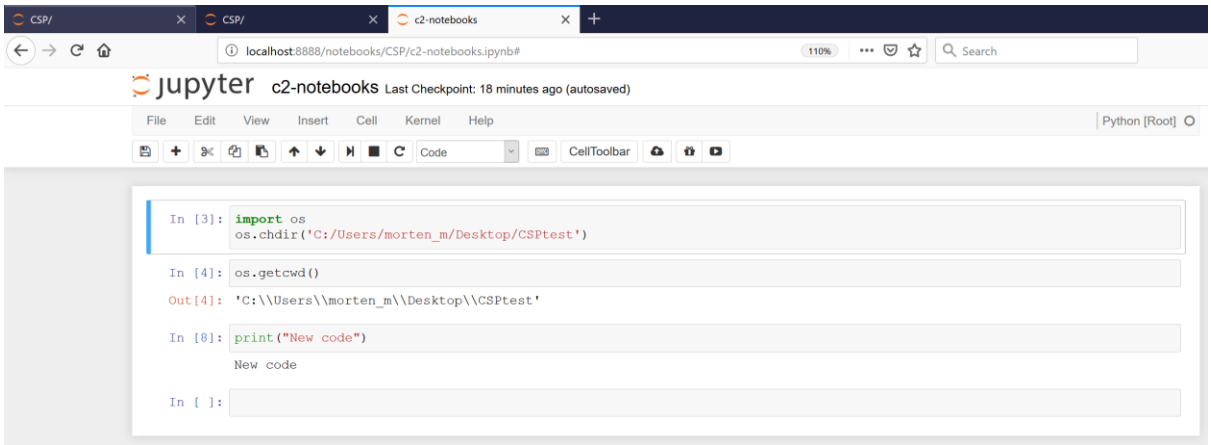
So we had working code at our last Checkpoint and non-working code now. It may be a good idea to revert back to our working code. Click on "File" and "Revert to Checkpoint". You may have more than one timestamp shown here (but probably it is just the one). Click on the most recent time (the one we created a few minutes ago). You should then see another modal which asks you to confirm and warns you that changes are not reversible (**Figure 24**):

Figure 24: Reverting Notebook to the Last Checkpoint



Click on the red "Revert" button and we should return to the previous version of our code:

Figure 25: Verifying Checkpoint



Application development typically involves several separate blocks of code (known as units) that are developed independently but need to work together. Rather than build everything in one go and hope it all works at the end, we want to test as we go along (this will be discussed further in section 3.7). This way we can isolate individual units or functionalities that we have changed/created and make sure these have worked. The ability to save to checkpoints is invaluable in this process. Take a hypothetical app we want to develop which will simply look up a customer's postcode from a database. This will involve a whole range of functionalities including a user-interface (UI), database look-up, and so on. We may first develop the part that asks for the user's surname, to pass to the database. We can build this part, verify it works, and then save as a checkpoint. We then start building a database look-up which may require us to change the code we have already written, tested and verified. If this then causes errors, we want an easy way to go back to the code state we know worked. Reverting to our checkpoint makes this easy and fast. In other words, use checkpoints whenever you have code that you know fully works, so that we can revert back to this point if we reach a time when it doesn't.

The last part of the "File" menu we'll discuss is the bottom option, "Close and Halt". Although it should be fairly obvious what this does, there is an important note. If we recall back to section 2.1, when we launched the Jupyter Notebook application we in fact launched both the browser-based Notebook, but also another window we called the kernel. While it is possible to close the Notebook by closing down the browser window(s), doing this does not shut down the application (because the kernel will still be running). If we want to shutdown both, i.e. to come back to our work later, we need to ensure we have saved(!) and then click on "Close and Halt" from the "File" menu. Both the Notebook and the kernel will now close, and we are free to click-to-close each window.

A brief note on kernels. A simple way to understand kernels are an interface between our computer (and its various resources such as processing (CPU) and file storage) and an application. When we are interacting with a Jupyter Notebook we are in fact accessing a browser-based app that will pass instructions to a Python implementation (IPython for Jupyter). When we launch the Notebook interface in our browser, a kernel is started in the background where Python can run (and access the computer's resources). In other words, much with many applications that are browser-based, we have a front-end (the Notebook browser interface) which speaks to a back-end (the IPython kernel). Both parts are needed to make the application work.

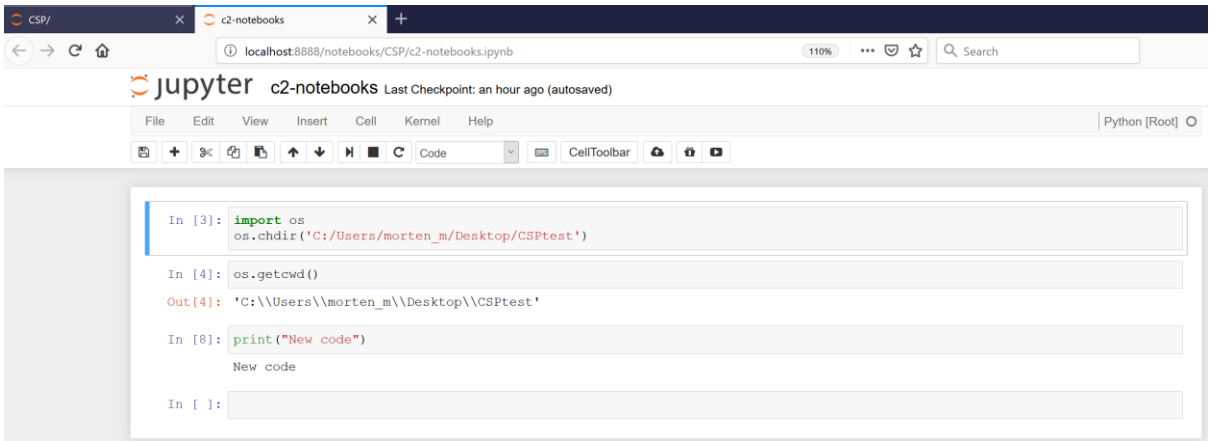
Having shut down Jupyter Notebooks we can now test that our earlier steps have actually worked – that we can open our earlier Notebook and resume our work. As we did earlier, search your system for the Jupyter Notebook and click to open (page 7). Again, this will launch both the kernel window, and, once the setup is complete, web interface (in your default browser).

Now we want to navigate to our folder, find the file we were working on, and clicking to open. For the example, on my machine, used in this chapter this means clicking on the new folder "CSP" and then clicking on the file we created "c2-notebooks.ipynb" (**Figure 26**):

Figure 26: Opening a Saved Notebook

This should open us back up at our last Checkpoint (**Figure 27**):

Figure 27: Accessing a Saved Notebook



To open up from a directory that is not set as default, probably the easiest way is to navigate to the folder, right-click on the file and select "open with". You'll then need to navigate to the Anaconda folder on your machine (in my case this is "C:/Anaconda3"), select the "Scripts" folder, and select "jupyter-notebook" – so the full path on my machine is "C:/Anaconda3/Scripts/jupyter-notebook". Ensure that you tick the option to always open file extensions of this type with this application so that you can avoid these steps next time (you can just navigate to the folder and click to open).

Edit Menu

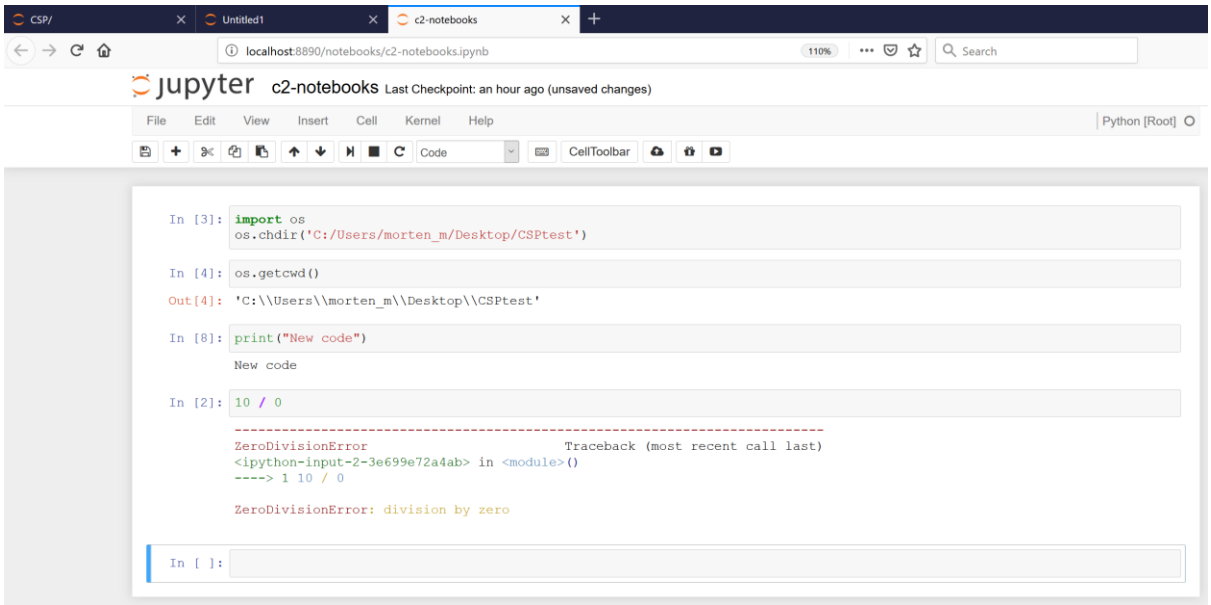
The "Edit" menu items mostly is useful for tidying up or "prettifying" your code. Typically there are only two functions I use here with any regularity. Firstly is the "Delete Cells" option. In the earlier discussion we talked about using Checkpoints to revert back and remove code with errors. In many cases, including the example we presented, this is overkill. "Delete Cells" provides a faster alternative.

To demo the functionality lets first create some code with an error. Type the following into your Notebook:

```
10 / 0
```

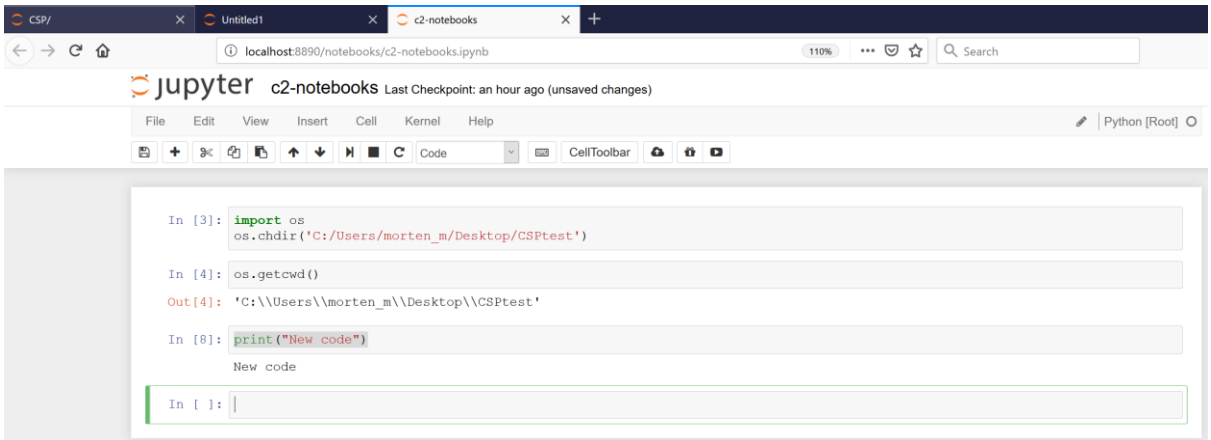
We are asking Python here to do some arithmetic – dividing ten by zero. Obviously, this is not possible and will return an error (**Figure 28**):

Figure 28: ZeroDivisonError



As you can see we have a "ZeroDivisionError", which is what we would want to see. Rather than reverting to the previous Checkpoint we will remove using the "Edit" menu. Click on the cell where we have written "10 / 0" (in the above, "In [2]"). Now click on "Delete Cells" from the "Edit" menu and our buggy code will be deleted (**Figure 29**):

Figure 29: Deleting Cells



The other function I regularly use is "Find and Replace". If you have a long piece of code, say >= 200 lines, often it can be time consuming to find the line you want to change. "Find and Replace" can be a quick way to get to the part you need. The "Replace" part can be useful, basically allowing you to change multiple entries at the same time. For instance, if we are printing "New code" lots of times in our script, and want to change this to "Newer code", this function

can help. However, you do need to be very careful here to ensure you don't replace something you didn't intend to. At a minimum, be sure to save the Checkpoint before doing this!

Kernel Menu

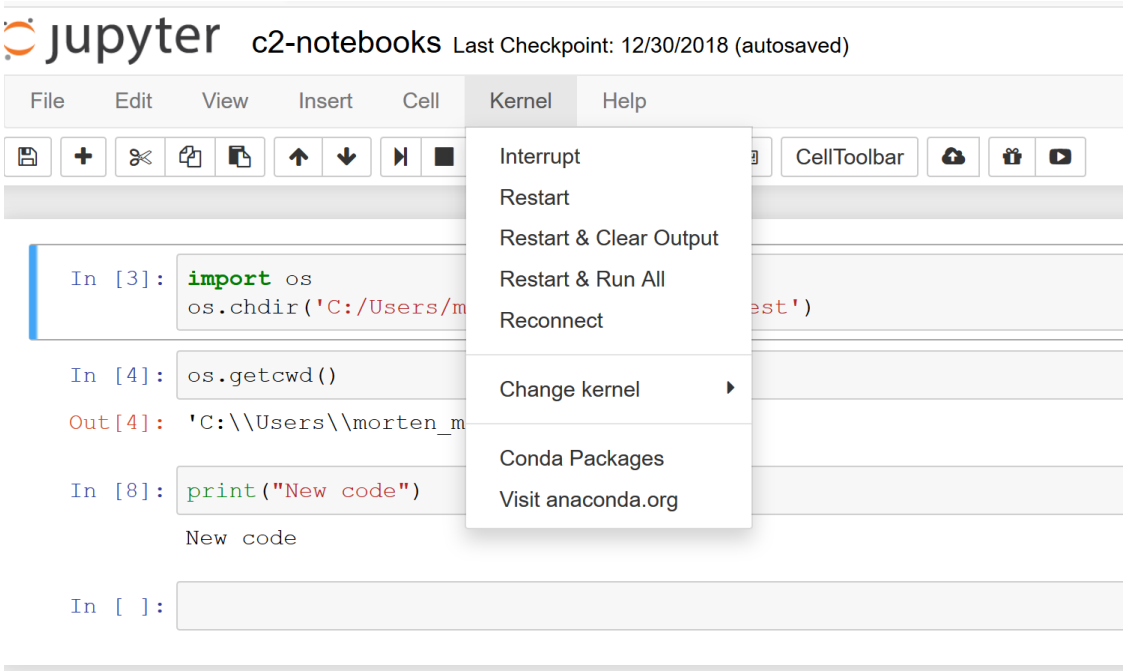
The kernel of the Notebook session was created during the application start up. We saw the initial steps in the black terminal window that opened during start up (**Figure 3**). The simplest explanation of a kernel is that this is the "computational engine" which operates the python code we type into the web browser. In other words, when we type some code, such as:

```
print("New code")
```

into the Jupyter Notebook browser, this is passed to our kernel and run from there. The result of this operation is then passed back to our browser and displayed on screen.

In many ways, we hope not to have to use the kernel menu, as typically we will only use this to interrupt, restart or reconnect to the kernel … i.e. we mostly will use it when things go wrong. This menu is shown in **Figure 30**.

Figure 30: The "Kernel" Menu



The first option, "Interrupt", is used to instruct the kernel to stop its current operation. This can be necessary if our code is taken an unacceptable amount of time to complete. The code will terminate as soon as is possible and will not return any incomplete result to the Notebook browser. Instead you will get some form of error message such as the example in **Figure 31**.

Figure 31: Error Message from Interrupting a Kernel

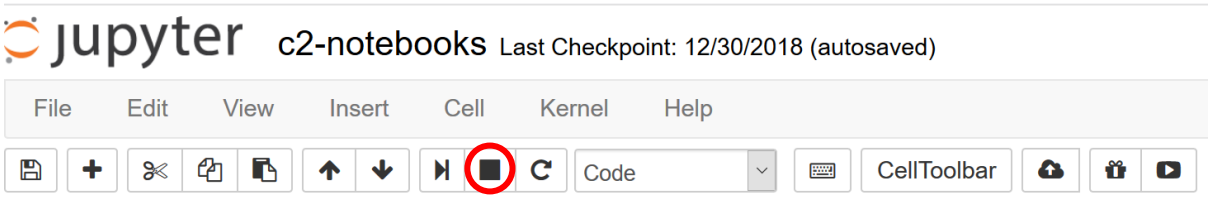
```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-3-857976f478cf> in <module>()
      1 for i in range(0,100000000):
----> 2     print("x")

C:\Anaconda3\lib\site-packages\ipykernel\iostream.py in write(self, string)
    315
    316             is_child = (not self._is_master_process())
--> 317             self._buffer.write(string)
    318             if is_child:
    319                 # newlines imply flush in subprocesses

ValueError: I/O operation on closed file
```

You can also interrupt the kernel using the stop butting in the top menu, as shown in **Figure 32**.

Figure 32: Interrupt Button in the GUI



In some cases interrupting the kernel may not be enough. In some instances the "interrupt" message will be queued behind another operation and will not force the kernel to stop in an acceptable timeframe. In this case it may be necessary to restart the kernel. Restarting the kernel is a more 'nuclear' approach to stopping slow running code, but this comes at a cost – namely losing not only the current operation but any variables stored in the session. Variables are explained in section 3.1 and 3.2, but, as a metaphor, this would be like exiting an e-commerce website by completely closing the browser rather than closing the current tab. In the case of the former, the whole of the shopping basket would be lost, in the case of the latter most sites will retain the items in the basket. There are three options to restarting the kernel given in the menu:

1. "Restart" – the kernel restarts (as above), but all of the outputs from the last operation that are shown in the Notebook browser are retained;
2. "Restart & Clear Output" – like (1) but the browser is cleared and returned to either the start (a new workbook) or the last checkpoint;
3. "Restart & Run All" – like (1) but after the kernel has restarted all the input code will be run from the top (usually from the input "In [1]" through to the last input which caused the problem to occur).

The final option worthy of mention is "Reconnect". Again, hopefully you will not need to use this, but sometimes the browser element of the Notebook and the kernel can lose their connection. Should this happen you would see an error message at the top of the browser version (and in the kernel). The "Reconnect" option will (hopefully) fix this and re-join the two.

Help Menu

The help menu has three relevant aspects. Firstly, at the bottom of the menu are links to the documentation of the Jupyter toolkit (including the documentation for the Notebook, IPython and others), as well as to other tools created by the same company (including libraries we will use in this book such as "Pandas"). Secondly, there is a list of useful keyboard shortcuts to speed up your work under "Keyboard Shortcuts". Finally, and the first item of the menu, is "User Interface Tour". This gives an interactive walkthrough of the interface using pop-ups/modals to show the different functionality. To finish this chapter, you can take this "tour" which will refresh on the items covered here, and introduce some additional items and functionality that are less commonly used (in my experience).

2.4. Additional Resources and Documentation

The following are a list of resources and documents on Python, Jupyter and related aspects:

- **The Python Website**: https://www.python.org/
- **Github**: https://github.com/
- **Sublime Text** (a code editor for all languages): https://www.sublimetext.com/
- **Project Jupyter**: https://jupyter.org/
- **Jupyter on Github**: https://github.com/jupyter
- **PyPi** (the Python Package Index): https://pypi.org/
- **Automate the Boring Stuff** (Python tutorial): https://automatetheboringstuff.com/
- **A Byte of Python** (Python tutorial): https://python.swaroopch.com/
- **Notes by a Nerd** (further, more advanced, Python articles): http://notesbyanerd.com/2017/12/29/essential-reads-for-any-python-programmer/
- **Stack Overflow** (a Q&A site for programming and development questions): https://stackoverflow.com/

3. Creating a Python Program

Now we have an understanding of the interface it's time to do the fun stuff – building Python programs! This chapter will walk through the basics of a Python program, different data types and basic algorithms/functions, and finally some best practice concepts for programming in Python and in other languages. As discussed in section 2.3, you are encouraged to keep things organised by creating different notebooks per chapter. Following the convention we used earlier, save the notebook from chapter two, and start a new notebook called "c3-notebooks".

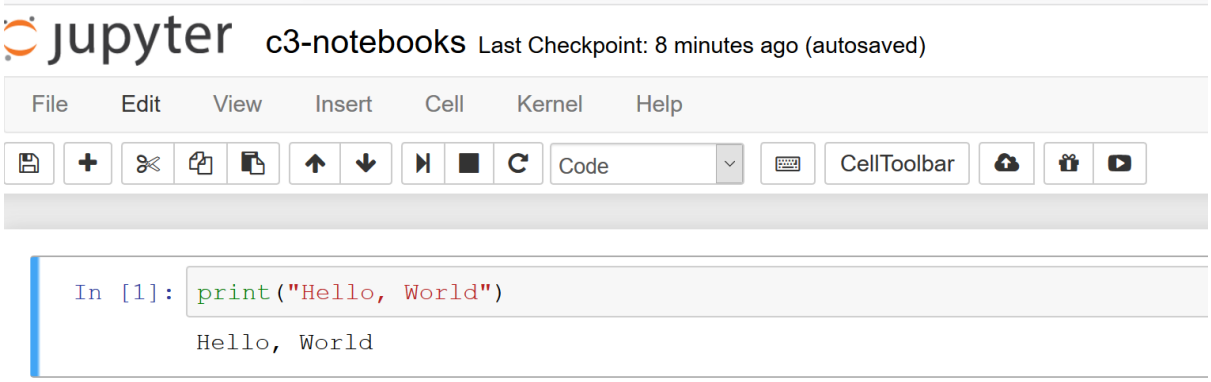
3.1. A First (and Second) Program

Convention has it that the first program we should produce is a "Hello, World" program – a simple program that can print this salutation. In fact we already have the necessary knowledge to do this, using the print function we introduced in section 2.3. In short, our first program, to keep up with convention, can be as simple as:

```
print("Hello, World")
```

If we run this code in our browser window, we should indeed see the salutation printed to screen, as in **Figure 33**.

Figure 33: The "Hello, World" Program

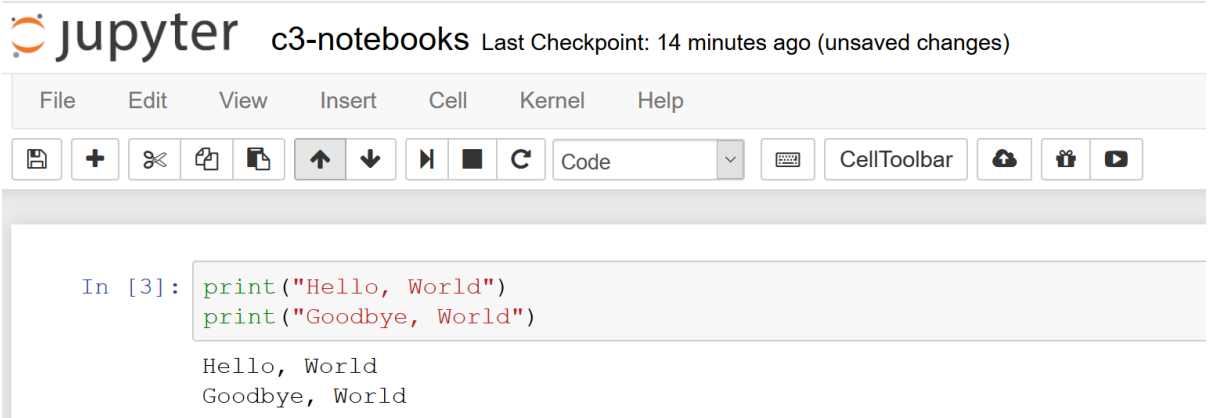


While this is a good start to the world of programming, we can look to do something a little more sophisticated. After we have greeted the World, we may also want to say "goodbye". A simple way to do this is:

```
print("Hello, World")
print("Goodbye, World")
```

Because the two lines are written in sequence, they are executed in sequence, meaning we get an output such as **Figure 34**:

Figure 34: The "Hello & Goodbye" Program



Because we are polite people, however, we may want to greet and say goodbye to others, not just the World as a whole. In other words, rather than treating these as a single block of text ("Hello, World") we may want to separate the parts of this sentence. To do so we will create a *variable*. A variable is a value which can change either based on our input or based on the running of the program. It is effectively a temporary store of some value that we intend to use.

A variable is generally a simple pair of variable name and the value being stored, assigned by the equals ("=") symbol. We can assign some variables as follows:

```
message_one = "Hello"
message_two = "Goodbye"
target_one = "World"
target_two = "James Pennington"
```

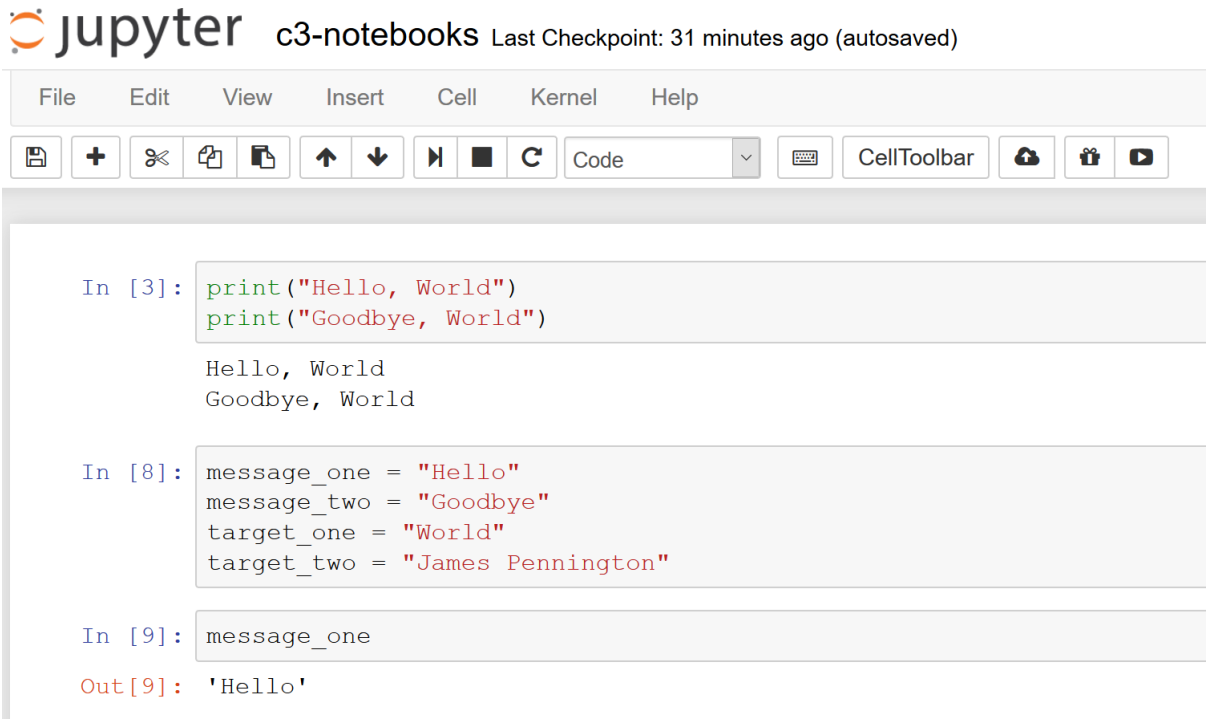
In this example we have created a variable we have named *message_one* and assigned a value of *"Hello"*. Note that our values, in this case, are all wrapped in quotation marks. As we discussed in section 2.3, the quotation marks tell Python that this is a text value (a string – discussed in section 3.2). However, although our variable names are also text, there are no quotation marks. This is because Python does not regard these as stings but as names.

Variable names can include both numbers and letters, but the first character must be a letter. For instance, "1_variable" will cause an error but "variable_1" will not. Note also, as with all programming, variable names are case sensitive. "Variable_1" is not the same thing as "variable_1". By convention, the first letter used would be lower case, but this is just a convention. It is important to use meaningful names that you can remember the

meaning of at a later time. "variable_1" will be meaningless to you if you return to the code in 2 months time; whereas "message_one" should be more memorable. We often have variable names that are more than one word, to make them more memorable. However, Python will expect the variable name to be a single phrase not a multiword phrase. It is best to employ a consistent system for managing this such as underscores ("message_one") or capital letters ("messageOne"). You cannot use a hyphen ("message-one") as Python interprets the hyphen as the minus symbol (i.e. it will try to subtract "one" from "message").

If we run this code in our Jupyter Notebook we will get no specific output (unless there is an error). However, if we type after this a variable name, as shown in **Figure 35**, the console will display the stored value:

Figure 35: Retrieving a Value Stored as a Variable

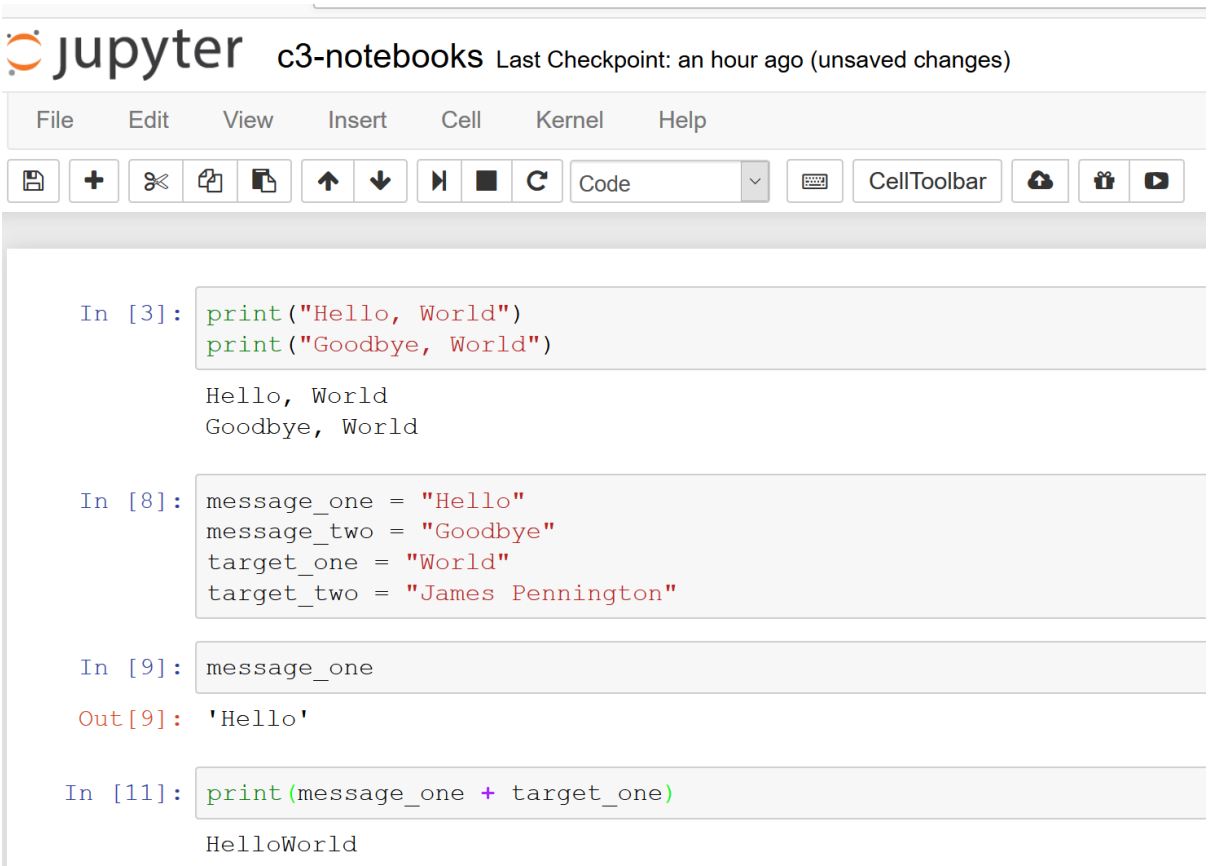


We can now use these variables elsewhere in our code. For instance, we can greet both the "World" and "James Pennington" with message_one ("Hello"). To do so we need to combine the two variables together in our print statement. Try the following code in the Jupyter Notepad:

```
print(message_one + target_one)
```

This should produce the following output:

Figure 36: Combining Variables in a Print Function



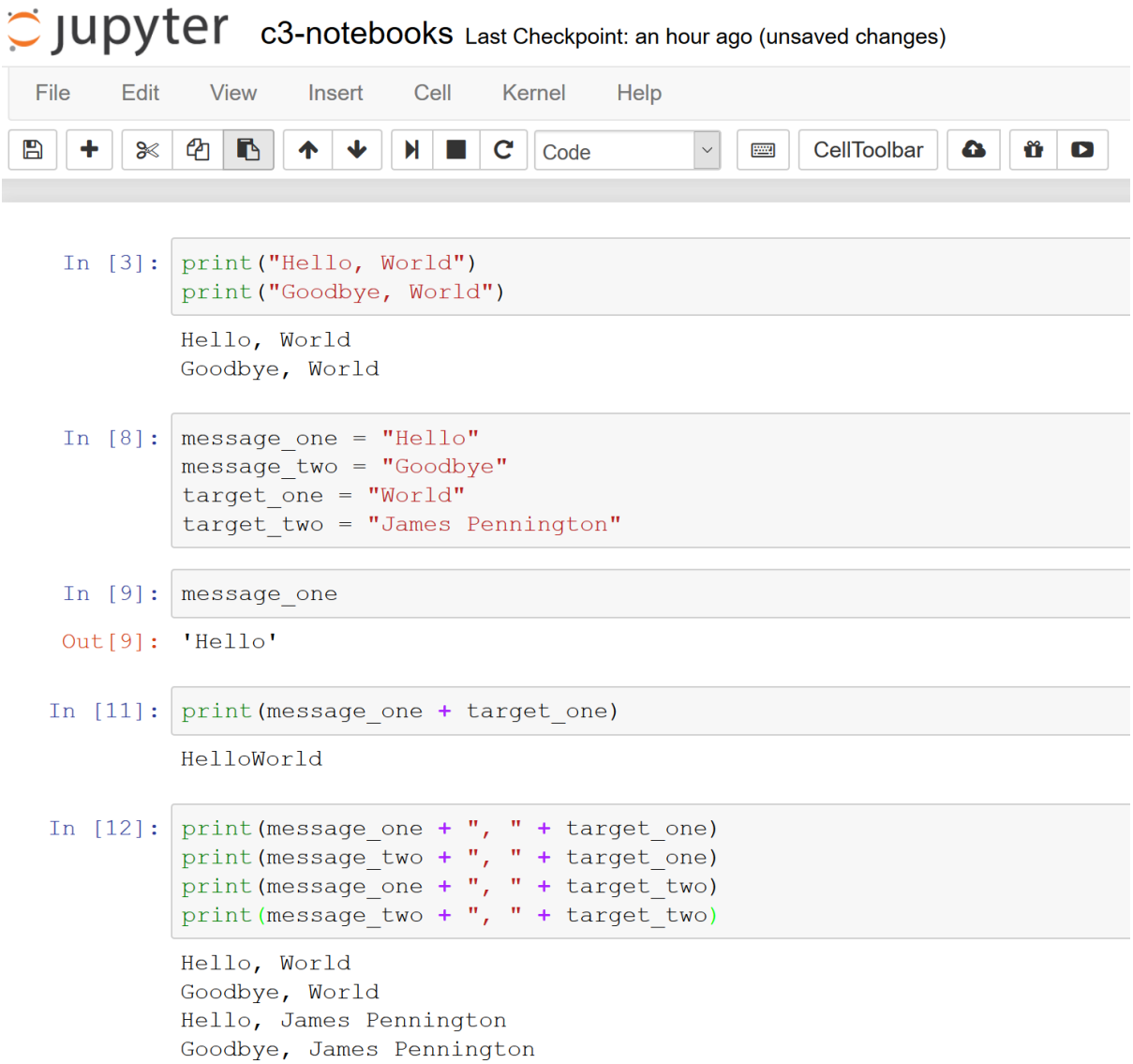
The print function has taken two inputs (our message_one and target_one variables) and combined them into the output because of the plus symbol ("+"). However, we have lost a little bit of presentation because of the lack of any spacing or punctuation. We can fix this by simply expanding the code. Try the following:

```
print(message_one + ", " + target_one)
print(message_two + ", " + target_one)
print(message_one + ", " + target_two)
print(message_two + ", " + target_two)
```

Here we have added an additional value in between each of our variables, again joined by using a plus symbol. This is the text value ", ". Note this includes a comma, but also a space after it. As we saw in Figure 36, Python's default behaviour is to join text values without any spacing. Therefore, we want to include the space in the value itself, to make our results present in a neater fashion

When this code is run, this should give the correct output, as shown in **Figure 37**.

Figure 37: Our Second Program



As noted above, we can easily change our variables. Go back to the line in the notebook where we assigned the value of "James Pennington" to target_two ("In [8]" in my case). Let's change this line of code to say "Gareth Edwards" and then run the code again in full. Note we need to run all the code again, as we want to run both the print statements ("In [12]" in my case) as well as running the variable assignments again ("In [8]"). If we just run the last set of commands ("In [12]") we will get the same output as the variable has not been changed. You can run everything by either highlighting all the code, or by navigating to "Cell > Run All" (see section 2.3). This should produce the output shown in **Figure 38**.

Figure 38: Our Second Program (changed value)

jupyter c3-notebooks Last Checkpoint: an hour ago (unsaved changes)

File Edit View Insert Cell Kernel Help

Code CellToolbar

```
In [16]: print("Hello, World")
         print("Goodbye, World")

         Hello, World
         Goodbye, World

In [17]: message_one = "Hello"
         message_two = "Goodbye"
         target_one = "World"
         target_two = "Gareth Edwards"

In [18]: message_one

Out[18]: 'Hello'

In [19]: print(message_one + target_one)

         HelloWorld

In [20]: print(message_one + ", " + target_one)
         print(message_two + ", " + target_one)
         print(message_one + ", " + target_two)
         print(message_two + ", " + target_two)

         Hello, World
         Goodbye, World
         Hello, Gareth Edwards
         Goodbye, Gareth Edwards
```

Congratulations! We have now created two successful programs, and also explored variables and variable assignment! If you have any issues with this section, you can pull (download) a Jupyter Notebook with a working version of this code from the Github repository (https://github.com/MJMortensonWarwick/msc_programming_python). The file for this exercise is titled "c3-notebooks-HelloWorld.ipynb".

3.2. Data Types

Essentially a Python program, like the two we created in the last section, are some function/operation, applied to some data. In the last example, by the end we were applying two operations (a join function marked by "+", sometimes known as *concatenation*, and a print

function). The data we used in all of these cases, and the first type of data we will discuss in this section, is text data, typically called 'strings' in computer programming.

String Data

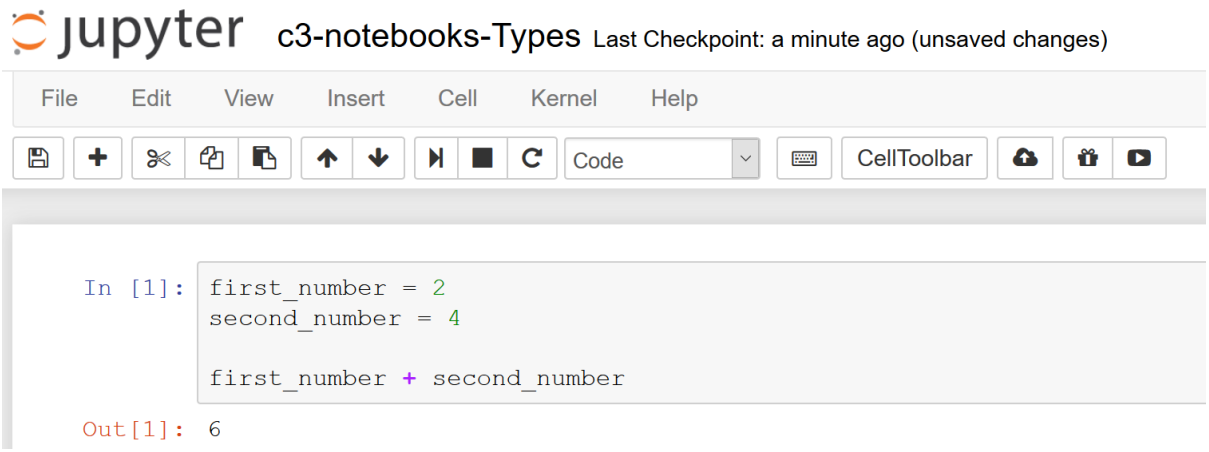
String data is any data we want to consider as a set of characters with no explicit numerical value. Often this includes letter characters (e.g. "World"), however it is not limited to this. We can include a mixture of text and numbers, or indeed our string could just be numbers. If we store numbers as strings in Python, then they can't be used in numerical calculations. Let's run the following operation:

```
first_number = 2
second_number = 4

first_number + second_number
```

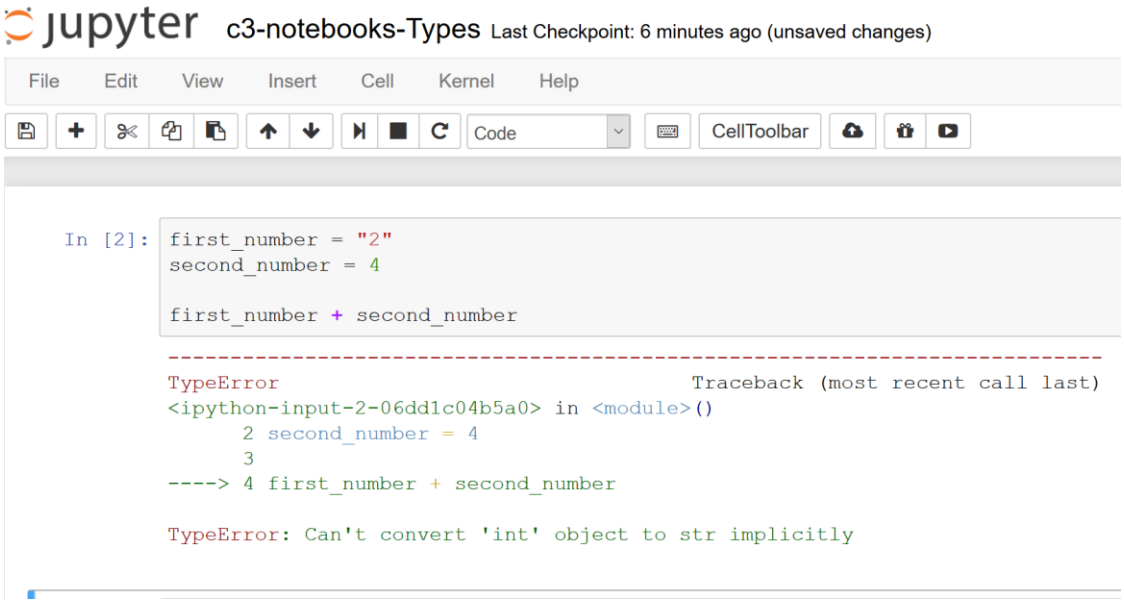
Running this code will produce the following output (**Figure 39**).

Figure 39: Addition in Python



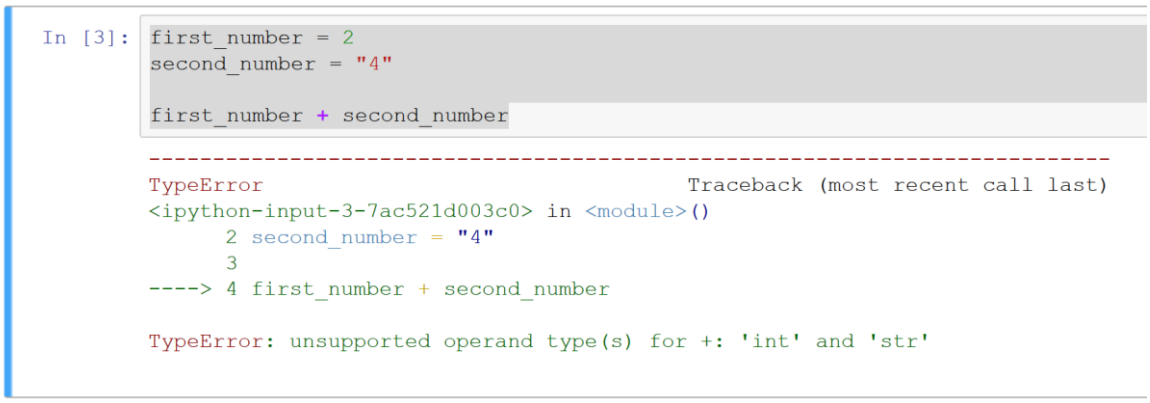
Again we have used variables, this time however the value we have assigned is both a number, and also is not wrapped in quotation marks (" "), which would indicate to Python our data is a string. Therefore, the operation asks Python to add *first_number* to *second_number* – i.e. 2 + 4 – which, of course, equals six. Let's what would happen if we were to change *first_number* from 2 to "2", shown in Figure 40 (remember to use "Cell > Run All").

Figure 40: Addition in Python (string plus integer)



Now we have a "TypeError" listed as "Can't convert 'int' object to str implicitly". To understand this error we first need to understand Python's short-hands. The reference to "int" is a short-hand for "integer" (i.e. a whole number) and "str" is the short-hand for "string". When we ask Python to add the two variables, the software considers the type of the first when working out what you want it to do. If the second variable does not match its expectation, it will throw up a type error. In this case, Python reads that *first_number* as a string, because of the quotation marks, so assumes that the plus symbol is being used for joining/concatenating the two variables as we did in the last section. I.e. it wants to create a string that is the join of *first_number* and *second_number*. However, when it then considers *second_number* it reads it as an integer value not as a string (and therefore throws the error). We can see that this throws a different error if we change the code again such that "2" becomes 2 again; and 4 becomes "4" (**Figure 41**).

Figure 41: Addition in Python (integer plus string)



Again we have a "TypeError" but this time the wording is slightly different, to indicate that the issue is slightly different. This time Python has read the first variable as an integer, and therefore has assumed that the plus symbol this time means mathematical addition. When presented with a string to add to an integer, there is no obvious method to follow, so the error is thrown (it is the equivalent of saying "2 + CSP").

In other words a string is considered by Python as essentially just a block of characters. This block of characters has an overall length, and can be indexed or sliced (see section 3.4 for more discussion of this), but the characters themselves have no specific value in and of themselves.

Additionally, a string need not only be letters and numbers, any character can be used from the machine's default character set (usually UTF-8 if using a UK machine) so long as they are wrapped in quotation marks. For instance, "%&x39!" would be a perfectly acceptable string. However, sometimes there can be problems where characters are also used to 'escape' the string (or the line). In our above examples, we have been indicating the start and the end of a string by using quotation marks (" "). What would happen though if we wanted to process a string which included such a quotation mark? Let's explore this in **Figure 42**.

Figure 42: Process Strings with Quotation Marks

```
In [6]: lecturer = "James "the hat" Pennington"

          File "<ipython-input-6-d5a53382a7b4>", line 1
            lecturer = "James "the hat" Pennington"
                               ^
        SyntaxError: invalid syntax
```

Python has not interpreted the whole input (*James "the hat" Pennington*) as a single phrase, it has broken it based on where the quotation marks are. In other words, it has read a string of *James_* and then read a bunch of other characters after it (hence the error). There are a couple ways of dealing with this. Firstly, we change the type of quotation mark from double quotes (" ") to single quotes (' '):

```
lecturer = 'James "the hat" Pennington'
```

Running this code will not produce any errors because our string is wrapped with single quotes and everything inside these quotes will be treated as a bunch of characters. However, this will not help us if our string has, or potentially could have, both single and double quotations (for

instance, the string *James "the 'top' hat" Pennington*). In Python the back-slash symbol can be used to instruct the language to treat the character after the symbol purely as a character and not as an instruction. The following code, **Figure 43**, will produce no errors.

```
lecturer = "James \"the 'top' hat\" Pennington"
```

Figure 43: Using Backslash to Ignore Escape Characters

```
In [17]: lecturer = "James \"the 'top' hat\" Pennington"
         lecturer

Out[17]: 'James "the \'top\' hat" Pennington'
```

This can be relevant in respect to other characters, discussed in more detail here (https://docs.python.org/3.7/tutorial/introduction.html?highlight=escape%20characters%20backslash#strings), including, of course, any text that includes back slashes! For instance a path directory in Windows of "C:\Users\morten_m\Documents\CSP" will throw an error, while "C:\\Users\\morten_m\\Documents\\CSP" will work.

The approach Python takes to the type problem, as discussed in this section, is typical of an interpreted language as opposed to a compiled language (such as Java or C). Although these are ambiguous terms, one of the key points is that Python executes each command in sequence (one at a time), and interprets what is required of it. When Python takes an input of a variable it interprets as a string, this then has bearing on how it interprets the plus symbol (it interprets it as a join/concatenation command). This typically differs in a compiled language, where the chosen type of every variable needs to be explicitly stated. For instance, in VBA code we may use the command "dim first_number as string" – meaning we are instructing the program to treat the first_number as a string rather than have the program interpret this assignment. In compiled languages, because we do not have this interpretation step, it is uncommon for a symbol to have multiple uses, in the fashion that Python uses "+" to both mean a mathematical addition and also a concatenation. There are implications on run time speed which will be discussed in class (if you are doing CSP), but as a general rule programs in interpreted languages take longer to run (because the machine needs to interpret every command), but less time to write (as we do not need to declare the type of every variable).

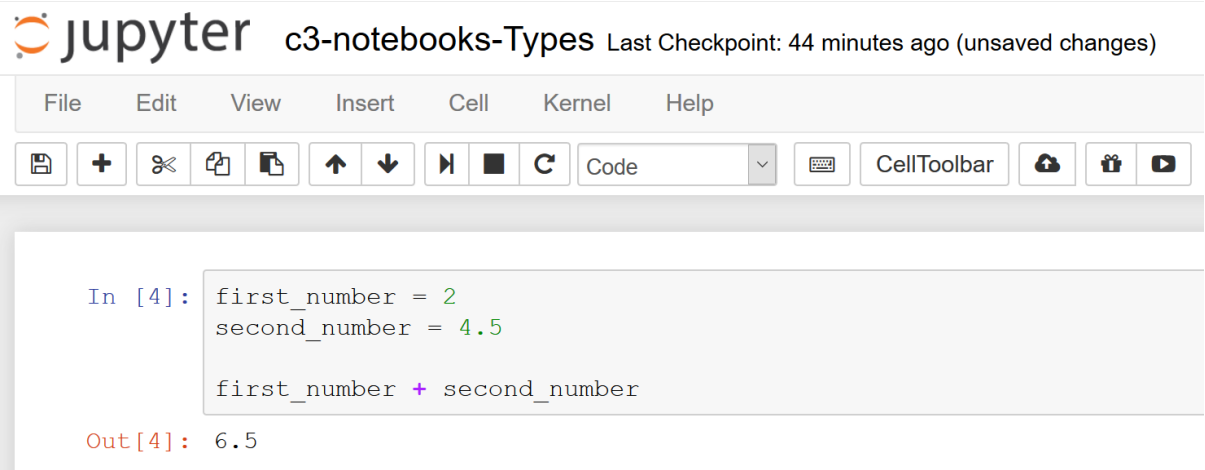
Integer and Float Data

The second type of data is one we have already seen, an integer (or whole number). Any number that is input to Python that has no quotation marks will be interpreted as a number. If that

number has no decimal place it will be interpreted as an integer. If the number has a decimal place, e.g. 3.45, it will be interpreted as a float (a number with a decimal).

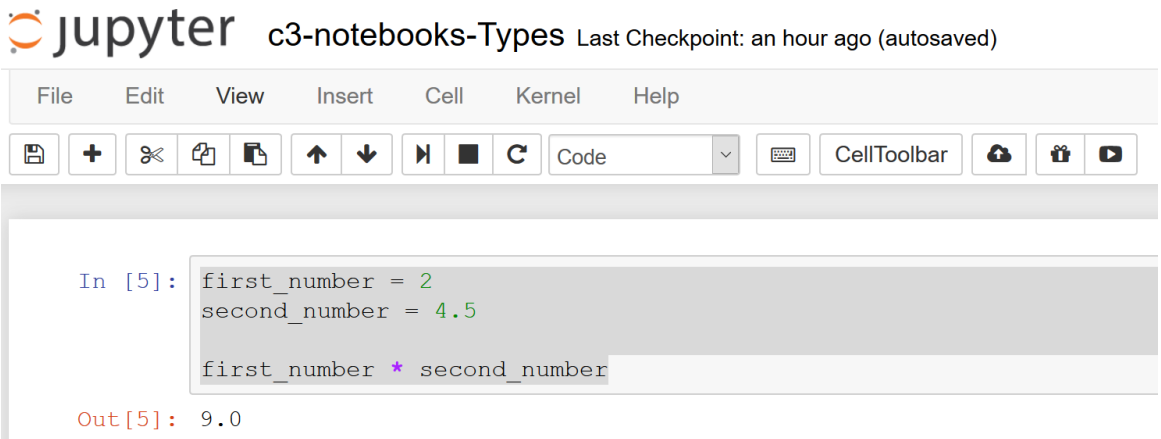
As before, let's do a bit of experimentation (**Figure 44**):

Figure 44: Integer and Float Addition



In this case, Python has interpreted *first_number* as an integer, and *second_number* as a float. However, perhaps surprisingly considering the previous example, the addition operation has ignored the first data type assignment (integer) when creating the output, instead creating a float. Despite this seeming inconsistency, however, it should be fairly clear that this is an appropriate output. Note that this will also be the case if we were to produce an output that is a whole number (has nothing after the decimal point), which we can verify by changing our equation from an addition to a multiplication, by using the asterisk symbol "*" (**Figure 45**).

Figure 45: Integer and Float Multiplication



Here we have asked Python to multiply 2 by 4.5, which obviously equals 9. However, Python has stored this as a float rather than an integer (9.0). In other words, Python's default behaviour is to treat integers and floats both as numbers, and when using both in the same operation to produce results at the lowest level of granularity. If both numbers are integers, it will return an integer. If either number is a float, because a float is more granular it will return as a float.

It is worth noting that the way that programming languages (and computers in general) deal with floating point numbers (floats) can quickly become a relatively complicated business! Although not necessary for these modules, the interested reader may want to refer to this blog post (http://www.lahey.com/float.htm) for a fuller discussion of this weird and wonderful world, and/or to the Python 3 documentation of how Python deals with these issues (https://docs.python.org/3/tutorial/floatingpoint.html).

Booleans

Another commonly used data type is Booleans, a measure of whether something is true or if it is false. A Boolean can only ever be one of these states. A Boolean value is assigned by typing either *True* or *False*, with the first letter capitalised and no quotation marks. For instance, consider the following two variables:

```
bored_boolean = True
bored_string = "True"
```

The variable *bored_string* will be treated by Python as a four-character string, in the fashion described above. However, *bored_boolean* is not a string but a state – a state of being true in this case. For a more illustrative example of where this would be used, consider the case of creating an account on a website such as Github. When a user creates an account we want them to verify their email, and until this is done we will not give them access to the account. When initialising the account it may be set up with a variable associated with it that looks something like:

```
account_is_verified = False
```

This variable can then be changed to *True* once the email has been verified (and therefore allowing the user to access their account).

Changing Types Dynamically

In some cases we may want to change the type of a variable within an operation. Fortunately, Python includes several inbuilt functions to do this such as "int()" to convert to integer; "float()"

to convert to float; and "str()" to convert to a string. Note, however, that this needs to be a logical conversion – we could not convert the word "James" to an integer.

We can use these functions in our code to change the type of a variable 'on the fly'. For instance, building on our earlier example, we can convert a value stored as an integer into a string, as shown in the below code and in **Figure 46**.

```
message_one = "Hello"
message_two = "Goodbye"
target_one = 2

print(message_one + ", " + str(target_one))
print(message_two + ", " + str(target_one))
```

Figure 46: Type Conversion using str( )

```
In [19]: message_one = "Hello"
         message_two = "Goodbye"
         target_one = 2

         print(message_one + ", " + str(target_one))
         print(message_two + ", " + str(target_one))

         Hello, 2
         Goodbye, 2
```

In this code, *target_one* which is the variable name for the value 2, stored as an integer, is converted in the print statement to a string. This means it can be printed without error. A couple of other notes regarding this. Firstly, this change (from integer to string) is only for the purposes of the print operation. *target_one* will still be storing its value (2) as an integer, and so this value could be used in subsequent mathematical calculations. If we wanted to permanently change this value from an integer to a string we would need to run something like:

```
target_one = 2
target_one = str(target_one)
```

Secondly, it is worth noting the double brackets that end the print commands in the code. We effectively have a function within a function – specifically a string function inside a print function. All functions, much like strings, need to be opened and closed using brackets. Of the two brackets that end each of these lines, the first closing bracket closes the "str( )" function, and the second closes the "print( )" function. Without a pair of brackets for each function the

code will not execute. Making sure that you open and close all brackets correctly is an important point to check when writing programs.

We can also use similar methods to convert numbers stored as strings into integers or floats, again, 'on the fly'. The following code and output (**Figure 47**) gives an example of this.

```
first_number = 2
second_number = "4"

first_number + int(second_number)
```

Figure 47: Type Conversion using int( )

```
In [20]: first_number = 2
         second_number = "4"

         first_number + int(second_number)

Out[20]: 6
```

Again, this does not permanently change the type of the variable *second_number*, it remains a string. However, it does convert it to an integer during our calculation to make the equation 2 + 4 = 6.

Another conversion we may want to use would be making transforming a number stored as a float into an integer. For instance, consider the following code and output (**Figure 48**).

```
third_number = 3.89
third_number = int(third_number)
third_number
```

Figure 48: Float to Integer Conversion

```
In [21]: third_number = 3.89
         third_number = int(third_number)
         third_number

Out[21]: 3
```

There are a couple of points to make about these operations. Firstly, because of the command:

```
third_number = int(third_number)
```

we have now permanently changed the value of *third_number* to an integer. Short of typing it in again, we have now lost the decimal values (.89) entirely. Secondly, those from a mathematics background may have been surprised by the output. Obviously conventional mathematics holds that 3.89 would round to 4, but the output given here is 3. This is, in fact, expected behaviour. Python, and indeed most languages, will make this conversion based on the sequence of digits not on the values *per se*. It will read the first part of the value as "3." and cut-off there – it makes no effort to evaluate whether the decimal part is >= 0.5 or otherwise. If you want to make a conversion that would round up or down, then the "round( )" function should be used. The round function takes two parameters within its brackets, the input value/variable, and the number of digits to round to, separated by a comma. For instance, the following code will produce the input shown in **Figure 49**.

```
third_number = 3.89
third_number_integer = int(round(third_number, 0))
third_number_smaller_float = round(third_number, 1)

print("third_number_integer = " + str(third_number_integer))
print("third_number_smaller_float = " + str(third_number_smaller_float))
```

Figure 49: Converting Floats using round( )

```
In [23]: third_number = 3.89
         third_number_integer = int(round(third_number, 0))
         third_number_smaller_float = round(third_number, 1)

         print("third_number_integer = " + str(third_number_integer))
         print("third_number_smaller_float = " + str(third_number_smaller_float))

         third_number_integer = 4
         third_number_smaller_float = 3.9
```

Again, a few notes on what we have done here. Firstly, in creating the *third_number_integer* variable, we have again added a function inside a function. On the inner function we ran *round(third_number, 0)*, meaning we took the variable we wanted to convert (*third_number*) as the first input, and the number of decimal places required (0) as the second. These are separated by a comma to tell the formula which part is which. If we ran this part alone it would return 4.0 as the round function cannot change the variable type, it just rounds up or down. By wrapping this function in the outer function, *int(…)*, we take the output of our inner function (4.0) and convert it to an input in the manner discussed above – resulting in our desired value of 4. As our

second converted variable, *third_number_smaller_float* is to remain as a float (albeit a float with one fewer decimal places) we do not need this extra function and conversion.

Secondly, we have also used a function within a function in the second part of our program, using a str( ) function within our print( ) function. This is the same process explored earlier – temporarily converting a numerical value to a string so we can print to screen. Again, the code used in this section is available from Github ("c3-notebooks-DataTypes.ipynb").

3.3. Python Lists

So far, all of our examples have been using single value variables (i.e. each variable has been used to label just one value – either a single string, integer or float). In many cases however, particularly in computational statistics and data science workloads, we will want to work with a set of data rather than a single value. Probably the most common type of data structure, and the subject of this first section on data structures, is a list. A Python list is an unordered (unsorted) collection of data, where the values are changeable and can be repeated. In other words, this is a flexible store of data which contain a range of values, each of which can be removed or modified.

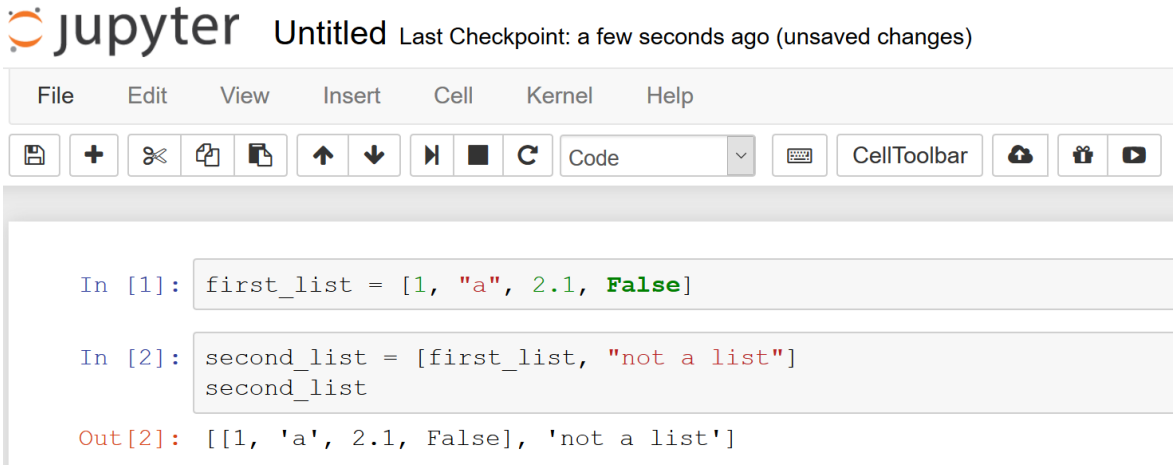
A list is created in Python by using square brackets, "[ ]", to wrap the values, each of which is separated by a comma. For example:

```
first_list = [1, "a", 2.1, False]
```

As can be seen, we can include any type of data inside a list (the above includes, in the order written, an integer, a string, a float and a Boolean). Indeed a list can also include a list. The following produces the output of **Figure 50**.

```
second_list = [first_list, "not a list"]
```

Figure 50: A Python List



Effectively our *second_list* is a list with two items. The second item is the string "not a list", and the first is the list we generated earlier (*first_list*), which in turn has the four items above.

Items in a list can be called by its index number – a process called indexing. It is worth noting that Python, like many programming languages counts from zero not from one. If we want to select the first item from *first_list* this will correspond to the value of zero; the second item would be a value of one; and so on. Indexing is done by listing the variable name of the list, and then the item required inside of square brackets. For instance, the code to index the third item of *first_list* is shown in **Figure 51**.

Figure 51: Indexing the Third Item in a List

```
In [3]: first_list[2]
Out[3]: 2.1
```

If we want more than one item, we can use a slice. Slicing uses the same notation as indexing, apart from we list a range between the square brackets. The range will start at the first number we want to include (remember, Python lists start at zero!) through to the item we want to stop at. This last number will not be included in the list. For example, the following will produce the output show in **Figure 52**.

```
first_list[1:3]
```

Figure 52: Slicing a Python List

```
In [4]: first_list[1:3]
Out[4]: ['a', 2.1]
```

These results can seem counter-intuitive so it is worth going over what we have asked Python to do. The slice "[1:3]" is asking Python to start with the second item (as our count starts at zero) and to include every item up until (but not including) the fourth item (the count starts at zero). In other words, we have asked for the second and third item, which is what was returned.

It is also possible to omit either a starting point of an ending point simply by leaving that value blank. For example, the following output shown in **Figure 53**.

Figure 53: Open Ended Slices

```
In [6]: first_list[1:]
Out[6]: ['a', 2.1, False]
```

Here we have asked Python to start at the second item, and include every item after it. Typing *first_list[:3]*, on the other hand, would select items zero, one and two from the list.

Finally, we can also ask Python to list all items except a fixed number from either end. For example, *first_list[:-1]* would be the equivalent of *first_list[0:3]*, displaying the following:

Figure 54: Slicing using Negative Indices

```
In [7]: first_list[:-1]
Out[7]: [1, 'a', 2.1]
```

Effectively we have asked Python for everything at the start (as there is nothing before the ":"), and all apart from the item minus one position from the end. The command *first_list[-1:]* would produce the item minus one position from the end, through the final item. In other words, it will return just the last item (False).

The final aspect of indexing and slicing we will cover here, concerns situations where we want to index an item inside a list. In this case we index the first item in the way above (using square brackets and the count of the item), and then include an item number or slice in a further pair of square brackets. For instance:

```
second_list[0][2]
```

will ask Python for the first item in *second_list* (which is the *first_list* object), and then the third item within that. If you are counting along, this will be the float 2.1, as verified in **Figure 55**.

Figure 55: Indexing a List inside a List

```
In [10]: second_list[0][2]
Out[10]: 2.1
```

Indeed, it is not just lists that can be indexed or sliced, in fact we can index/slice a string object as well. In this case we are counting the number of characters (again with a count that starts at zero not at one!). For instance:

```
third_list = [["James", "the", "hat", "Pennington"], "not a list"]
third_list[0][0][:3]
```

In this example we have again created a list of two items, where the first item is itself a list. This list within a list consists of four items each of which is a string. Our indexing, the second line of code, now goes to three levels. First we are selecting the first item of the list, which is the list of strings. Secondly we are selecting the first item of this list (the word "James"). Finally, we are asking for all the characters at the start of this string up to, but not including, the 4th character (remember, Python starts counting at zero!). In other words, the first, second, and third characters in the first item of the list that is the first item of the *third_list* object. This gives the output shown in **Figure 56**.

Figure 56: Slicing a String Object

```
In [11]: third_list = [["James", "the", "hat", "Pennington"], "not a list"]
         third_list[0][0][:3]
Out[11]: 'Jam'
```

Finally for lists, we will discuss some of the functions that can be used to modify a list. If we recall from the start of this section, lists are changeable in Python, as are the items within it. For example, we can index an item in a list, and simply replace it with a new one. When we created *first_list*, we set the last value to be the Boolean value of *False*. To change this, we can simply type the following:

```
first_list[3] = True
```

Here we are telling Python to take the 4th item of the list (remember … yeah you remember now) and change it from *False* to *True*. Indeed it would not need to be change to another Boolean, it could have been changed to a string, and integer or even another list. We can verify that the change has been made by the output in **Figure 57**.

Figure 57: Changing a Value in a List

```
In [12]: first_list[3] = True
         first_list

Out[12]: [1, 'a', 2.1, True]
```

We can also add items to our list. The function *list.append( )* – where *list* is the name of our list – will add another item to the end of the list. For instance:

```
first_list.append(['James', 'Pennington'])
```

will append an item to the end of *first_list*, in this case another list that contains two strings. We can verify the output in **Figure 58**.

Figure 58: Appending an Item to a List

```
In [13]: first_list.append(['James', 'Pennington'])
         first_list

Out[13]: [1, 'a', 2.1, True, ['James', 'Pennington']]
```

While append is useful when we don't care about order, in some cases we will want to insert our new item into a specific place in the list. Here we use the *list.insert( )* function. This function takes two inputs: firstly the position we want to insert to, and secondly the item we want to insert. So *first_list.insert(1, "Gareth")* will insert the string "Gareth" into the second position of the list. We verify this has worked in **Figure 59**.

Figure 59: Inserting an Item into a Specific Position in a List

```
In [14]: first_list.insert(1, "Gareth")
         first_list

Out[14]: [1, 'Gareth', 'a', 2.1, True, ['James', 'Pennington']]
```

We can also uses functions to remove items. As the opposite of append, the function *list.pop( )* will remove the last item of the list, as shown in **Figure 60**.

Figure 60: Removing the Last Item from a List

```
In [3]: first_list.pop()
        first_list

Out[3]: [1, 'Gareth', 'a', 2.1, True]
```

Note, the last item in *first_list* was itself a list. The *pop( )* command removes the item entirely, and therefore the whole list. However, unlike *append( )*, we can also use *pop( )* to remove a specific item by its location. For instance, *first_list.pop(1)* will remove the second item from out list:

Figure 61: Removing an Item from a List by its Position

```
In [4]: first_list.pop(1)
        first_list

Out[4]: [1, 'a', 2.1, True]
```

Finally, we may want to remove an item by its value rather than its positon. This can be very helpful when we are dealing with very long lists! Instead of *pop( )*, here we use *list.remove( )* where the value we want to remove is listed between the brackets. As a note here, if we recall from the start of this section, Python lists allow for duplicated values. The *remove( )* function will delete only the first instance of the value in the list. For instance:

```
fourth_list = [1, 5, 1, 4, 2, 1, 3]
fourth_list.remove(1)
```

will delete only the first occurrence of the value of 1 – in this case, the first item of the list. This is verified in **Figure 62**.

Figure 62: Removing the First Occurrence of a Value in a List

```
In [18]: fourth_list = [1, 5, 1, 4, 2, 1, 3]
         fourth_list.remove(1)
         fourth_list

Out[18]: [5, 1, 4, 2, 1, 3]
```

There are several other functions that can be used on lists (see the documentation for more details: https://docs.python.org/3/tutorial/datastructures.html). We will cover just two more here, as they are the ones that are most useful in this space.

The first is the sort function. If we recall the start of this section, Python lists are unordered by default. The ordering of the values is based on where they were added not on their values. Often

it is useful to be able to index based on value not just on an arbitrary position in the list. As an example, let's assume I have a list of PMA marks associated with a student:

```
pma_marks = [72, 54, 62, 66, 71]
```

If I want to know the students top two marks I have to read the whole list and work this out in my head. Granted this is relatively easily done in a list this short, but if this were a much longer list, or indeed a list of lists (multiple students), things will get trickier. It is valuable for me to be able to sort the list, which I can do using *list.sort( )*, shown in **Figure 63**.

Figure 63: Sorting a Python List

```
In [20]: pma_marks = [72, 54, 62, 66, 71]
         pma_marks.sort()
         pma_marks

Out[20]: [54, 62, 66, 71, 72]
```

Note that the list automatically sorts into ascending order (smallest numbers first). This is because ascending is the default value. If preferred, I can sort in reverse (descending) order by passing the command *reverse=True*. Not that *True* is capital "T" and no quotation marks, meaning that this is a Boolean. A sort in descending order, followed by indexing just the first two items based on my initial query (find the highest two marks for the student) is shown in **Figure 64**.

Figure 64: Reverse Sorting and Indexing a List

```
In [21]: pma_marks.sort(reverse=True)
         pma_marks[:2]

Out[21]: [72, 71]
```

Finally for this section, we'll discuss the *count( )* function. As the name suggests, this function will count the occurrence of a value in a list. Recall our *fourth_list* shown in Figure 62. Even after we removed the first occurrence, this list still has multiple occurrence of the value of 1. We can use *list.count( )* to calculate how many, shown in **Figure 65**.

Figure 65: Counting the Occurrences of a Value in a List

```
In [22]: fourth_list.count(1)

Out[22]: 2
```

As expected, we count two occurrences. Again, in a list of six items we don't really need a function to do this, however, it should be fairly clear how this will be much more useful when dealing with very long lists. As before, the source code used in this section is available from Github, titled "c3-notebooks-Lists.ipynb".

3.4. Other Data Structures: Dictionaries, tuples and sets

Although lists are very common, they are not the only data structure we will encounter. The second most common, and arguably the most useful of all, are dictionaries.

Dictionaries

Dictionaries share some characteristics of lists, except every item is in fact a pair: a key (effectively a name for a stored item) and a value (the item being stored). In this way they are similar to the variables we have been creating throughout this chapter. We can look up a value in a dictionary by its name (and vice versa – although this is not really the point).

Unlike lists, the order of items in a dictionary is irrelevant, and essentially arbitrary. We do not want to use indexes in this case, as we are finding our values by their key not by their position. Some examples of where we may want to use dictionaries:

- User passwords: we could use a username as a key to look up a password value (obviously don't do this as it is not a secure way to manage logins);
- Student PMA marks: we could use a student's ID (key) to find their PMA marks (value);
- Email address lookup: we enter a person's name (key) to get their email address (value).

In fact, the use cases are plentiful and dictionaries are an important part of programming. If you have a technical background, or have done modules such as *Big Data, Analytics & Visualisation* or *Information Communication Technology* you will probably see similarities between this way of working and working with databases.

Where a list in Python is created using square brackets, a dictionary is built with curly brackets – { }. As above, within these brackets we need to pass two objects, the key and the value, for each item we add to the dictionary. Each item is separated by a comma, as before, and each key/value combination by a colon. Here is a simple example:

```
first_dictionary = {'James Pennington':100000, 'Mark Bonnett':23000, 'Emily
Davis':24000}
```

This dictionary, which could possibly show something like salaries as a random example, has three objects (people). For each object we have a key (their name) and a value (a number which could or could not be their salary).

We can retrieve values from the dictionary by passing the key. This is done by using the syntax *dictionary[key]*, as shown in **Figure 66**.

Figure 66: A Python Dictionary

```
In [1]: first_dictionary = {'James Pennington':100000, 'Mark Bonnett':23000, 'Emily Davis':24000}
        first_dictionary['James Pennington']

Out[1]: 100000
```

As can be seen, the dictionary has looked up the value (100,000) based on the key given in the square brackets ("James Pennington"). We can add new items to a list in a very similar way, by again using the square brackets containing the new key, but this time following this with an equals symbol followed by the new value. For instance,

Figure 67: Adding New Items to a Dictionary

```
In [5]: first_dictionary['Gareth Edwards'] = 24000
        first_dictionary

Out[5]: {'Emily Davis': 24000,
         'Gareth Edwards': 24000,
         'James Pennington': 100000,
         'Mark Bonnett': 23000}
```

Another key difference between a list and a dictionary is that we could not have duplicated items (keys) in the latter. If we ran similar code as above for an existing key (say "Mark Bonnett") but with a different value, this would not add an additional "Mark Bonnett", it would just update the value stored against this key. This is demonstrated in **Figure 68**.

Figure 68: Updating a Dictionary Item

```
In [6]: first_dictionary['Mark Bonnett'] = 22000
        first_dictionary

Out[6]: {'Emily Davis': 24000,
         'Gareth Edwards': 24000,
         'James Pennington': 100000,
         'Mark Bonnett': 22000}
```

As expected, the value associated with "Mark Bonnett" changes, rather than an additional value being added.

Again there are multiple types that can be used inside a dictionary, not just integers. Indeed the value stored could be another dictionary or a list. For instance, let's say we wanted to add another person to our dictionary, and if the values were salaries, this person had two salaries. It may be better to then add this as a list, as shown in **Figure 69**.

Figure 69: Adding a List as a Value in a Dictionary

```
In [7]: first_dictionary['Michael Mortenson'] = [24000, 36000]
        first_dictionary

Out[7]: {'Emily Davis': 24000,
         'Gareth Edwards': 24000,
         'James Pennington': 100000,
         'Mark Bonnett': 22000,
         'Michael Mortenson': [24000, 36000]}
```

Using a dictionary inside a dictionary opens up even more use-cases. We can expand our dictionary concept above to a full human resources record. For instance:

```
second_dictionary = {'Michael Mortenson':{'DOB':'13/06/1981',
'HighestEd':'PhD', 'NAME_REMOVED':[24000, 36000]}}
```

We can call specific values from our nested dictionary (dictionary inside a dictionary) using two sets of square brackets as we did when indexing (section 3.3). This is demonstrated in **Figure 70**.

Figure 70: Calling Items in a Nested Dictionary

```
In [11]: second_dictionary = {'Michael Mortenson':{'DOB':'13/06/1981', 'HighestEd':'PhD',\
                                                   'NAME_REMOVED':[24000, 36000]}}
         second_dictionary['Michael Mortenson']['DOB']

Out[11]: '13/06/1981'
```

Note the use of the backslash in this example to add a return to the line of code. Recall that backslash means that the program ignores the item that comes after it when processing the code. This allows me to move the last key/value pair to a new line to make it easier to read without breaking the code.

Sometimes we would want to remove an item from a dictionary, for instance if someone in our dictionary no longer worked for the company. In dictionaries we do this using the *del* function (short for delete). For instance:

```
del first_dictionary['Gareth Edwards']
```

As before, we use the dictionary name and the key to delete in square brackets, but note that the *del* function has no brackets unlike every other function we have encountered. This will remove entirely this key from the dictionary, and also the value stored against it, as shown in **Figure 71**.

Figure 71: Deleting an Item from a Dictionary

```
In [8]: del first_dictionary['Gareth Edwards']
        first_dictionary

Out[8]: {'Emily Davis': 24000,
         'James Pennington': 100000,
         'Mark Bonnett': 22000,
         'Michael Mortenson': [24000, 36000]}
```

Although indexing a dictionary is not really relevant, that is not to say that sorting would not be. This is done using the *sorted( )* function, as shown in **Figure 72**.

Figure 72: Sorting a Dictionary

```
In [9]: sorted(first_dictionary)

Out[9]: ['Emily Davis', 'James Pennington', 'Mark Bonnett', 'Michael Mortenson']
```

As can be seen, the dictionary is now sorted into the order of the keys. Note that as each key is a single string, this is sorted by the first character (the first letter of the first name). If we want to sort by surname the string would need to be in the order "SURNAME, FORENAME". While this has some uses, it is perhaps more useful to sort by values. Unfortunately this is a little more complex, but will be something we return to later.

Tuples and Sets

There are two other data structures we'll touch on, although they are less commonly used. The first is tuples. Tuples are a lot like lists, except they have some slightly different properties which we will touch on in due course. A tuple is (typically) made using normal brackets. For instance:

```
first_tuple = ("a", 2, 5.5, False)
```

As a note, in fact we do not need the round brackets for single tuples only for nested tuples. So, ***first_tuple = "a", 2, 5.5, False*** *would work in exactly the same way. Any time we have a nested tuple (a tuple inside a*

tuple, or a tuple inside a list or dictionary), the brackets become necessary. I generally find it easier to always use brackets as it is quicker to read the code and understand the structure being used.

As with lists, we are creating an unordered structure of single items (rather than the key/value system of dictionaries) which can have repetition. Again we index (or slice) from a tuple in the same way as we would with a list, as shown in **Figure 73** (remember to count from zero!).

```
first_tuple[1]
```

Figure 73: A Python Tuple

```
In [14]: first_tuple = ("a", 2, 5.5, False)
         first_tuple[1]

Out[14]: 2
```

One key difference between tuples and lists is that a tuple is *immutable*. This means that we cannot change items in a tuple in the way we would change an item in a list. This is demonstrated in the following, which will throw an error (**Figure 74**).

Figure 74: Tuples are Immutable

```
In [15]: first_tuple[3] = True

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-bc9144b61236> in <module>()
----> 1 first_tuple[3] = True

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

As can be seen, we get a TypeError that tells us that "[a] 'tuple' object does not support item assignment" – i.e. we can't assign a new value to a tuple item. It is possible to include mutable items inside a tuple. For instance, if an item in a tuple is a list, it is possible to change items inside the list, by indexing in the usual fashion. Indeed, we can also sort a list inside a tuple (using the code *tuple.sorted( )* – although this will only sort mutable items (like lists) inside the tuple not the tuple itself). For an example, **Figure 75** shows the code to insert a new item inside a list (mutable) inside a tuple (immutable).

Figure 75: Changing Mutable Items inside a Tuple

```
In [18]: second_tuple = (["James", "Pennington"], 2, "not a list")
         second_tuple[0].insert(1, "Julian")
         second_tuple

Out[18]: (['James', 'Julian', 'Pennington'], 2, 'not a list')
```

This demonstrates something of the main use case of tuples over lists. If we want a list to be immutable (for instance, if we do not want a user to be able to change these values), then a tuple will be preferable. If we want to change this value, we need to recreate the whole tuple object (**Figure 76**):

Figure 76: Changing a Tuple

```
In [17]: first_tuple = ("a", 2, 5.5, True)
         first_tuple

Out[17]: ('a', 2, 5.5, True)
```

The final data type is sets. Again sets have similarity with lists but a set can only have one instance of each item (no duplication). Again, this is the main use case – scenarios where we want objects to either be in a set (once) or not in a set. For instance, if we want to do membership tests. As an example, maybe I want to keep a set of students who have submitted a PMA so I can check if a student is late. Sets use the curly brackets, { }, like dictionaries, but have no key/value pairs (i.e. no colon separating values). An example set of student IDs would be:

```
first_set = {"u10000", "u10001", "u10002"}
```

Unlike a list, indexing is not really relevant to sets. We do not care about the position of items, just whether they are in a set or not in a set. **Figure 77** shows an error message you would get if you try to index a set.

Figure 77: TypeError when Indexing a Set

```
In [20]: first_set = {"u10000", "u10001", "u10002"}
         first_set[1]

         ---------------------------------------------------------------------------
         TypeError                                 Traceback (most recent call last)
         <ipython-input-20-5d1ca31d77f8> in <module>()
               1 first_set = {"u10000", "u10001", "u10002"}
         ----> 2 first_set[1]

         TypeError: 'set' object does not support indexing
```

We can add new items (students) to our list by using either *set.add( )* for single items and *set.update( )* if we have multiple items. For instance:

```
first_set.add('u10004')
first_set.update(['u10005', 'u10007'])
```

As can be seen, we are using *set.update( )* to add a list of items. However, the behaviour here may not be quite as expected, as shown in **Figure 78**.

Figure 78: Adding and Updating a Set

```
In [21]: first_set.add('u10004')
         first_set.update(['u10005', 'u10007'])
         first_set

Out[21]: {'u10000', 'u10001', 'u10002', 'u10004', 'u10005', 'u10007'}
```

When updating a list to a set, the list itself does not become an item in the set, rather its individual values are added. In other words, the list is "unpacked". If we consider the main use case of sets, checking if an item is in a set or not in a set, this makes some sense.

We can also remove items by using the *set.discard( )* function. As an example:

```
first_set.discard('u10004')
```

Note, you can also use set.remove( ) to achieve a similar effect. The key difference is that an item is not in the set, set.remove( ) will result in an error, while set.discard( ) will not.

This will remove the item for the set as shown in **Figure 79**.

Figure 79: Removing Item from a Set using discard( )

```
In [22]: first_set.discard('u10004')
         first_set

Out[22]: {'u10000', 'u10001', 'u10002', 'u10005', 'u10007'}
```

Finally we consider the main use case for sets, testing whether an item is a member of the set or not. Sets allow us to very quickly perform these operations. For instance, I may wish to test if the student 'u10000' has submitted their PMA (is in the *first_set* object). I can do this with the following code, which produces the output shown in **Figure 80**.

```
"u10000" in first_set
```

Figure 80: Testing Set Membership

```
In [23]: "u10000" in first_set
Out[23]: True
```

This test returns a Boolean – either *True* or *False*. As "u10000" is in *first_set*, the response in *True*. If we were to test "u10003", this is not in our set so the response would be *False*.

As before, all the code used in this section is available from Github. The title of this notebook is "c3-notebooks-OtherDataStructures.ipynb".

3.5. If Statements and Loops

So far we have covered only single-line operations. All of our programs have taken some data and performed a single operation on it, we haven't worked on anything that has a *control flow* between multiple operations or conditions. This will be the subject of this section.

The first of these is an "if statement". If statements allow us to examine an item of data in comparison to some condition, and take a different action dependent on the result. As a simple example, we have an e-commerce website with a section for "Orders" that allows a customer to check what outstanding orders they have to be delivered. When a customer comes to this section, we want to do one of two things:

1. If there are outstanding orders, we want to display these to the customer;
2. If there are no outstanding orders, we want to display a message that says "no outstanding orders".

To do this, we need to apply a logic test that says "if orders exist do (1), if no orders exist do (2)". Obviously this will depend on some data, so, building on the last section, let's add this first:

```
customers = {'James':{'name':'James Pennington', \
                              'open_orders':1, \
                              'orders':['o10001', 'o10002', 'o10004']
                        }, \
                        'Gareth':{'name':'Gareth Edwards', \
                              'open_orders':0, \
                              'orders':[]
                        }, \
                        'Mark':{'name':'Mark Bonnett', \
                              'open_orders':1, \
                              'orders':['o10003']
                        }, \
```

```
                        'Emily':{'name':'Emily Davis', \
                                 'open_orders':0, \
                                 'orders':[]
                        }
}
```

As can be seen, we have built a nested dictionary that, at the top level, includes four customers where their first name is the key. (If this does not make sense, go back and review section 3.4). The value is another dictionary that contains three items:

1. 'name' – A string;
2. 'open_orders' – An integer that says how many orders are open;
3. 'orders' – A list of the order IDs that are open.

If we call the dictionary object (*customers*), we get the following output (**Figure 81**).

Figure 81: Customer Dictionary Output

```
In [5]: customers

Out[5]: {'Emily': {'name': 'Emily Davis', 'open_orders': 0, 'orders': []},
         'Gareth': {'name': 'Gareth Edwards', 'open_orders': 0, 'orders': []},
         'James': {'name': 'James Pennington',
          'open_orders': 3,
          'orders': ['o10001', 'o10002', 'o10004']},
         'Mark': {'name': 'Mark Bonnett', 'open_orders': 1, 'orders': ['o10003']}}
```

Recalling our initial brief, we want to determine if an individual customer has any open orders (i.e. *open_orders* > 0). We can do this with an if statement – the first part of which is in the form (don't type this in yet!):

```
if customers['James']['open_orders'] > 0:
```

This applies the logical test discussed by looking up the number of open orders associated with 'James', and testing if it is greater than zero. The form is the keyword of the function ("if"), followed by the data item (James' open orders), followed by the condition ("> 0", greater than zero), and finally a colon (":"). A list of operators (like ">" for greater than, is included in section 3.8). If it passes the test we want to take a specific action. To do so we expand the code as follows (don't type this in yet!):

```
if customers['James']['open_orders'] > 0:
        print('You have ' + str(customers['James']['open_orders']) + ' open
orders.')
```

Again, there is nothing necessarily new here – if we pass the test we want to use the print function, and we are joining a message ("You have X open orders") with the a string representation of the number of 'open_orders' associated with 'James' (replacing the "X"). (If this does not make sense, go back and review section 3.1).

However, there is a note about indentation and Python. Unlike in some other languages, Python will often use indentation to interpret the program. Our "if" statement ends with a colon, and Python will expect the next line (the thing to do if it passes the test) to be indented by a tab. If you use Jupyter Notebooks to write your code, or modern code editors such as Sublime Text (https://www.sublimetext.com/), these indentations will be added for you, but this is something that can (and probably will) go wrong if you use something like Word to write your code.

We have now created our test, and have an appropriate action if the customer does have any existing orders (print the number of orders to screen), but we do not have any action for the opposite case, if the customer has no orders. We can do this by adding an "else" after the first action, as follows:

```
if customers['James']['open_orders'] > 0:
        print('You have ' + str(customers['James']['open_orders']) + ' open
orders.')
else:
        print('You have no open orders')
```

A few things to note here. Firstly, the else command is not indented – it is back at the far left. If you are typing the code in, you will need to delete the indentation with backspace before typing "else:". Secondly, the "else" is again followed by a colon (":"). This effectively means that else is another logic test – we are instructing Python to take the action that follows (printing that they have no open orders) if it has failed the initial logic test (if open_orders > 0). If we run this code we should get the output shown in **Figure 82**.

Figure 82: A Python If Statement for a Customer with Open Orders

```
In [6]: if customers['James']['open_orders'] > 0:
            print('You have ' + str(customers['James']['open_orders']) + ' open orders.')
        else:
            print('You have no open orders')

        You have 3 open orders.
```

As can be seen, the program confirms that "James" does have open orders, and then prints the correct number of them (3 open orders). We can also check that it works in the opposite way, by entering a customer with no open orders (Emily):

Figure 83: A Python If Statement for a Customer without Open Orders

```
In [7]: if customers['Emily']['open_orders'] > 0:
            print('You have ' + str(customers['Emily']['open_orders']) + ' open orders.')
        else:
            print('You have no open orders')

        You have no open orders
```

You will note that we had to change the customer "James" to "Emily" in two places – the initial if statement and then also in the print statement. This is a little inefficient so let's make a change to the code:

```
customer = 'Emily'

if customers[customer]['open_orders'] > 0:
        print('You have ' + str(customers[customer]['open_orders']) + '
open orders.')
else:
        print('You have no open orders')
```

Now we have a variable in which we can set the customer name, and the if statement and print statement no longer require an update as they are pointed at this variable name. If we want to change the customer, we only need to change the one line, *customer = 'Emily'*, to whatever we want (e.g. *customer = 'Gareth'*) and the code will work. Test it out yourself!

We now have a working if statement, which can carry out our logic test we need, and then display a different output depending on the output of the test. However, the success or otherwise of the logic test is dependent on the *open_orders* data point being correct. What should happen if *open_orders* is listed as zero, but there are orders IDs stored in the *orders* list?

One solution is to also print the list of orders in this case. We can do this by adding an additional if statement to our loop. After we conduct our first test (is *open_orders* > 0), we will then conduct the second test (are there any orders in the *orders* list), and if the answer to both of these is "no" we then print the "no orders" message.

To do this we need to introduce two new functions. The first will form part of our test, we want to count how many orders are in the *orders* list. We can do this with the *len( )* function (where "len" is short for "length". *len( )* can be used to measure the length of a string, but it can also measure the length (number of items) of a list. Let's try the following, which produces the output shown in **Figure 84**.

```
len(customers['James']['orders'])
```

Figure 84: Counting Items in a List using len( )

```
In [9]: len(customers['James']['orders'])
Out[9]: 3
```

There are three orders in James' *orders* list, and this is what the function returns. Note, unlike indexing (which starts at zero), *len( )* does a physical count where zero would be no items.

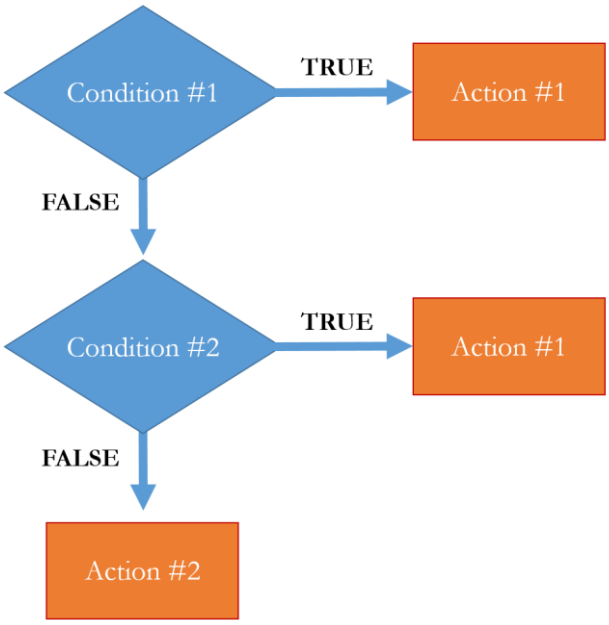
The second thing we need to introduce is an additional if inside our if, that only is tested if the first test is failed. In some languages this would be a "else if", but in Python this is shortened to *elif*. Try the following code:

```
customer = 'Emily'

if customers[customer]['open_orders'] > 0:
        print('You have ' + str(customers[customer]['open_orders']) + '
open orders.')
elif len(customers[customer]['orders']) > 0:
        print('You have ' + str(customers[customer]['open_orders']) + '
open orders.')
else:
        print('You have no open orders')
```

Now we have this additional test (is the *orders* list longer than zero) that will only be conducted if the first test is failed. If either of the tests pass, we produce the same output (the message saying how many orders the customers has); if they both fail we produce a different output (the no orders message). This logic flow is shown in **Figure 85**.

Figure 85: The Logic Flow of our Nested If Statement



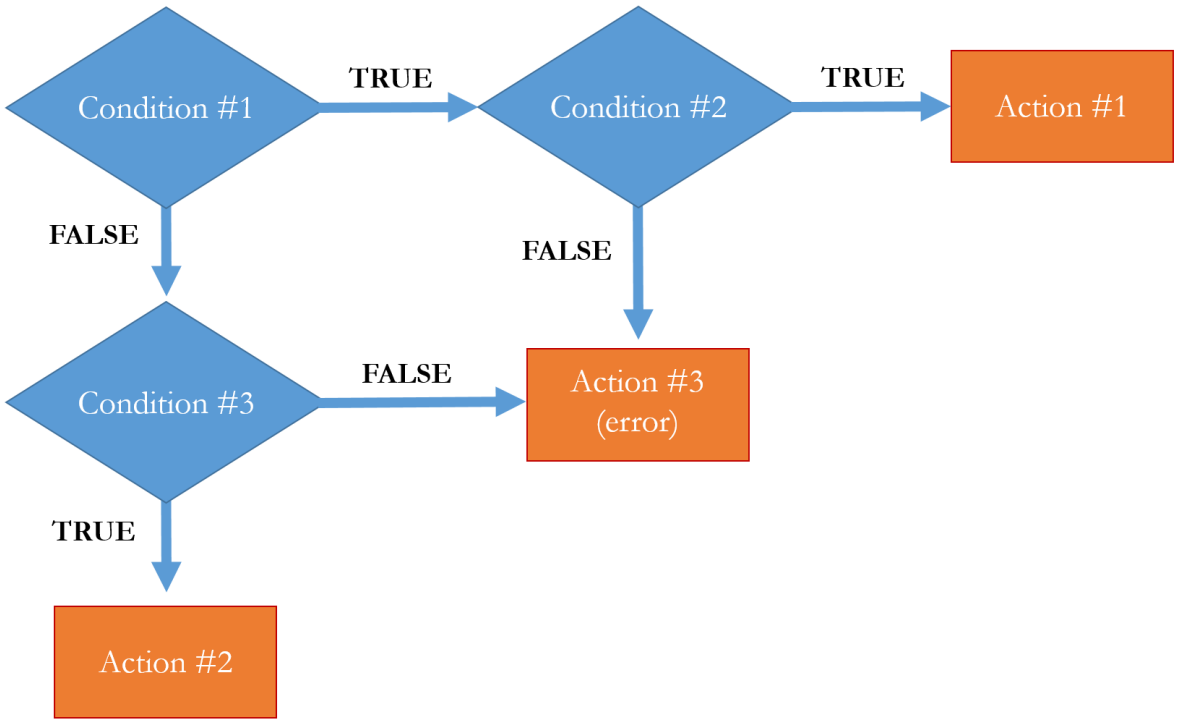
We can test this code and verify that it works. However, this may not be the optimal solution. If the system has zero recorded for *open_orders*, but there are orders in the *orders* list, this would suggest there is some kind of problem. Rather than ignoring the problem, which is what we do now, maybe we would want to inform the customer there is an error. Consider the following:

```
customer = 'Emily'

if customers[customer]['open_orders'] > 0:
        if len(customers[customer]['orders']) > 0:
                print('You have ' + str(customers[customer]['open_orders'])
+ ' open orders.')
        else:
                print('There is an error')
else:
        if len(customers[customer]['orders']) < 1:
                print('You have no open orders')
        else:
                print('There is an error')
```

In this example we are conducting an if test inside an if test. First we check if there are open orders and then we check if the length of the orders list is zero (either by checking if it is greater than (“>”) zero, or in the second part if it is less than (“<”) one). If these results are contradictory, we throw up an error message to the customer, if not, we show the appropriate message. This logic flow is shown in **Figure 86**.

Figure 86: The Logic Flow of our Second Nested If Statement



Again, we can verify our code works by running in the Notebook (shown in **Figure 87**).

Figure 87: Output of our Second Nested If Statement

```
In [12]: customer = 'Emily'

         if customers[customer]['open_orders'] > 0:
             if len(customers[customer]['orders']) > 0:
                 print('You have ' + str(customers[customer]['open_orders']) + ' open orders.')
             else:
                 print('There is an error')
         else:
             if len(customers[customer]['orders']) < 1:
                 print('You have no open orders')
             else:
                 print('There is an error')

         You have no open orders
```

We now have working nested if statements that can also handle errors. However, it will only work on one customer at a time – whatever we set as the *customer* variable. This is fine if we are using this on our website for a customer to self-check, however, there may be other uses of this data. For instance, another use case maybe for our operations team to check all customers to identify all open orders. To do this we can wrap our nested if statements in a *for loop*.

A for loop will go throw every item passed to it, and apply the same operation to each. This can be a set of items or some data structure like a list or a dictionary. The general syntax for a for

loop is the function ("for") followed by the name used in the function for each item (can be any name at all), followed by the items (could be a list or a dictionary), followed again by a colon (":"). We will find it relatively easy to add a for loop to our code as follows:

```
for customer in customers:
        if customers[customer]['open_orders'] > 0:
                if len(customers[customer]['orders']) > 0:
                        print(customers[customer]['name'] + \
                                ' has ' + \
                                str(customers[customer]['open_orders']) + \
                                ' open orders.')
                else:
                        print('There is an error')
        else:
                if len(customers[customer]['orders']) < 1:
                        print(customers[customer]['name'] + \
                                ' has no open orders')
                else:
                        print('There is an error')
```

Fundamentally we have the same nested if statements as before, the only change is that we are also printing the customer name each time, except it is wrapped in this for statement. Note that because our 'name' value is already a string, we do not need to wrap it in a *str( )* function.

Everything after the for statement again needs to be indented. Note, in the initial line *for customer in customers:* the *customers* part refers to our *customers* dictionary which we will iterate through. The *customer* part can be any name, but it is this name that needs to be referred to later in the text. For instance, the below (replacing *customer* for i) would work in the same way:

```
for i in customers:
        if customers[i]['open_orders'] > 0:
                if len(customers[i]['orders']) > 0:
                        print(customers[i]['name'] + \
                                ' has ' + \
                                str(customers[i]['open_orders']) + \
                                ' open orders.')
                else:
                        print('There is an error')
        else:
                if len(customers[i]['orders']) < 1:
                        print(customers[i]['name'] + \
                                ' has no open orders')
                else:
                        print('There is an error')
```

If we run our new code we should get the following output (**Figure 88**).

Figure 88: A For Loop

```
In [14]: for customer in customers:
             if customers[customer]['open_orders'] > 0:
                 if len(customers[customer]['orders']) > 0:
                     print(customers[customer]['name'] + \
                         ' has ' + str(customers[customer]['open_orders']) + \
                         ' open orders.')
                 else:
                     print('There is an error')
             else:
                 if len(customers[customer]['orders']) < 1:
                     print(customers[customer]['name'] +' has no open orders')
                 else:
                     print('There is an error')

         James Pennington has 3 open orders.
         Emily Davis has no open orders
         Gareth Edwards has no open orders
         Mark Bonnett has 1 open orders.
```

The for loop we have used here works well for any scenario where we have a data structure like a list or a dictionary we want to iterate through. However, this will not always be the case. Sometimes we may just want to run through the loop a set number of times. In this case the *range( )* function is useful. *range( )* takes two inputs, the starting number and the last number (as with indexing this is the number we want to go up to but not include). The below shows this inside a for loop, with the output shown in **Figure 89**.

```
for i in range(1, 10):
        print("This is number #" + str(i))
```

Figure 89: A For Loop using the range( ) Function

```
In [16]: for i in range(1, 10):
             print("This is number #" + str(i))

         This is number #1
         This is number #2
         This is number #3
         This is number #4
         This is number #5
         This is number #6
         This is number #7
         This is number #8
         This is number #9
```

One other type of loop is not based on a fixed number (either given via the *range( )* function or based on the number of items passed to the loop) but based on a condition. Let's say we are holding an event, and we want to restrict access to 10 participants maximum (granted we could just do this in a *range(1, 11)* function, but let's just say we don't want to do that). We will create a variable called *capacity* which will be a Boolean. At the start of this exercise we have no participants so obviously we are not at capacity we will set this value to be *False*:

```
capacity = False
```

However, as we add new participants, or more specifically when we add 10 participants, we will reach this capacity and the value will need to be changed to *True*. At this point we want to stop our loop and not allow any more participants. We can build in this logic using a *while* loop. This is a loop that is based on a logic condition, such that when it ends the loop it will test against the logic condition and if it still passes the condition it will loop through again. Consider the following code:

```
capacity = False
participants = 0
while capacity == False:
        participants = participants + 1
        print('There are ' + str(participants) + ' participants')
        if participants >= 10:
                capacity = True
```

Alongside our *capacity* variable which starts as *False*, we also have a variable for *participants*, which starts at zero. As we go through our loop, each time a new participant will be accepted. We do this by making the *participants* variable equal the number of participants stored plus one more (*participants* = *participants* + *1*). When then print to screen the number of participants using a *str( )* function as before.

After that we are going to apply an in if test as before. Our test asks "is the number of participants greater than or equal to (">=") 10. If it is, we want to change the *capacity* variable to *True*, we are now at capacity. At this point we want our loop to end because we cannot accept any more participants. The actual condition that controls this, the *while* statement of the first line of the loop, introduces a new symbol, the double equals ("=="). This is symbolises a logical test of do the two things equal each other. This is different from the single equals ("=") which we have been using to assign objects to names. In other words, we are asking 'does capacity equal *False*'. The output of the loop is shown in **Figure 90**.

Figure 90: A While Loop

```
In [20]: capacity = False
         participants = 0

         while capacity == False:
             participants = participants + 1
             print('There are ' + str(participants) + ' participants')
             if participants >= 10:
                 capacity = True

         There are 1 participants
         There are 2 participants
         There are 3 participants
         There are 4 participants
         There are 5 participants
         There are 6 participants
         There are 7 participants
         There are 8 participants
         There are 9 participants
         There are 10 participants
```

As we see, the while loop has worked as expected and run exactly 10 times. Each time it has added one participant to the *participant* variable (count), printed the number of participants to screen, and then conducted our if test. The if test fails (the participant count is under 10) every time until the 10th time. At this point the if test passes (there are 10 participants), and the *capacity* variable is changed to *True*. When we return to the top of the loop the *while* condition no longer holds – now capacity is equal to *True* (i.e. it is no longer the case that capacity equals *False*). Consequently, the loop ends.

*As a note, while loops can easily go wrong if there is no proper way to break the loop and the while condition. If we removed from the above code the part that changes capacity to True (**capacity = True**), then we would have what is called an infinite loop – the loop would just go on forever. If you find yourself in an infinite loop, you will need to interrupt or reconnect the session using the methods discussed in section 2.3. If you are using an interactive Python shell (like IPython) then Ctrl+C or Cmd+C will interrupt the loop.*

As before, the Notebook used in this section is available from Github. The file is titled "c3-notebooks-IFs+Loops.ipynb".

3.6. Functions

As we indicated earlier, one of the key efforts in programming is trying to work in as efficient a way as possible. Often programming we would want to use the same code in multiple places, and it would be more efficient to have this saved as a function, so we can call the function as a single line of code rather than type everything out again.

Let's resume our e-commerce example. We created a dictionary of our customers and their open orders earlier. As we are using a new notebook for this section we will need to recreate this:

```
customers = {'James':{'name':'James Pennington', \
                              'open_orders':1, \
                              'orders':['o10001', 'o10002', 'o10004']
                      }, \
                      'Gareth':{'name':'Gareth Edwards', \
                              'open_orders':0, \
                              'orders':[]
                      }, \
                      'Mark':{'name':'Mark Bonnett', \
                              'open_orders':1, \
                              'orders':['o10003']
                      }, \
                      'Emily':{'name':'Emily Davis', \
                              'open_orders':0, \
                              'orders':[]
                      }
}
customers['James']
```

We discussed earlier a use case where a customer would want to login, go to the "Orders" page of the website, and view their open orders. However, this is probably not the only part of our system where we would want to be able to make such a check. For instance, we may want to enable our sales staff to view open orders before visiting a client; our customer services team may need to view this information to deal with customer queries; our accounts team may need to review orders for invoicing purposes; and so on. We want to be able to include this functionality on all these separate pages of our staff intranet site.

Let's begin by creating the code to do this. We'll also need a dictionary of what the orders represent:

```
orders = {'o10001':{'date':'2019/01/10', \
                              'product':'Social Media Detector', \
                              'quantity':'1'
               }, \
               'o10002':{'date':'2019/01/13', \
                              'product':'Realistic Man\'s wig', \
                              'quantity':'2' \
               }, \
               'o10003':{'date':'2019/01/14', \
                              'product':'Weather\'s Originals', \
                              'quantity':'10' \
```

```
                }, \
                'o10004':{'date':'2019/01/15', \
                                'product':'Brown Shoes', \
                                'quantity':'2' \
                }
}
orders['o10001']
```

Now we can build a function that can take the name of a customer and print a list of all their orders. To begin with we'll write a program to show the orders of all customers using a for loop:

```
for customer in customers:
        if len(customers[customer]['orders']) > 0:
                print('Customer: ' + str(customer))
                for order in customers[customer]['orders']:
                        print('Date: ' + str(orders[order]['date']))
                        print('Product: ' + orders[order]['product'])
                        print('Quantity: ' + orders[order]['quantity'])
                print('\n')
        else:
                print('Customer: ' + str(customer) + ' has no orders')
                print('\n')
```

The majority of this functionality should be familiar to you from earlier sections. We have made use of for loops, if statements, *len( )* for measuring the length of a string, and creating print strings by concatenation/joining. There is one part that will not be familiar which is the *print('\n')* command. '*\n*' is the Python code for new line, so effectively we are asking Python to print a line break before going on to the next part of the for loop.

We can test our codes work by running it in our notebook, which should produce the following output (**Figure 91**):

Figure 91: Printing all Customer Orders using a For Loop

```
In [10]: for customer in customers:
             if len(customers[customer]['orders']) > 0:
                 print('Customer: ' + str(customer))
                 for order in customers[customer]['orders']:
                     print('Date: ' + str(orders[order]['date']))
                     print('Product: ' + orders[order]['product'])
                     print('Quantity: ' + orders[order]['quantity'])
                 print('\n')
             else:
                 print('Customer: ' + str(customer) + ' has no orders')
                 print('\n')

Customer: James
Date: 2019/01/10
Product: Social Media Detector
Quantity: 1
Date: 2019/01/13
Product: Realistic Man's wig
Quantity: 2
Date: 2019/01/15
Product: Brown Shoes
Quantity: 2


Customer: Gareth has no orders


Customer: Mark
Date: 2019/01/14
Product: Weather's Originals
Quantity: 10


Customer: Emily has no orders
```

As per the above, between each customer we have a blank line (line break) because of our *print('\n')* command. We also see that the if statement means if a customer has no orders we print this to screen, and if not we loop through each order printing the details.

We have two problems with this code in respect to our stated goal. While this is not a particularly long program (11 lines of code) we don't want to have to write this code on every page where the functionality will be used. If we do this, we'd also have the issue of having to rewrite this code on every page any time we have an update to the functionality. If we can store this as a single function we would only have 1 line of code on each page, and one place where updates need to be applied. Secondly, our program will loop through every customer. What we want is to be able to pass the name of a single customer, and return just their results. We can do this with a function.

Firstly, we need to tell Python that we are building a function. We can do so like this:

```
def order_printer(customer):
```

The *def* part (short for "define") tells Python this is a function, and that it has the name "order_printer". We can use any name we like but we want a name that is logical and describes the purpose of the function. After the function name, within the brackets, we tell Python if the function should expect a data input. This can be blank if the function requires no data to passed. If you recall section 3.3 on lists, we used the *list.pop( )* function. If we have no value in between the brackets of this function, Python removes the last value in the list. We could design our function like this, but in our case here we definitely want there to be a data item passed – the customer name we want to print orders for. The name we use here doesn't matter, for instance it could be the letter "i". Within the function we use this name whenever we want to refer to the data item being passed, in the same way as "*customer*" in our "*for customer in customers:*" loop. Let's look at the full code:

```
def order_printer(customer):
        if len(customers[customer]['orders']) > 0:
                print('Customer: ' + str(customer))
                for order in customers[customer]['orders']:
                        print('Date: ' + str(orders[order]['date']))
                        print('Product: ' + orders[order]['product'])
                        print('Quantity: ' + orders[order]['quantity'])
                print('\n')
        else:
                print('Customer: ' + str(customer) + ' has no orders')
                print('\n')
```

In fact it is exactly the same code as the code we used in the for loop, except we changed the first line to make it a function definition rather than a loop. If we want to call the function all we need to do is type the function name, and the data we wish to pass to it:

```
order_printer('James')
```

We can test the functionality in the notebook, for both '*James*', who has orders, and '*Emily*', who hasn't. This is shown in **Figure 92**.

Figure 92: The order_printer Custom Function

```
In [5]: order_printer('James')

        Customer: James
        Date: 2019/01/10
        Product: Social Media Detector
        Quantity: 1
        Date: 2019/01/13
        Product: Realistic Man's wig
        Quantity: 2
        Date: 2019/01/15
        Product: Brown Shoes
        Quantity: 2


In [6]: order_printer('Emily')

        Customer: Emily has no orders
```

We can now use the *order_printer* function anywhere in our code and it will run the assigned set of code on our data. For completion we will update the function with a bit of extra code so that we can deal with situations where the data input is not in our *customers* dictionary. Fortunately, this is fairly easy to do by just wrapping it in an if condition that check if the customer name is in the customer dictionary like so:

```
if customer in customers:
```

This means that we can pass a more friendly message if they are not, rather than the error message Python would pass. The final code, and friendly message, is shown in **Figure 93**.

Figure 93: Adding Error Handing to the order_printer Customer Function

```
In [7]: def order_printer(customer):
            if customer in customers:
                if len(customers[customer]['orders']) > 0:
                    print('Customer: ' + str(customer))
                    for order in customers[customer]['orders']:
                        print('Date: ' + str(orders[order]['date']))
                        print('Product: ' + orders[order]['product'])
                        print('Quantity: ' + orders[order]['quantity'])
                    print('\n')
                else:
                    print('Customer: ' + str(customer) + ' has no orders')
                    print('\n')
            else:
                print('Customer name not found. Please try again')

In [8]: order_printer('John')

        Customer name not found. Please try again
```

Alongside custom functions we can design ourselves, Python has a large number of in-built functions that can be used. We have already encountered several of these such as *print( )*, *str( )*, *len( )*, and others. The full list of these, and their definitions, are available here: https://docs.python.org/3.3/library/functions.html. As before, the notebook associated with this section is available from Github. The filename is "c3-notebooks-Functions.ipynb".

3.7. Programming Best Practice

Over the course of this chapter we have covered some of the common patterns and methods for writing Python programs. As we have gone through, we have touched upon some concepts of best practice such as error handing and using functions rather than repeating code. There are a few other tips for writing programs that are worth discussing.

Firstly, we will address comments. Currently, we have been writing programs in a lazy way without any comments. As our programs are very short and are not going to be reused, this is OK, but in practice we are likely to want to write longer, more complicated programs, and potentially to come back to these at later times. As currently written, we may have a hard time doing this, because we will probably have forgotten what the code does.

Python allows us to add comments that can allow us to write an explanation of our code which Python will ignore, but are useful when a human is reading the code. There are two methods to doing this in Python, either by wrapping the comment in triple apostrophes on either side (''') or by adding a hash before the comment (#) like so:

```
''' This is a comment '''
# This is also a comment
```

I tend to use hashes more often, but triple quotation marks can be useful if we want to add a comment inside a line of code. To demonstrate commenting code, we will add commentary to our custom *order_printer( )* function:

```
# order printer function to take a customer name, and check their current orders
def order_printer(customer): # customer is the customer's first name
        # check the customer is in the dictionary
        if customer in customers:
                # check the length of the orders value (the # of orders) is > 0
                if len(customers[customer]['orders']) > 0:
                        # print the customer name
                        print('Customer: ' + str(customer))
                        # loop through the customer's orders
                        for order in customers[customer]['orders']:
                                # print each of the values stored.
```

```
                                print('Date: ' + str(orders[order]['date']))
                                print('Product: ' + orders[order]['product'])
                                print('Quantity: ' + orders[order]['quantity'])
                        # print a line break after the customer orders
                        print('\n')
                else:
                        # if the customer has no orders print this to screen
                        print('Customer: ' + str(customer) + ' has no orders')
                        # print a line break after the customer orders
                        print('\n')
        else:
                # if the customer is not in the customers dictionary print error
                print('Customer name not found. Please try again')
```

Another useful tip, when writing long programs, is to write them in small chunks that can be tested rather than waiting until the end to test everything (and then potentially having a hard time working out what is wrong). For instance, when making our *order_printer( )* function we can test the individual parts as we go a long rather than at the end. We can use *print( )* functions to show the results to screen and to check that everything is working as expected. For instance, the following would be a way to check the orders loop works (note – we need to specify a variable for *customer* as we have taken the code outside the function so it is not being passed):

```
customer = 'James'
for order in customers[customer]['orders']:
        print(str(order))
```

If this works (which it does), we print a list of every order number associated with the customer 'James'. If it doesn't, we can fix it before adding it to our main function.

Also considering the *order_printer( )* function, we have made an assumption in the code that the dictionary will include all the variables (*date*, *product* and *quantity*). As we have made that dictionary ourselves we are fairly confident it will, but in practice this dictionary would probably actually be a database, and be based on real data. In practice as well, there is a very real possibility that some data may be missing for whatever reason. If this does happen, the code will crash and send an error message, which is probably not very helpful.

We can deal with this using a *try* statement. *try* statements, as the name suggests, will try a line of code and run it if it works. If it doesn't work, we can then give Python a different instruction such as printing a friendly error message or moving on to the next record. We can see this in a modified version of the function:

```
# order printer function to take a customer name, and check
```

```
# all their current orders
def order_printer(customer): # customer is the customer's first name
        # check the customer is in the dictionary
        if customer in customers:
                # check that the length of the orders value
                # (the # of orders) is > 0
                if len(customers[customer]['orders']) > 0:
                        # print the customer name
                        print('Customer: ' + str(customer))
                        # loop through the customer's orders
                        for order in customers[customer]['orders']:
                                # print each of the values stored.
                                # wrapped in try statements in case the value
                                # is missing
                                try:
                                        print('Date: ' + \
                                                str(orders[order]['date']))
                                except:
                                        continue
                                try:
                                        print('Product: ' + \
                                                orders[order]['product'])
                                except:
                                        continue
                                try:
                                        print('Quantity: ' + \
                                                orders[order]['quantity'])
                                except:
                                        continue
                        # print a lne break after the customer orders
                        print('\n')
                else:
                        # if the customer has no orders print this to screen
                        print('Customer: ' + str(customer) + ' has no orders')
                        # print a line break after the customer orders
                        print('\n')
        else:
                # if the customer is not in the customers dictionary
                # print an error message
                print('Customer name not found. Please try again')
```

Similarly to an if condition, we have added a pairing instruction to the *try* which is *except*. The code that follows *except* is what we want Python to do if the *try* fails. In this case we are instructing Python to *continue*, which it means we move on to the next item in the for loop. An alternative would be *pass*, which would tell Python to ignore this particular order:

```
# order printer function to take a customer name, and check
# all their current orders
def order_printer(customer): # customer is the customer's first name
        # check the customer is in the dictionary
```

```
        if customer in customers:
                # check that the length of the orders value
                # (the # of orders) is > 0
                if len(customers[customer]['orders']) > 0:
                        # print the customer name
                        print('Customer: ' + str(customer))
                        # loop through the customer's orders
                        for order in customers[customer]['orders']:
                                # print each of the values stored.
                                # wrapped in try statements in case the value
                                # is missing
                                try:
                                        print('Date: ' + \
                                                str(orders[order]['date']))
                                        print('Product: ' + \
                                                orders[order]['product'])
                                        print('Quantity: ' + \
                                                orders[order]['quantity'])
                                except:
                                        pass
                        # print a lne break after the customer orders
                        print('\n')
                else:
                        # if the customer has no orders print this to screen
                        print('Customer: ' + str(customer) + ' has no orders')
                        # print a lne break after the customer orders
                        print('\n')
        else:
                # if the customer is not in the customers dictionary
                # print an error message
                print('Customer name not found. Please try again')
```

Note that rather than having a *try* and *except* pair for every item in orders, we have only one *try* before all three and the *except* after. This is because we want Python to consider if all three items exist and only print if they do.

Finally, we can instruct Python that we want to interrupt the program and stop by using *break* in place of *continue* or *pass*. Typically we would want to print something to the user if doing so:

```
try:
        print('Date: ' + str(orders[order]['date']))
        print('Product: ' + orders[order]['product'])
        print('Quantity: ' + orders[order]['quantity'])
except:
        print('There was an error')
        break
```

Note however that this will not tell us what the error is, whereas just letting the code run with any "try" and "except" pair would. In practice we would also want to extract the actual error using the "sys" package.

For more on *try* and *except* see the following: https://docs.python.org/3/tutorial/errors.html and https://docs.python.org/3/library/exceptions.html.

3.8. Python Operators

The following is a list of Python operators:

Operator	***Description***	***Usage***
=	Variable assignment	a = 5
+	Addition	5 + 2 *(equals 7)*
-	Subtraction	4 - 1 *(equals 3)*
*	Multiplication	2 * 3 *(equals 6)*
/	Division	8 / 2 *(equals 4)*
**	Exponent	2 ** 3 *(equals 8)*
==	Are two items the same	3 == 6 / 2 (*True*)
!=	Are two items not the same	3 != 6 / 2 (*False*)
>	Is the value on the left greater than the value on the right	3 > 2 (*True*)
<	Is the value on the left less than the value on the right	3 < 2 (*False*)
>=	Is the value on the left greater than, or equal to, the value on the right	3 >= 3 (*True*)
<=	Is the value on the left less than, or equal to, the value on the right	3 <= 4 (*True*)
and	Are both the left item and the right item *True*	True and False *(False)*
or	Are either the left or the right item *True*	True or False *(True)*
is	Evaluates if a variable is a value	a = 2 a is 2 (*True)*
not	Evaluates if the opposite is true	a = 2 a is not 3 (*True*)
+=	Adds a value to an existing variable	a = 2 a += 1 (*equals 3*)
-=	Subtracts a value from an existing variable	a = 2

		a -= 1 (*equals 1*)
*=	Multiplies an existing variable by a value and stores this as the variable	a = 2 a *= 2 (*equals 4*)
/=	Divides an existing variable by a value and stores this as the variable	a = 4 a /= 2 (*equals 2*)

3.9. Exercises

In order to test that you have understood these concepts the following exercises will help you practice. If you are struggling to make them work contact me (m.mortenson@warwick.ac.uk) and I can help you debug or share example code.

1. Create a nested dictionary of students (3 students or more) and their PMA marks for three modules, based on IDs. Using this dictionary write a for loop that will print the name of each of their modules and the score achieved. Additionally, you will need to establish if the grade is a "fail", "pass", "merit" or "distinction" and print this to screen.
2. Using the above dictionary again, write a program that will calculate the average mark for each student and then print the name of the student with the highest average mark.
3. Write a function that can add a new module and PMA mark for student in the above dictionary. This program will need to update their average mark item.
4. Write a function that can allow a student to query this dictionary to determine the average mark they will require if they want to achieve a specific result (i.e. if the student wants to achieve a distinction, what do they need to average on their remaining PMAs?) The function should take three inputs: the student ID; the overall result desired; and the number of modules left. The function should, taking into account how many modules they have completed and how many are left, provide an average grade that specific student will need to attain to get their desired result. An example input/output would be:

```
mark_checker("u10001", "distinction", 5)
```

"Student 10001 requires an average of 68.2 marks per PMA to achieve a distinction"

4. Packages and Modules

Alongside Python's in-built function, and functions that we build ourselves (section 4.6), Python provides access to a large number of *modules* and *packages* (the equivalent of libraries in other

languages). These modules/packages give us pre-built functionality that can be used to help with a wide range of tasks, including statistics and data management. Some of these come automatically with our Python distribution, but unlike the in-built functions need to be called (imported into the session).

For an example, will use the Python module *random*. As the name suggest, *random* provides random number functionality for our Python programs. We can import the whole of the *random* library with the command:

```
import random
```

This gives us access to all the functionality available. To demo its use, let's build a single program that replicates the popular 1980s British quiz show "Play Your Cards Right" (https://en.wikipedia.org/wiki/Play_Your_Cards_Right).

In this game a player will draw a card from the deck where "2" is the lowest card and "Ace" is the highest (effectively 14). The player then will have to guess if the next card will be higher or lower. We will assume that the player will base this purely on the most logical answer and if the card is between two and seven they will call "higher". If the card is between nine and ace they will call "lower". If the card is eight they will pick either "higher" or "lower" at random. Every time they get it right they win £100 and get to go again. If they get it wrong, or the new card is the same as the existing card they lose and whatever money that have won is their final award. There will be two players and the winner is the one who wins the most money.

Note: we are simplifying the game so that we will ignore that there are actually only four of each number, one for each suit, and assuming there are infinite of each number.

The code looks like this:

```
import random

players = ['Player #1', 'Player #2']
game_continue = True
money_dictionary = {}

def print_name(card_drawn):
        if card_drawn == 11:
                card_drawn_print = "Jack"
        elif card_drawn == 12:
                card_drawn_print = "Queen"
        elif card_drawn == 13:
```

```
                card_drawn_print = "King"
        elif card_drawn == 14:
                card_drawn_print = "Ace"
        else:
                card_drawn_print = str(card_drawn)
        return card_drawn_print

for player in players:
        game_continue = True
        prize = 0
        while game_continue == True:
                card_drawn = random.randint(2, 14)
                print("The card is a " + print_name(card_drawn))
                if card_drawn < 8:
                        guess = "higher"
                elif card_drawn > 8:
                        guess = "lower"
                else:
                        if random.randint(1, 2) == 1:
                                guess = "lower"
                        else:
                                guess = "higher"
                print(player + " guesses " + guess)
                new_card = random.randint(2, 14)
                print("The card is a " + print_name(new_card))
                if new_card > card_drawn and guess == "higher":
                        prize += 100
                        print(player + " wins £100")
                elif new_card < card_drawn and guess == "lower":
                        prize += 100
                        print(player + " wins £100")
                else:
                        print(player + " loses!")
                        print("Total prize money is £" + str(prize))
                        money_dictionary[player] = prize
                        game_continue = False

if money_dictionary['Player #1'] > money_dictionary['Player #2']:
        print("Player #1 wins with a total prize of £" + \
                str(money_dictionary['Player #1']))
elif money_dictionary['Player #1'] < money_dictionary['Player #2']:
        print("Player #2 wins with a total prize of £" + \
                str(money_dictionary['Player #2']))
else:
        print("It's a tie. No-one wins")
```

Much of this should be familiar to but there are a few points to make. Firstly, is our *print_name( )* function. While in the most part this works like the function we created in section 3.7, in this we have introduced a final command of *return card_drawn_print*. This means that the function will return this calculated value when called.

Secondly, when we compare the two cards (*if new card* > *card_drawn*) we include an *and* statement so that both of these conditions need to be true for the test to pass. Thirdly, as detailed in the operators shown in section 3.8, we increase the value of *prize* using the command *prize += 100* (meaning the prize is increased by 100 if the player wins).

The final "new" code in this example relates to the *random* library. At various points we make calls to the *random.randint( )* function. *randint* is a shortened version of "random integer" and, as this suggests, the function will draw a random number between the first value passed and the last value passed. So *random.randint(2, 14)* will draw a random integer between 2 and 14.

We can evaluate that the program works by running it in the notebook. The output will obviously vary (it's based on random numbers), but a possible output is shown in **Figure 94**.

Figure 94: Example Output from our Play Your Cards Right Simulator

```
The card is a 6
Player #1 guesses higher
The card is a 10
Player #1 wins £100
The card is a Ace
Player #1 guesses lower
The card is a 8
Player #1 wins £100
The card is a Ace
Player #1 guesses lower
The card is a 5
Player #1 wins £100
The card is a 3
Player #1 guesses higher
The card is a 3
Player #1 loses!
Total prize money is £300
The card is a Jack
Player #2 guesses lower
The card is a Queen
Player #2 loses!
Total prize money is £0
Player #1 wins with a total prize of £300
```

In this example we have only used one function available in *random* (there are lots of them), the *randint( )* function. If we are not planning on using any other functions we can import the package in a different way:

```
from random import randint
```

The benefit is that rather than having to call *random.randint( )* we can call just *randint( )* like so:

```
new_card = randint(2, 14)
```

We can also import modules with a specific name, such as :

```
from random import randint as num_generator
```

which allows us to call the function as:

```
card_drawn = num_generator(2, 14)
```

We won't repeat the full code here for each of these examples, but they are available in the notebook on Github, titled "c4-notebooks-Modules.ipynb". The full list of modules available in a Python 3.x installation is shown here: https://docs.python.org/3/py-modindex.html.

4.1. Installing with Pip

Alongside the modules which come with our Python installation, we can install further packages – which are basically modules that come from other sources. While we import them into our session in the same way as modules, we need to first install them on to our machine.

There are multiple ways to do this, including directly from binaries/zip files/tar balls, or via the Python plugin *easyinstall*, but by far and away the easiest and most common approach is to use *pip*, which comes pre-installed with this version of Python. The Anaconda distribution provides a really easy way to use *pip* – directly from our notebook.

As a note, installing using pip via other distributions of Python can be a little more complicated. You often will need to install packages as raw system code – via the CMD on a Windows machine or via the terminal on Linux/Mac OS. Typically the command would be something like ***[path-to-Python]/Python3.7 –m pip install [package name]*** *where "path-to-Python" is the folder where Python3.7 lives and "package name" is the package you want to install.*

Let's demo this by installing and using a third-party package into Python. The package we'll use is a slightly obscure, but fun and useful *flashtext* (https://flashtext.readthedocs.io/en/latest/). We can install *flashtext* directly in the notebook window using:

```
!pip install flashtext
```

This will produce an output similar to the below (**Figure 95**):

Figure 95: Installing flashtext in the Notebook Interface

```
In [5]: !pip install flashtext

Collecting flashtext
  Downloading https://files.pythonhosted.org/packages/81/d8/2cd0656eae456d615c2f1efbc
e09/flashtext-2.7.tar.gz
Building wheels for collected packages: flashtext
  Running setup.py bdist_wheel for flashtext: started
  Running setup.py bdist_wheel for flashtext: finished with status 'done'
  Stored in directory: C:\Users\morten_m\AppData\Local\pip\Cache\wheels\37\db\d7\fe74
d81ab6b3d3128c
Successfully built flashtext
Installing collected packages: flashtext
Successfully installed flashtext-2.7

You are using pip version 8.1.2, however version 19.0.3 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.
```

We don't need to read the full text here but we are interested to know that the package installed – to which the line near the bottom "Successfully installed flashtext-2.7" tells us that this is the case. You will also see the red box telling us to upgrade pip to the latest version. Although though not really necessary, and you'll need to do it with an administrator password / access, it does show us another use of pip – for upgrading packages to the latest version. Let's do that for a package that comes with our distribution which we will use later in the book – *numpy* (numerical Python - http://www.numpy.org/). We need slightly different code to that written in red box, because we are using the notebook interface, which will produce an output like in **Figure 96**:

```
!pip install --upgrade numpy
```

Figure 96: Upgrading a Package using pip

```
In [2]: !pip install --upgrade numpy

Collecting numpy
  Downloading https://files.pythonhosted.org/packages/0b/
9c2/numpy-1.16.2-cp35-cp35m-win_amd64.whl (11.9MB)
Installing collected packages: numpy
  Found existing installation: numpy 1.15.4
    Uninstalling numpy-1.15.4:
      Successfully uninstalled numpy-1.15.4
Successfully installed numpy-1.16.2
```

Again, we only really care about the text at the end which says that we have "uninstalled numpy-1.15.4" and have "Successfully installed numpy-1.16.2".

Let's return to *flashtext* and run an example using it (read the documentation for other tutorials and functionalities - https://flashtext.readthedocs.io/en/latest/). We're going to write a simple

program that can find specified in keywords in a block of text. To start with we need to import the package like we did with modules:

```
from flashtext import KeywordProcessor
keyword_processor = KeywordProcessor()
```

Here we have imported the *KeywordProcessor( )* function from the package, and renamed it as *keyword_processor*. This just allows us to use it in a slightly different way where we don't need to use brackets after.

Next we will add some keywords that we are interested in and then process some text. We will use the module outline for *Computational Statistics with Python*, and are interested in the words:

```
keyword_processor.add_keyword('computational', 'Computers')
keyword_processor.add_keyword('computer', 'Computers')
keyword_processor.add_keyword('high performance computing', 'Computers')
keyword_processor.add_keyword('statistics', 'Statistics')
keyword_processor.add_keyword('statistical', 'Statistics')
keyword_processor.add_keyword('python', 'Python')
```

The words we will search for (the first word listed each time) are obviously fairly relevant, and are a combination of single words (called unigrams) and multi-word phrases (called n-grams) – each word/phrase is separated by quote marks as before. We pass them to *flashtext* using the *keyword_processor* variable we created, and the function *add_keyword( )*. After we pass each keyword we give it a category (*Computers, Statistics* or *Python*) which we will use to count occurrences.

We then want to pass the text. Because computers will not consider an uppercase letter as the same as lower case letter, if our text includes "Python" that will not be a match with "python". Therefore, we will also convert our text to lower case using the *lower()* function. The code for this is as follows:

```
text = 'This module aims to introduce students to many of the advanced
statistical techniques made possible by innovations  in computing and
modern processing power. This includes Markov chain Monte Carlo approaches,
probabilistic methods, Bayesian statistics, dimension reduction and high
performance computing'
text = text.lower()
```

This will produce a lower case version of our text as shown in **Figure 97**.

Figure 97: Converting Text to Lower Case using lower()

```
In [9]: text = 'This module aims to introduce students to many of the adva
        text = text.lower()
        text

Out[9]: 'this module aims to introduce students to many of the advanced st
        n computing and modern processing power. this includes markov chai
        yesian statistics, dimension reduction and high performance comput
```

We can see that "Markov" has been converted to "markov" so we can verify this has worked.

Lastly, we will count the keywords that occur (by their category), to produce the output shown in **Figure 98**:

```
keywords_found = keyword_processor.extract_keywords(text)
```

Figure 98: Find Keywords using flashtext

```
In [10]: keywords_found = keyword_processor.extract_keywords(text)
         keywords_found

Out[10]: ['Statistics', 'Statistics', 'Computers']
```

We have found two keywords we associated with *Statistics* (1x "statistical" and 1x "statistics") and one keyword with *Computers* ("high performance computing").

The code for this exercise is available in the Github download, titled "c4-notebooks-Packages.ipynb". If you have time, you can play with some of the other functions in the package (https://flashtext.readthedocs.io/en/latest/).

4.2. Creating Modules

In section 3.6 we created a custom function in Python which prints a customer's orders, the *order_printer( )* function. We discussed the benefits of using this as a function rather than repeating the code behind it in multiple places in our program. However, this only works if the function description is in the same program or notebook as the code which calls it. Often we want to share this function across multiple programs/notebooks. Creating it as a custom module that can be imported into a notebook is a solution to this problem.

Firstly, we'll start a new notebook and copy our functions code in, with one more addition. We will also tell the function to aspect two additional inputs: *customers* (our customer dictionary) and *orders* (our orders dictionary). The code will be as follows:

```
# order printer function to take a customer name, and check
# all their current orders
def order_printer(customer, customers, orders):
        # check the customer is in the dictionary
        if customer in customers:
                # check that the length of the orders value
                # (the # of orders) is > 0
                if len(customers[customer]['orders']) > 0:
                        # print the customer name
                        print('Customer: ' + str(customer))
                        # loop through the customer's orders
                        for order in customers[customer]['orders']:
                                # print each of the values stored.
                                # wrapped in try statements in case the value
                                # is missing
                                try:
                                        print('Date: ' + \
                                                str(orders[order]['date']))
                                        print('Product: ' + \
                                                orders[order]['product'])
                                        print('Quantity: ' + \
                                                orders[order]['quantity'])
                                except:
                                        pass
                        # print a lne break after the customer orders
                        print('\n')
                else:
                        # if the customer has no orders print this to screen
                        print('Customer: ' + str(customer) + ' has no orders')
                        # print a lne break after the customer orders
                        print('\n')
        else:
                # if the customer is not in the customers dictionary
                # print an error message
                print('Customer name not found. Please try again')
```

Once we have done this we will rename the notebook as *customer_order_printer*. Rather than save this file though we will save it as a Python file. To do this navigate to "File > Download as" and select "Python". This will download as a file called "customer_order_printer.py" (".py" indicates that this is a Python file. We need to then move this to the folder where our Notebooks are saved (in my case "C:\Users\morten_m\Documents\CSP"). Now our module is available to us to use in a new program.

To test this we can open a new notebook. We will need to add our data (the two dictionaries to this notebook again to make it available:

```
customers = {'James':{'name':'James Pennington', \
                              'open_orders':1, \
                              'orders':['o10001', 'o10002', 'o10004']
                      }, \
                      'Gareth':{'name':'Gareth Edwards', \
                              'open_orders':0, \
                              'orders':[]
                      }, \
                      'Mark':{'name':'Mark Bonnett', \
                              'open_orders':1, \
                              'orders':['o10003']
                      }, \
                      'Emily':{'name':'Emily Davis', \
                              'open_orders':0, \
                              'orders':[]
                      }
}
```

```
orders = {'o10001':{'date':'2019/01/10', \
                              'product':'Social Media Detector', \
                              'quantity':'1'
               }, \
               'o10002':{'date':'2019/01/13', \
                              'product':'Realistic Man\'s wig', \
                              'quantity':'2' \
               }, \
               'o10003':{'date':'2019/01/14', \
                              'product':'Weather\'s Originals', \
                              'quantity':'10' \
               }, \
               'o10004':{'date':'2019/01/15', \
                              'product':'Brown Shoes', \
                              'quantity':'2' \
               }
}
```

With the data available we need to import the module. The command to this should look familiar:

```
from customer_order_printer import order_printer
```

The *from* part needs to refer to the name of our file (we named the file *customer_order_printer.py*) and the *import* to the name of our custom function (*order_printer*). We can now call, and test, the function in the normal way (only with the name of our *customers* and *orders* dictionaries):

```
order_printer('Mark', customers, orders)
```

This produces the following output (**Figure 99**), confirming that everything works:

Figure 99: Using our Custom Module

```
In [3]: from customer_order_printer import order_printer
        order_printer('Mark', customers, orders)

        Customer: Mark
        Date: 2019/01/14
        Product: Weather's Originals
        Quantity: 10
```

We have now created a custom module that can be called in this fashion from any Python program we write. We just need to ensure that the module code is saved in the same folder that we are currently working in. The code to import and run the module is available as "c4-notebooks-CustomModules.ipynb", and the Python module is "custom_order_printer.py", both in the Github download.

4.3. Exercises

Try the following exercises to test your familiarity with modules and packages. Again, if there are any problems, email me at m.mortenson@warwick.ac.uk.

1. Install *fuzzywuzzy* (silly name – useful and simple library). Work through the examples given in the docs: https://github.com/seatgeek/fuzzywuzzy
2. Take the custom function you created at the end of chapter three. this as a module and call it from another program.

5. Summary and Further Reading

This book has provided some introduction to the basics of programming (many aspects such as if conditions and for loops are common to all programming), to Python, and working (in general) with modules and packages. The second (shorter) book presents more specific content to support the module you are studying. The second book for *Computer Programming* will focus on

materials for building applications, while the second book for *Computational Statistics with Python* focuses on data management and statistical operations.

Although only meant as an introduction, there are a huge number of techniques, packages/libraries and directions of study that you can learn from here, you have now done by far the hardest bit – getting started. Now that you have these fundamentals in place, you should feel confident to download a new package or module and (using the online documentation) be able to start using it. Equally, you should feel much more confident picking up another language (e.g. Java or C), and starting to work through this – yes, the code and language rules change, but the underlying principles remain the same.

In respect to further reading, I will point you back to the original list of resources presented in section two (reproduced here), along with some of the packages we have referenced in the chapters since then:

- **The Python Website**: https://www.python.org/
- **Github**: https://github.com/
- **Sublime Text** (a code editor for all languages): https://www.sublimetext.com/
- **Project Jupyter**: https://jupyter.org/
- **Jupyter on Github**: https://github.com/jupyter
- **PyPi** (the Python Package Index): https://pypi.org/
- **Automate the Boring Stuff** (Python tutorial): https://automatetheboringstuff.com/
- **A Byte of Python** (Python tutorial): https://python.swaroopch.com/
- **Notes by a Nerd** (further, more advanced, Python articles): http://notesbyanerd.com/2017/12/29/essential-reads-for-any-python-programmer/
- **Stack Overflow** (a Q&A site for programming and development questions): https://stackoverflow.com/
- **Pandas:** https://pandas.pydata.org/
- **Numpy:** http://www.numpy.org/
- **Scipy:** https://www.scipy.org/
- **RoboBrowser** (Python screen-scraper): https://robobrowser.readthedocs.io/en/latest/readme.html
- **BeautifulSoup** (Python package to read text data from websites): https://www.crummy.com/software/BeautifulSoup/bs4/doc/

- **Flask** (Python micro-framework for building web apps): http://flask.pocoo.org/
- **Django** (Python website framework): https://www.djangoproject.com/
- **SQLAlchemy** (Python toolkit for interfacing with relational databases): https://www.sqlalchemy.org/.

Thanks for reading and look forward to seeing you on the module!