

Getting Started with the Scientific PYthon Development EnviRonment 4 (SPYDER4) Rev 002 January 2020

Dr Philip Yip

This pdf document is still a work in progress...

The sections on installing Anaconda, getting started with the spyder IDE, core python and numpy, an introduction to object orientated programming and the pandas library should give a good introduction for those who are just getting started.

This pdf document is geared towards beginners in the scientific and engineering fields and looks at the use of spyder 4 open source software.

Home Page for this pdf:

<https://dellwindowsreinstallationguide.com/python/>

Contents

Installation of Python with the Spyder 4 IDE Using the Anaconda Distribution.....	7
Anaconda Installation Windows 10 (Version 1909).....	7
User Accounts	7
Anaconda Installation	7
Updating the Anaconda Installation Using the Anaconda Powershell Prompt	11
Kite Installation	14
Spyder Dependency Check.....	17
Anaconda Installation Linux (Fedora 31/Manjaro 18/Mint 19.3/Ubuntu 19.10)	18
Anaconda Installation with Terminal	19
Updating the Anaconda Installation Using the Terminal.....	22
Launching Spyder from the Terminal.....	23
Kite Installation	24
Spyder Dependency Check.....	25
Spyder Preferences	26
Fundamental Python.....	29
Python as a Basic Calculator	29
Accessing Previous Commands.....	31
Clearing the iPython Console	32
Clearing the Console	32
Restarting the Kernel	34
Variable Assignment	35
Variable Names	35
Numeric Variables.....	36
Deleting Variables	38
Variable Reassignment.....	39
String Variables	44
Boolean Variables	48
Concatenation (+) and Duplication (*).....	49
Working with Python Scripts.....	50
The Script File.....	51
Saving the Script File	52
Running a Script File.....	53
The <code>print</code> Function - Printing Variables to the Console.....	54
Comments.....	56
Section Break	57

Formatted String – Strings and Variables	60
Relative String – Relative File Paths	62
The input Function – Gathering Input from the Console	64
Collections.....	69
Lists	69
Concatenation (+) and Duplication (*).....	70
Indexing.....	71
Slicing	75
List Methods	78
Nested Lists.....	84
Mutability.....	87
Tuples.....	90
Dictionaries	91
Sets.....	98
Single Value Lists, Tuples, Sets and Dictionaries.....	103
Conditional Logic.....	105
Logical Operators	105
Combining Logical Operators.....	106
Logical Operators and the Assignment Operator	109
Inverting Logical Operators.....	110
if, elif (else if), else	111
Nested if, elif, else.....	117
Working with Multiple Python Scripts	118
Creating a Nested Script	118
Creating Custom Functions.....	120
Input Argument.....	123
Positional Input Arguments	125
Keyword Input Arguments.....	126
Positional Arguments and Keyword Input Arguments	128
Document Strings.....	129
Return Statement.....	131
Nested Functions	137
Loops.....	140
While Loop	140
For Loop	143
Numerical Lists.....	152

The NUMeric PYthon Library (NUMPY).....	154
Numbers Decimal (Humans) vs Binary (Computers)	156
Decimal Counting System vs Binary Counting System.....	157
8 Bit Integer Unsigned	163
American Standard Code for Information Interchange (ASCII)	163
Hexadecimal Counting System.....	165
8 Bit Integer Signed.....	166
Bits and Bytes.....	167
16 Bit Integer Unsigned	167
16 Bit Integer Signed.....	167
Scientific Notation.....	168
Significant Figures	170
Float 16 (Half Precision) Float.....	171
Comparison	176
Datatype and Number of Bytes	177
2D Plots	179
Matplotlib – The Plotting Library	179
Line Plot	181
Scatter Plot.....	194
How we Encode Color	196
Spelling of Color	196
Theory of Light	197
Additive Addition of Light	198
[r,g,b] integers or [r,g,b,a] floats.....	199
#rrggbb Hex system	200
Microsoft Office Standard Colors.....	200
Subtractive Addition of Light	203
Matrices	203
Practical Applications of Matrices	203
Lists vs NumPy Arrays	206
Creation of Matrices	212
Element by Element Operations.....	219
Functions for Rapid Array Generation	225
Concatenation of NumPy Arrays.....	229
Sorting Data in Matrices	234
Statistical Functions	248

Element by Element Multiplication vs Array Multiplication	262
Element by Element Division/Interpolation	267
The SClentific PYthon Library (SCIPY).....	279
scipy.linalg.....	279
scipy.stats.....	281
scipy.random.....	282
scipy.interpolate	287
Object Orientated Programming and Classes.....	290
Introduction to the Turtle Library.....	290
Creating an Instance of a Class	293
Geometric Shapes, Geometry, Angles and For Loops	298
Pandas – Python and Data Analysis Library (PANDAS)	309
Attributes	311
Methods.....	313
Indexing a Pandas Series (Column) – Square Bracket Indexing vs Dot Attribute Lookup	318
Creating a New Series (Column)	323
Renaming Series (Column).....	324
Deleting a Series (Column) or Index (Row)	331
Adding an Index (Row).....	333
Sorting Data	334
Filtering	338
Categorical	342
Ordering Categories.....	347
The Cut Function.....	348
Rename Categories	350
Adding Categories.....	351
Selecting by Category.....	353
Grouping by Category	354
Selecting an Index	355
Selecting a Cell	357
Renaming Indexes.....	358
Comma Separated Values (CSV) and Tab Delimited (TXT) Files	360
Reading and Writing to CSV, TXT or XLSX File.....	364
Missing Data.....	377
Operating System Module	384

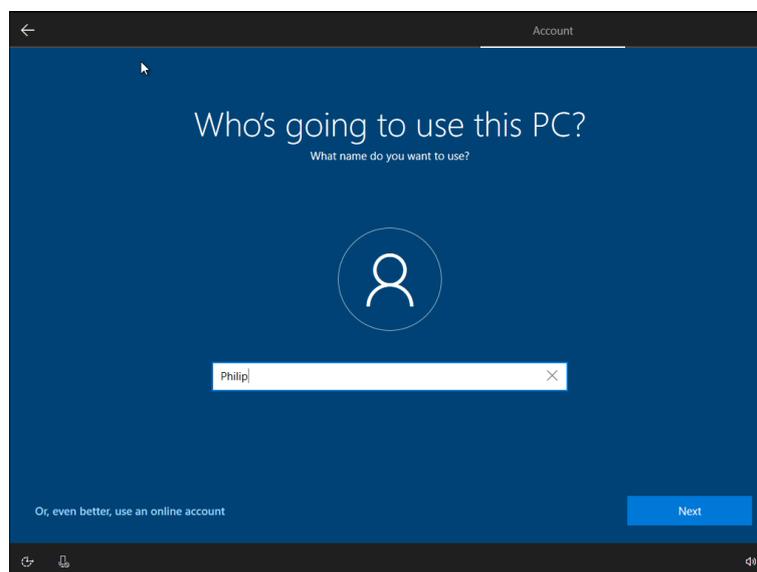
Installation of Python with the Spyder 4 IDE Using the Anaconda Distribution

There are many different Integrated Development Environments for Python each with their own merits and drawbacks. In this book we will use the Spyder IDE which is geared towards the scientific fields. The Spyder IDE is bundled with the Anaconda installer and as it is amongst the easiest distribution to install cross platform on Windows 10, Mac OSX and Linux. In my case I will install it on Windows 10 and Linux Fedora.

Anaconda Installation Windows 10 (Version 1909)

User Accounts

If installing Windows 10 using an Offline Account. It is recommended to use only your first name without any special characters such as spaces as Anaconda may not work properly if installed within a user directory that contains a space.



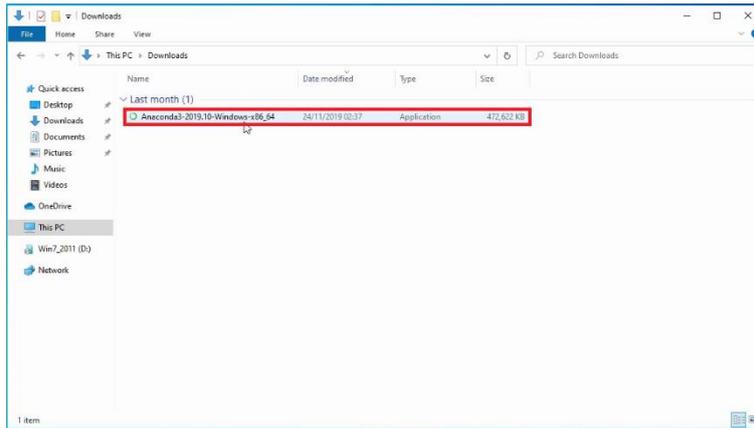
This problem does not occur with a Microsoft Account because it takes the first five characters of your email for the name of your user folder and emails cannot contain spaces.

Anaconda Installation

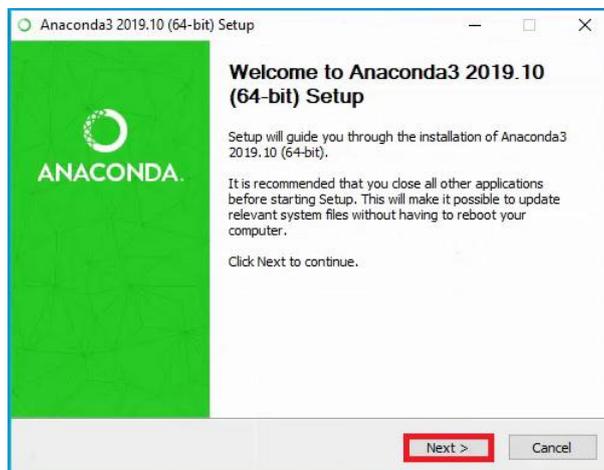
Anaconda is available to download from the Anaconda official website:

<https://www.anaconda.com/distribution/>

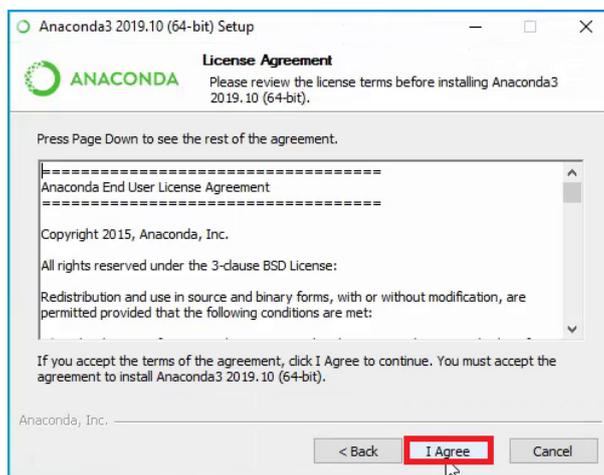
Double click the Anaconda installer:



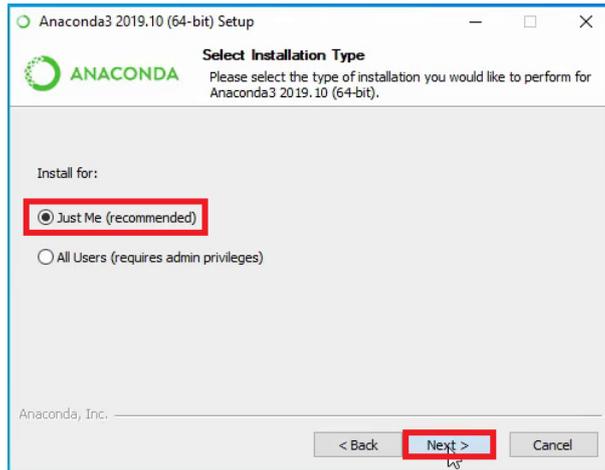
Select Next:



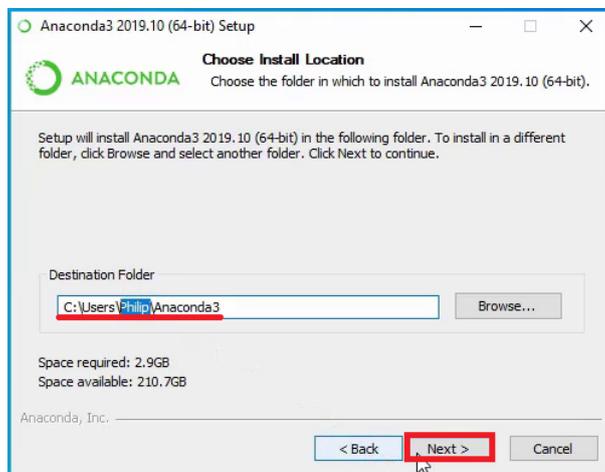
Read and Accept the User Agreement:



In the next screen, select Just Me (recommended) and select Next:



In the next screen you will be prompted for the location to install Anaconda.



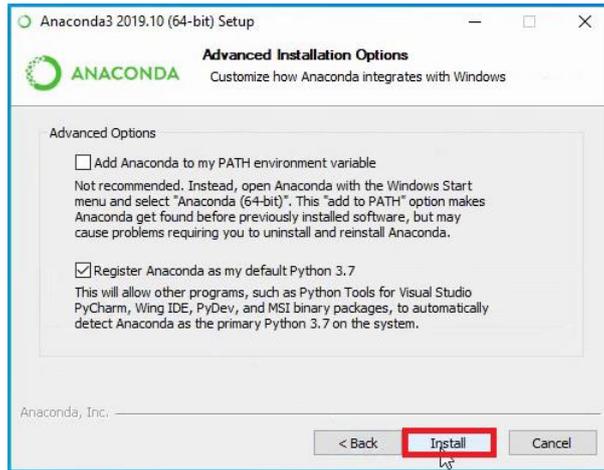
By default, it is within your user account. In my case I am signed in with a Local user account Philip so the installation directory will be:

`C:\Users\Philip\Anaconda3`

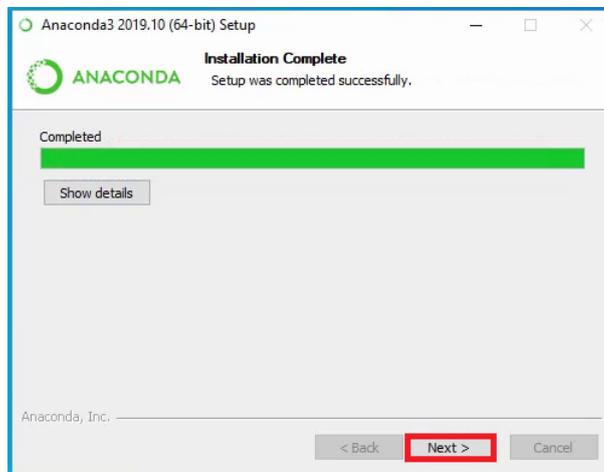
Note if your user account has a space in it then some Python libraries may not load correctly. You will need to specify a location without a space.

If a Microsoft Account is used, typically the first 5 digits of your email will be used and as emails do not contain spaces this shouldn't be a problem.

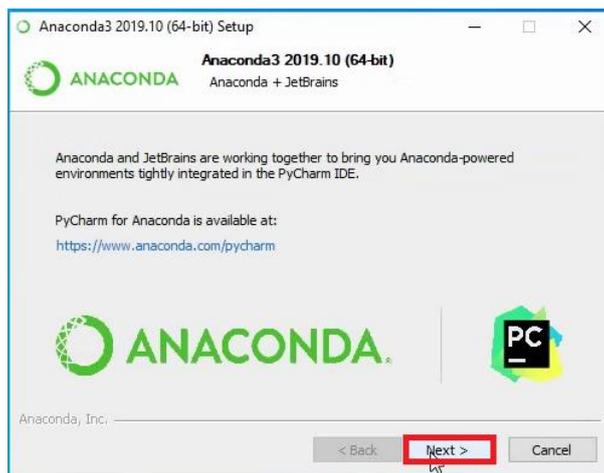
In the next screen you can optionally add Anaconda to my PATH environmental variable and Register Anaconda as your default Python 3.7. Only the second checkbox should be checked by default. It is recommended to leave these settings as be and select Install.



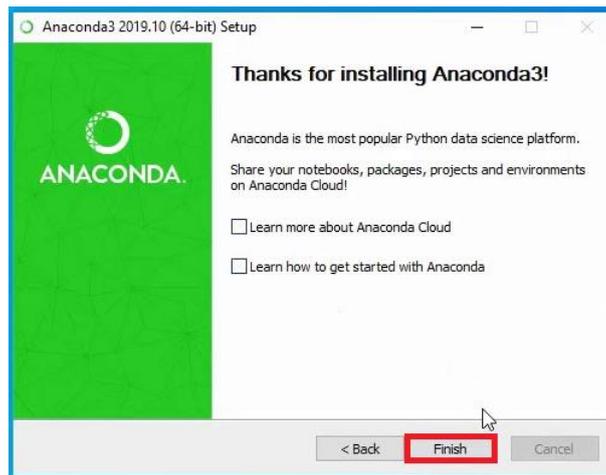
You should be informed that the setup is successfully finished. Select next:



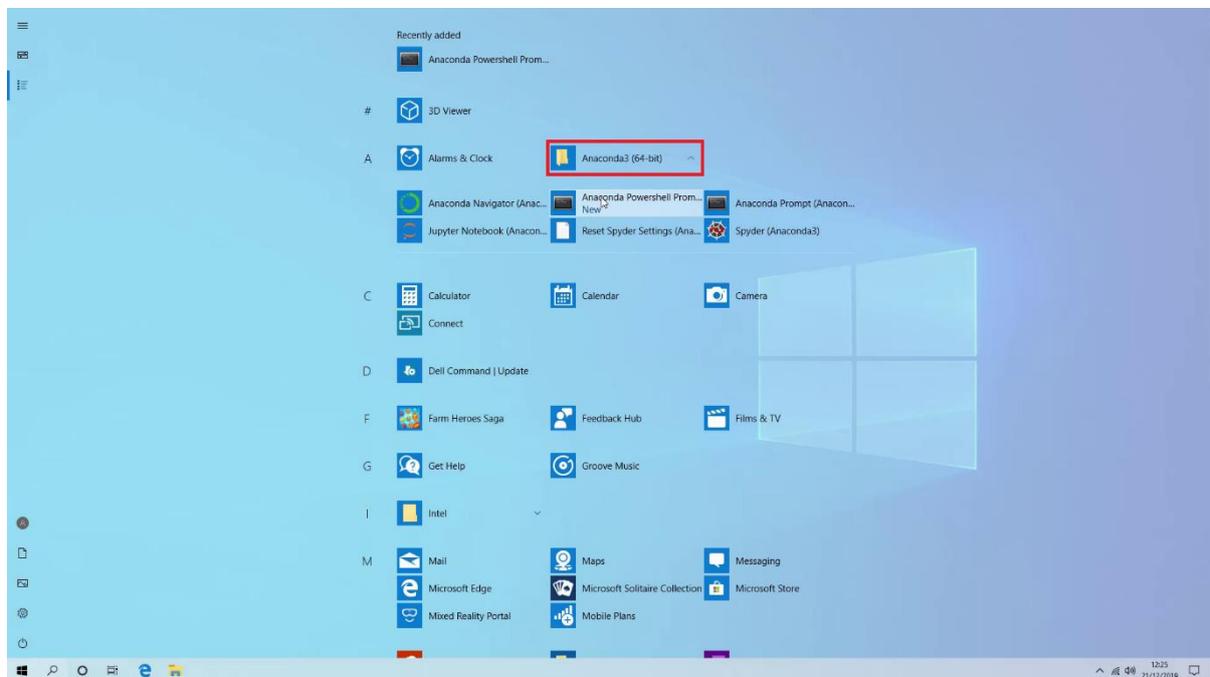
Then next again:



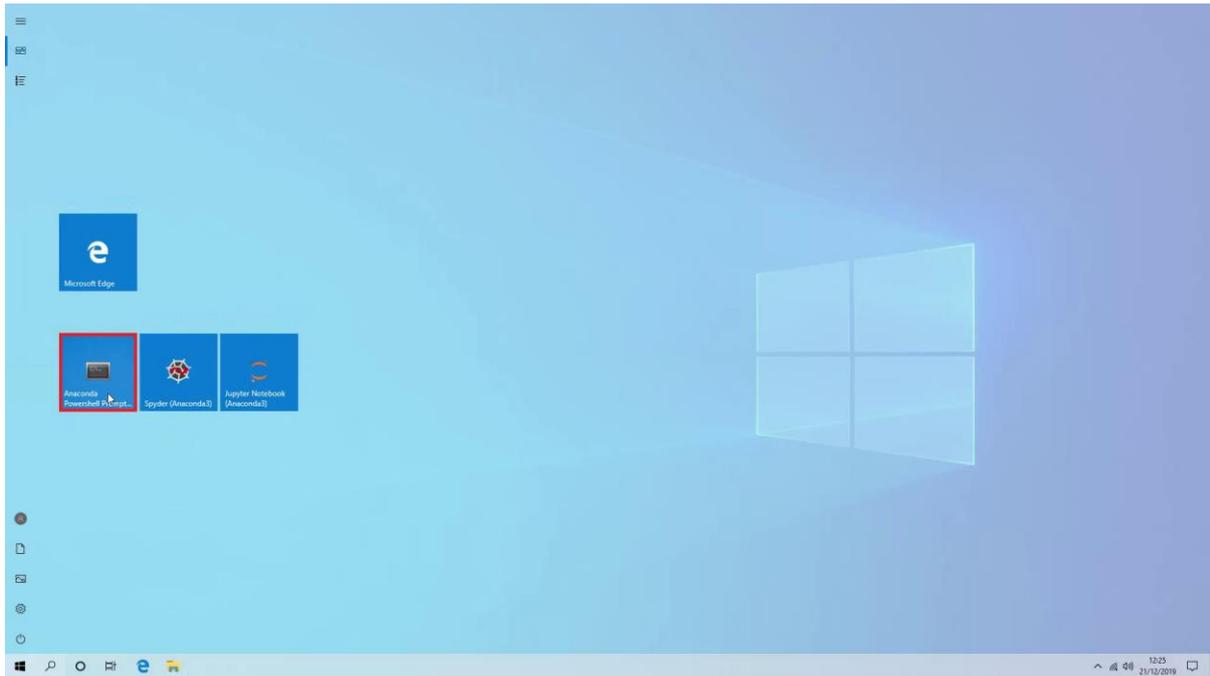
Then Finish:



Anaconda installation should be complete. It is recommended to pin the Anaconda Powershell Prompt, Spyder and Jupyter Notebooks to your start screen.



Updating the Anaconda Installation Using the Anaconda Powershell Prompt
Although we have downloaded the latest standalone installer it is out of date and does not include Spyder 4. We need to update this, to do this select the Anaconda Powershell Prompt:

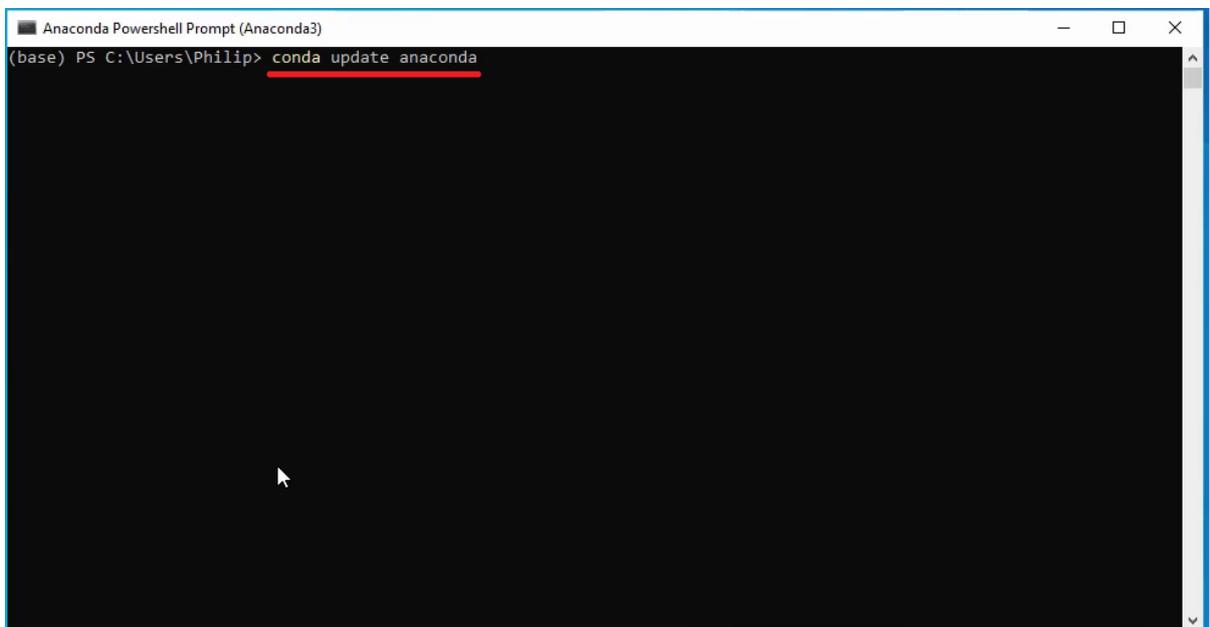


Type in

Type in the following command.

```
conda update anaconda
```

Then press `[Enter]` to execute the above line of code.



When prompted to proceed with the update input `y` followed by `[Enter]` to proceed. Note do not worry about some modules being downgraded the Anaconda will sometimes revert to more stable versions.

```

Anaconda Powershell Prompt (Anaconda3)
scipy                1.3.1-py37h29ff71c_0 --> 1.3.2-py37h29ff71c_0
seaborn              pkgs/main/win-64::seaborn-0.9.0-py37_0 --> pkgs/main/noarch::seaborn-0.9.0-py91ea838_1
setuptools           41.4.0-py37_0 --> 42.0.2-py37_0
six                  1.12.0-py37_0 --> 1.13.0-py37_0
soupsieve            1.9.3-py37_0 --> 1.9.5-py37_0
sphinx               2.2.0-py_0 --> 2.3.0-py_0
spyder               3.3.6-py37_0 --> 4.0.0-py37_0
spyder-kernels       0.5.2-py37_0 --> 1.8.1-py37_0
sqlalchemy           1.3.9-py37he774522_0 --> 1.3.11-py37he774522_0
sqlite               3.30.0-he774522_0 --> 3.30.1-he774522_0
sympy                1.4-py37_0 --> 1.5-py37_0
tblib                1.4.0-py_0 --> 1.6.0-py_0
terminado            0.8.2-py37_0 --> 0.8.3-py37_0
testpath             pkgs/main/win-64::testpath-0.4.2-py37~ --> pkgs/main/noarch::testpath-0.4.4-py_0
tqdm                 4.36.1-py_0 --> 4.40.2-py_0
urllib3              1.24.2-py37_0 --> 1.25.7-py37_0
vs2015_runtime       14.16.27012-hf0eaf9b_0 --> 14.16.27012-hf0eaf9b_1
xlswriter            1.2.1-py_0 --> 1.2.6-py_0
xlwings              0.15.10-py37_0 --> 0.16.3-py37_0

The following packages will be DOWNGRADED:

anaconda              2019.10-py37_0 --> custom-py37_1
jedi                  0.15.1-py37_0 --> 0.14.1-py37_0
jupyter_console      6.0.0-py37_0 --> 5.2.0-py37_1
pycosat              0.6.3-py37ha6e2cd_0 --> 0.6.3-py37he774522_0

Proceed ([y]/n)? y

```

Wait for Anaconda to download and install the new files. Once it has a new command line will show. You can now close the Anaconda Powershell Prompt.

```

Anaconda Powershell Prompt (Anaconda3)
naconda3\cwp.py', 'C:\Users\Philip\Anaconda3', 'C:\Users\Philip\Anaconda3\pythonw.exe', 'C:\Users\Philip\Anaconda3\Scripts\spyder-script.py']
DEBUG menuinst_win32:create(323): Shortcut cmd is C:\Users\Philip\Anaconda3\python.exe, args are ['C:\Users\Philip\Anaconda3\cwp.py', 'C:\Users\Philip\Anaconda3', 'C:\Users\Philip\Anaconda3\python.exe', 'C:\Users\Philip\Anaconda3\Scripts\spyder-script.py', '--reset']
DEBUG menuinst_win32:__init__(199): Menu: name: 'Anaconda${PY_VER} ${PLATFORM}', prefix: 'C:\Users\Philip\Anaconda3', env_name: 'None', mode: 'user', used_mode: 'user'
DEBUG menuinst_win32:create(323): Shortcut cmd is C:\Users\Philip\Anaconda3\python.exe, args are ['C:\Users\Philip\Anaconda3\cwp.py', 'C:\Users\Philip\Anaconda3', 'C:\Users\Philip\Anaconda3\python.exe', 'C:\Users\Philip\Anaconda3\Scripts\jupyter-notebook-script.py', "%USERPROFILE%/"]
DEBUG menuinst_win32:__init__(199): Menu: name: 'Anaconda${PY_VER} ${PLATFORM}', prefix: 'C:\Users\Philip\Anaconda3', env_name: 'None', mode: 'user', used_mode: 'user'
DEBUG menuinst_win32:create(323): Shortcut cmd is C:\Users\Philip\Anaconda3\python.exe, args are ['C:\Users\Philip\Anaconda3\cwp.py', 'C:\Users\Philip\Anaconda3', 'C:\Users\Philip\Anaconda3\python.exe', 'C:\Users\Philip\Anaconda3\Scripts\jupyter-notebook-script.py', "%USERPROFILE%/"]
- DEBUG menuinst_win32:__init__(199): Menu: name: 'Anaconda${PY_VER} ${PLATFORM}', prefix: 'C:\Users\Philip\Anaconda3', env_name: 'None', mode: 'user', used_mode: 'user'
DEBUG menuinst_win32:create(323): Shortcut cmd is C:\Users\Philip\Anaconda3\python.exe, args are ['C:\Users\Philip\Anaconda3\cwp.py', 'C:\Users\Philip\Anaconda3', 'C:\Users\Philip\Anaconda3\python.exe', 'C:\Users\Philip\Anaconda3\Scripts\jupyter-notebook-script.py', "%USERPROFILE%/"]
\ DEBUG menuinst_win32:__init__(199): Menu: name: 'Anaconda${PY_VER} ${PLATFORM}', prefix: 'C:\Users\Philip\Anaconda3', env_name: 'None', mode: 'user', used_mode: 'user'
DEBUG menuinst_win32:create(323): Shortcut cmd is C:\Users\Philip\Anaconda3\pythonw.exe, args are ['C:\Users\Philip\Anaconda3\cwp.py', 'C:\Users\Philip\Anaconda3', 'C:\Users\Philip\Anaconda3\pythonw.exe', 'C:\Users\Philip\Anaconda3\Scripts\spyder-script.py']
DEBUG menuinst_win32:create(323): Shortcut cmd is C:\Users\Philip\Anaconda3\python.exe, args are ['C:\Users\Philip\Anaconda3\cwp.py', 'C:\Users\Philip\Anaconda3', 'C:\Users\Philip\Anaconda3\python.exe', 'C:\Users\Philip\Anaconda3\Scripts\spyder-script.py', '--reset']
done
(base) PS C:\Users\Philip>

```

Note with an Anaconda installation is **not** recommended to use any commands of the form:

```
pip install module
```

Instead use:

```
conda install module
```

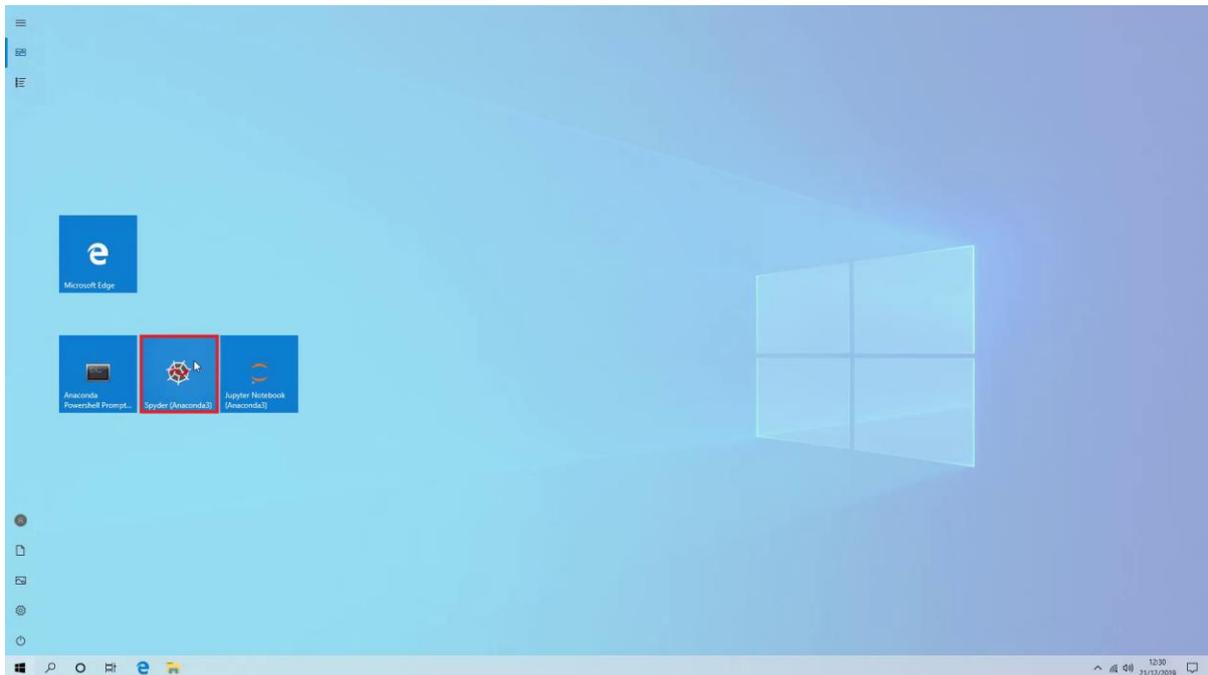
Using `pip install` commands may break the Anaconda installation.

Note if you are not offered Spyder 4, type in:

```
conda install spyder=4.0.1
```

Kite Installation

Launch Spyder.



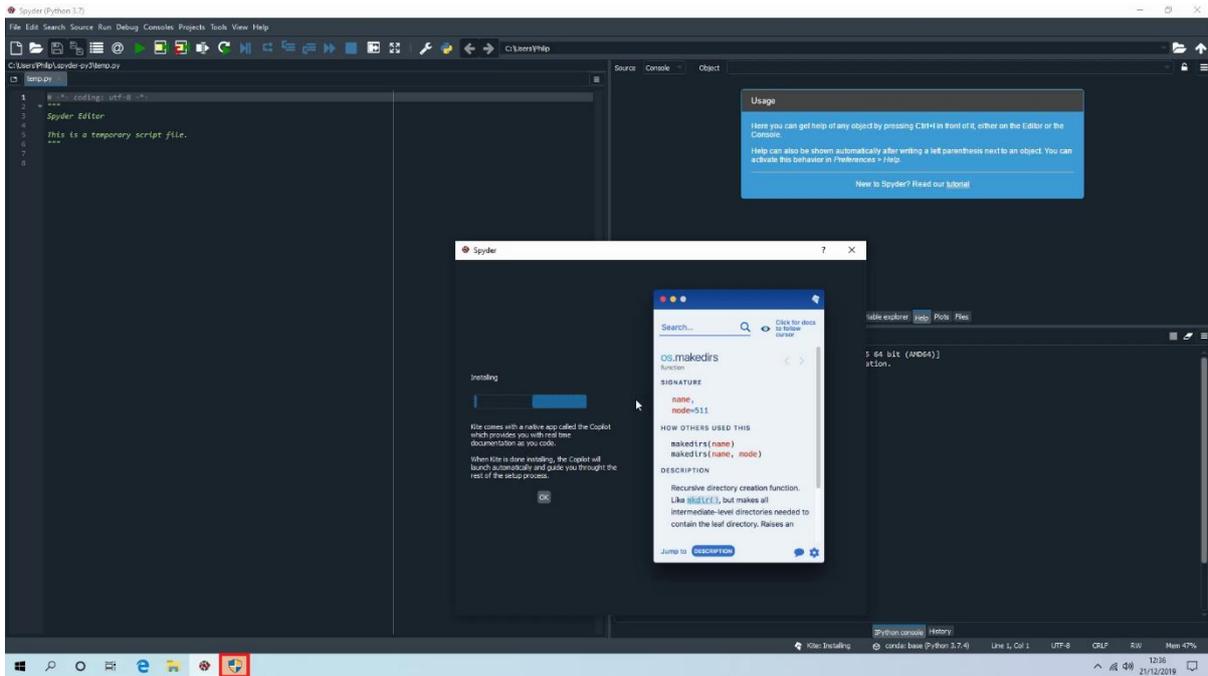
During the first launch of Spyder you will be prompted to install Kite. Select Install Kite.



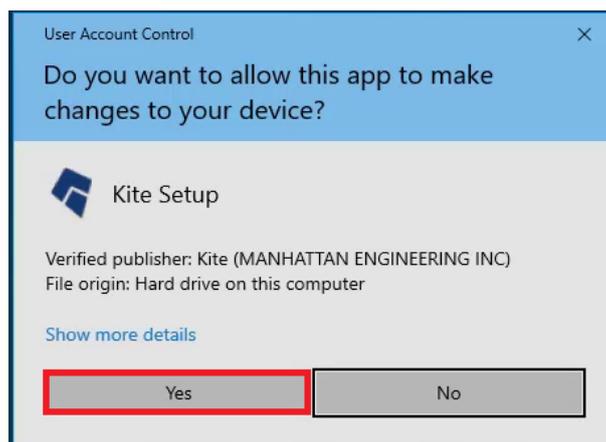
If you get an error message download and install in manually from:

<https://kite.com/download/>

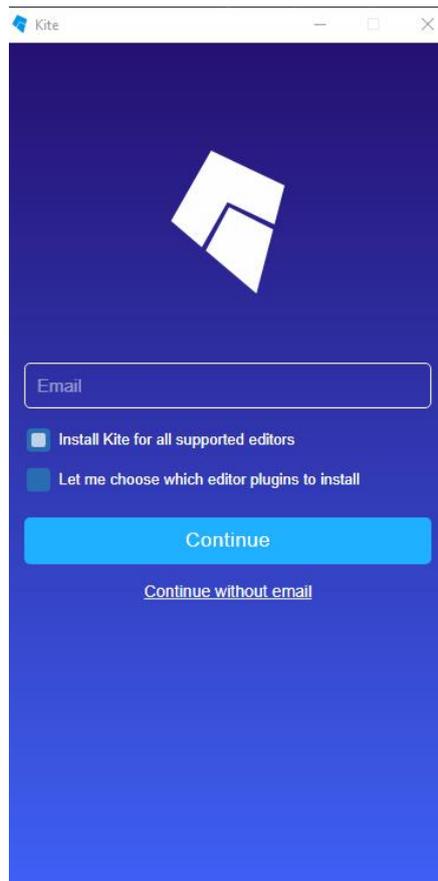
The installation of Kite requires administrator privileges and will hang until the user account control prompt is selected.



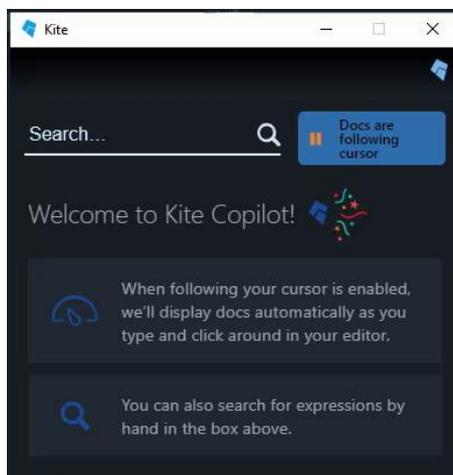
Accept this user account control prompt.



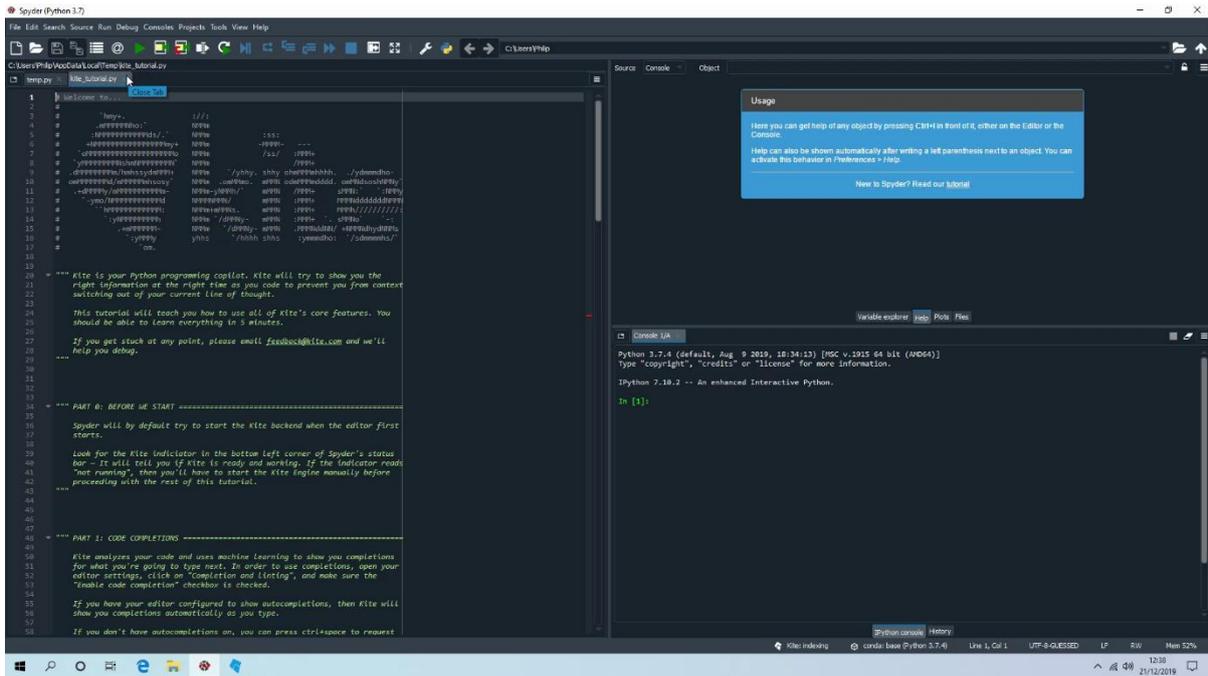
Input your details and select Continue or select Continue without Email.



Kite will be installed.

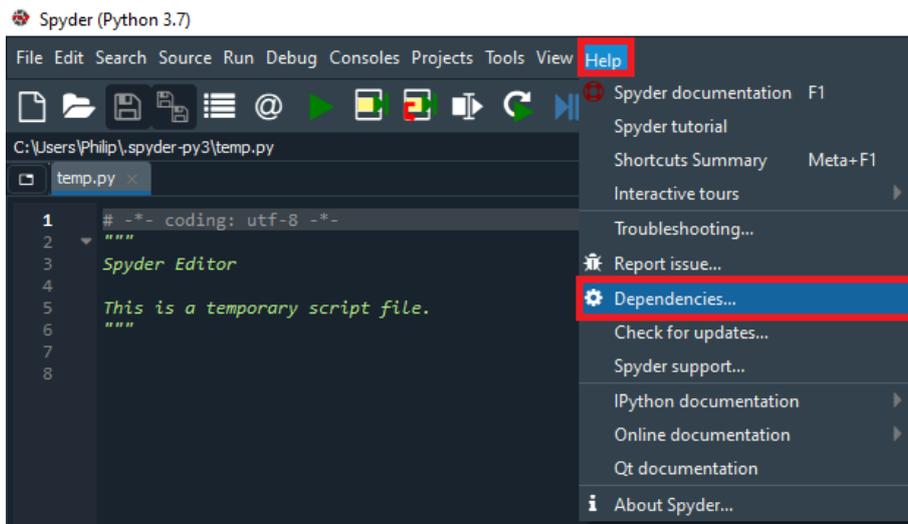


A Kite welcome script will display:

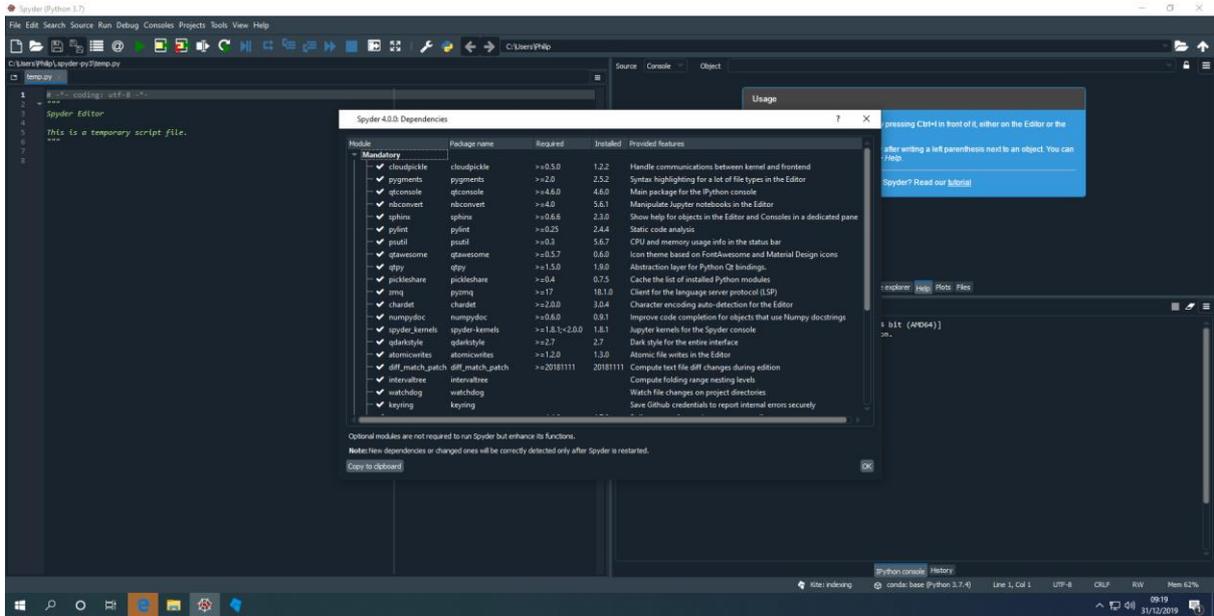


Spyder Dependency Check

It is advised to check the Dependencies before using Spyder. To do this select Help then Dependencies:

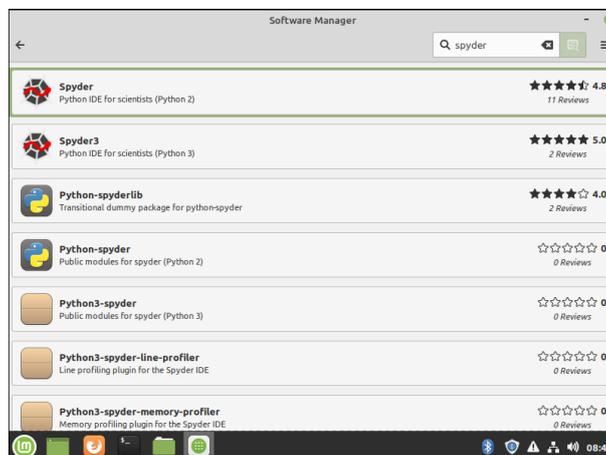
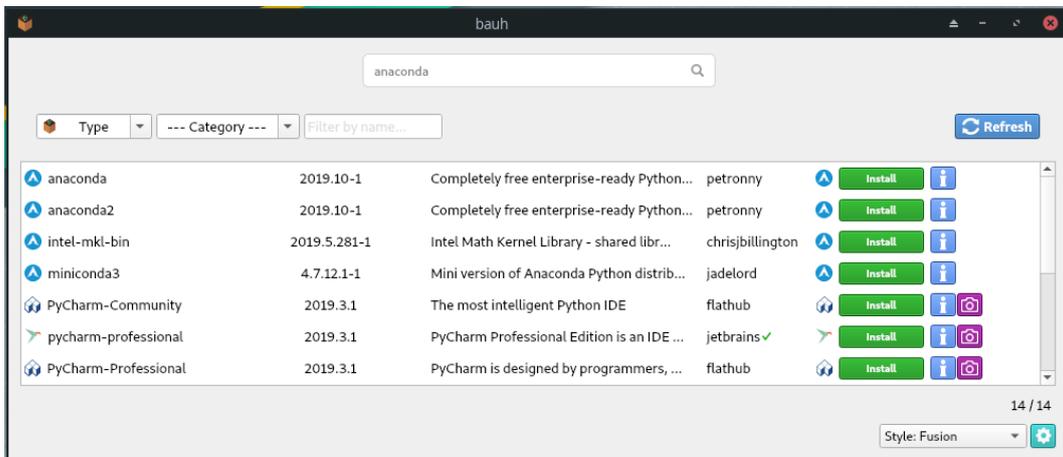


All Mandatory and Optional dependencies should be checked.



Anaconda Installation Linux (Fedora 31/Manjaro 18/Mint 19.3/Ubuntu 19.10)

Note many of the Linux distributions such as Manjaro 18/Mint 19.3 have Spyder and Anaconda listed in the Software/Applications Manager. These either do not work or fail to install, or in the case of Spyder, install an older version Spyder 3 without all the required dependencies. It is therefore recommended to avoid using the Applications Manager to install Anaconda. I have tested the installation on the four above distributions using screenshots from Fedora 31 as an example.

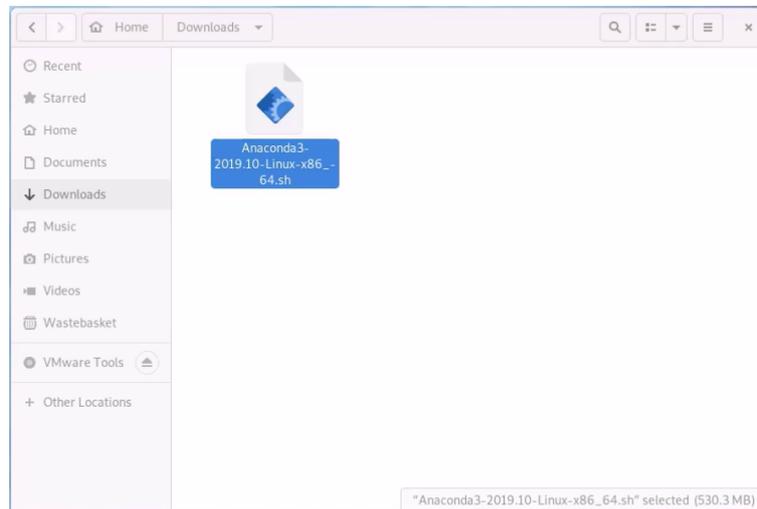


Anaconda Installation with Terminal

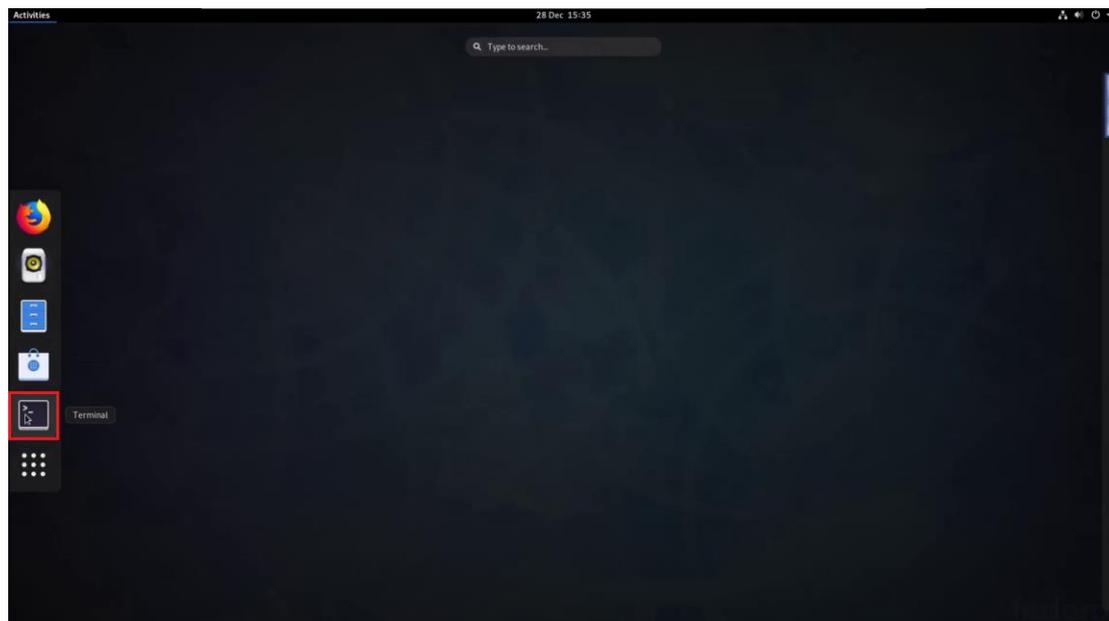
Anaconda is available to download from the Anaconda official website:

<https://www.anaconda.com/distribution/>

In Downloads you will have a `Anaconda3-2019.10-Linux-x86_64.sh` file (the date may be newer if a newer version is available).



To install it you will need to use the terminal.



You will need to change the directory to your downloads folder. Type in:

```
cd Downloads
```

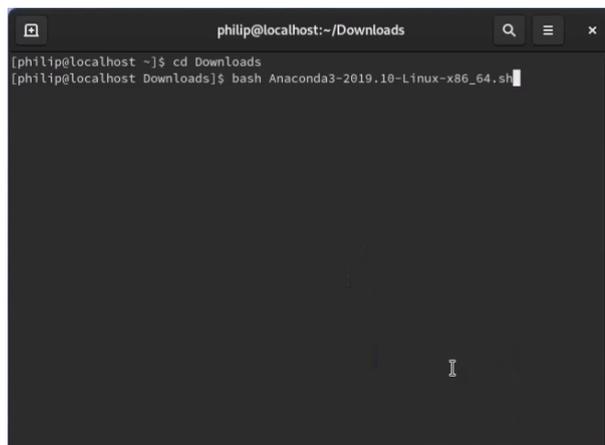


```
philip@localhost:~  
[philip@localhost ~]$ cd Downloads
```

Now type in

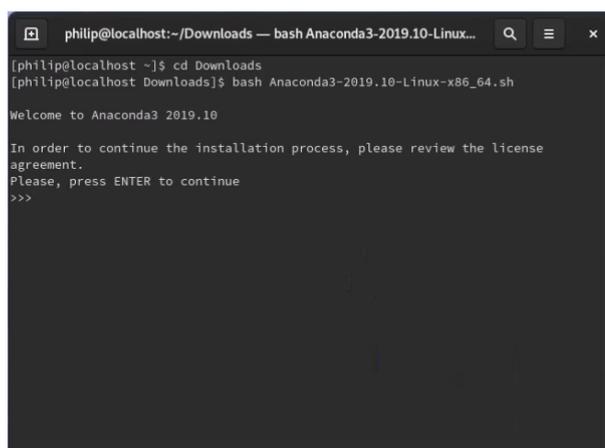
```
bash Anaconda3-2019.10-Linux-x86_64.sh
```

If your file has a newer date use that instead.



```
philip@localhost:~/Downloads  
[philip@localhost ~]$ cd Downloads  
[philip@localhost Downloads]$ bash Anaconda3-2019.10-Linux-x86_64.sh
```

Press {Enter} multiple times to scroll through the license agreement:



```
philip@localhost:~/Downloads — bash Anaconda3-2019.10-Linux...  
[philip@localhost ~]$ cd Downloads  
[philip@localhost Downloads]$ bash Anaconda3-2019.10-Linux-x86_64.sh  
  
Welcome to Anaconda3 2019.10  
  
In order to continue the installation process, please review the license  
agreement.  
Please, press ENTER to continue  
>>>
```

Type in **yes** to accept the agreement and continue:


```
philip@localhost:~/Downloads
no change /home/philip/anaconda3/shell/condabin/conda-hook.ps1
no change /home/philip/anaconda3/lib/python3.7/site-packages/xontrib/conda.x
sh
no change /home/philip/anaconda3/etc/profile.d/conda.csh
modified /home/philip/.bashrc

==> For changes to take effect, close and re-open your current shell. <==

If you'd prefer that conda's base environment not be activated on startup,
set the auto_activate_base parameter to false:

conda config --set auto_activate_base false

Thank you for installing Anaconda3!

=====

Anaconda and JetBrains are working together to bring you Anaconda-powered
environments tightly integrated in the PyCharm IDE.

PyCharm for Anaconda is available at:
https://www.anaconda.com/pycharm

[philip@localhost Downloads]$
```

Updating the Anaconda Installation Using the Terminal

Although we have downloaded the latest standalone installer it is out of date and does not include Spyder 4. We need to update this.

If you are in the same terminal type in:

```
cd
```

This will take you home. Alternatively, just open up a new terminal.

Type in:

```
conda update anaconda
```

```
philip@localhost:~
(base) [philip@localhost ~]$ conda update anaconda
```

Type in **y**. Note do not worry about some modules being downgraded the Anaconda will sometimes revert to more stable versions.

```

philip@localhost:~ — /home/philip/anaconda3/bin/python /home/...
47c_0
sqlite 3.30.0-h7b6447c_0 --> 3.30.1-h7b6447c_0
0
sympy 1.4-py37_0 --> 1.5-py37_0
tbb 2019.4-hfd86e86_0 --> 2019.8-hfd86e86_0
0
tblib 1.4.0-py_0 --> 1.6.0-py_0
terminado 0.8.2-py37_0 --> 0.8.3-py37_0
testpath pkgs/main/linux-64::testpath-0.4.2-py~ --> pkgs/main/noarch
::testpath-0.4.4-py_0
tqdm 4.36.1-py_0 --> 4.40.2-py_0
urllib3 1.24.2-py37_0 --> 1.25.7-py37_0
wurlitzer 1.0.3-py37_0 --> 2.0.0-py37_0
xlswriter 1.2.1-py_0 --> 1.2.6-py_0

The following packages will be DOWNGRADED:

anaconda 2019.10-py37_0 --> custom-py37_1
jedi 0.15.1-py37_0 --> 0.14.1-py37_0
pycosat 0.6.3-py37h14c3975_0 --> 0.6.3-py37h7b644
7c_0

Proceed ([y]/n)? y

```

Your Anaconda installation will now be up to date.

```

philip@localhost:~
pandas-0.25.3 | 8.8 MB | ##### | 100%
yapf-0.28.0 | 120 KB | ##### | 100%
setuptools-42.0.2 | 518 KB | ##### | 100%
contextlib2-0.6.0.py | 16 KB | ##### | 100%
pysistent-0.15.6 | 93 KB | ##### | 100%
watchdog-0.9.0 | 93 KB | ##### | 100%
autopep8-1.4.4 | 41 KB | ##### | 100%
nbconvert-5.6.1 | 459 KB | ##### | 100%
sqlalchemy-1.3.11 | 1.4 MB | ##### | 100%
bokeh-1.4.0 | 8.2 MB | ##### | 100%
urllib3-1.25.7 | 169 KB | ##### | 100%
glib-2.63.1 | 2.9 MB | ##### | 100%
pytables-3.6.1 | 1.2 MB | ##### | 100%
spyder-kernels-1.8.1 | 92 KB | ##### | 100%
lxml-4.4.2 | 1.4 MB | ##### | 100%
python-language-serv | 75 KB | ##### | 100%
pytest-5.3.2 | 370 KB | ##### | 100%
bitarray-1.2.0 | 82 KB | ##### | 100%
tqdm-4.40.2 | 53 KB | ##### | 100%
fsspec-0.6.2 | 53 KB | ##### | 100%
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
(base) [philip@localhost ~]$

```

Note with an Anaconda installation is recommended to avoid any commands of the form:

```
pip install module
```

And instead use:

```
conda install module
```

Using `pip install` commands may break the Anaconda installation.

Note if you are not offered Spyder 4, type in:

```
conda install spyder=4.0.1
```

Launching Spyder from the Terminal

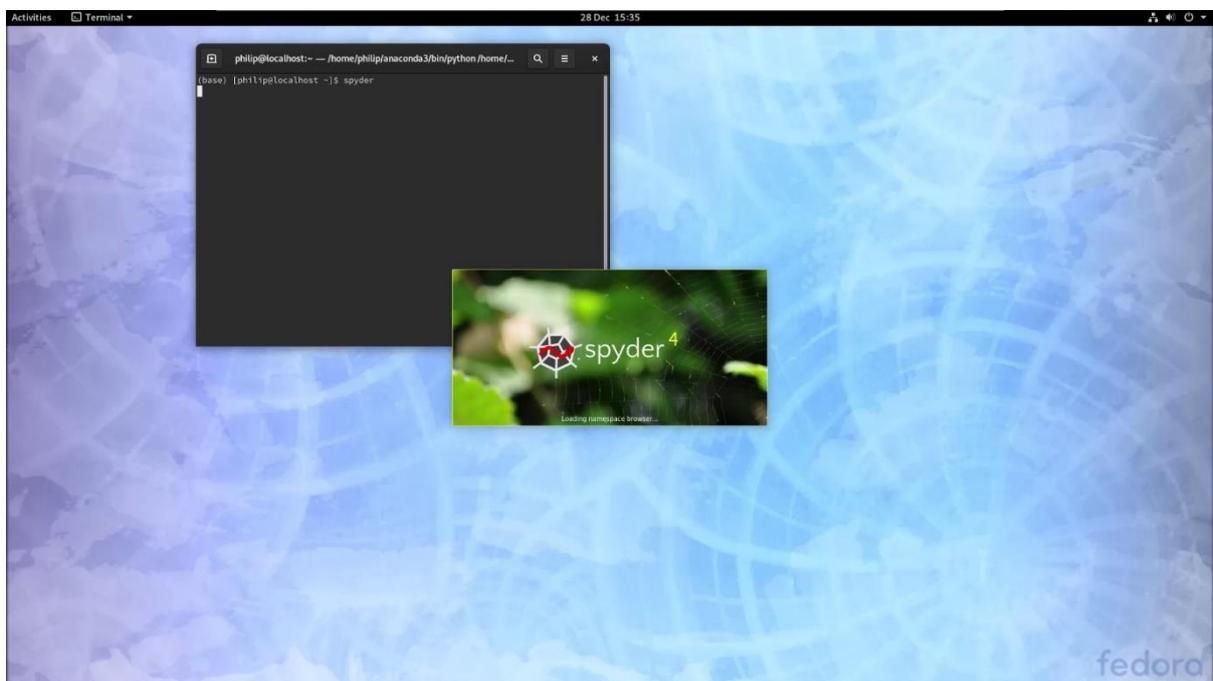
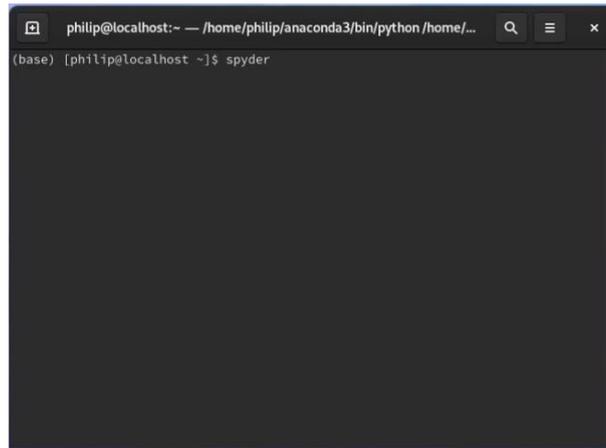
On Linux you will get no shortcuts to launch Spyder 4 or Jupyter notebooks. You must launch these programs via the terminal. If the terminal is open type in

```
cd
```

To return home. Otherwise open a new terminal.

To launch Spyder type in:

```
spyder
```



Likewise, although we won't cover this IDE further, to launch Jupyter Notebooks type in:

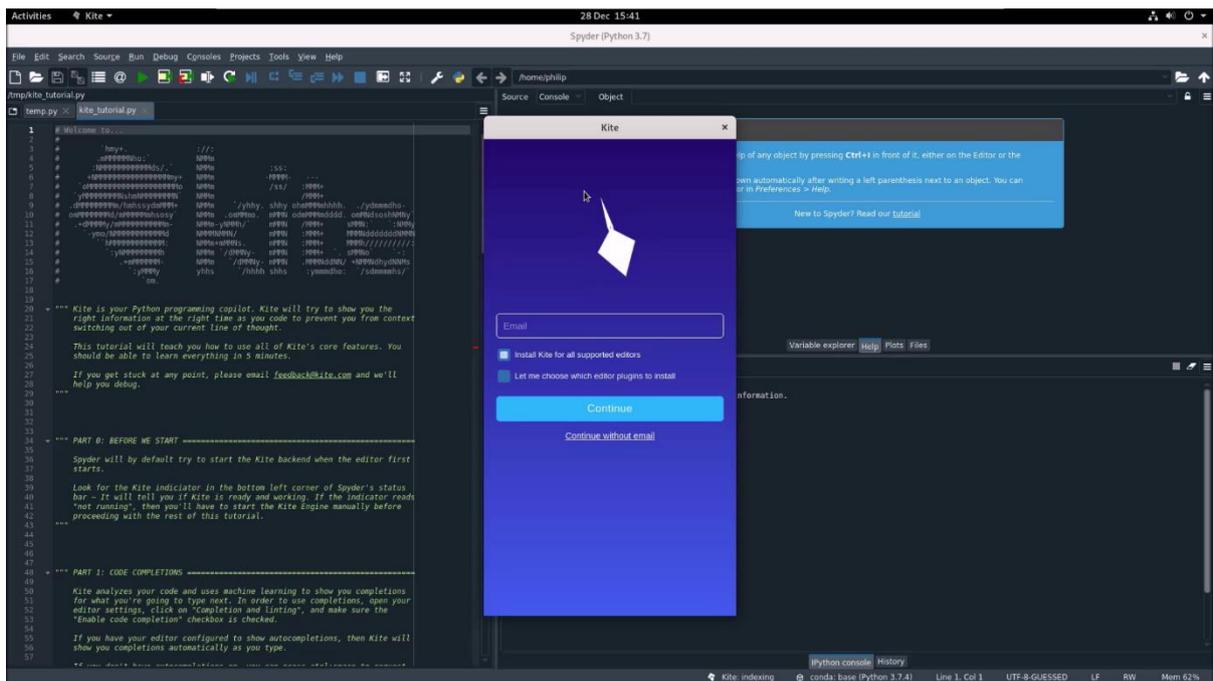
```
jupyter notebook
```

Kite Installation

During the first launch of Spyder 4 you will be prompted to install Kite. Select Install Kite and wait for the install to finish:

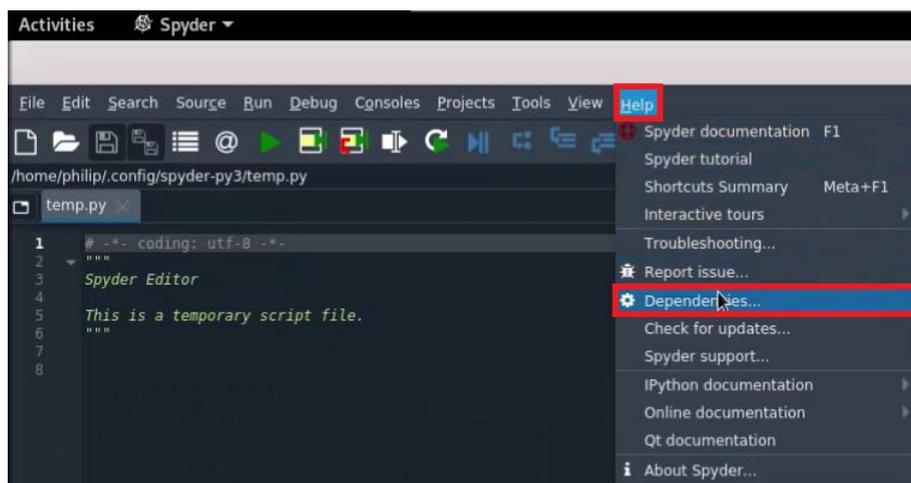


Input your details and select Continue or select Continue without Email.

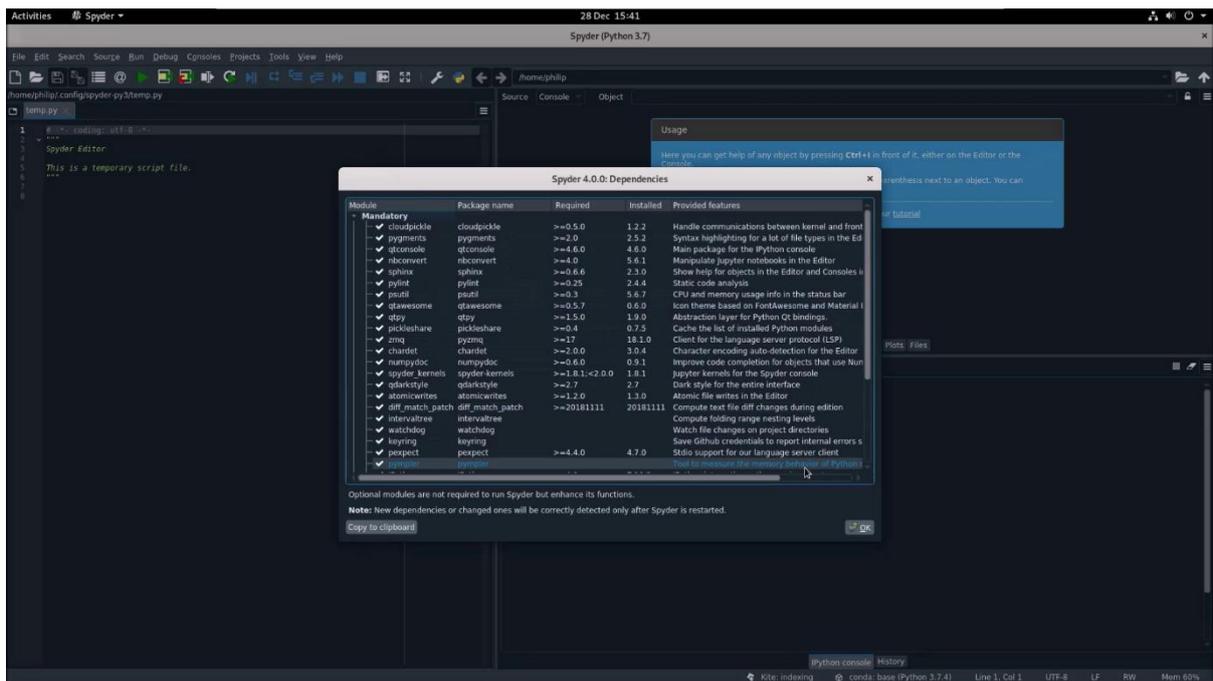


Spyder Dependency Check

It is advised to check the Dependencies before using Spyder. To do this select Help then Dependencies:

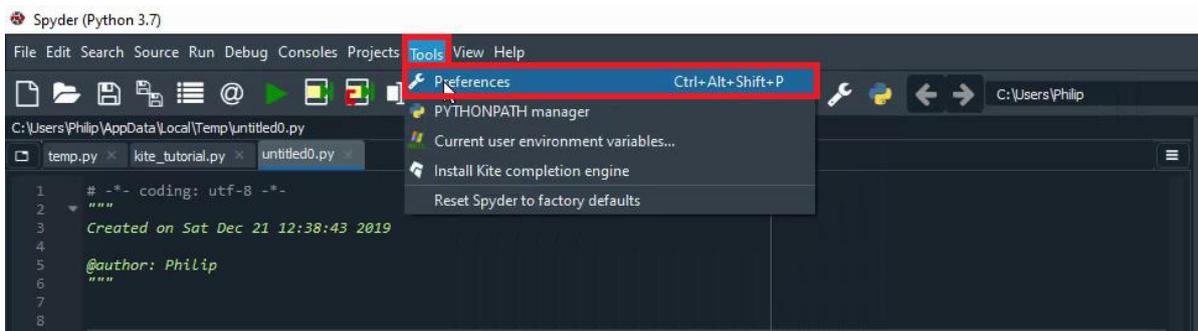


All Mandatory and Optional dependencies should be checked.

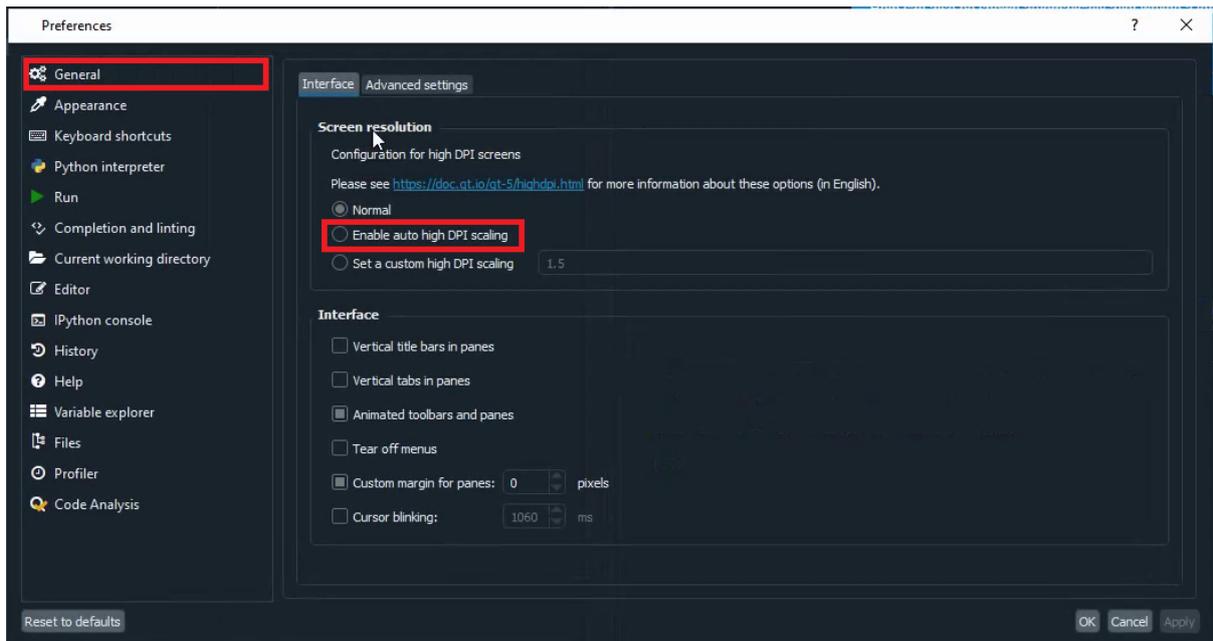


Spyder Preferences

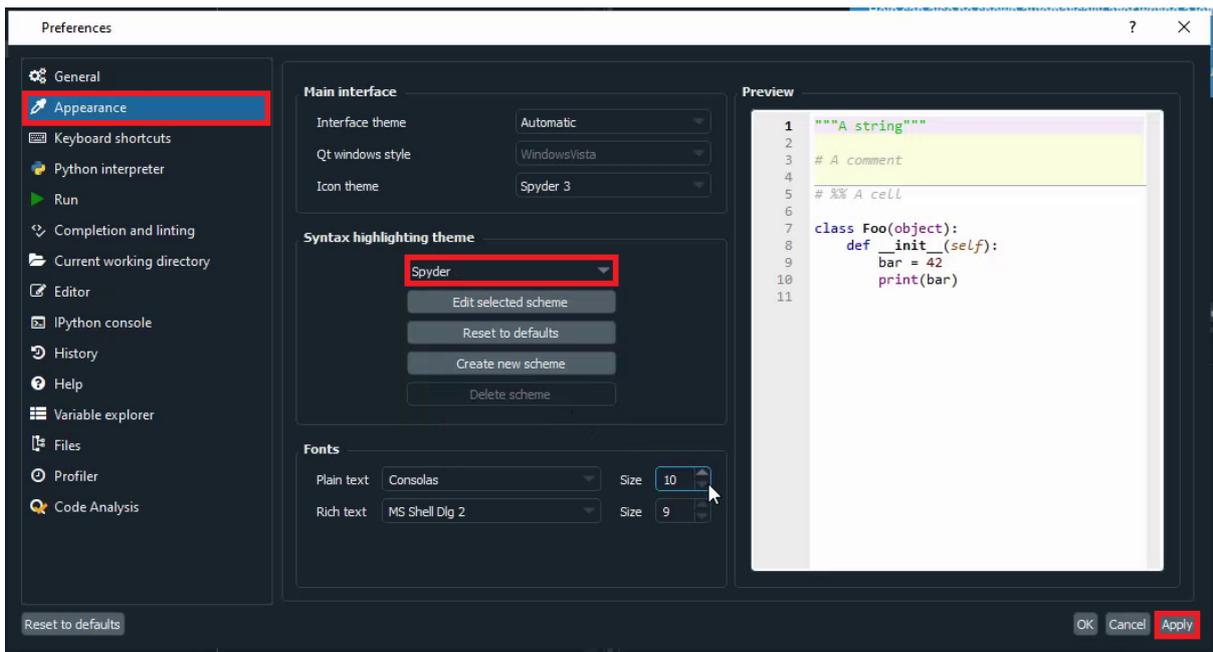
Select Tools and then Preferences:



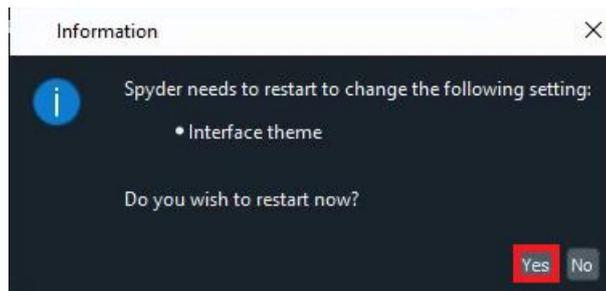
If you have a high resolution screen like my XPS 13 9365 does, in the general tab select Enable auto high DPI scaling.



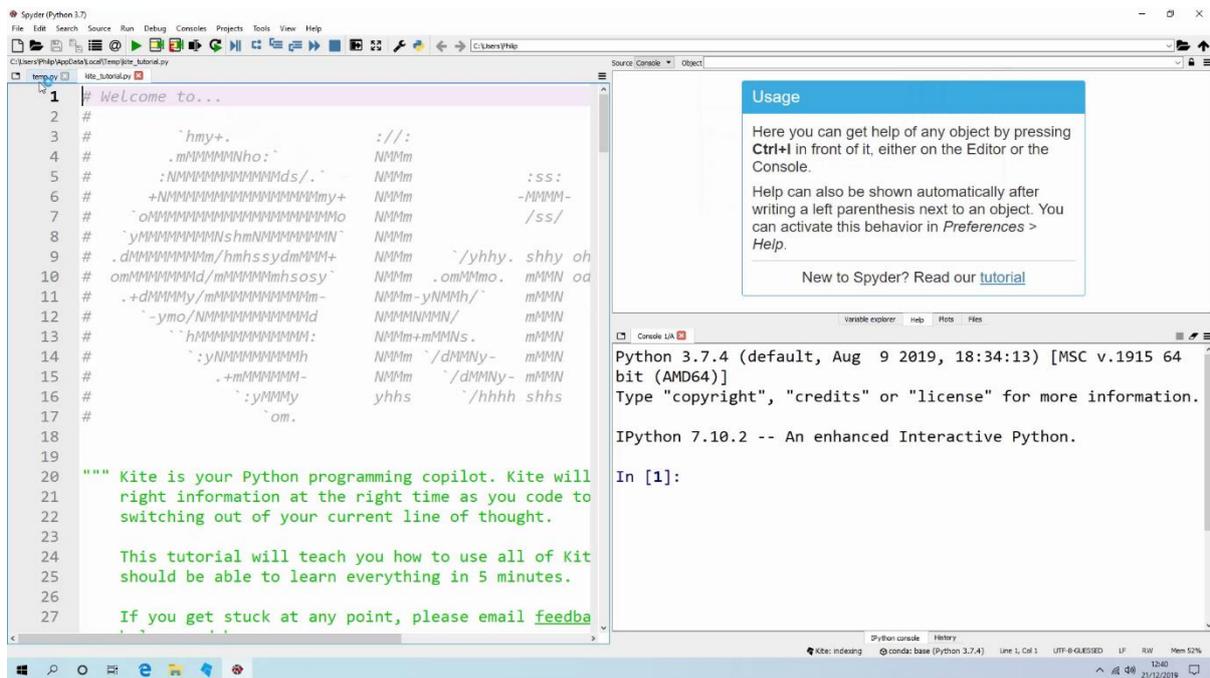
If you wish to use the Spyder light theme as I will use in this book then go to the Appearance tab and select Spyder opposed to Spyder dark under Syntax highlighting theme. Then select Apply:



You will be prompted to restart Spyder.



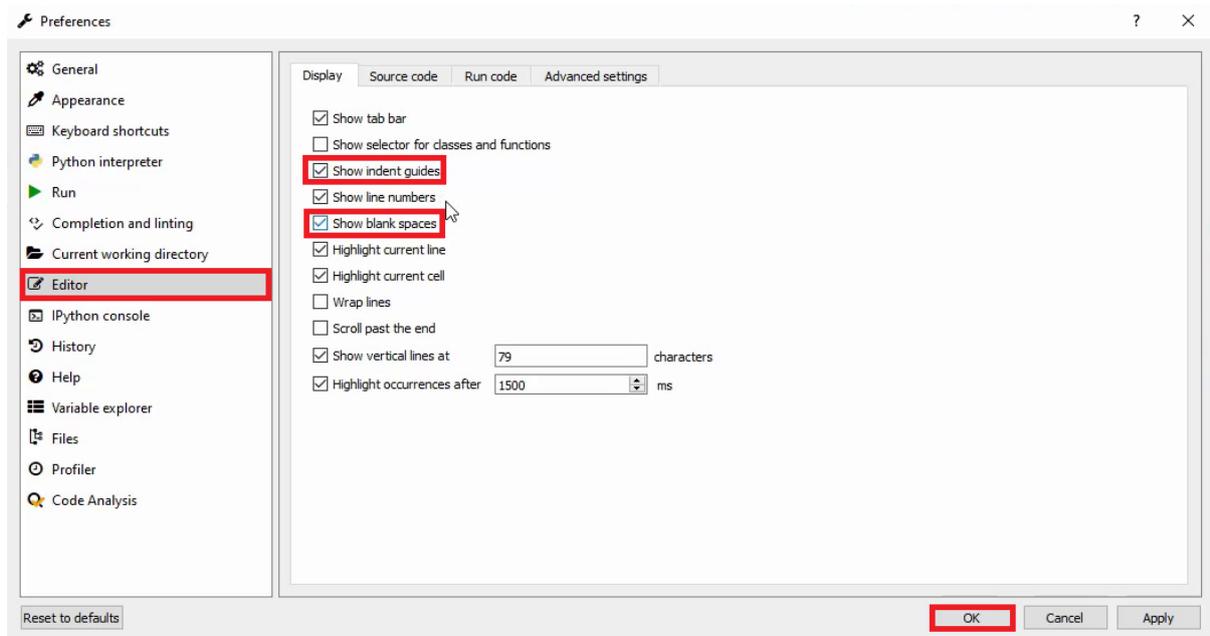
The Spyder light theme will now be in use:



Return to Tools and Preferences:



In the Editor settings you can optionally select show indent guides and show blank spaces:



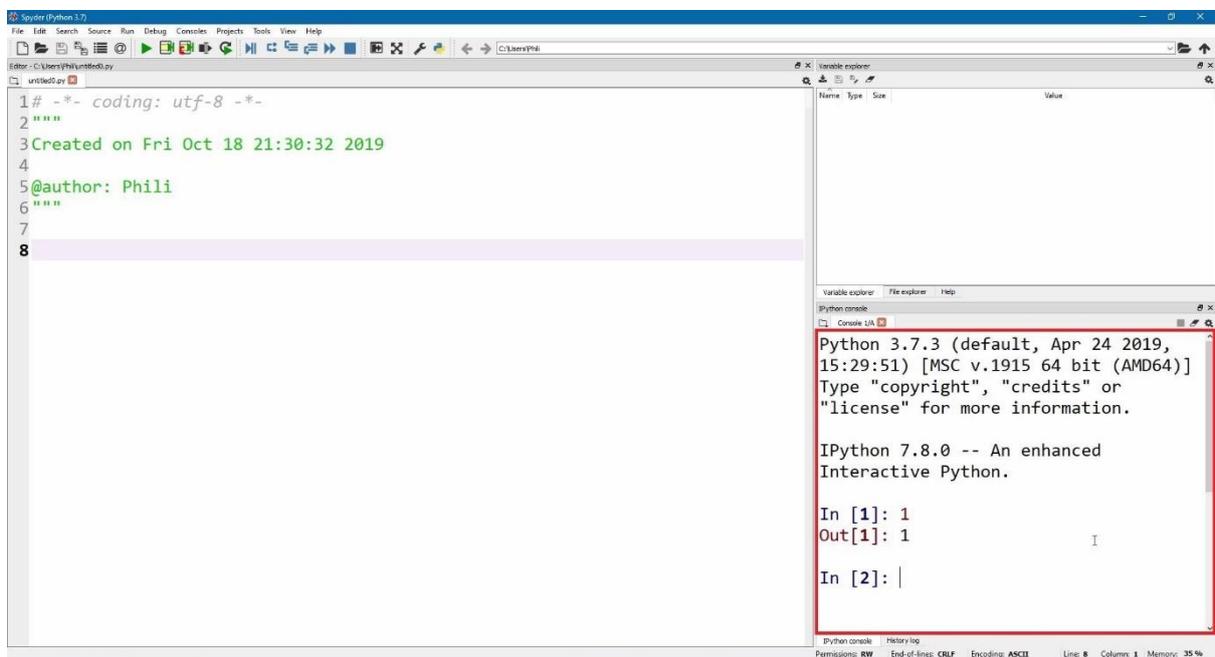
Fundamental Python

Python as a Basic Calculator

In this section we will use Python as a very basic calculator. Important symbols are

Key	Description
0, 1, 2, 3, 4, 5, 6, 7, 8, 9	the numbers
.	the decimal point
+	addition operator
-	subtraction operator
*	multiplication operator
**	exponentiation operator
/	division - returning a float
//	division - returning only a complete integer
%	division - returning the remainder as an integer
()	parenthesis – do the operation in the brackets first

Let's start with the Python Console to type in single lines of code to use Python as a basic calculator:



```
1 # -*- coding: utf-8 -*-
2
3 Created on Fri Oct 18 21:30:32 2019
4
5 @author: Phili
6
7
8
```

```
Python 3.7.3 (default, Apr 24 2019,
15:29:51) [MSC v.1915 64 bit (AMD64)]
Type "copyright", "credits" or
"license" for more information.

IPython 7.8.0 -- An enhanced
Interactive Python.

In [1]: 1
Out[1]: 1

In [2]: |
```

Like an ordinary calculator we can type numbers into the iPython console, to finish a line of code, in this case just the number 1 press [Enter]. Typing a number alone will just return the number as shown above. We can type in integer numbers (numbers which are whole and have no decimal point) abbreviated int for short.

1

We can also type in numbers which contain a decimal point, these are known as floating point numbers or floats for short. Note we use the full stop *i. d* to indicate the decimal point where *i* is the integer and *d* is the decimal.

```
2.03
```

Often, we are dealing with very either very large numbers such as 3×10^8 , which is the speed of light or very small numbers such as 6.63×10^{-34} which is the Planck's constant. To do so we use scientific notation. To denote an exponent $n \times 10^p$ we use the letter *nep* so for the speed of light and Planck's constant we can use:

```
3e8  
6.63e-34
```

We can also use a number of operators:

`+` for addition. For example:

```
2+4
```

This will return the scalar integer of value `6`. If one or both of the numbers had been a float it would have returned a float for example:

```
2.0+4
```

Will return the floating point number (number with a decimal point) of value `6.0`.

`-` for subtraction. For example:

```
2-4
```

Will return the scalar integer of value `-2`.

`*` for multiplication. For example:

```
4*2
```

Will return the scalar integer of value `8`.

`/` for float division. For example:

```
9/2
```

Will return the float of value `4.5`.

`//` for integer division. For example:

```
9//2
```

Will return only complete integers. In this case the value `4`.

`%` to return the integer remainder for integer division known as the quotient.

```
9%2
```

Will return the remainder `1`.

`**` for exponentiation. For example:

```
3**2
```

computes 3^2 which returns the integer value of 9. This can also be used to calculate the nth root for example:

```
9**0.5
```

computes $\sqrt{9} = 9^{0.5}$ which returns a float of 3.0. Note a float is returned as 0.5 is a float.

() is used for parenthesis. For example compare the values:

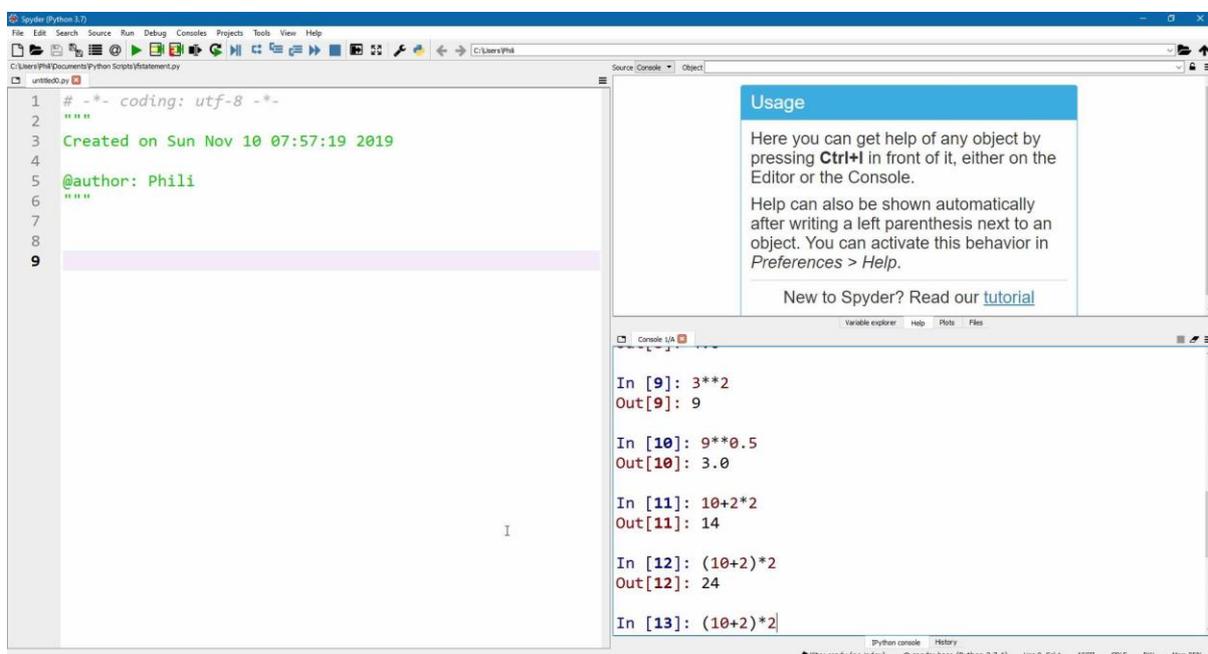
```
10+2*2
```

```
(10+2)*2
```

which return the values 14 and 24 respectively.

Accessing Previous Commands

The up arrow \uparrow may be used to access the previous command typed.



```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Sun Nov 10 07:57:19 2019
4
5 @author: Phili
6 """
7
8
9
```

```
In [9]: 3**2
Out[9]: 9

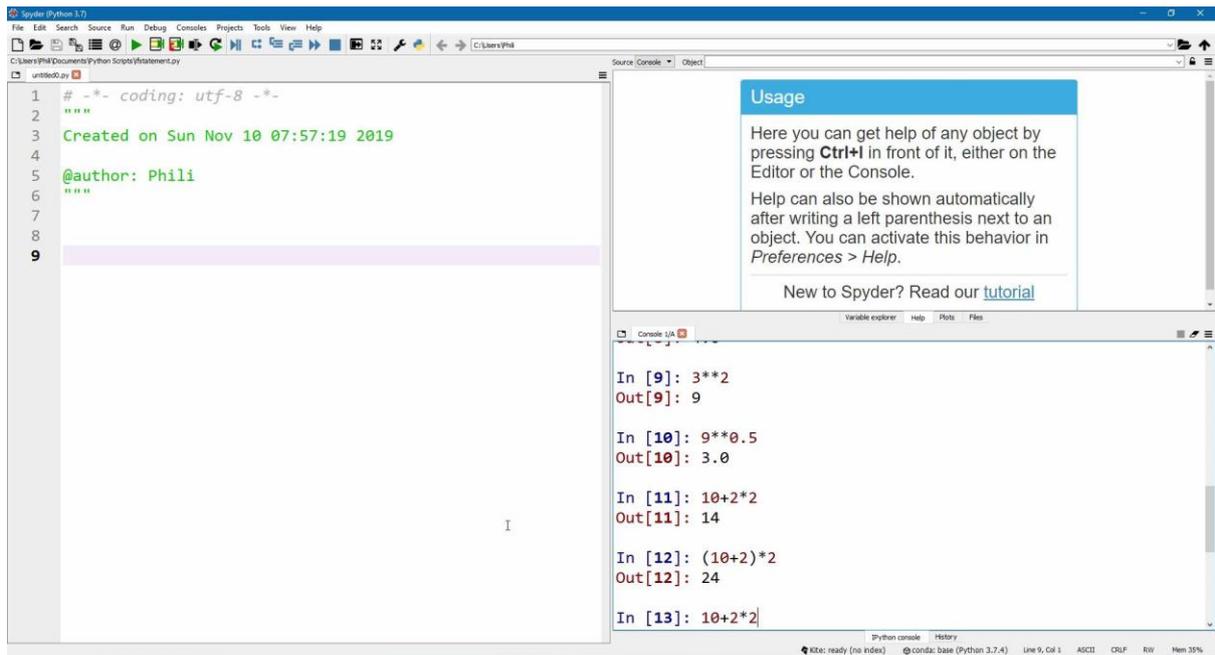
In [10]: 9**0.5
Out[10]: 3.0

In [11]: 10+2*2
Out[11]: 14

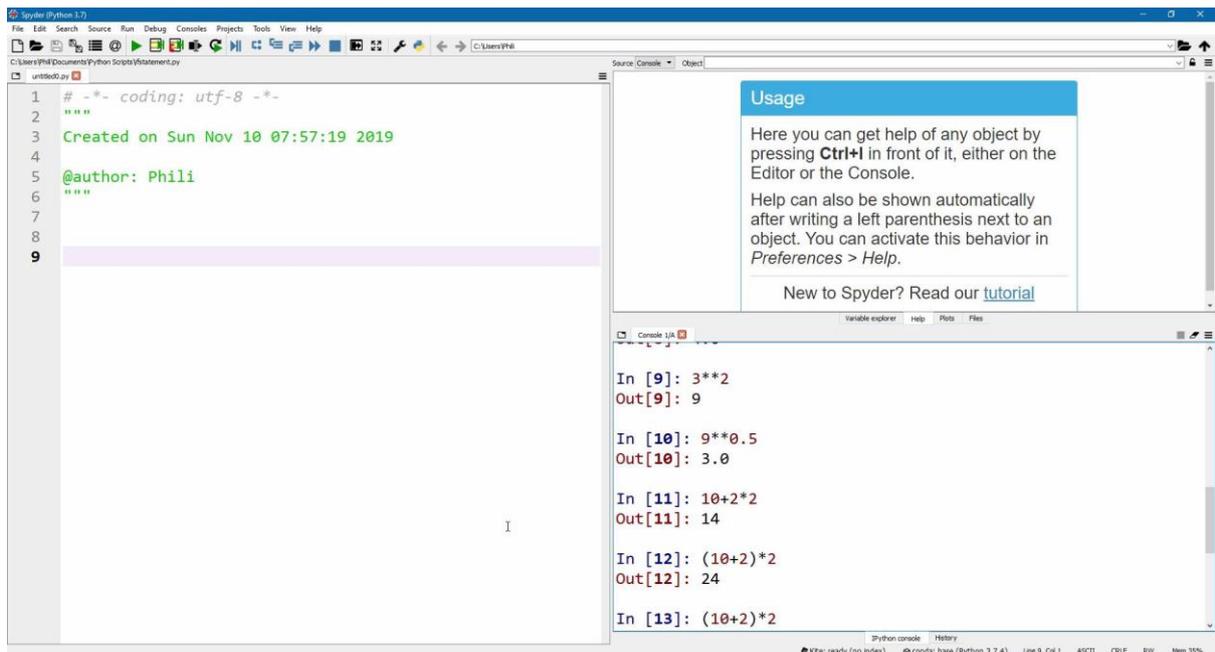
In [12]: (10+2)*2
Out[12]: 24

In [13]: (10+2)*2
```

If pressed again it will get the second previous command.



Pressing the down arrow  will return you to the last command.

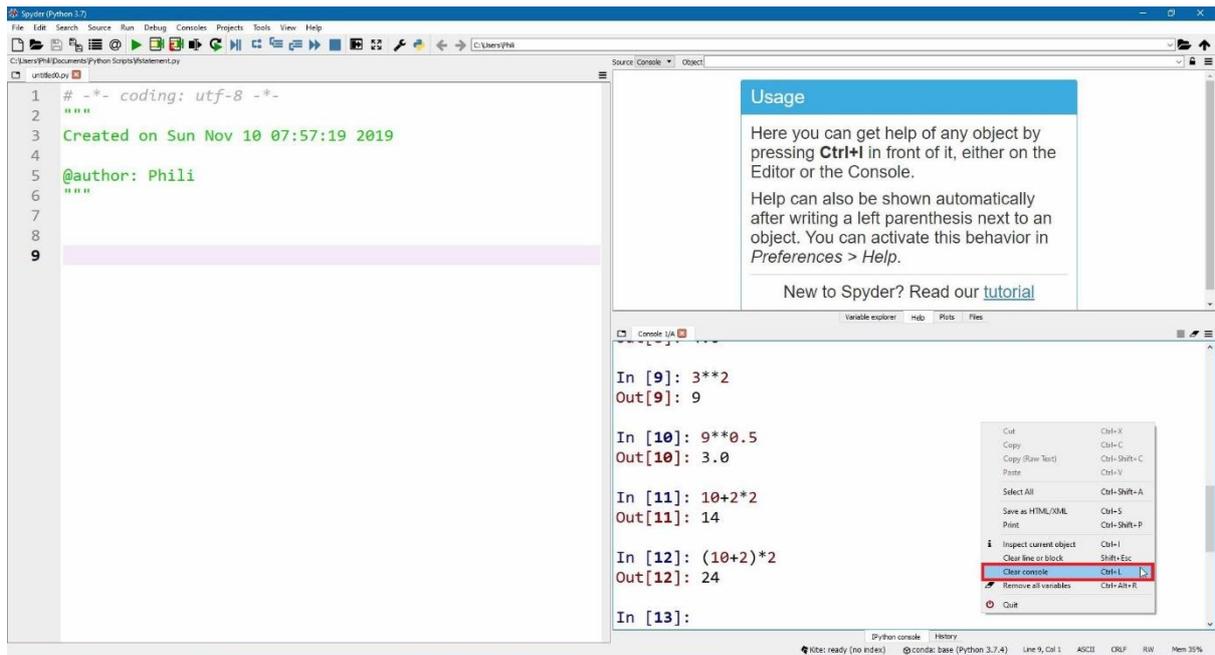


You can use these keys to scroll through all the previously used commands.

Clearing the iPython Console

Clearing the Console

It is possible to clear the commands printed on the console by right clicking the Console and selecting clear console. There is also the shortcut key `Ctrl+L` (upper case L or lower case l work).

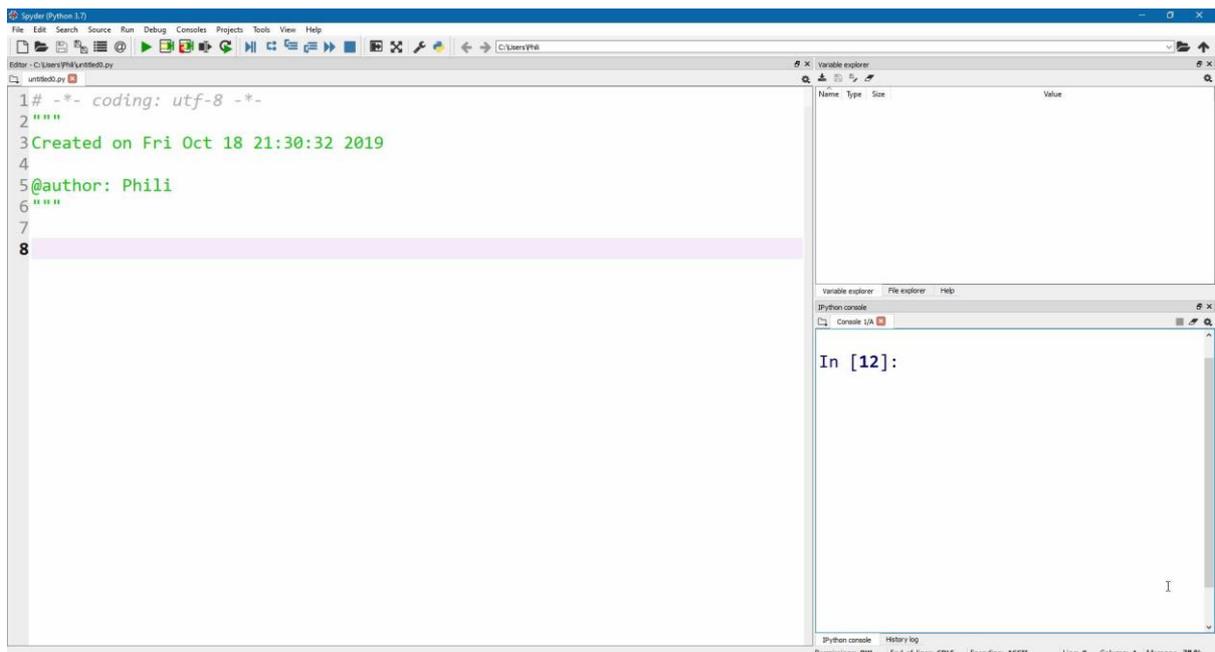


Clearing the console will not remove any libraries loaded and will not clear any variables created (so far, we haven't looked at these). The command line in the console will also continue numerically from the last command. In this case, the last command was 12, the instruction to Clear the Console was taken as line 13 and so line 14 will display.

The following command

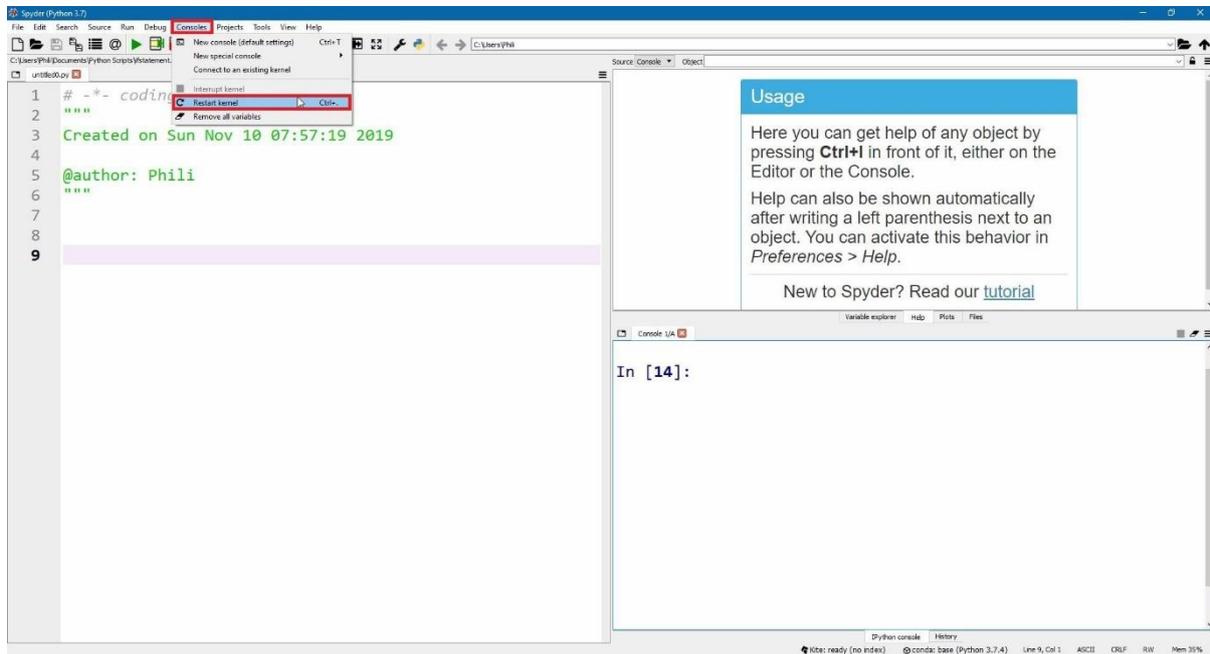
```
14. clear
```

Also clears the console and is what happens in the background when Clear Console is selected via the context menu.



Restarting the Kernel

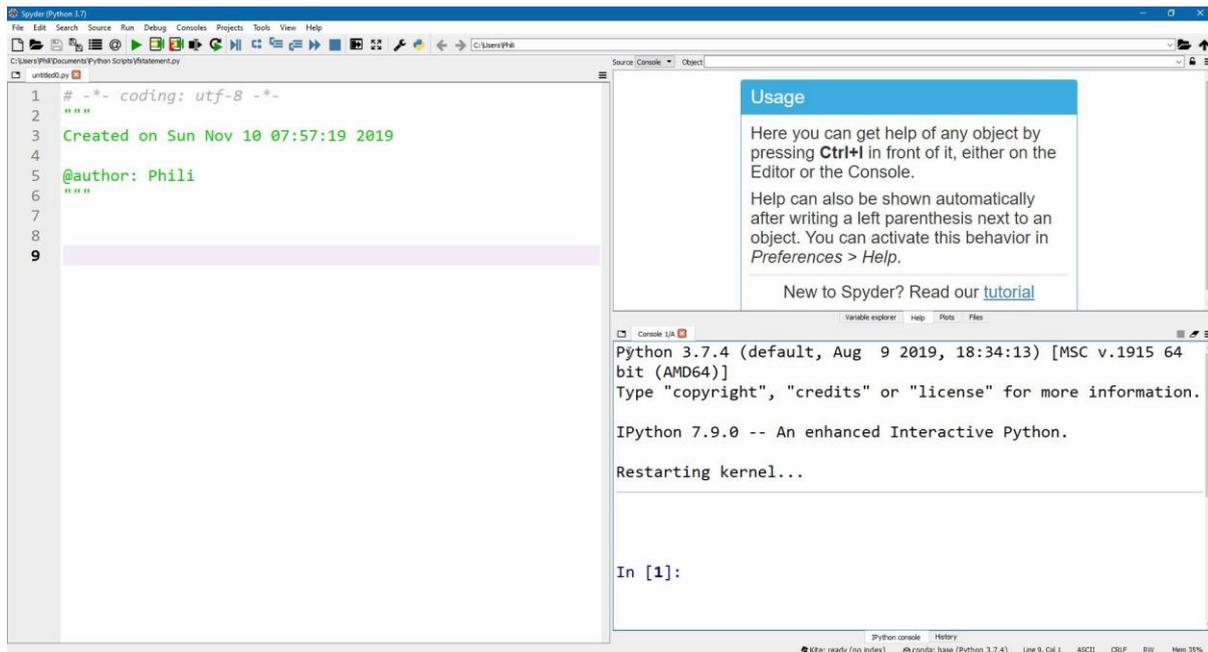
It is also possible to restart the Kernel which will close any imported libraries, remove any variable assigned and reset the line to [1]. To do so, select Console and then Restart Kernel, the shortcut key **Ctrl+.** is also available.



Accept the warning dialogue:



And wait a moment for the Kernel to be restarted.



Variable Assignment

Variable Names

In Python, values may be assigned using the assignment operator =. To the left hand side is the variable name which should consist of all lower case values and no special characters with the exception to the underscore _ which should be used in place of a space.

Good variable names

```
my_var=1
```

This variable name is descriptive is all lower case with an underscore _ and is short and concise.

```
a=2
```

This variable name is short and concise which can be useful for a personal quick algebraic expression or a loop but may need to be documented in more detail in more complicated and shared code.

```
a2=2.1
```

This variable name is once again short and concise and can be used in relation to variable a. Note the variable name has to start with a lower case letter.

Frowned upon but allowed variable names

```
My_Var=1
```

Python is case sensitive, so if `my_var` is called you will get the error message `NameError: name 'my_var' is not defined`.

```
My_really_amazing_variable=1
```

This variable name is really descriptive however has 28 characters which means one is needlessly typing in characters and thus far more likely to make a typo.

Disallowed Variable names

```
2=1
```

This does not make sense, the number 2 cannot be assigned the value 1 as all mathematics would be broken. If this is attempted, you will get the error `SyntaxError: can't assign to literal`.

```
2a=1
```

Variable names also cannot begin with numbers, you will get `SyntaxError: invalid syntax`.

```
my var=1
```

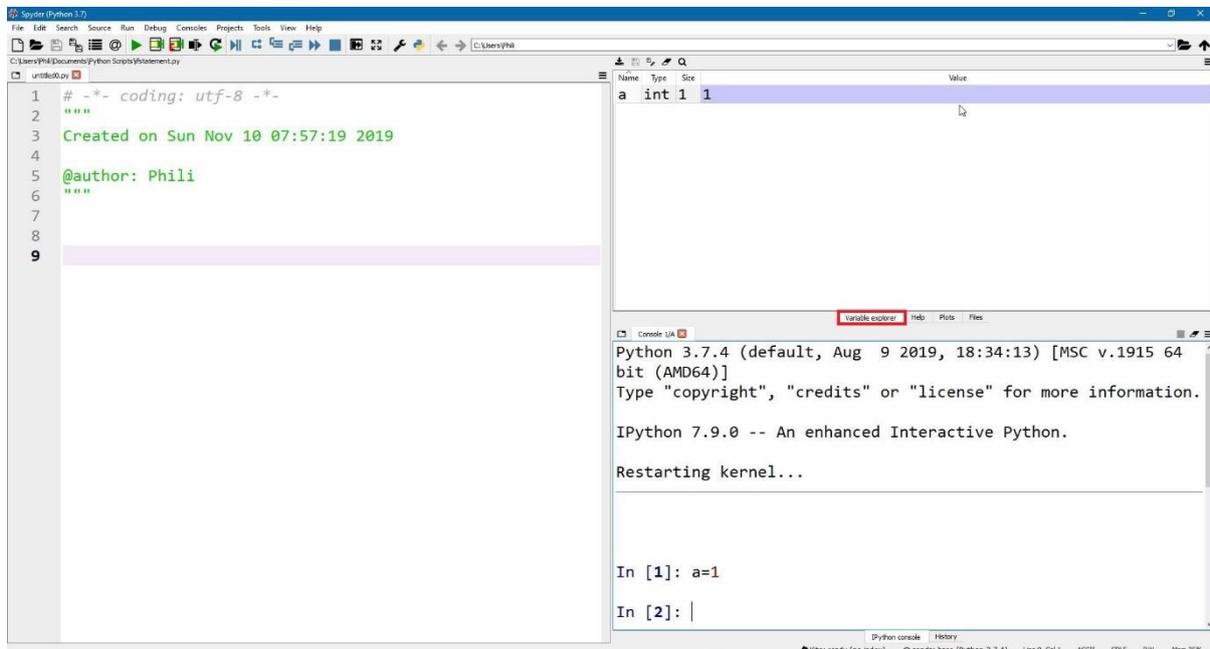
Variable names cannot have spaces, if you attempt to call a variable name with a space you will get `SyntaxError: invalid syntax` which is why the underscore `_` is used opposed to the space.

Numeric Variables

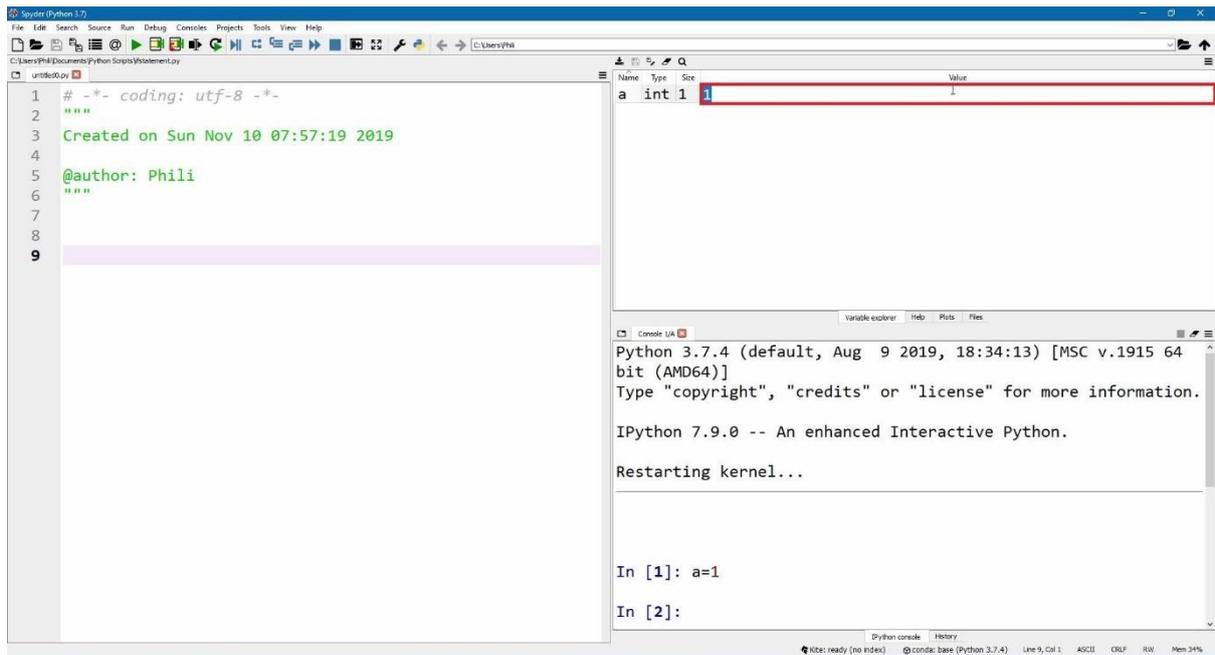
In this stage we will use the assignment operator to assign the variable name `a` to the value `1`.

```
a=1
```

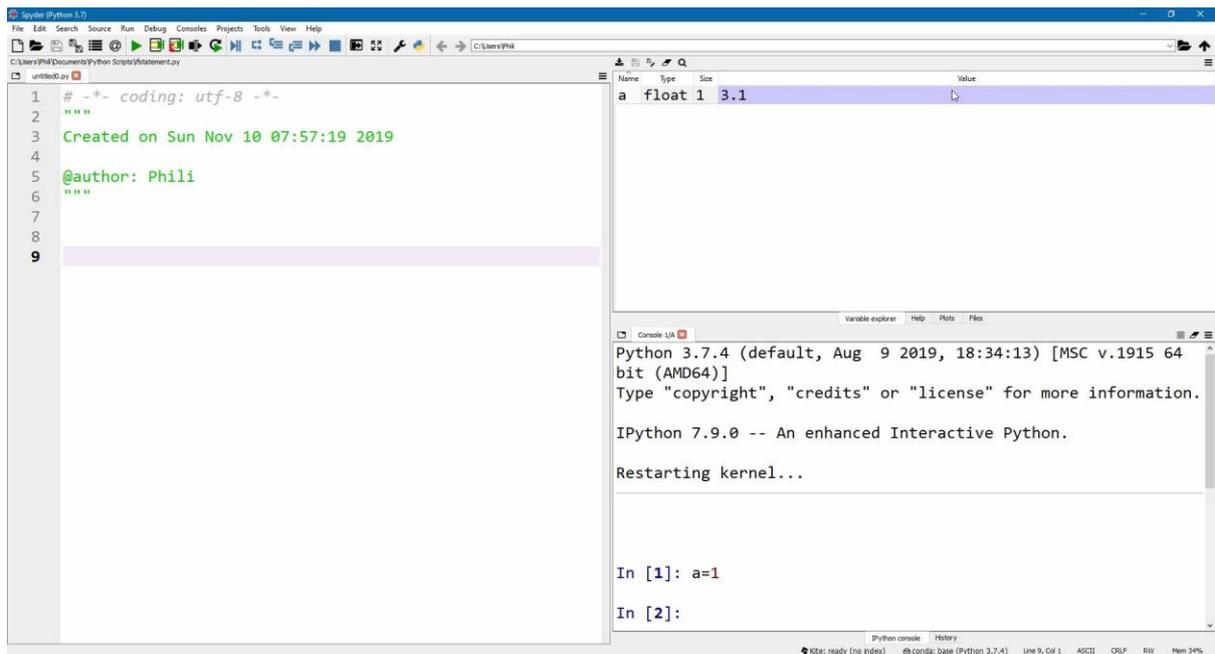
This assigned variable will Display in the Variable Explorer. Select the Variable Explorer tab.



The Name will show as `a`, the Type as `int` (because it is a whole number) and the Value will be `1`. One can double click into the value field and amend it.



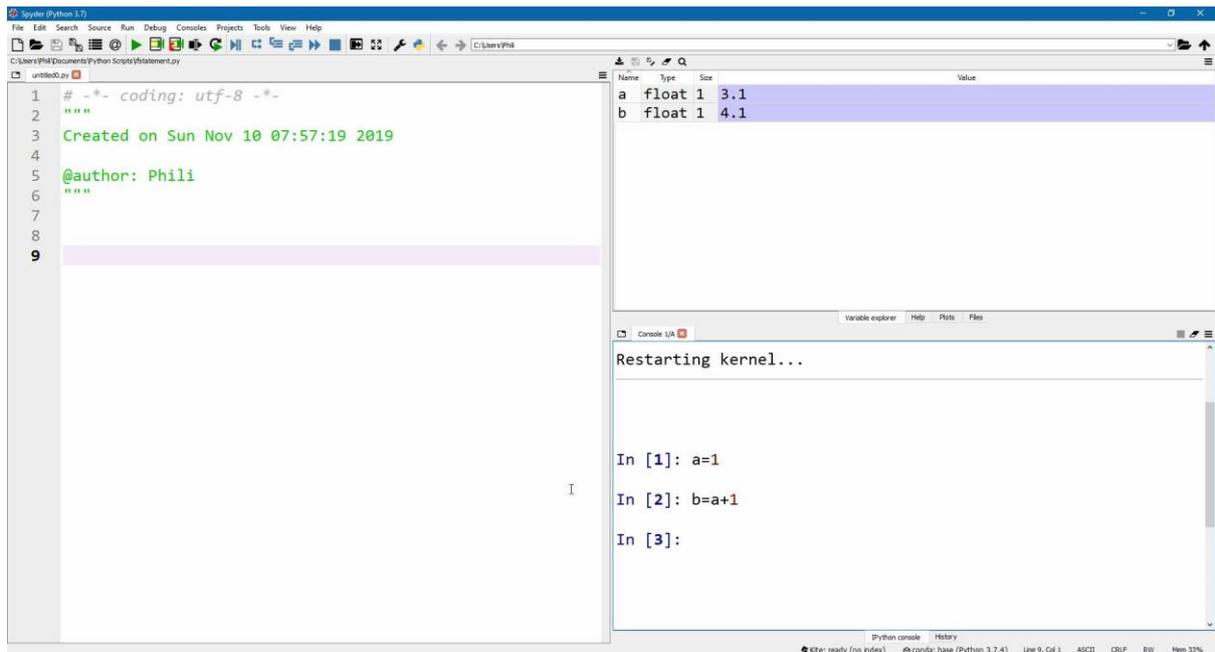
Let's type in another number, this time with a decimal point `3.1` and we'll see that the Type is updated to be a float because this is a number which has a decimal place.



We can now use this variable in as part of an algebraic calculation. For example

```
b=a+1
```

Here the new variable Name `b` is assigned the value of the current Value of `a` added to `1`. When this calculation was called the Value of `a` is `3.1` so this is equivalent to `3.1+1` which is of course `4.1`.



The new Variable `b` shows up alphabetically below `a` on the variable explorer.

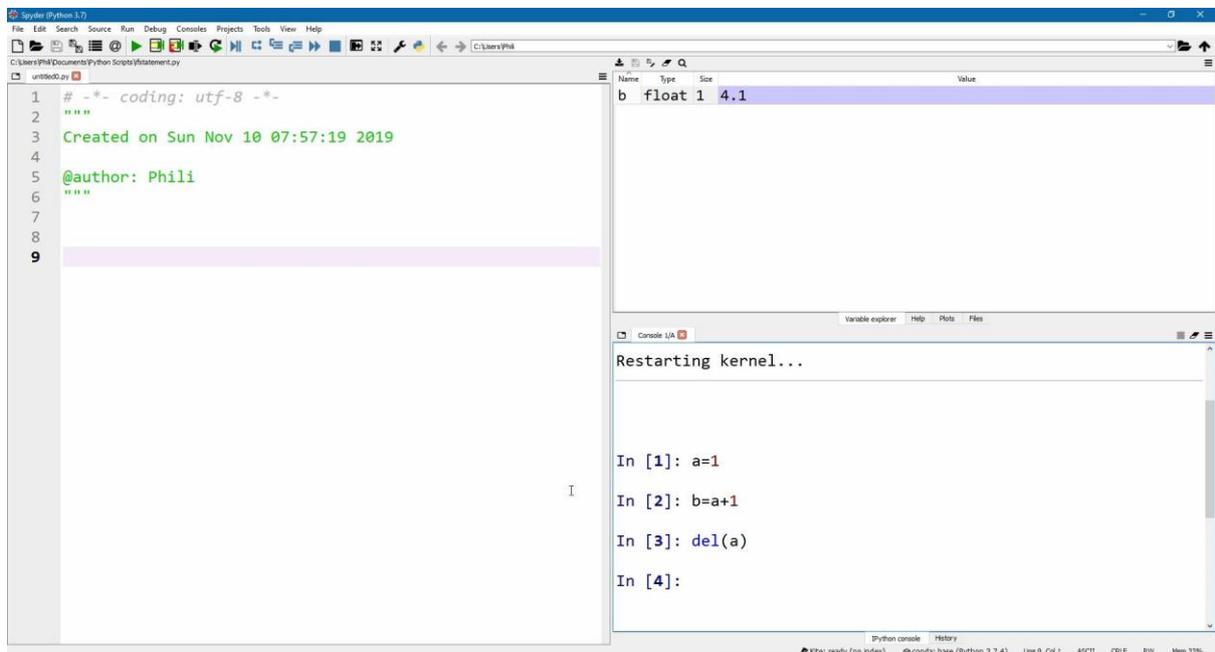
Deleting Variables

To remove the variable for example the variable `b` we can use the `del` function.

Let's delete the variable `a`

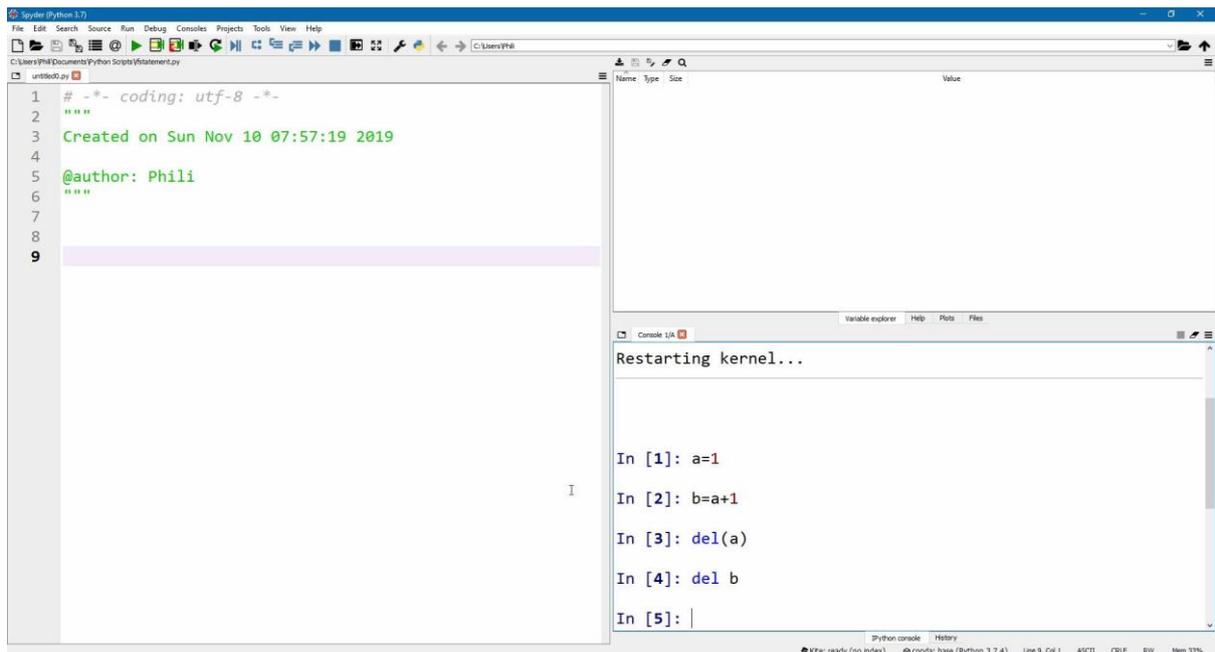
```
del (a)
```

We'll see the `a` disappear in the variable explorer.



Note the colors around numbers `4.1` and the special function `del`. Functions are normally called followed by parenthesis `()` for input arguments, in our case the input argument is the variable `a` however as the function `del` is so commonly used it can also be called with a space opposed to brackets. We can demonstrate this by deleting the variable `b` also.

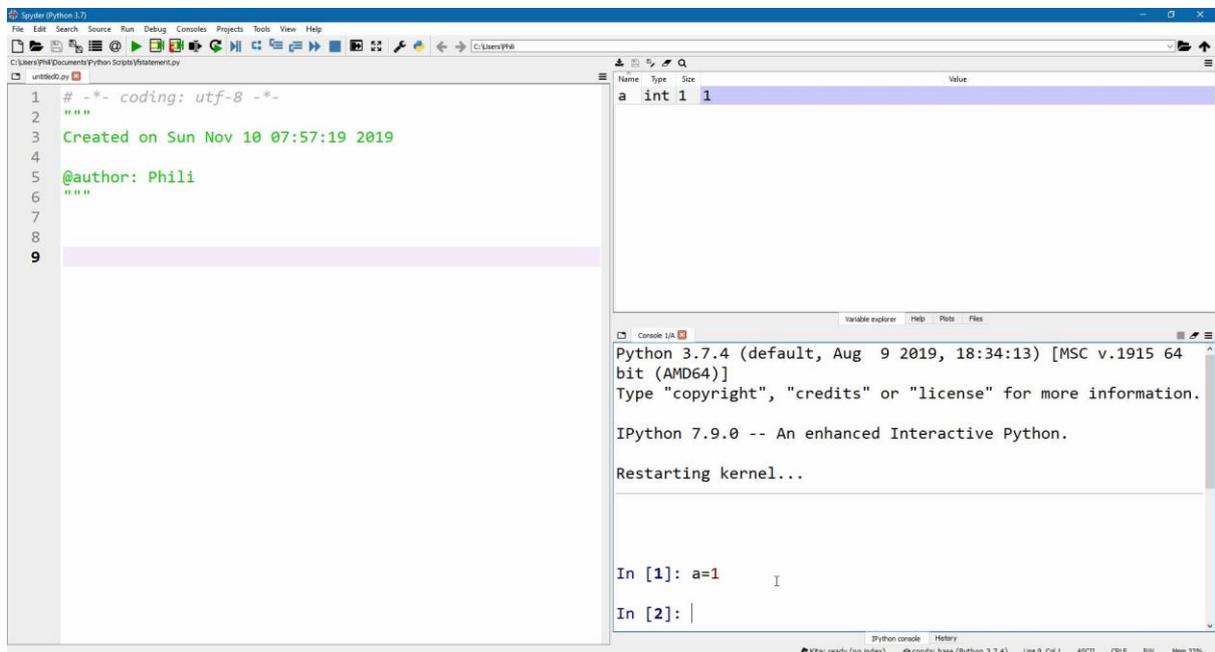
```
del b
```



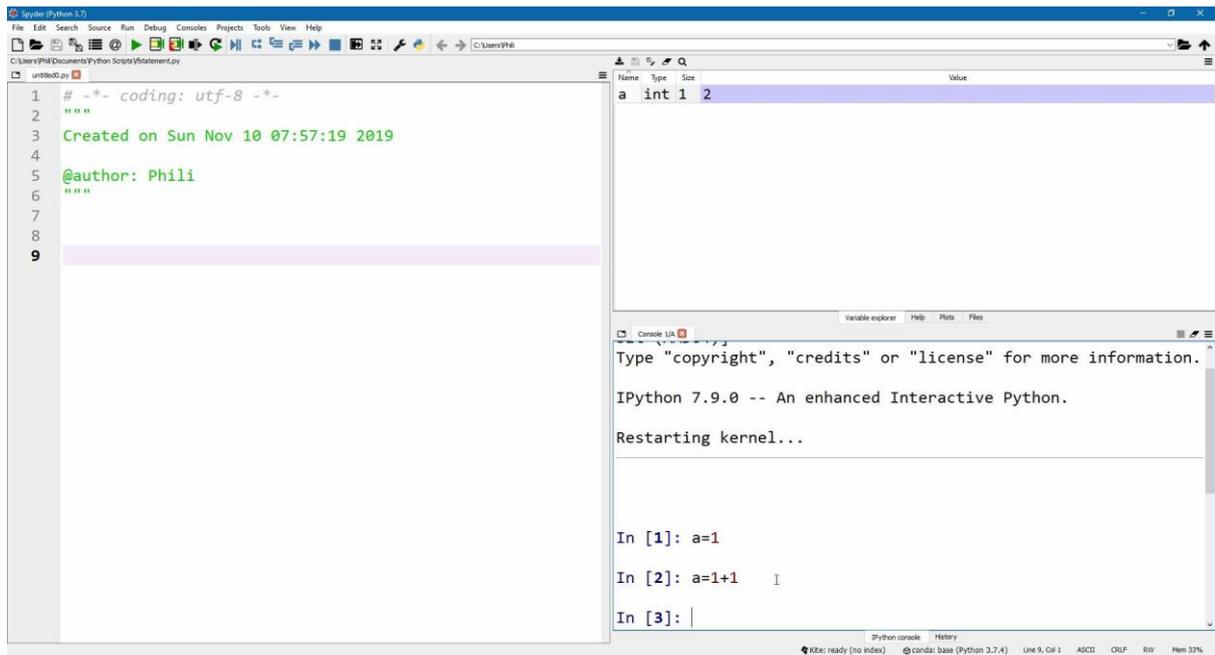
Variable Reassignment

We have seen the update, or reassignment of a variable by clicking on its value in the variable explorer. It is also possible to reassign it by using the assignment command. Let's create a new variable name `a` and assign it to `1`.

```
a=1
```



Now let's assign `a` to the value `1+1`.

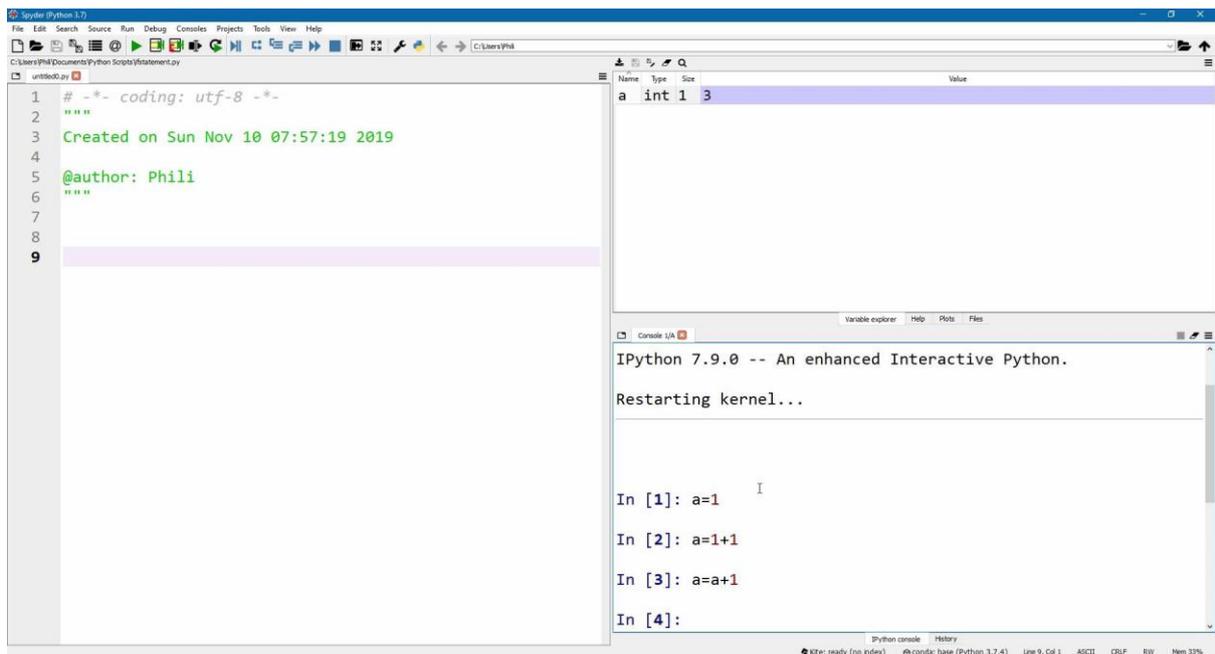


```
a=1+1
```

Since `a` was already created this is known as reassignment, the updated value is now `2`. As we have seen above it is possible to perform an algebraic expression involving a variable and assigning it to a new variable name. However, it is also possible to create an algebraic expression involving a variable and then reassign the value to the original name this is known as variable reassignment. For example:

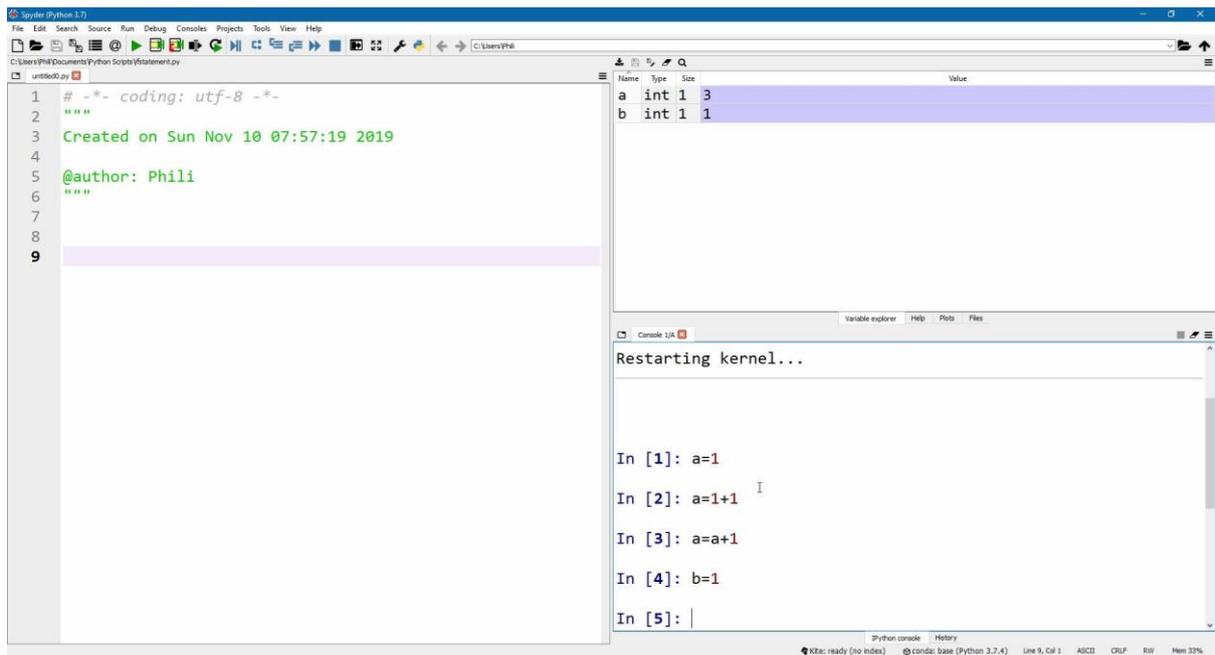
```
a=a+1
```

When the assignment operator `=` is used, the mathematics to the right hand is carried out first. In this case the value of `a` which is `2` is added to `1` which creates a new value of `3` that is then reassigned to the variable `a`.



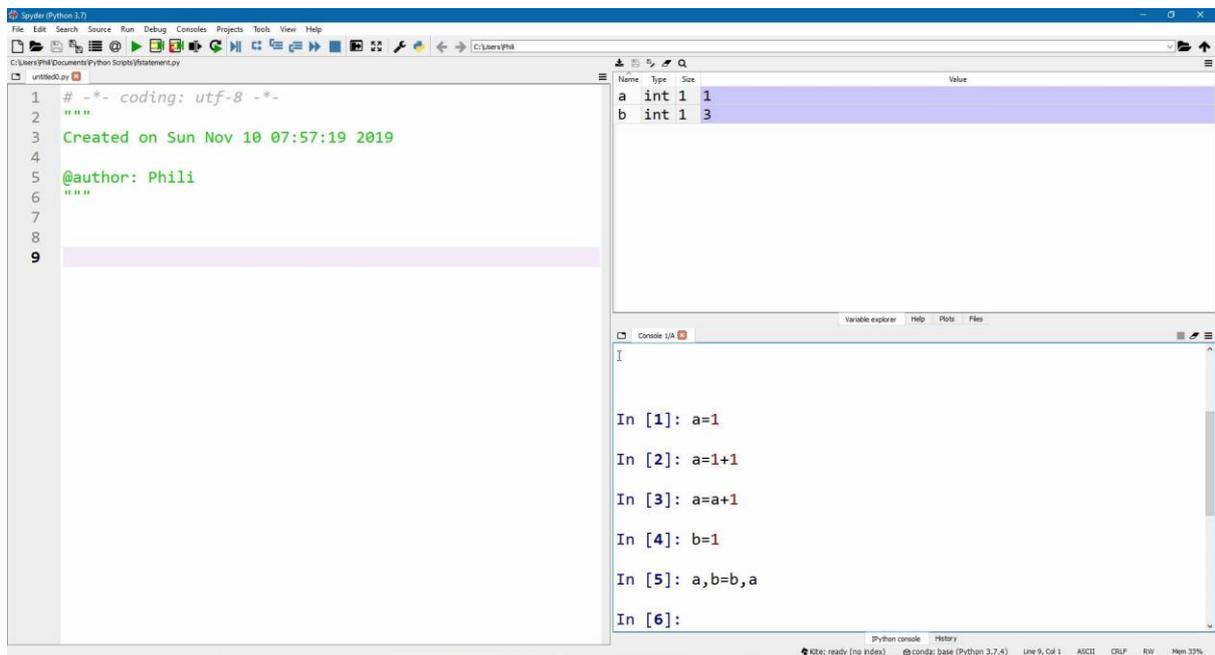
Let's now create the variable `b` and set it to `1`.

```
b=1
```

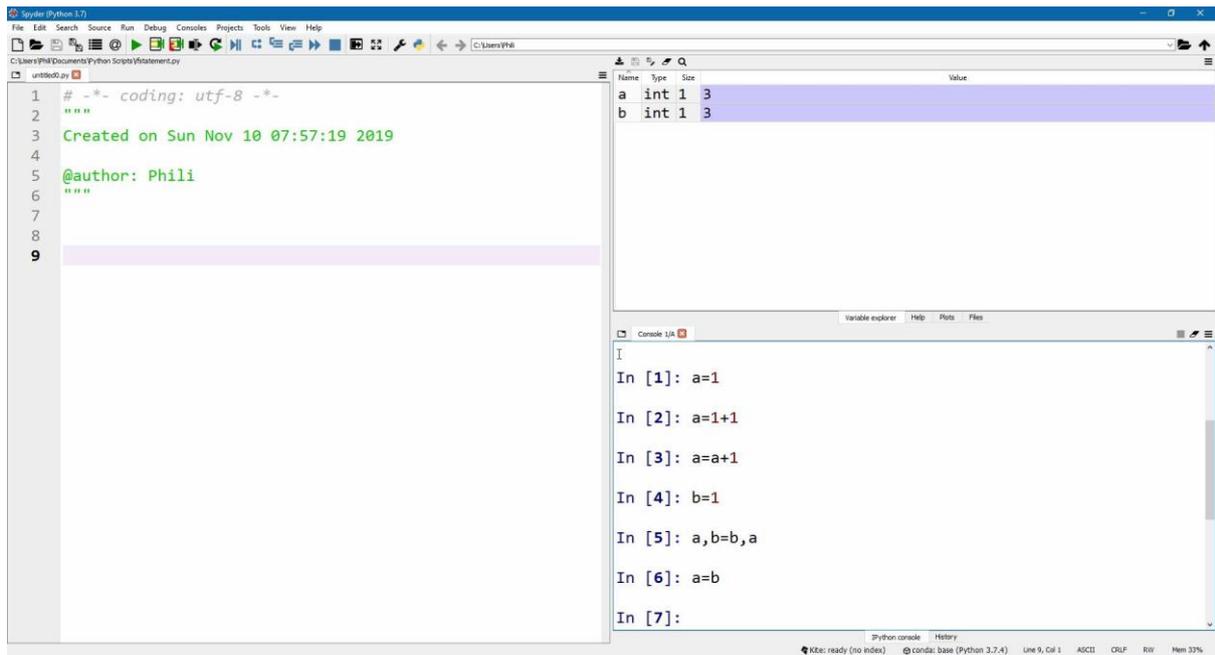


Supposing we now want to reassign `a` to equal the original value of `b` and we also want `b` to have the original value of `a`. This can be done by use of a comma `,`.

```
a, b=b, a
```

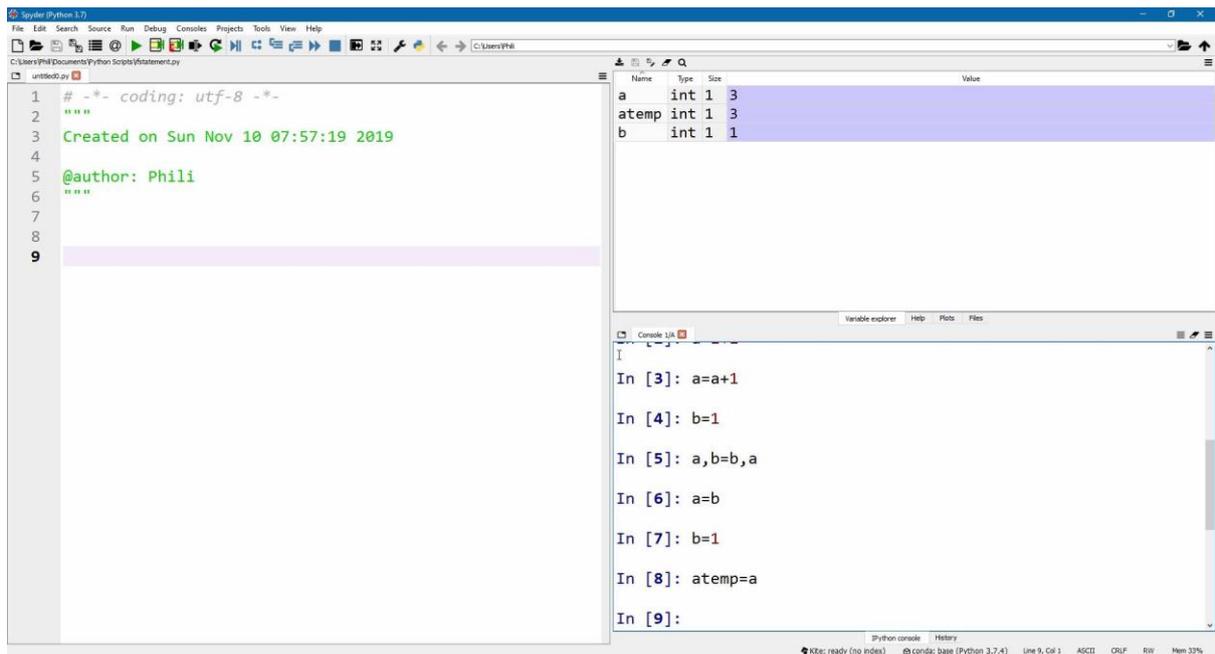


However, it should be noted that if only the value of `a` is assigned to the value of `b` will result in the loss of the original value of `a` or alternatively assigning `b` to the value of `a` will result in the loss of the original value of `b`.



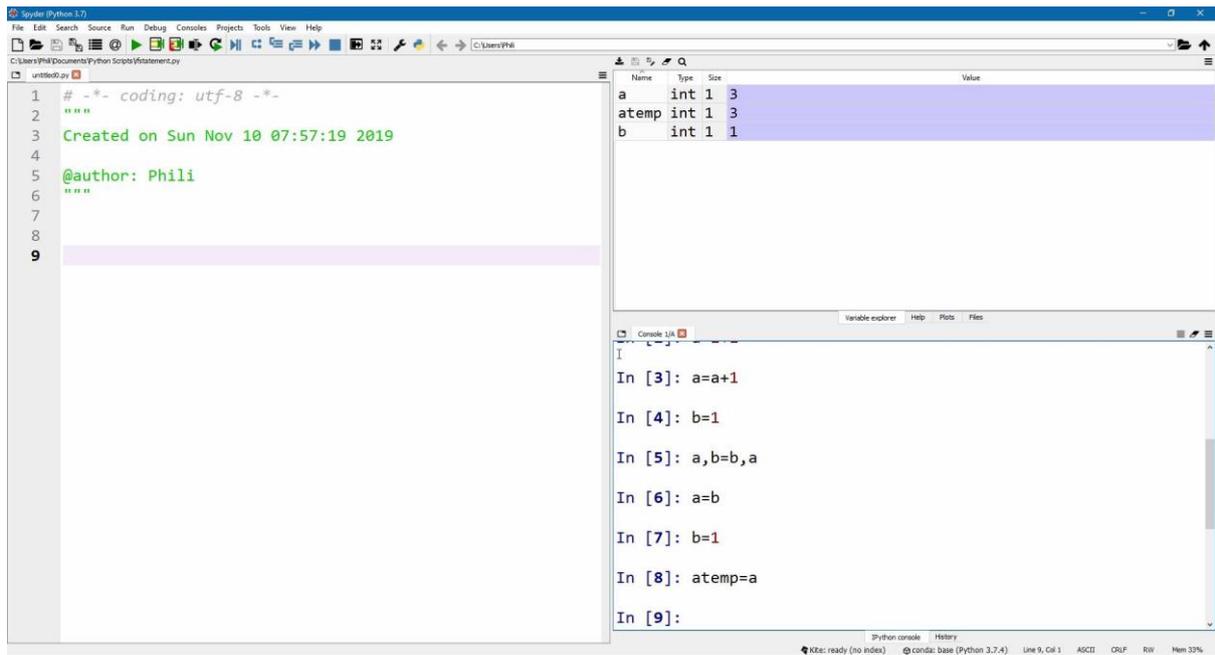
Setting `b=1` we can swap the variables by introduction of a temporary variable.

```
a=temp=a
```

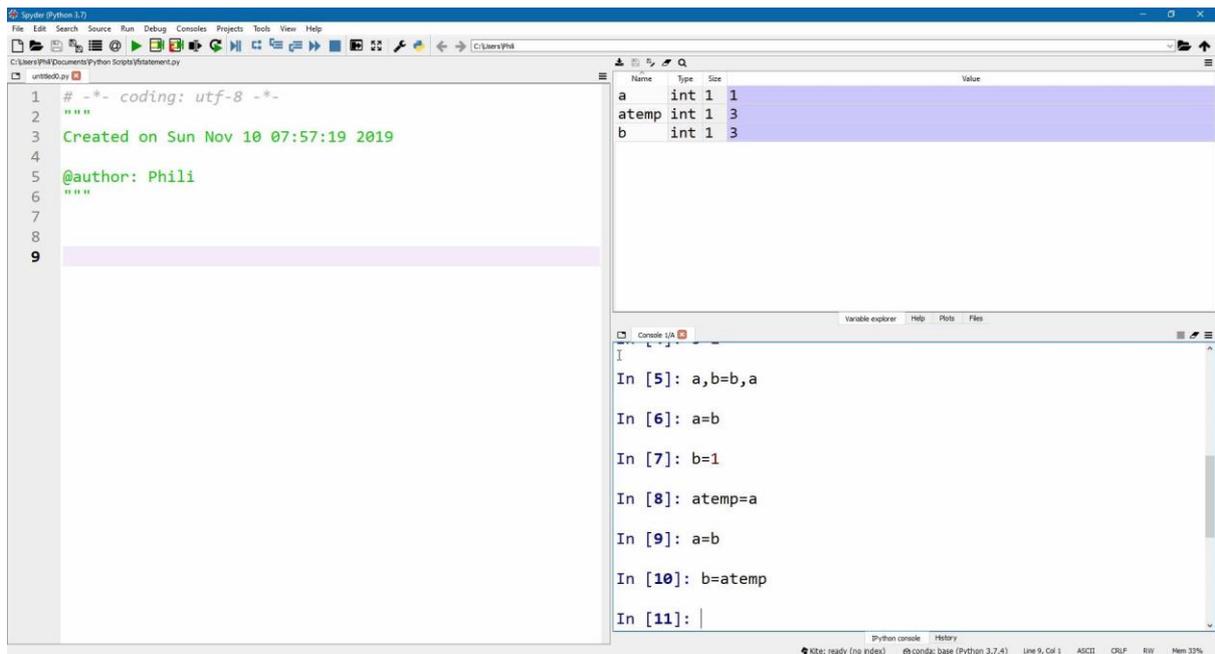


Now `a` can safely be assigned to the value of `b` without loss of the original value of `a`.

```
a=b
```

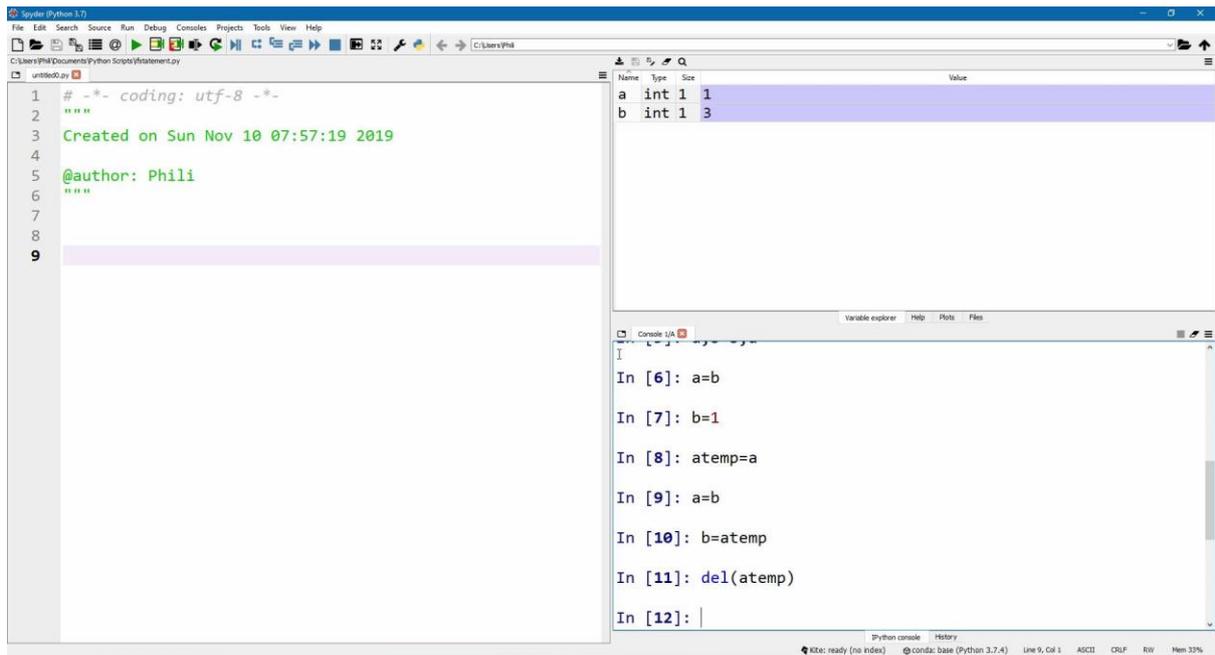


Then `b` can be assigned to the value of `atemp`.



And finally, `atemp` can be deleted.

`del (atemp)`

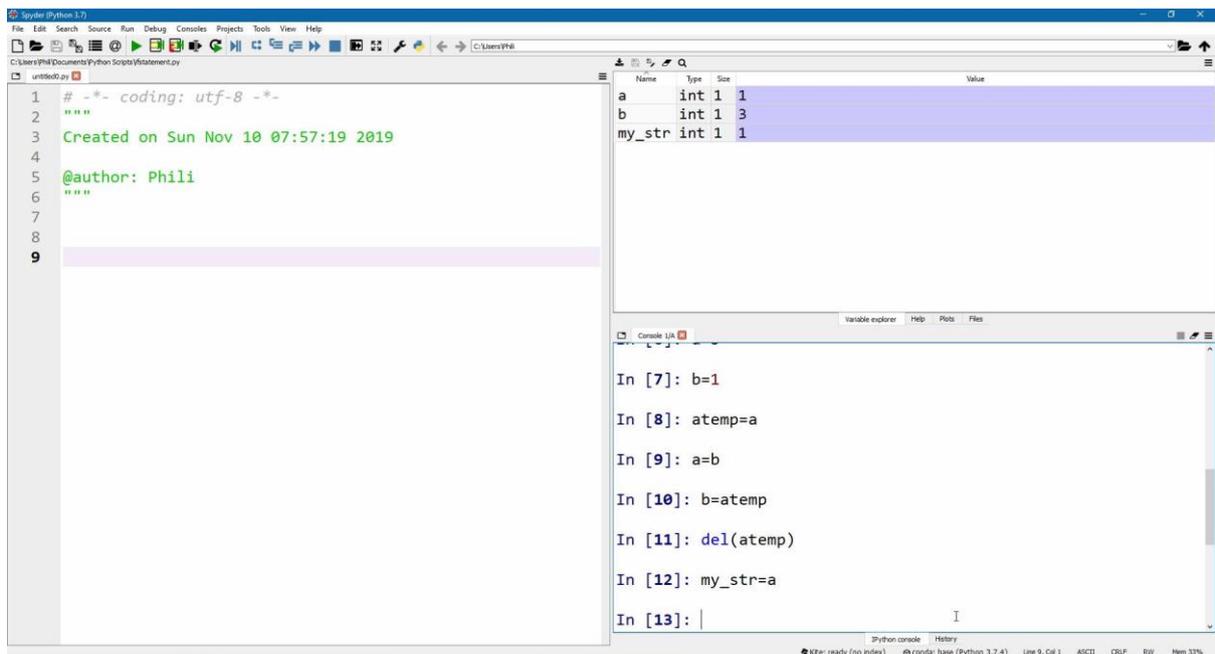


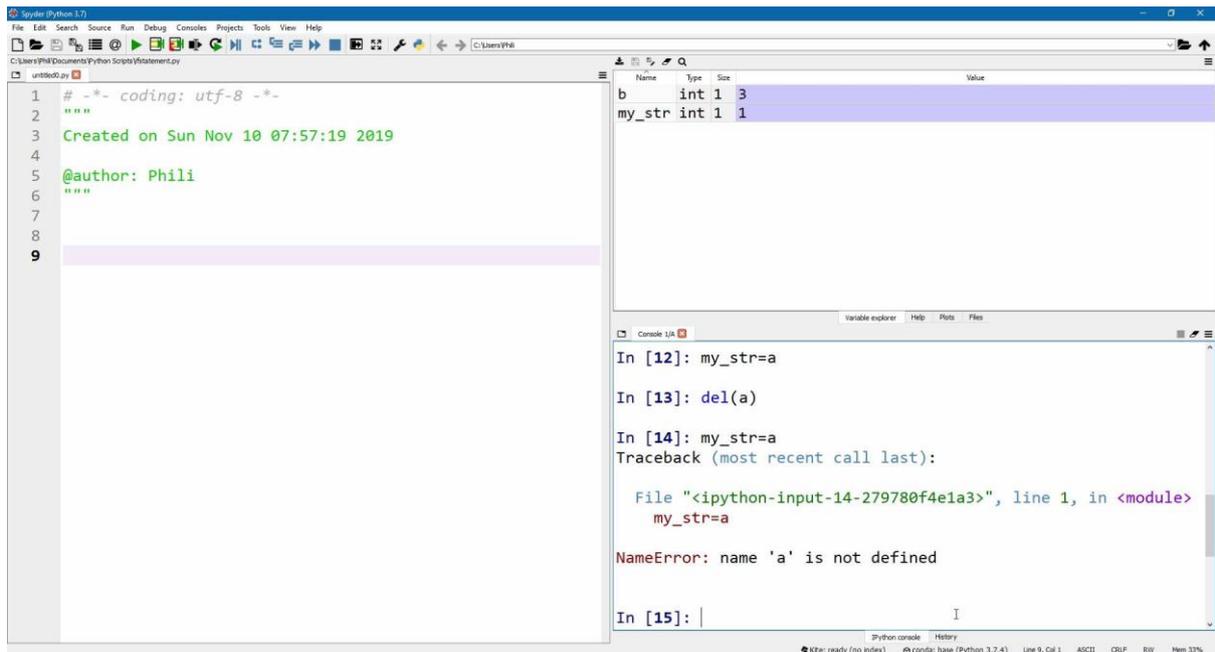
String Variables

So far, we have only looked at numeric variables, it is also possible to assign a letter, word, words which include numbers, special characters and are spaced by a full stop which are collectively known as strings of characters often abbreviated strings or str. Strings are usually enclosed in single quotations however Python also allows the use of double quotations. Note the difference between.

```
my_str=a
```

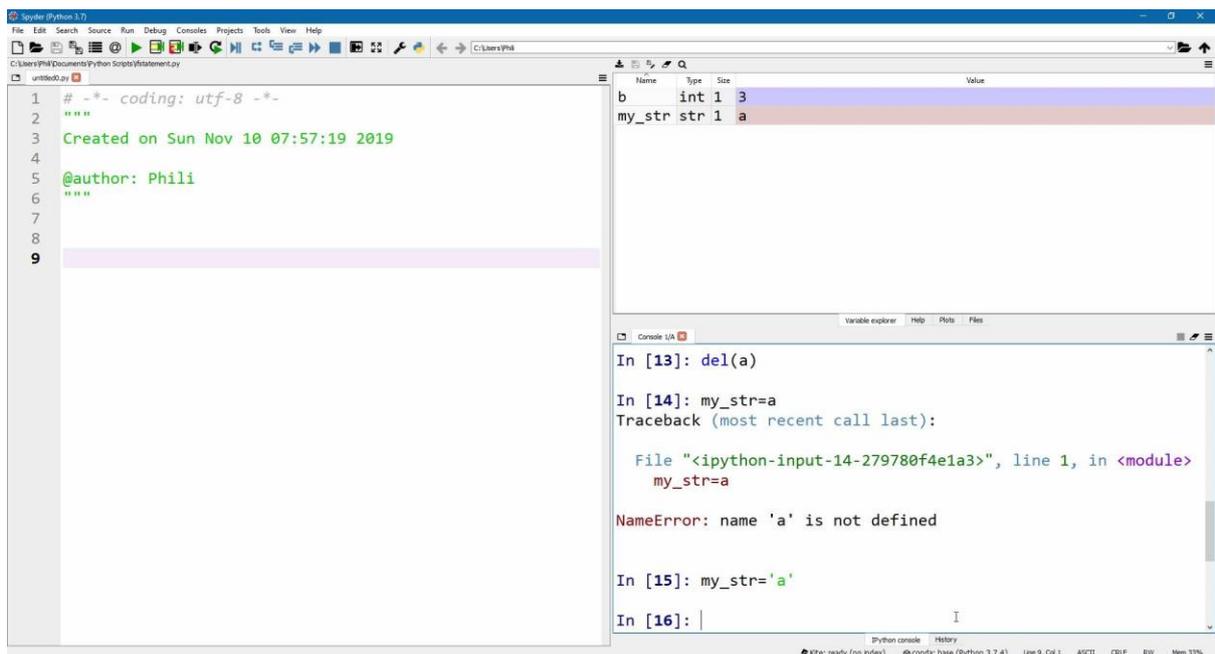
Which assigns the value of the variable `a` to the new variable name `my_str` or alternatively if the variable `a` does not exist throws up the error `NameError: name 'a' is not defined` for example if we delete the variable `a` and then run the same command.





In contrast when `my_str` is assigned to a str, the str is enclosed in quotations and highlighted green 'a'.

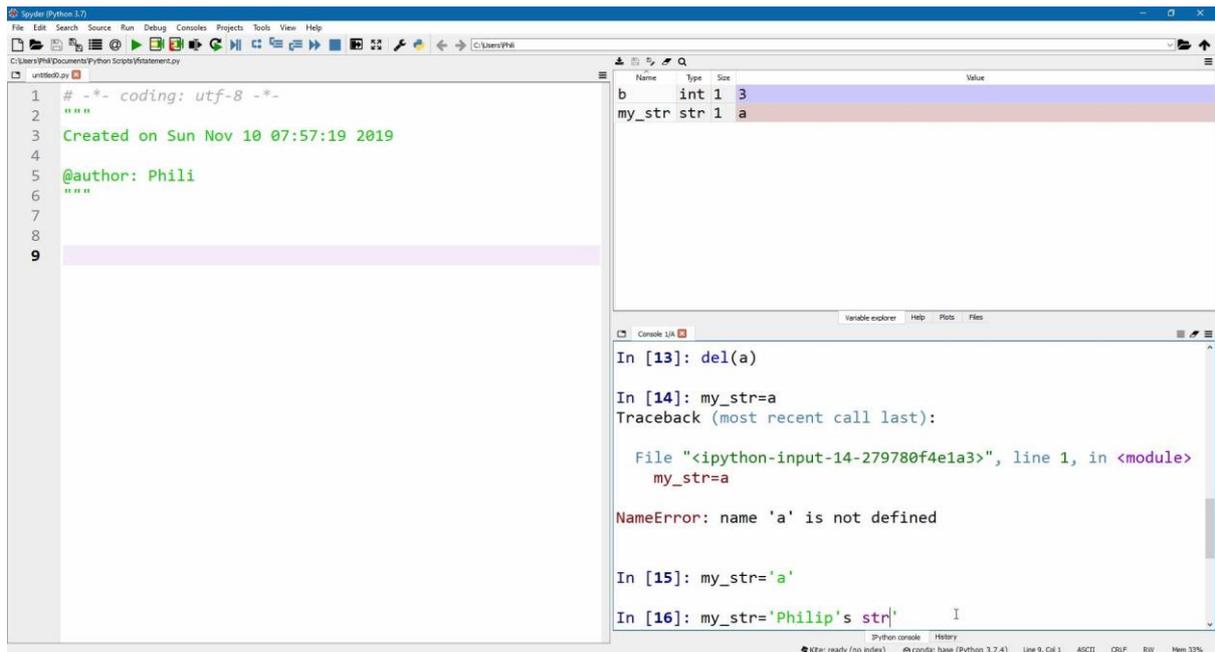
```
my_str='a'
```



There are several special characters in strings such as the quotation marks themselves. This can be seen if we try to type in the following string:

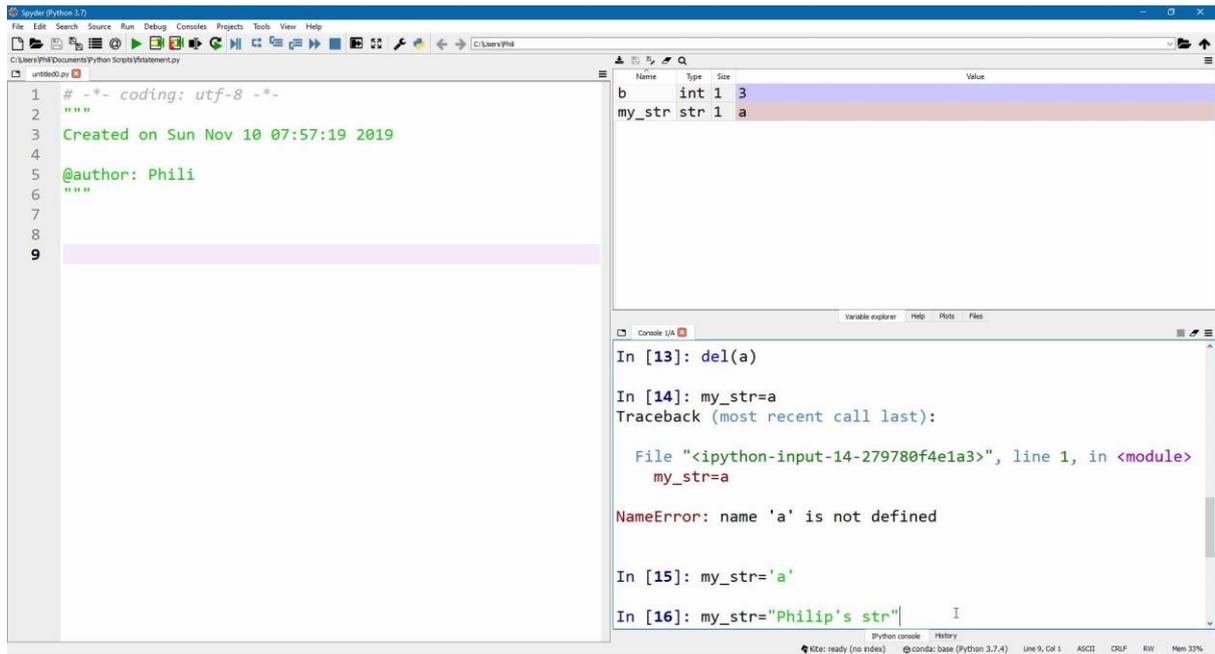
```
my_str='Philip's str'
```

The string is ended at 'philip' then s is taken outside the string, str is taken as a function and a new string begins at '.



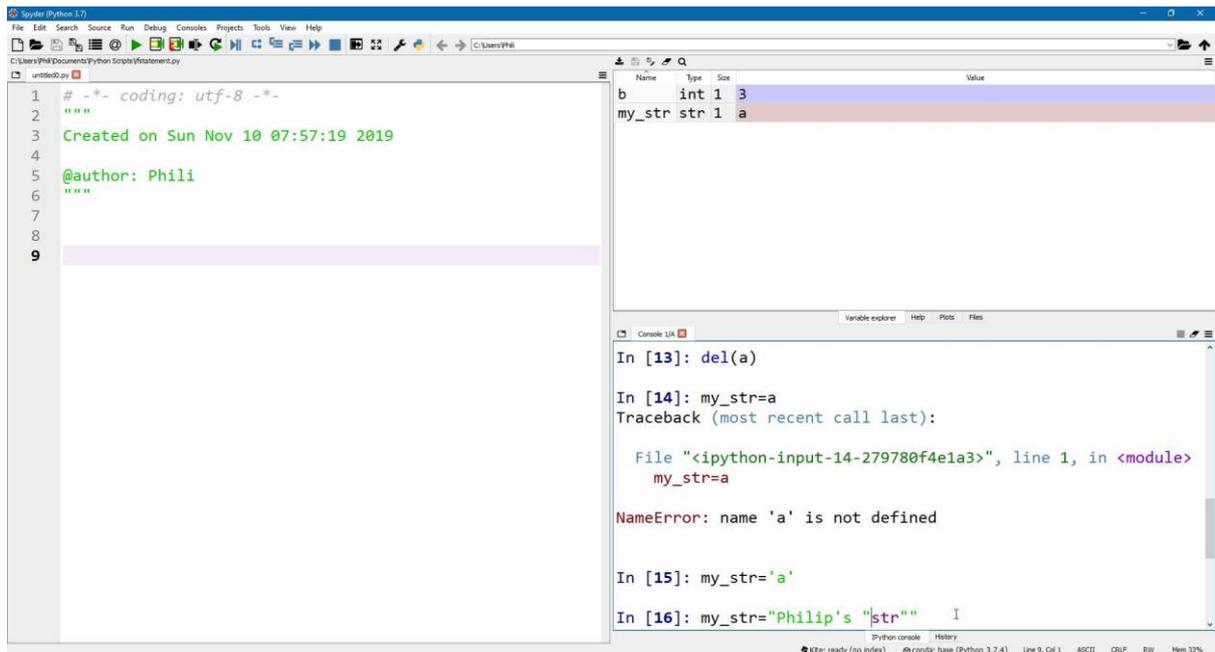
For this reason, the above can be written as

```
my_str="Philip's str"
```



However if part of a string is enclosed in quotation marks for example:

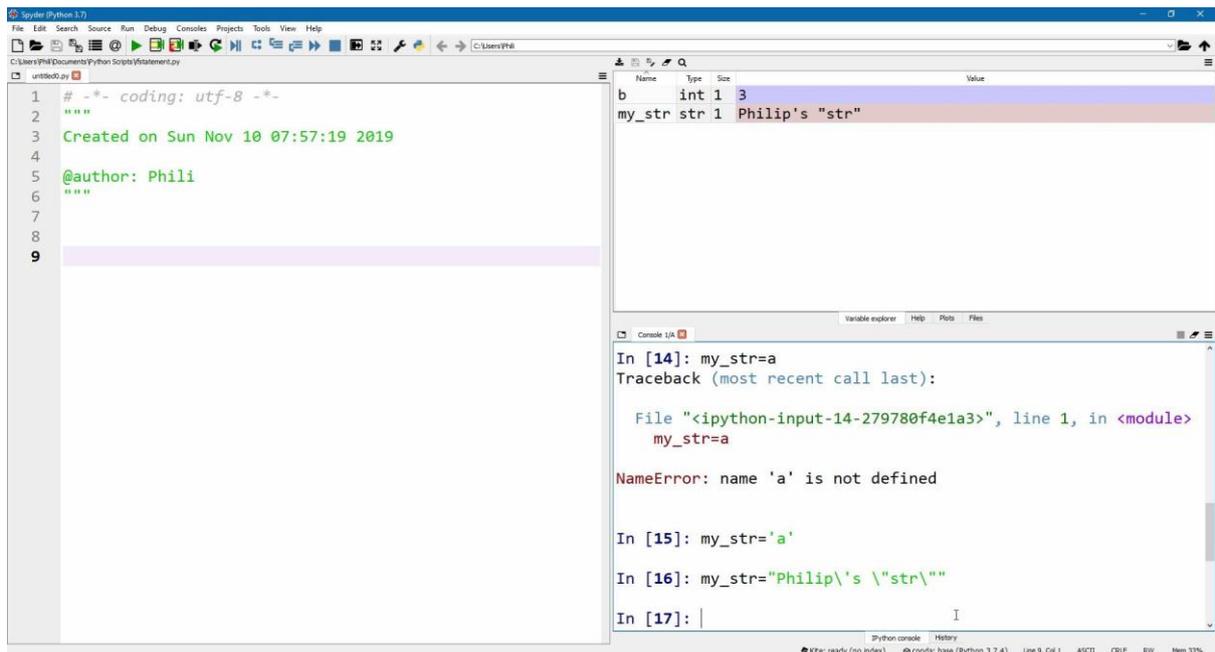
```
my_str="Philip's "str""
```



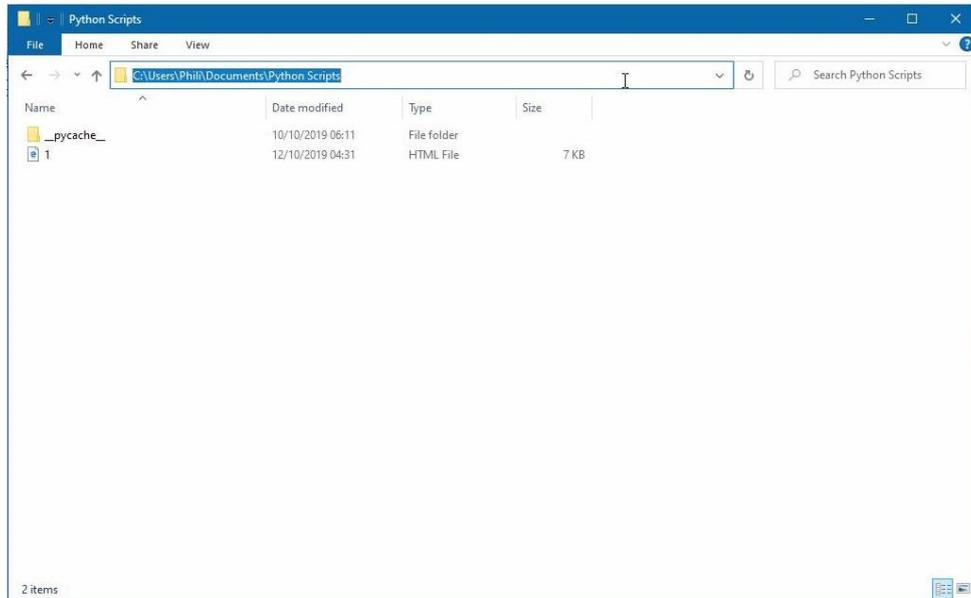
Then once again, the string is closed prematurely. We can get around this by using the special character the left slash `\` which is used for `\'` and `\"`

```
my_str="Philip\'s \"str\""

```



The left slash `\` can also be used for `\n` which gives a new line, a `\t` horizontal tab, a `\v` vertical tab. However, the file path in the Windows Operating System also uses the `\` to indicate a directory (folder). For example:



`C:\Users\Phili\Documents\Python Scripts`

In order to set this as a string, we need to use the left slash twice `\\`.

```
file_path='C:\\Users\\Phili'
```

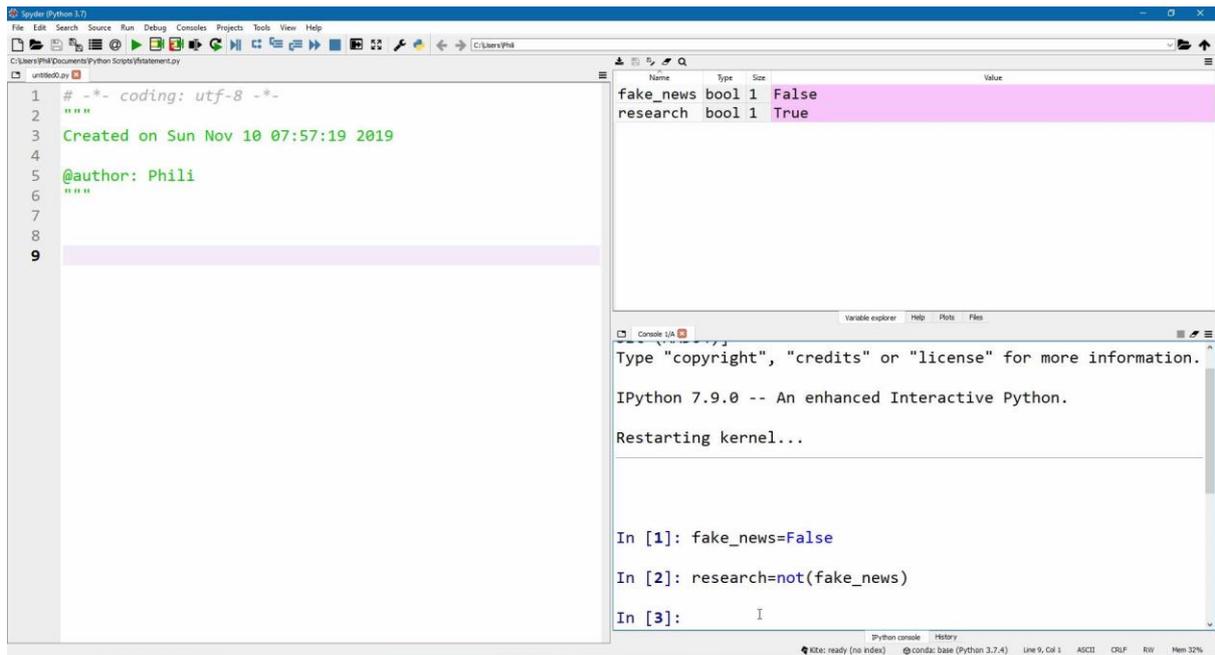
Boolean Variables

So far, we have looked at numerical data types int, flats and string data types another commonly used datatype is the Boolean datatype which has either the value `True` or `False`. We can assign a variable name `fake_news` and assign it to `False`.

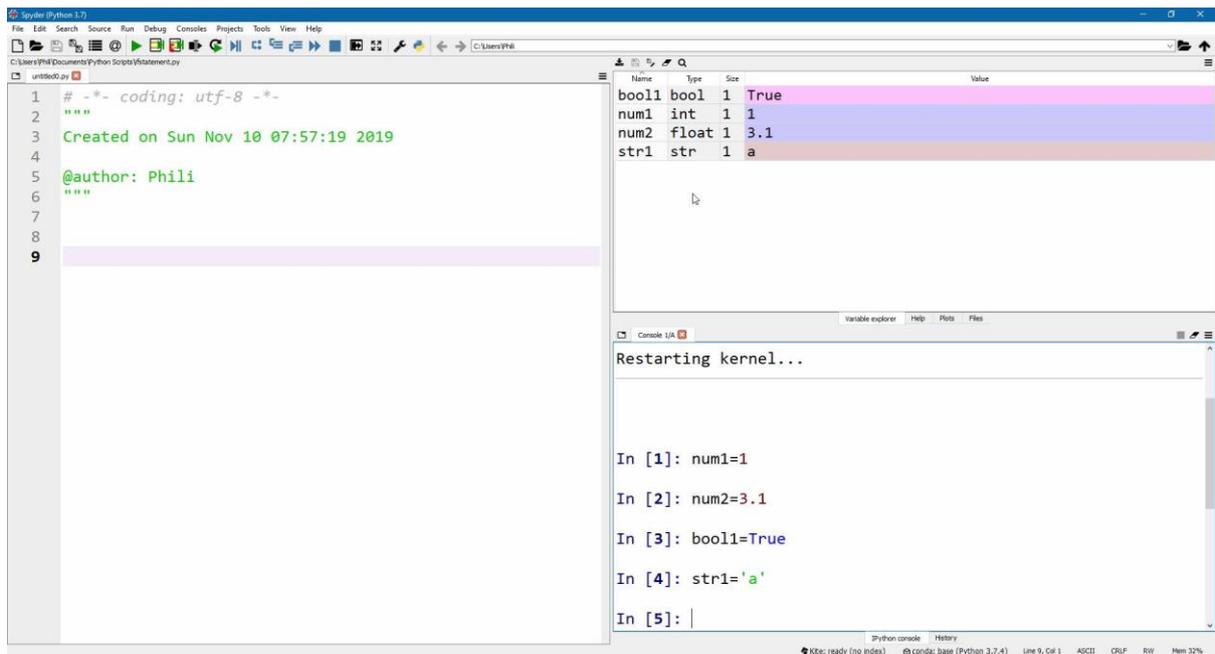
```
fake_news=False
```

Note the capitalisation and the color and how this varies to a string which would be green or variable which would be plain.

We can create a new variable `research` and assign it to a function `not()` with the input argument being `fake_news`. The function `not()` inverts the Boolean value, exactly like the word not.



Note the different coloring schemes for each value on the variable explorer.



Concatenation (+) and Duplication (*)

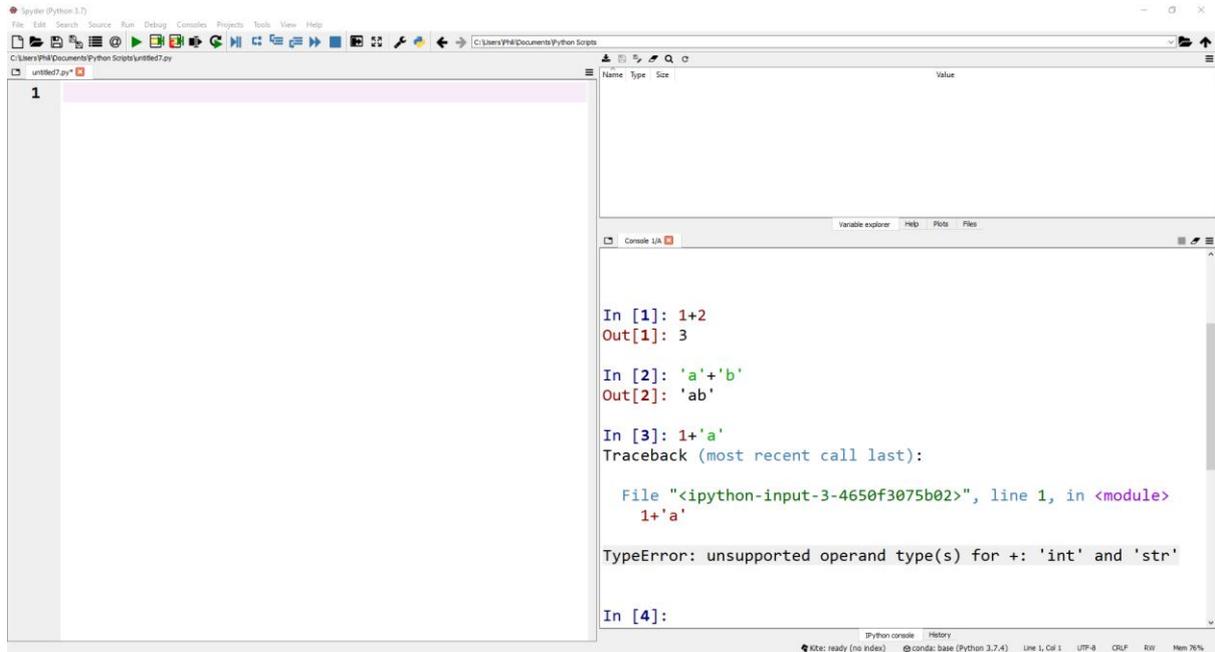
For numeric values the `+` operator adds the 0th variable to the 1st variable. For example:

```
1+2
```

On the other hand when the `+` operator can be used with two strings it concatenates them. For example:

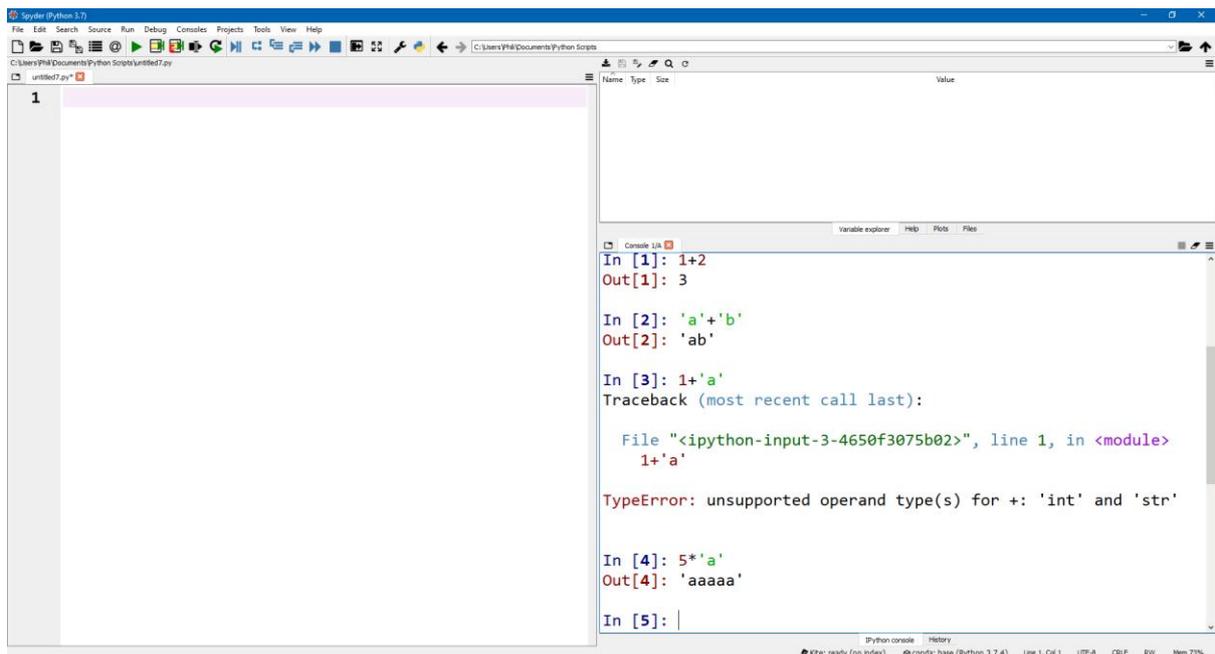
```
'a'+'b'
```

Therefore, it does not make sense to use the `+` operator on a string and a number. Attempting to do so will return an error `TypeError: unsupported operand type(s) for +: 'int' and 'str'`.



An integer can however be used with the `*` on a string. In this case the original string will repeat itself by the scalar used in the multiplication:

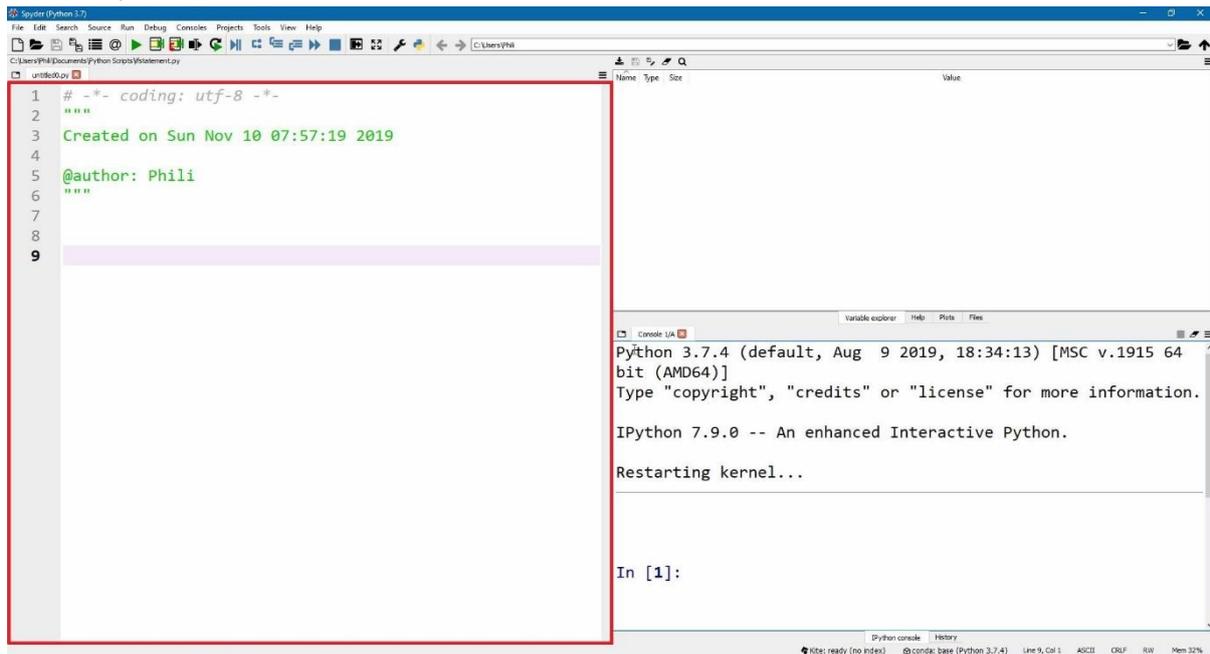
```
5 * 'a'
```



Working with Python Scripts

So far, every piece of code has been typed individually into the console and executed line by line by pressing `[Enter]` and this means when Spyder is closed, or the iPython Kernel is restarted that all data is lost. Instead we can type all of code into a script file.

The Script File



Note to the left-hand side that each line is numbered. The line number is important when it comes to reading code and when Python tells you about an error it will reference the line number. The default script file starts with a line 1 that begins with a # and is all colored grey. The # denotes a comment and anything following the # to the end of the line is a comment and ignored by the IPython console.

```
1. #
```

It is also possible to Comment/Uncomment a line out by highlighting it and selecting Edit then Comment/Uncomment. There is also the shortcut key `Ctrl+1`. It is possible to add a comment to the end of a line but it is far more common to have comments on individual lines.

After this comment line, there is a series of lines with triple quotes. These are used as documentation strings, for the user to read, or reference using help, once again this will be ignored by the IPython console. These will all be colored green.

```
2. """
3.
4.
5.
6. """
```

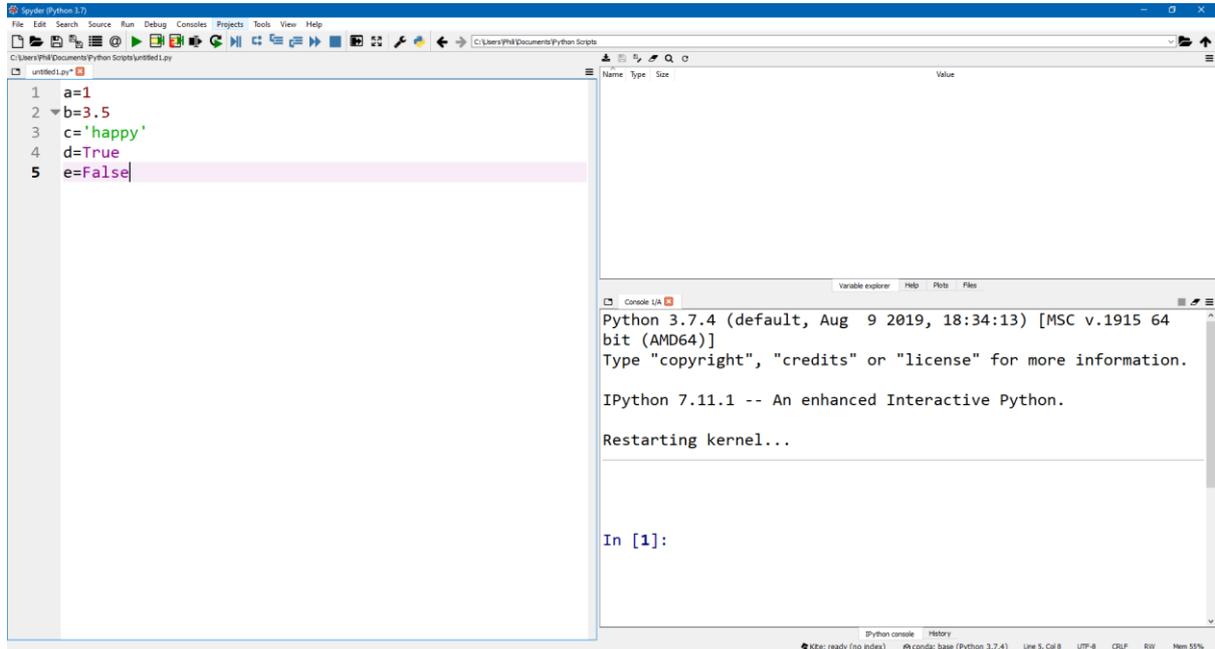
In our basic script, we can remove these comments and we can directly just create a series of variables.

```
1. a=1
2. b=3.5
3. c='happy'
4. d=True
5. e=False
```

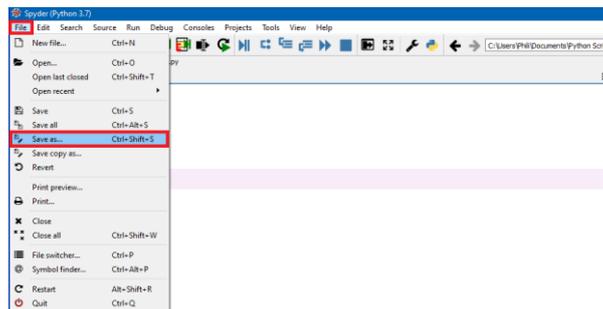
Note that none of these lines of code are executed until the script file is run.

Saving the Script File

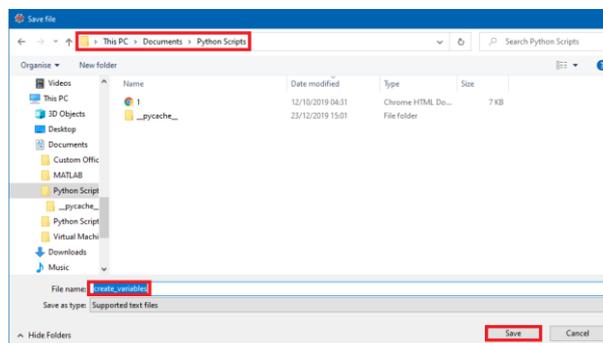
By default our script will be called `untitled1.py` and marked with a `*`. The `*` denotes changes are unsaved.



To save the script to a file, select `File` → `Save As...`



Select your folder to save the script file in and then name the script and select save. In this case I will use the script name `create_variables`. Script names should be named, following the same convention as Variable Names. They will automatically end in an extension `.py`.



The file name of the script will display as a script tab. The currently selected script tabs file location will display above.

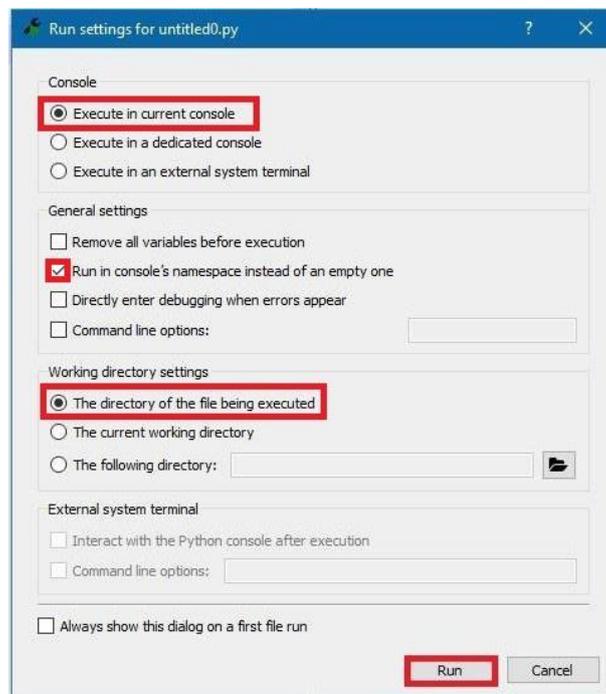
```
1 a=1
2 b=3.5
3 c='happy'
4 d=True
5 e=False
```

Running a Script File

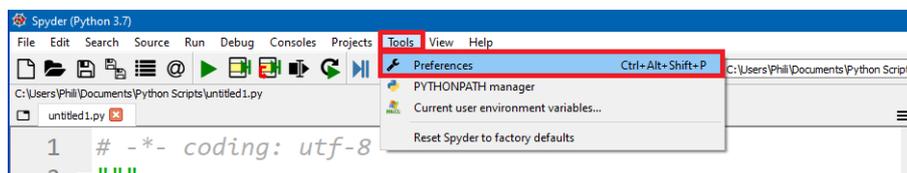
To run the script, select the run button, this also has a shortcut key **F5**.

```
1 a=1
2 b=3.5
3 c='happy'
4 d=True
5 e=False
```

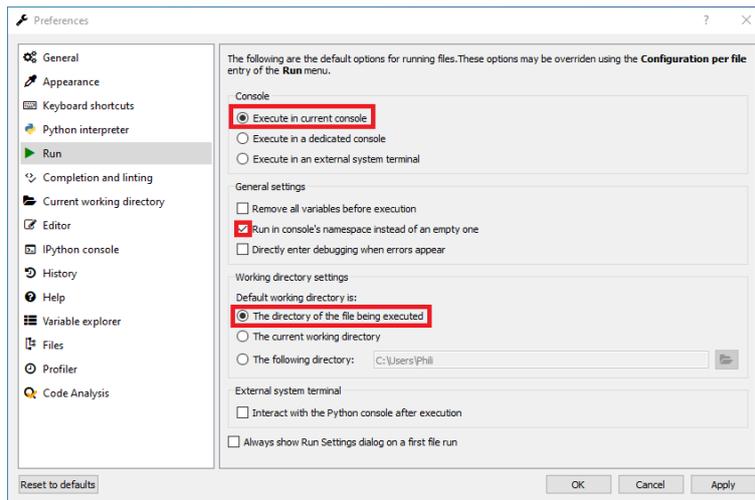
During the first launch of any scripts, you will be prompted for run settings. These can be changed according to your own preferences. In this case I recommend using the following settings:



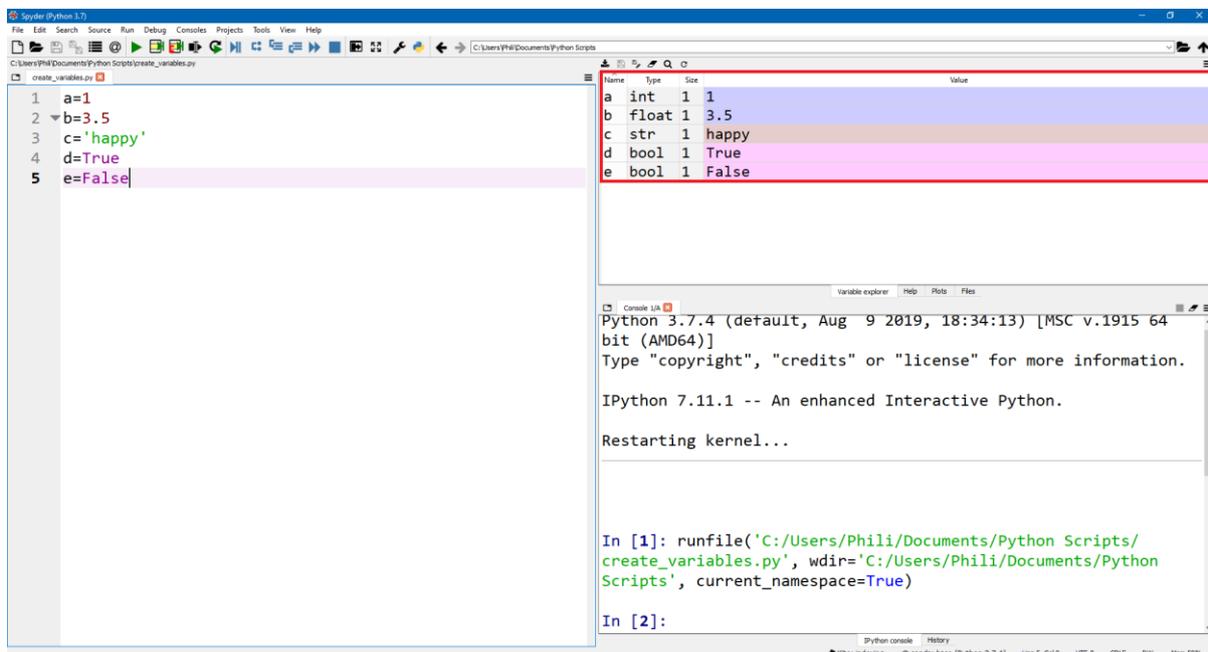
These settings can later be changed by going to Tools → Preferences:



Then selecting your settings and then applying them.



Now the commands in the script are run. Note however that none of the values are printed in the console, like they were when typed in the console and ran line by line individually. The code is executed however, and the variables display in the variable explorer.

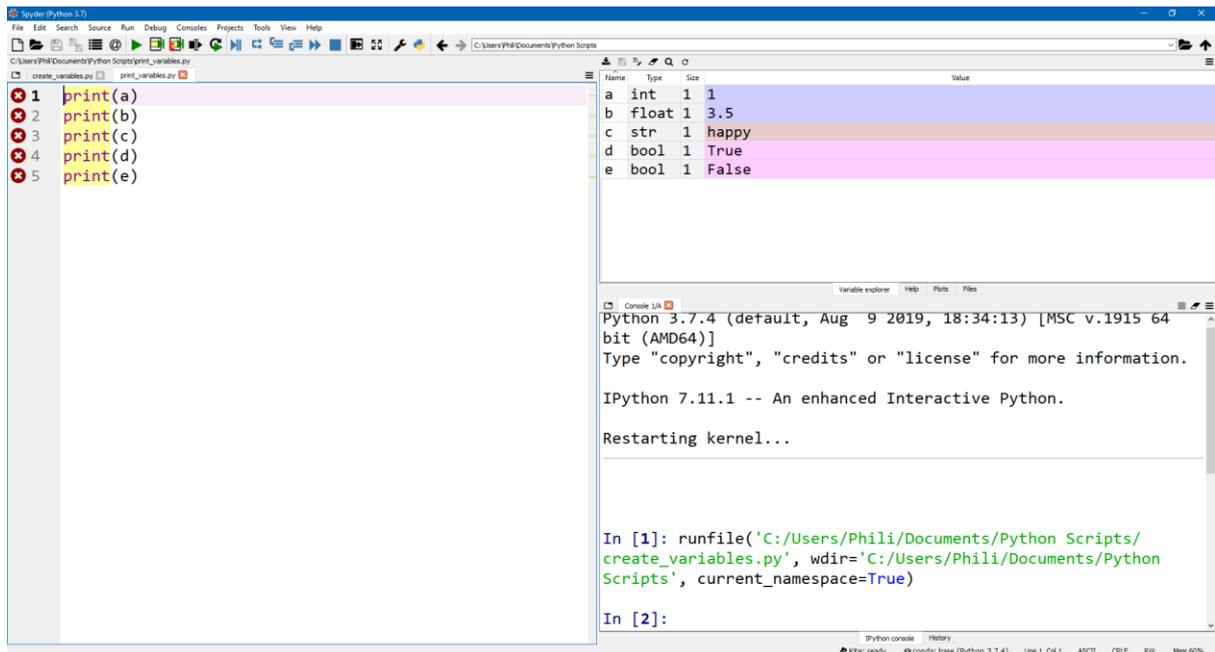


The `print` Function - Printing Variables to the Console

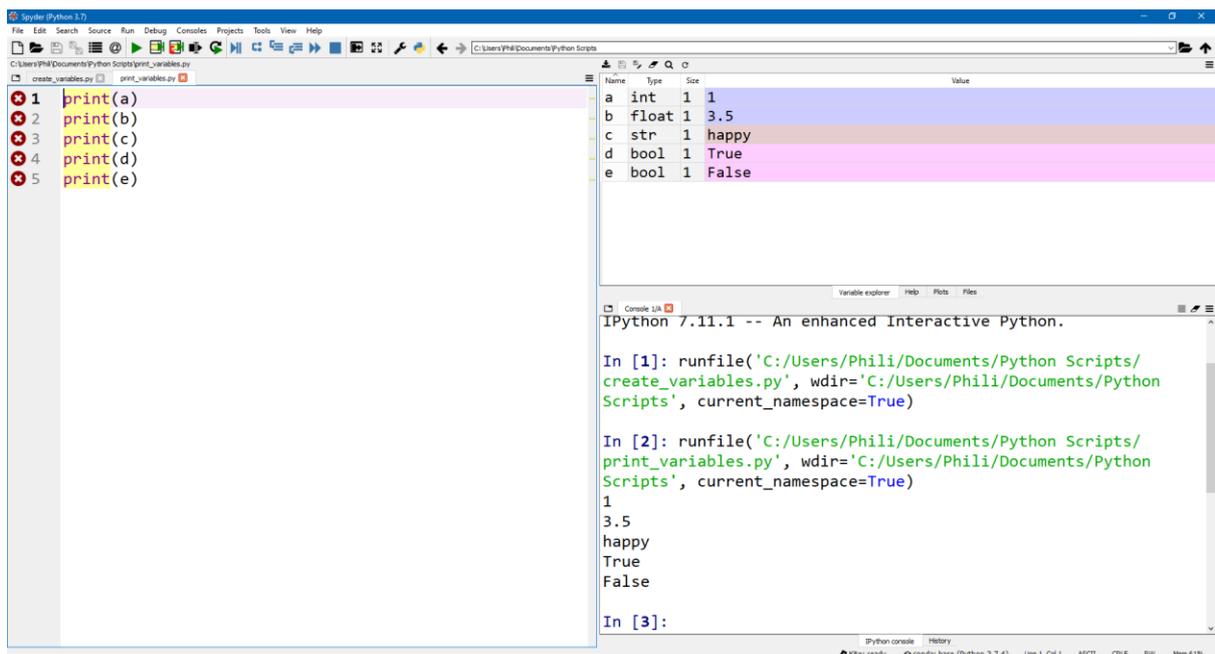
To print a variable to a console we can use the `print()` function. The variable to be printed is enclosed in the parenthesis () which denote that the variable is the input argument to the `print()` function.

By default, Spyder assumes, the setting Run in console's namespace instead of an empty one is unchecked. So, if a second script file `print_variables` is created with the commands they will all be marked with an error, because they are not created in the same file:

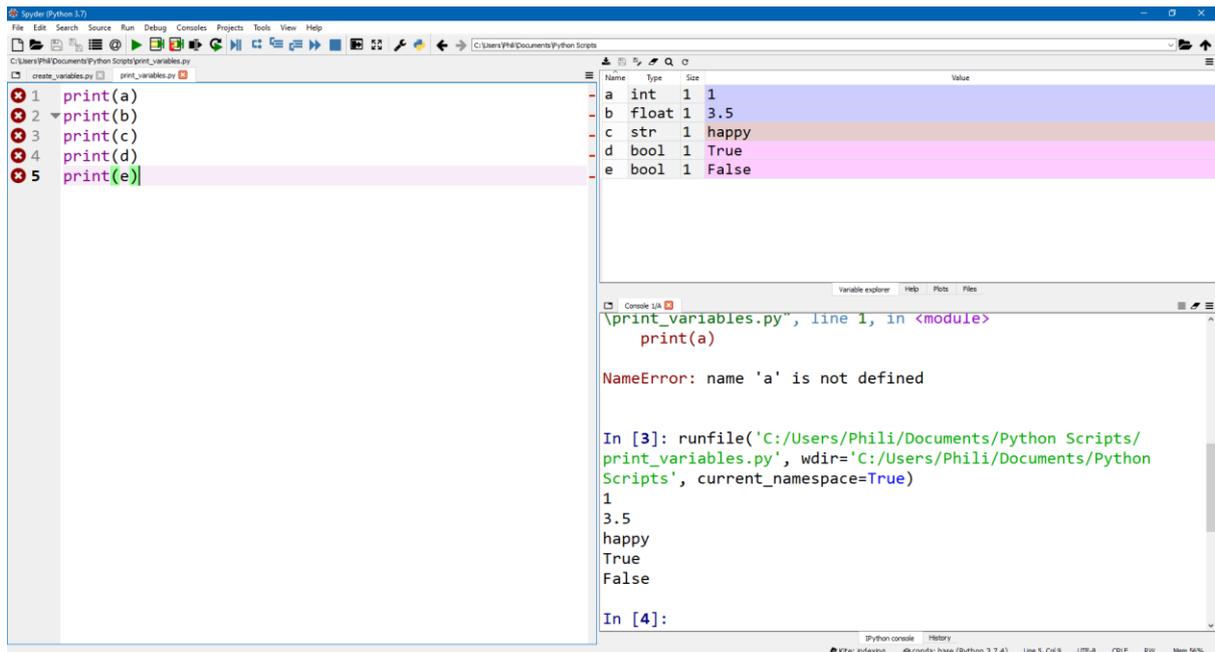
1. `print(a)`
2. `print(b)`
3. `print(c)`
4. `print(d)`
5. `print(e)`



However, this script runs fine because the script `create_variables` was previously executed meaning the five variables to be printed are in the name space already.



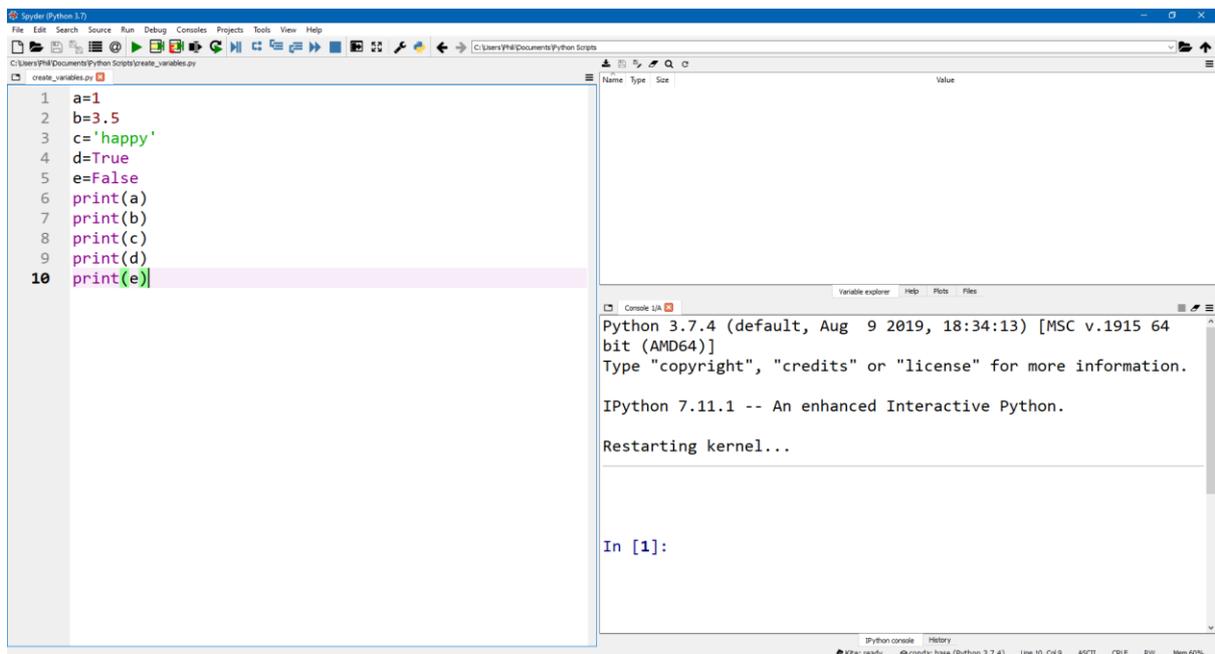
If the setting Run in console's namespace instead of an empty one is unchecked or if `create_variables` is not ran before `print_variables` the following error will occur **NameError: name 'a' is not defined** in line 1. The script will stop executing on line 1 where it found the error (so only this error will be mentioned).



Care should be taken when using a script to generate variables or data and then using a subsequent script to process or analyse variables because errors will be flagged up in the manner shown above. For this reason Spyder usually unchecks the setting Run in console's namespace instead of an empty one is unchecked however this can cause confusion for users beginning who see variables populated in the variable explorer window and can access them by typing commands in the console but cannot access them in a separate script.

Comments

If we combine the two script files above, we will see that there are no errors marked on lines 6-10 because the variables are assigned earlier on in the same script file.



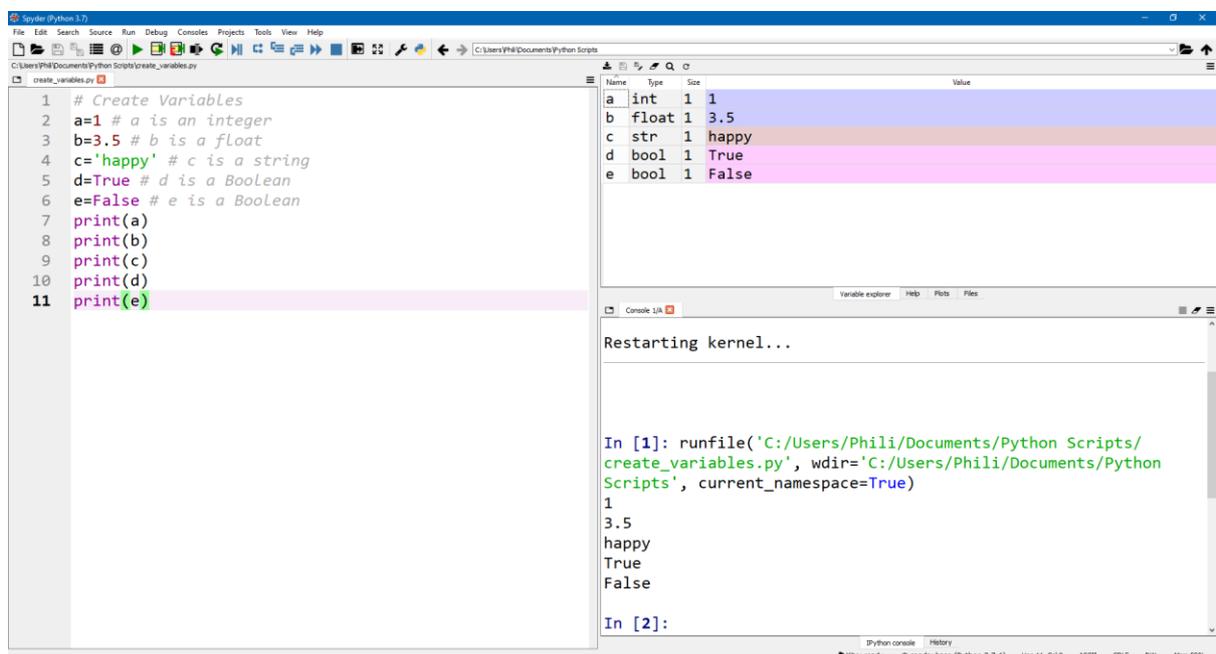
We can add comments to this script by using a `#` and then typing in our comment. These can be applied to full lines or at the end of a line:

```

1. # Create Variables
2. a=1 # a is an integer
3. b=3.5 # b is a float
4. c='happy' # c is a string
5. d=True # d is a Boolean
6. e=False # e is a Boolean
7. print(a)
8. print(b)
9. print(c)
10. print(d)
11. print(e)

```

Note that the comments are all greyed out and are ignored by Python, they are only in place to help the user when reading the script file.



Section Break

Sections can be made using the section break. The name of the section is a comment ignored by Python.

```
# %% comment
```

We can update our code above and split it into 2 sections.

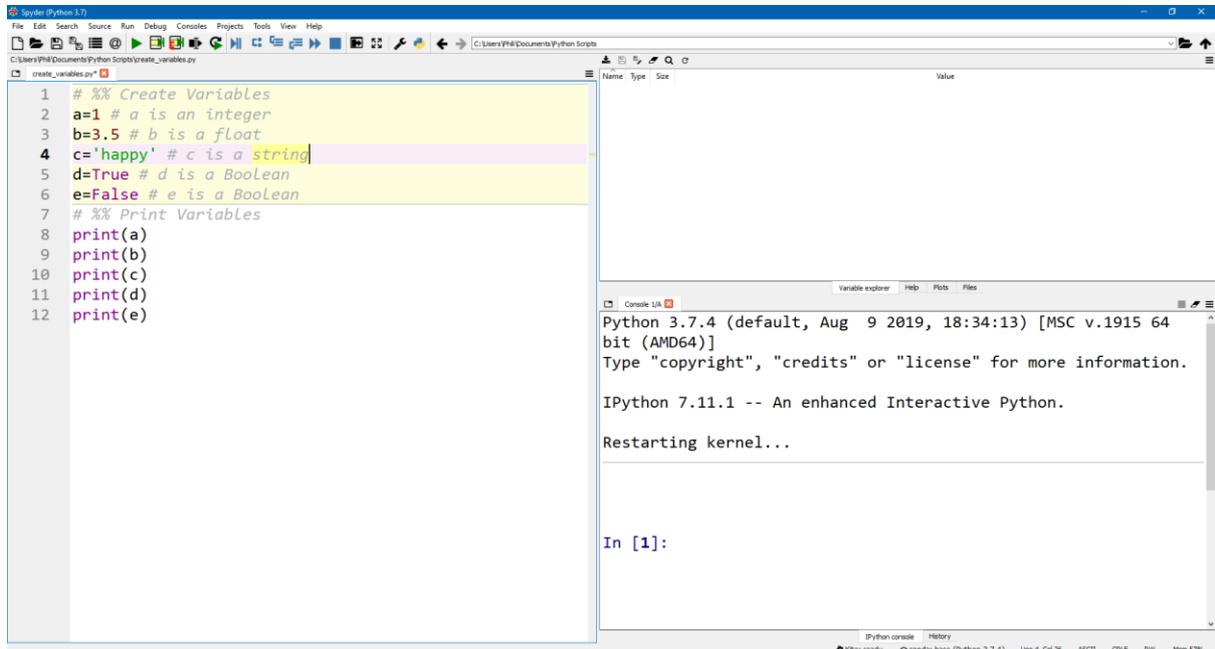
```

1. # %% Create Variables
2. a=1 # a is an integer
3. b=3.5 # b is a float
4. c='happy' # c is a string
5. d=True # d is a Boolean
6. e=False # e is a Boolean
7. # %% Print Variables
8. print(a)
9. print(b)

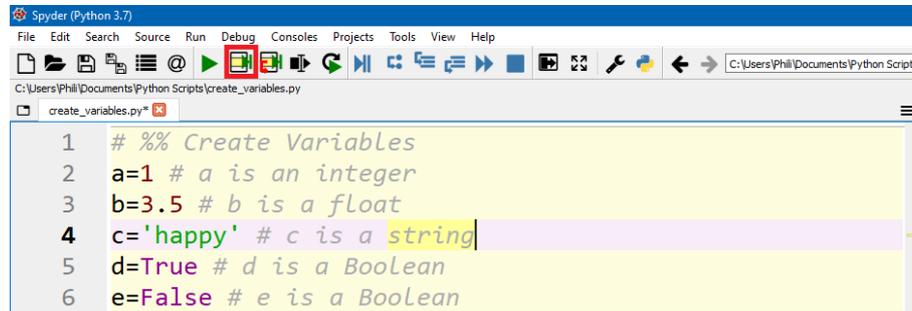
```

10. `print(c)`
11. `print(d)`
12. `print(e)`

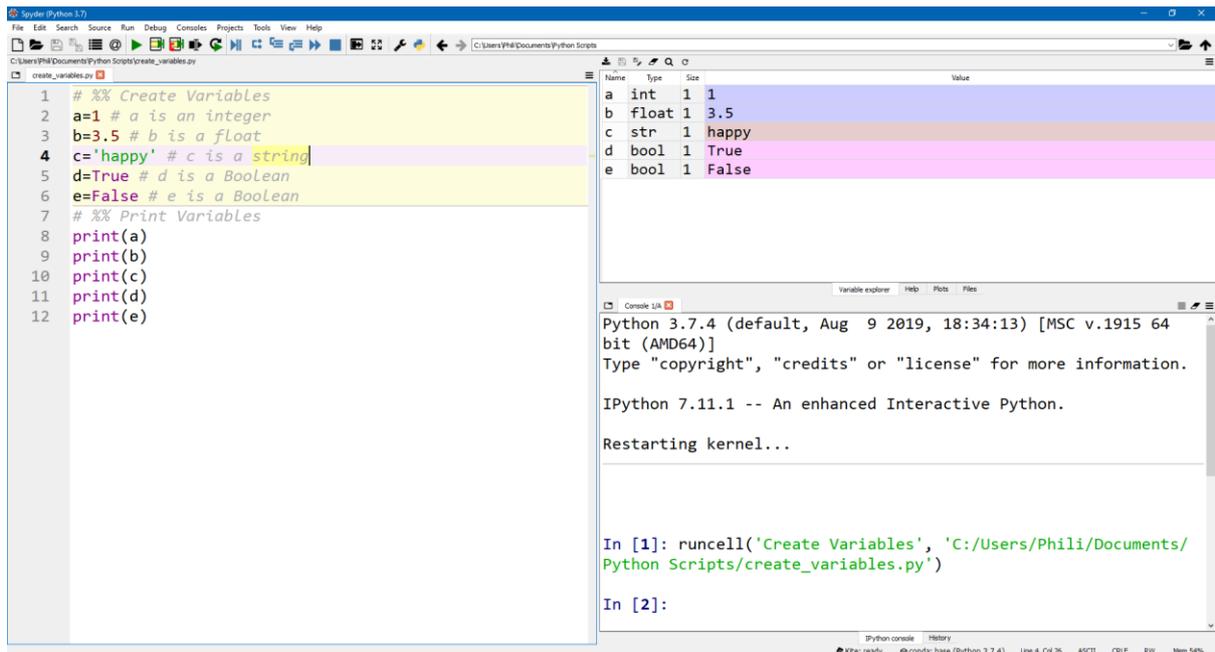
Note how the currently selected section is highlighted in yellow.



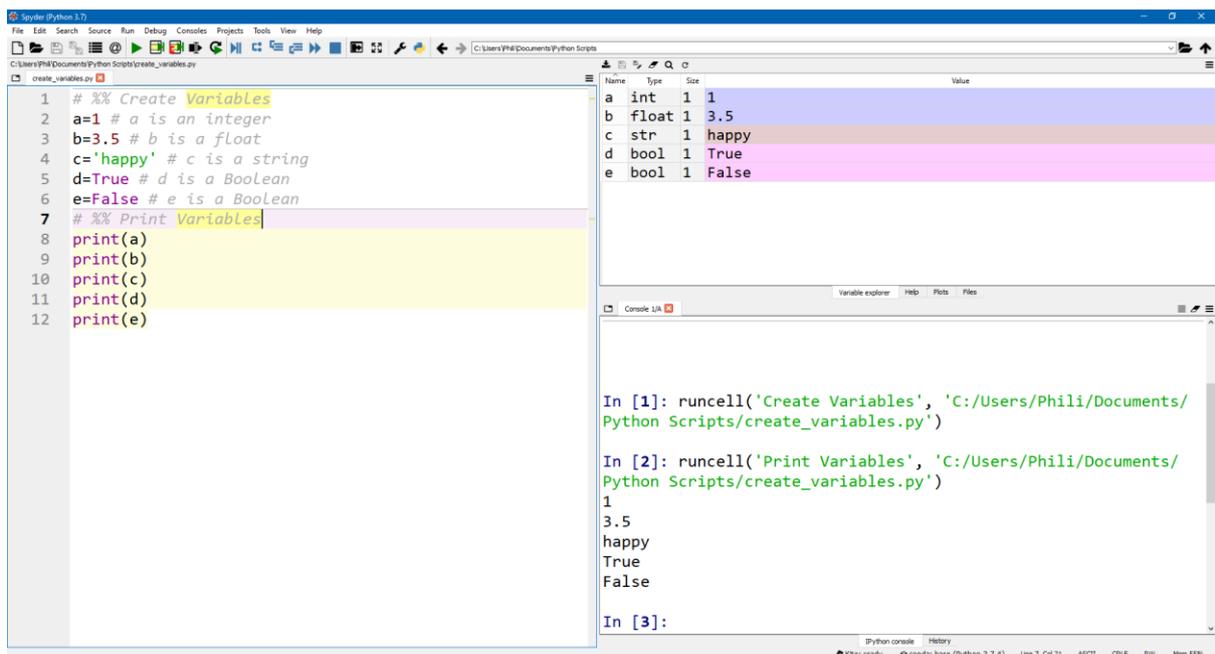
The code can be run section by section by selecting the run section button:



In this case the first section will be ran:



And then we can run the second section:



Note in the case of this script, if the second section is ran before the first section, an error will be generated `NameError: name 'a' is not defined` in line 8. We have seen this before.

```

1  # %% Create Variables
2  a=1 # a is an integer
3  b=3.5 # b is a float
4  c='happy' # c is a string
5  d=True # d is a Boolean
6  e=False # e is a Boolean
7  # %% Print Variables
8  print(a)
9  print(b)
10 print(c)
11 print(d)
12 print(e)

```

```

In [1]: runcell('Print Variables', 'C:/Users/Phili/Documents/Python Scripts/create_variables.py')
Traceback (most recent call last):

  File "C:/Users/Phili/Documents/Python Scripts/create_variables.py", line 8, in <module>
    print(a)
NameError: name 'a' is not defined

In [2]:

```

Formatted String – Strings and Variables

Strings can also be typed directly inside the `print()` function, recall that these are enclosed in `' '`. Once a string ends we can follow it using a comma `,` and then call a variable name. We can then follow this with another comma `,` and another string. We can repeat this procedure until we have built up the sentence including the variables that we desire.

1. `a=1`
2. `b=3.5`
3. `c='happy'`
4. `d=True`
5. `e=False`
6. `print('my first number is ',a,' and my second is ',b, '.')`

```

1  a=1
2  b=3.5
3  c='happy'
4  d=True
5  e=False
6  print('my first number is ',a,' and my second is ',b, '.')

```

Name	Type	Size	Value
a	int	1	1
b	float	1	3.5
c	str	1	happy
d	bool	1	True
e	bool	1	False

```

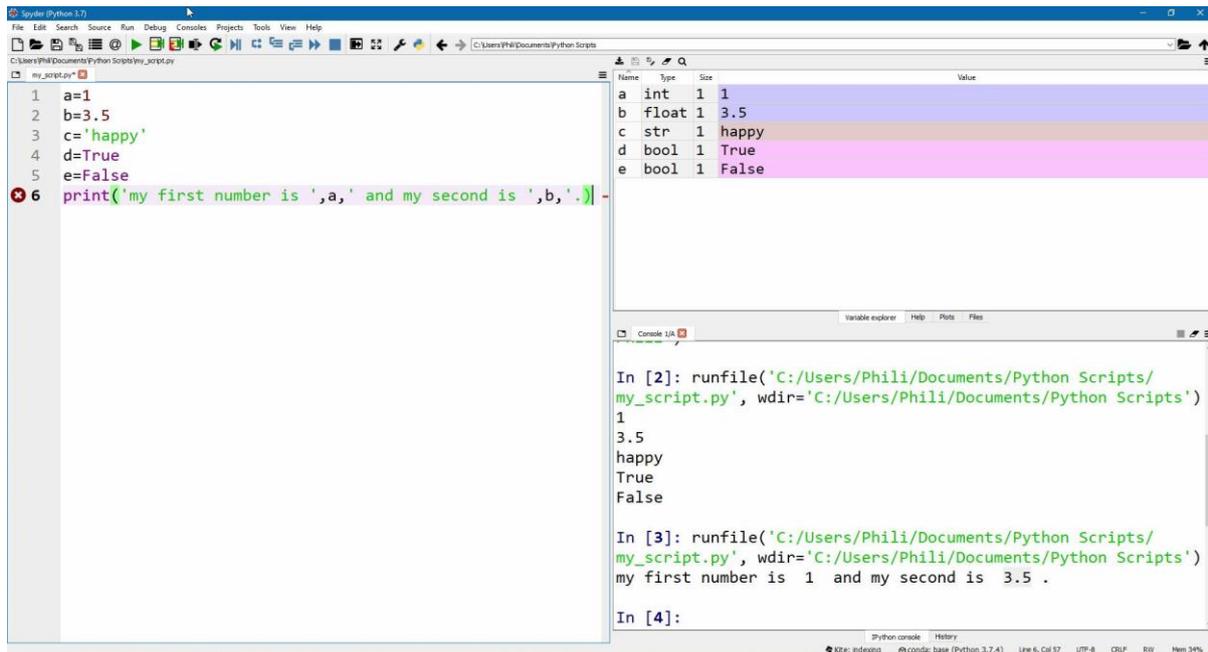
In [2]: runfile('C:/Users/Phili/Documents/Python Scripts/my_script.py', wdir='C:/Users/Phili/Documents/Python Scripts')
1
3.5
happy
True
False

In [3]: runfile('C:/Users/Phili/Documents/Python Scripts/my_script.py', wdir='C:/Users/Phili/Documents/Python Scripts')
my first number is 1 and my second is 3.5 .

In [4]:

```

If we mess up the code for example by forgetting the `'` in front of the `)`, there is a good chance that the spyder IDE will detect an error and display a warning sign on the line.

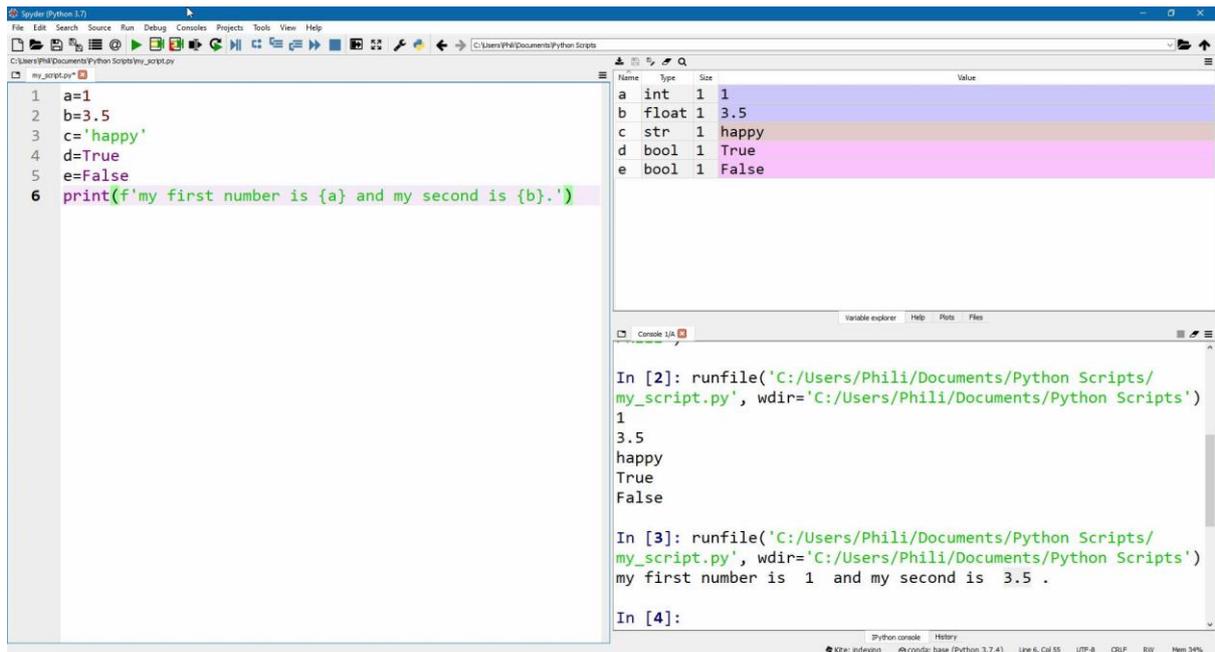


Typing text with variables like above can become a bit difficult to follow, so Python also has the ability to input formatted strings into the console. A formatted string begins with a `f` and any variable name is enclosed in a `{}`.

```
1. a=1
2. b=3.5
3. c='happy'
4. d=True
5. e=False
6. print(f'my first number is {a} and my second is {b}.')
```

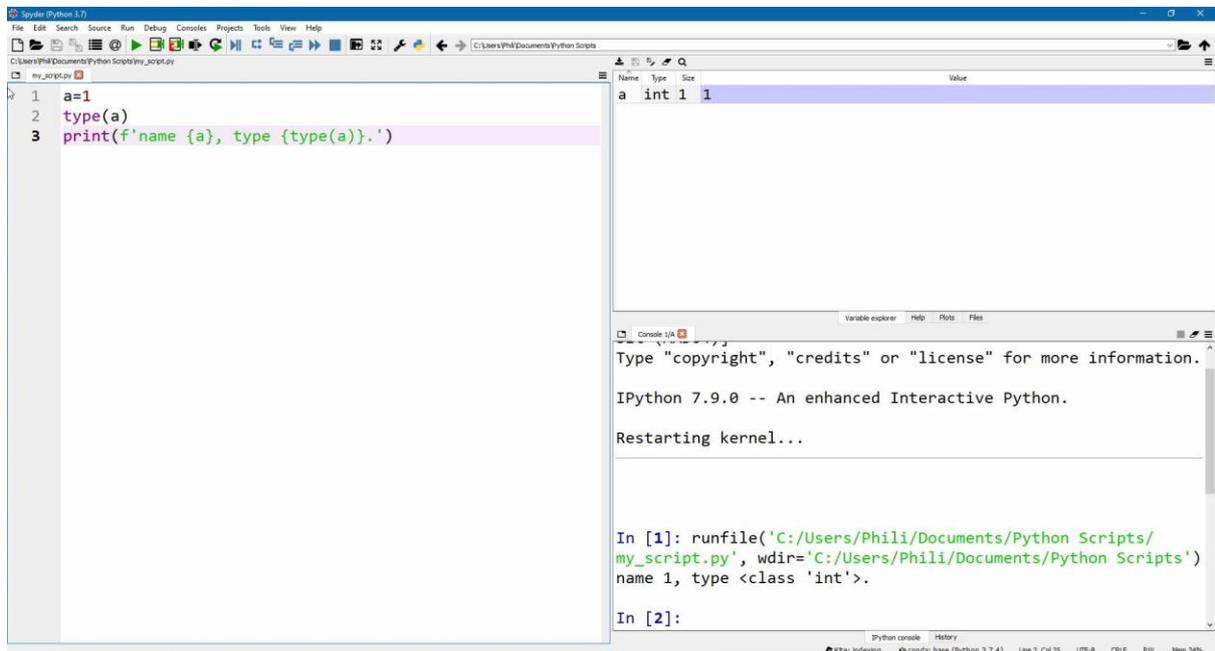
Formatted strings are relatively new and the Spyder IDE doesn't color code the variables as black, they instead remain green.

```
print(f'my first number is {a} and my second is {b}.')
```



The function `type()` can also be used to determine the variable type where the input argument is the variable name being examined. In this case we use the function in line 2 however because the right hand side is not assigned to the variable name, it does not display in the variable explorer as a new variable and because it is not within a print command it does not print to the iPython console. In line 3, it is embedded in a formatted string within the `print()` command so does display.

1. `a=1`
2. `type(a)`
3. `print(f'name {a}, type {type(a)}')`

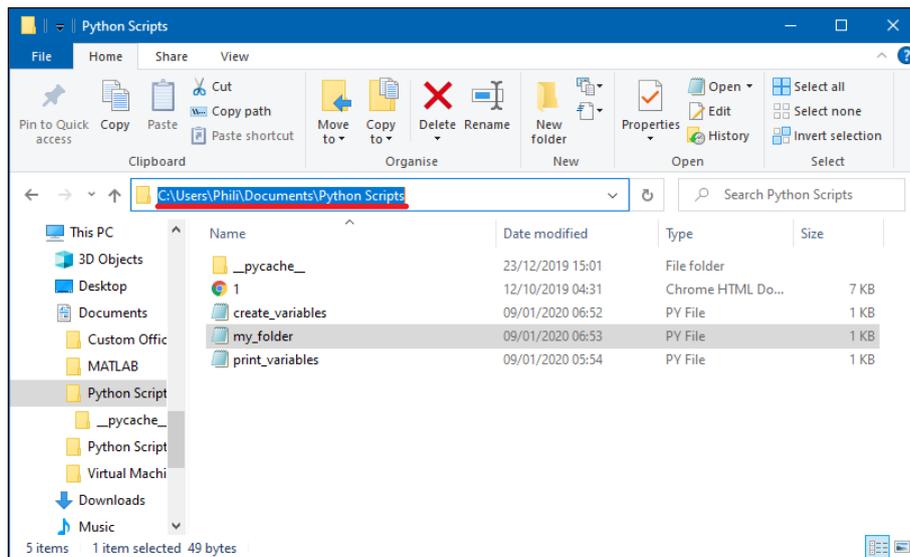


Relative String – Relative File Paths

The folder where I created the last script file has the location:

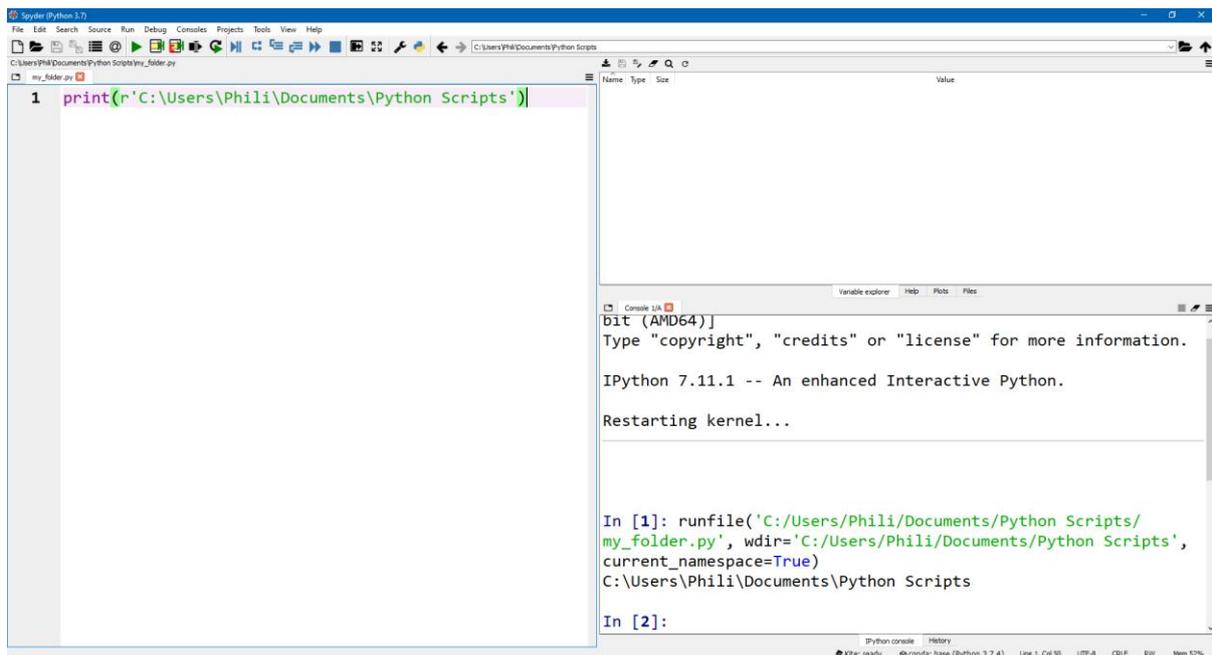
```
'C:\Users\Phili\Documents\Python Scripts'
```

It can be copied and pasted from windows explorer:

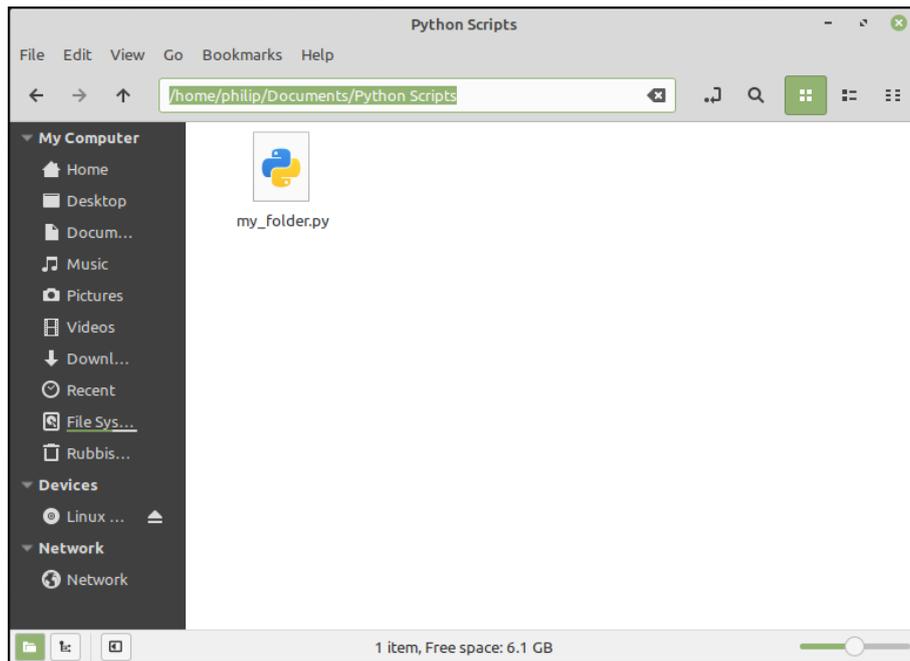


However, as we know `\` is a special character used for `\n` (new line) and `\t` (tab) and `\"` (quotation) for example. As these are commonly used, we have a relative string which has a similar form to a formatted string, except we use `r` and the rest of the input argument is the file path in quotations.

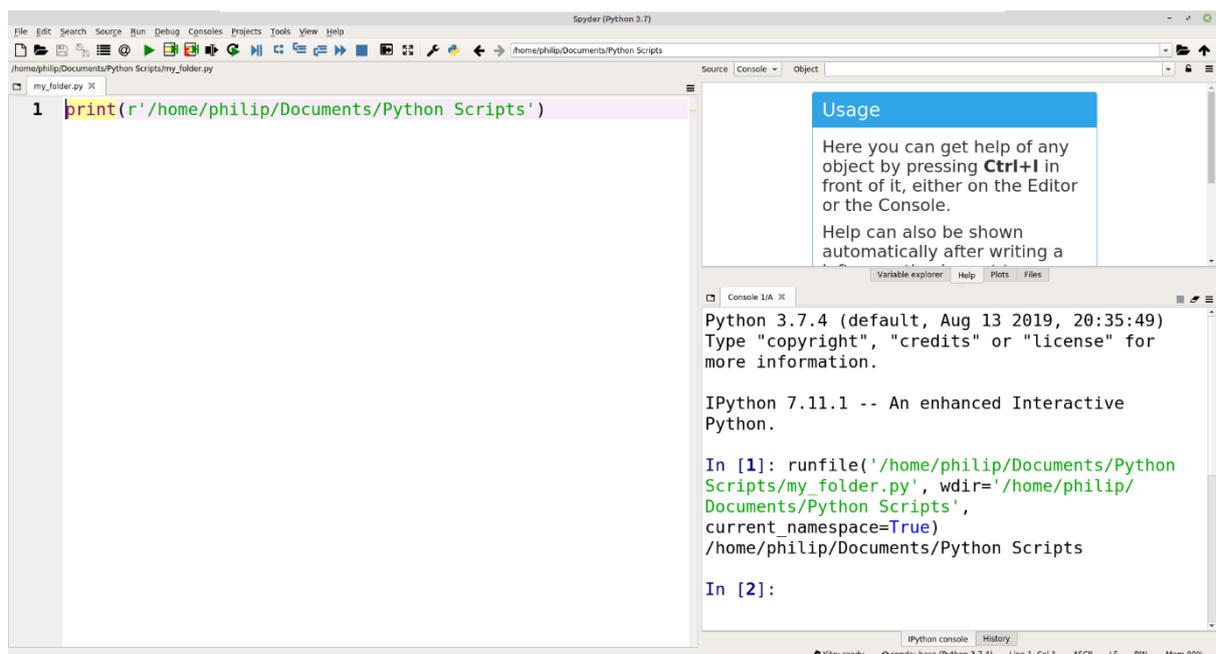
```
print(r'C:\Users\Phili\Documents\Python Scripts')
```



In Linux the folder structure also uses `/` opposed to `\` but it can still be accessed using a formatted string.



```
print(r'/home/philip/Documents/Python Scripts')
```



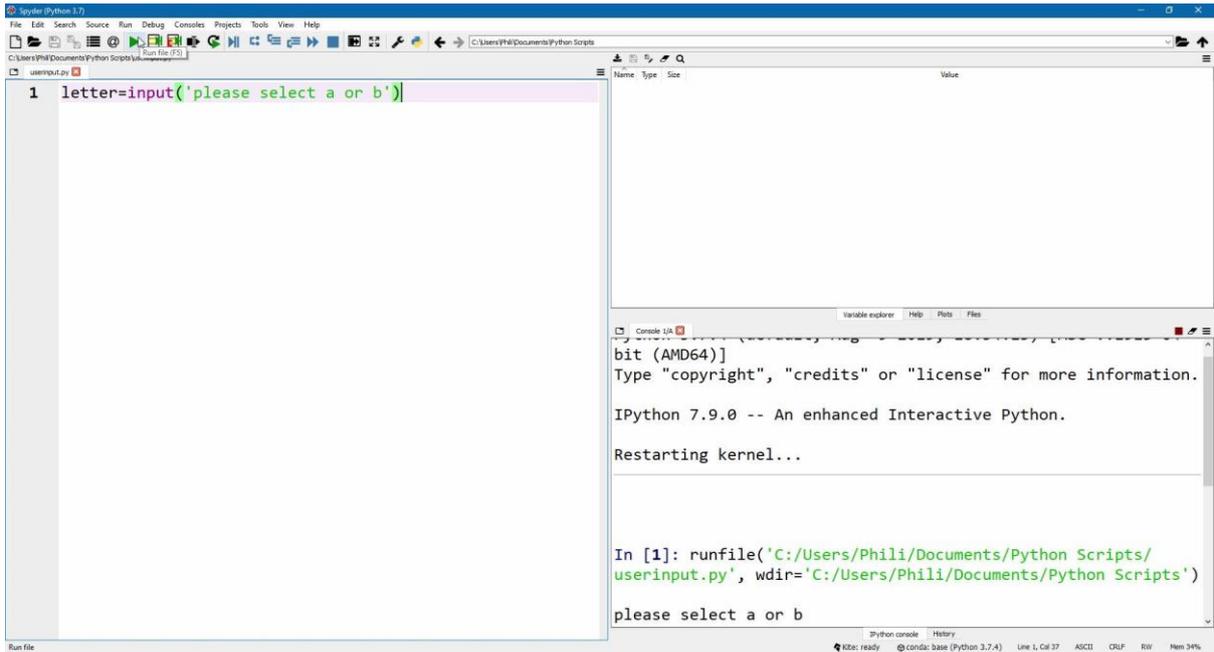
The `input` Function – Gathering Input from the Console

The function `input` can be used to gather input from the user. The input argument is a string of text known as a prompt and it may be assigned to an output variable using the assignment operator.

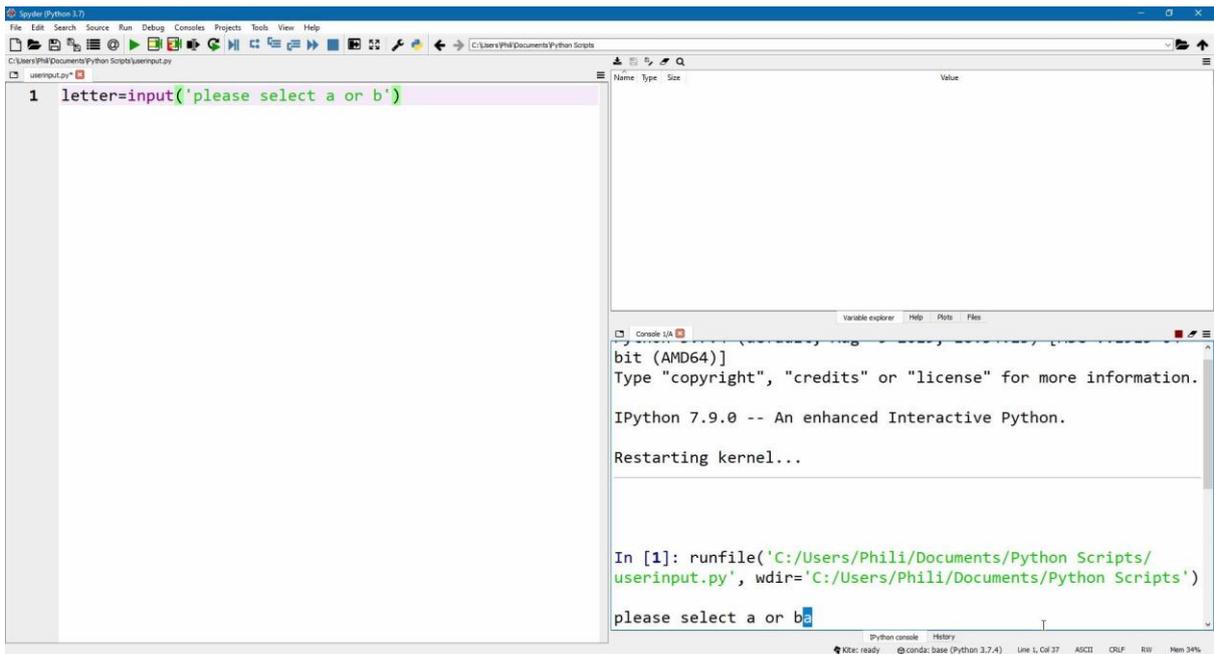
```
variable=input('input dialogue')
```

Let's create the input dialogue.

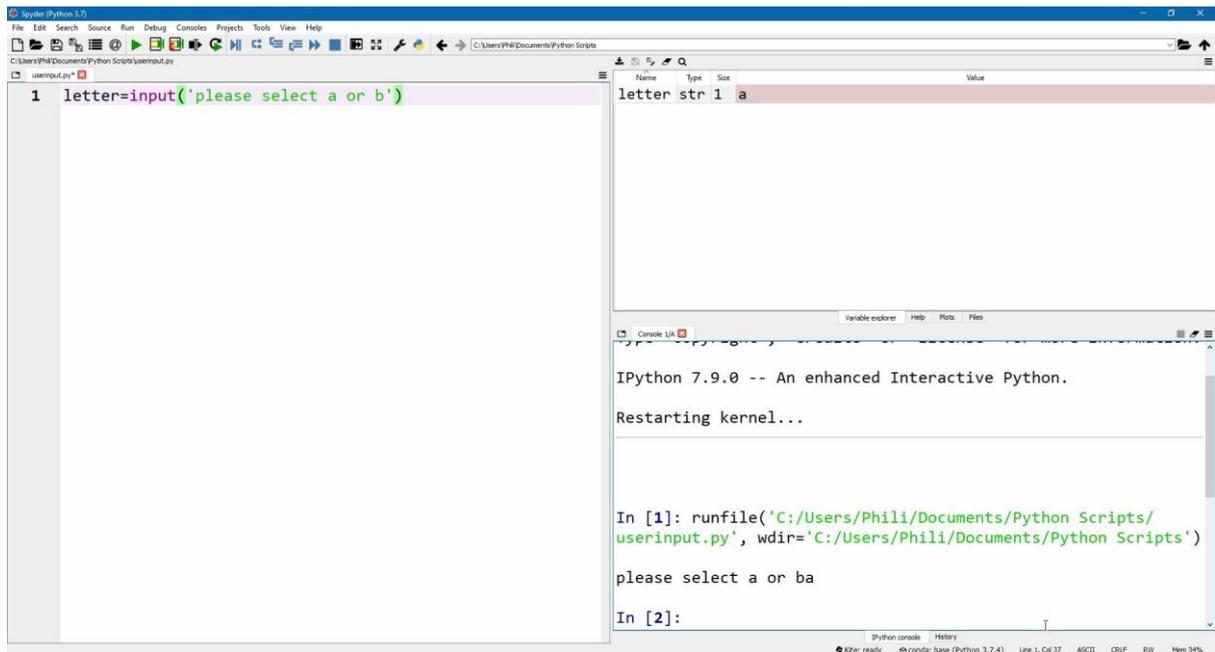
```
letter=input('please select a or b')
```



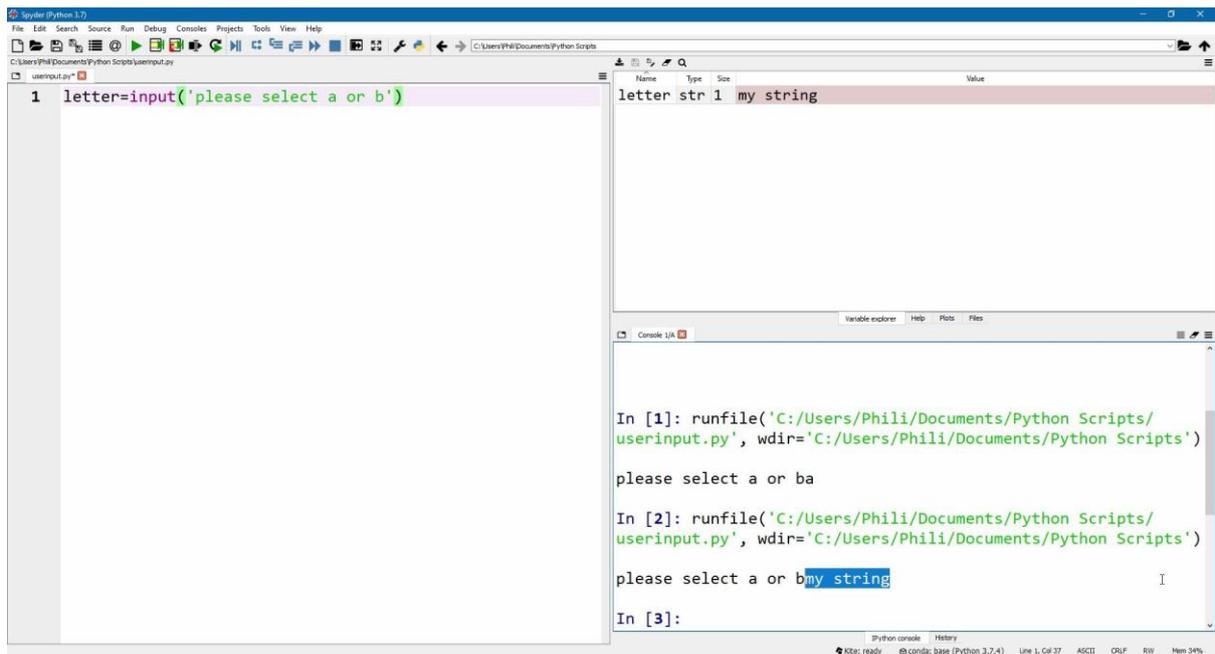
Note however when the user inputs their value it is right up against the prompt, to amend this a space should be placed at the end of the prompt.



With the selection of `a`, the function creates an output that is a string `'a'` and in this case is assigned to the variable name `letter`.



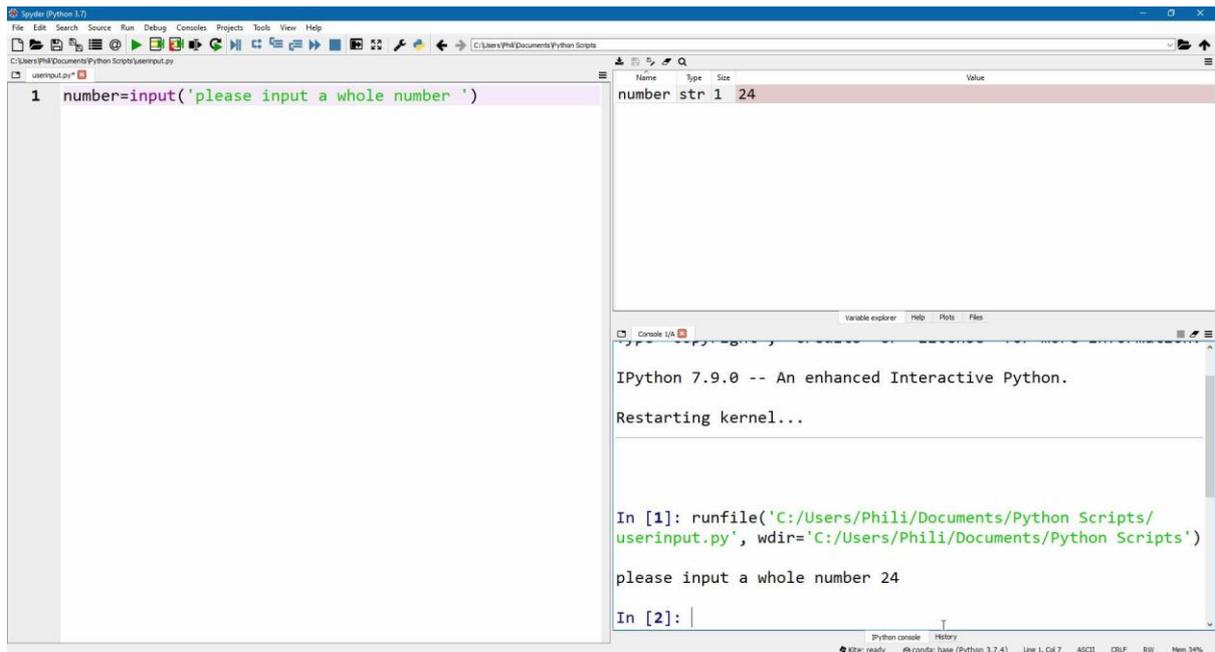
Although the instructions prompted the user for a letter, the user can of course type in whatever they want and whatever the user types in will be stored as a string.



This means if we ask for a number input:

```
number=input('please input a whole number')
```

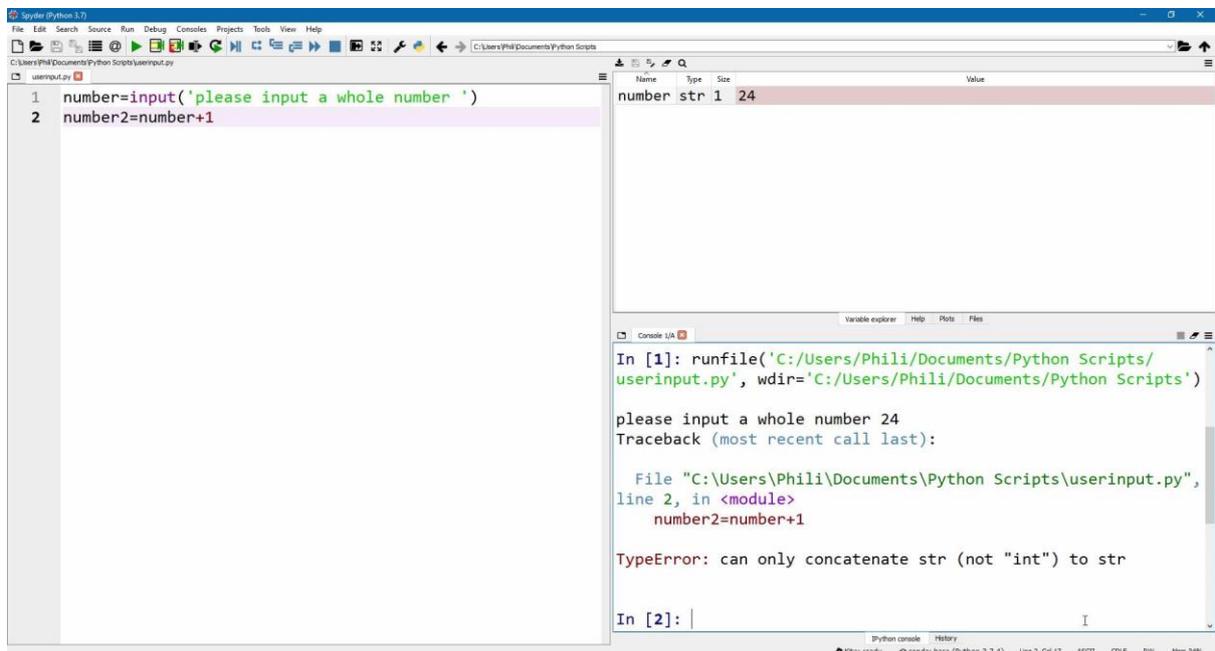
And the number 24 is input, then the output argument will be assigned to the string '24'.



This will be problematic if this string is used with a numerical calculation later on for instance:

```
1. number=input('please input a whole number')
2. number2=number+1
```

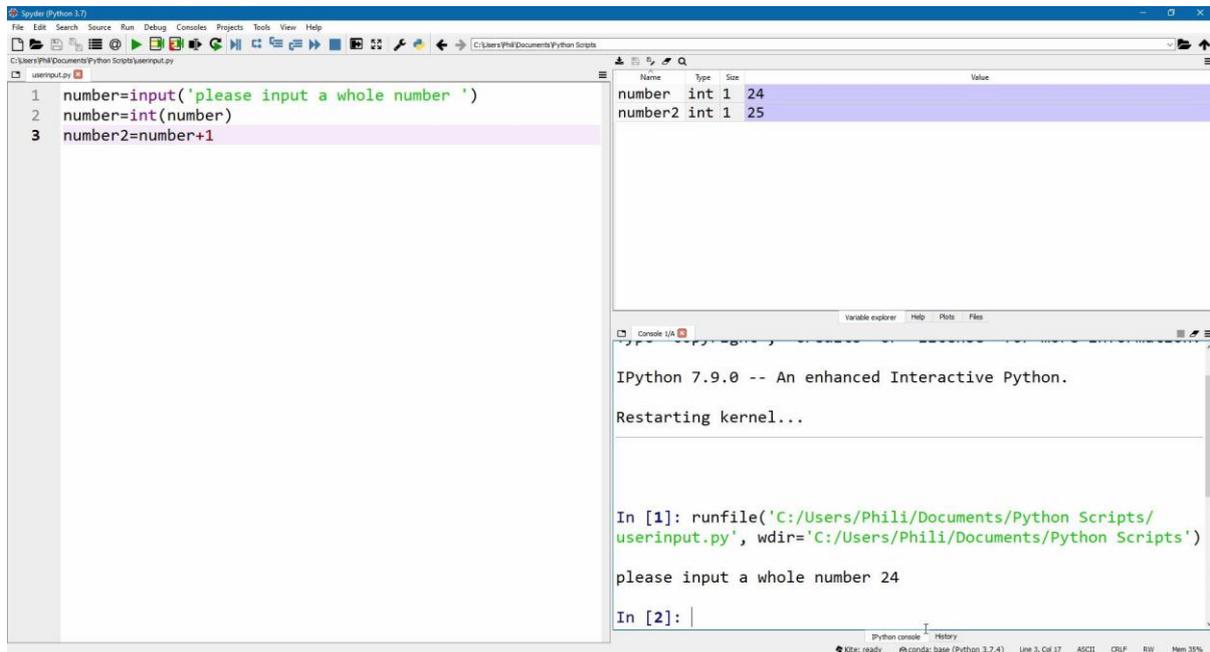
in this case when a mathematical expression is used with this string, the error message `TypeError: can only concatenate str (not "int") to str` appears.



To get around this, the function `int` can be used to convert the string of a number into an integer, that is a whole number. Alternatively the function `float` can be used to convert the string of a number to a floating point number, that is a number with a decimal place.

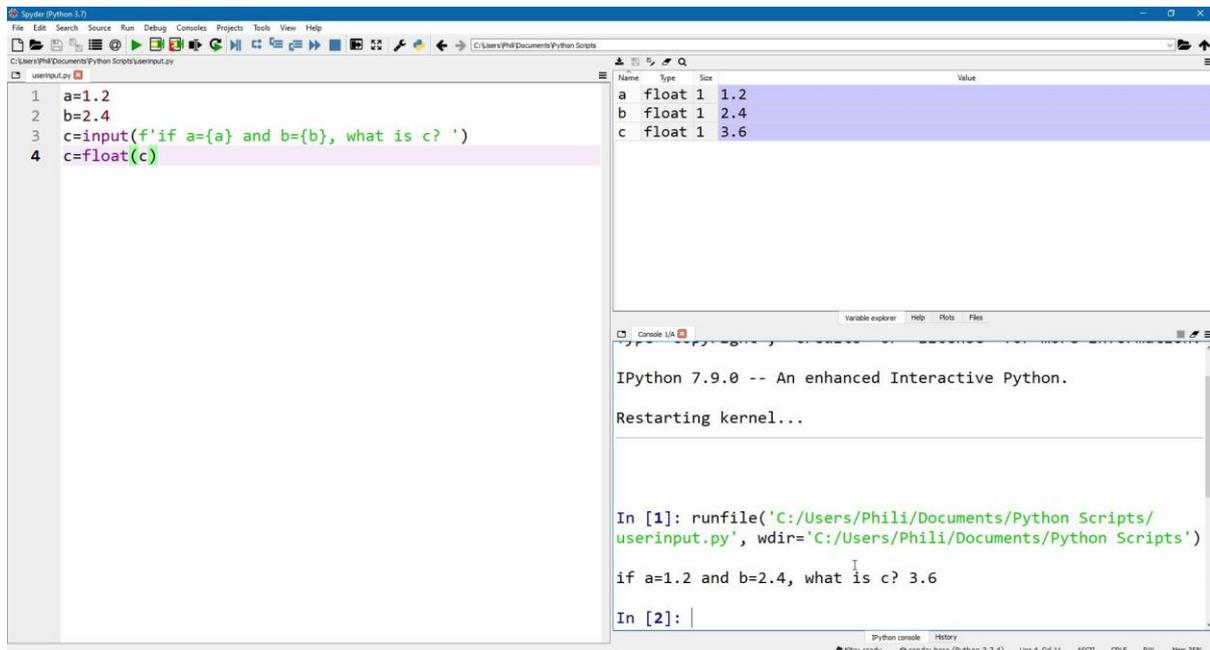
```
1. number=input('please input a whole number')
2. number=int(number)
3. number2=number+1
```

In line 2, the input argument to the function `int` is `number` which is a string however once this function has been carried out, the value which is now an integer is reassigned to the variable name `number`.



Formatted strings can also be used as input arguments for example:

1. `a=1.2`
2. `b=2.4`
3. `c=input(f'if a={a} and b={b}, what is c? ')`
4. `c=float(c)`



Collections

Lists

A Python list has the same sort of form as a shopping list where we list a collection of strings to remind us what to buy when we go shopping.

- apples
- bananas
- grapes
- oranges
- pears

In Python to enclose a list we use square brackets `[]`. For example:

```
1. empty_list=[]
```

We can add the first item to our list as a string:

```
1. shop_list=['apples']
```

To add the next item to our list we need to separate it out from the first, to do this we need a character as a delimiter or separator and in Python we use the comma `,` as the delimiter.

```
1. shop_list=['apples', 'bananas']
```

If we like, following the delimiter we can also press [Enter] to begin the next item on a new line. The Spyder IDE will know that you are continuing a list and will automatically indent the next line.

```
1. shop_list=['apples',  
2.           'bananas']
```

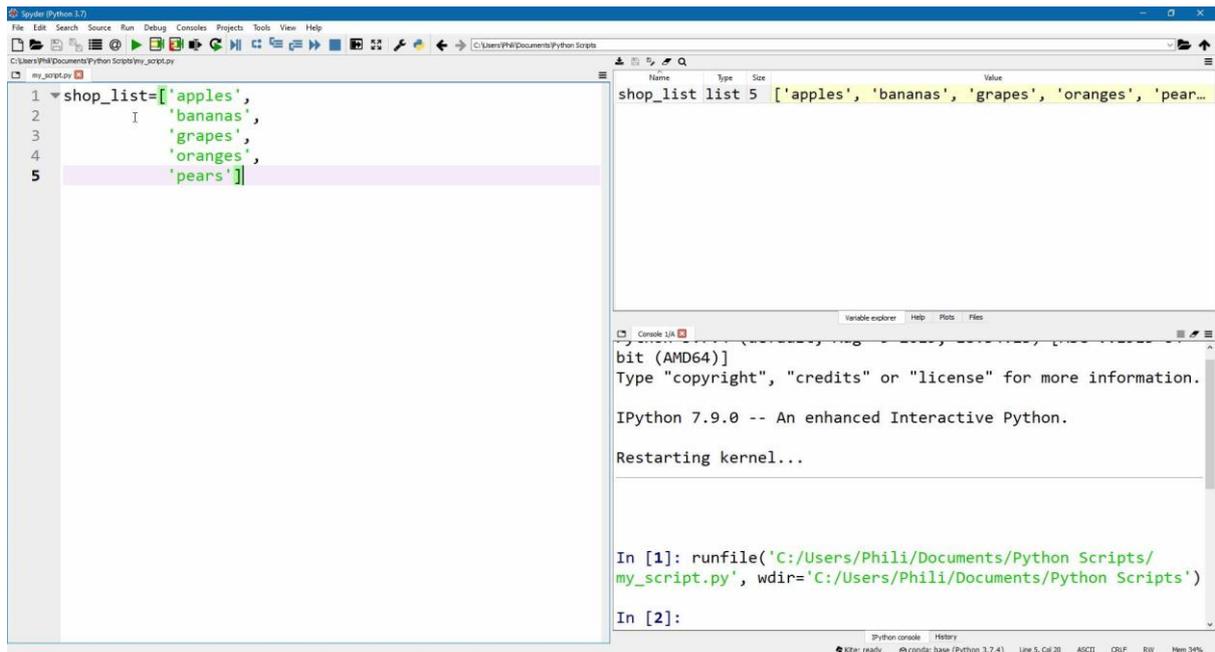
We repeat the process until we have added everything within our list. For example:

```
1. shop_list=['apples', 'bananas', 'grapes', 'oranges', 'pears']
```

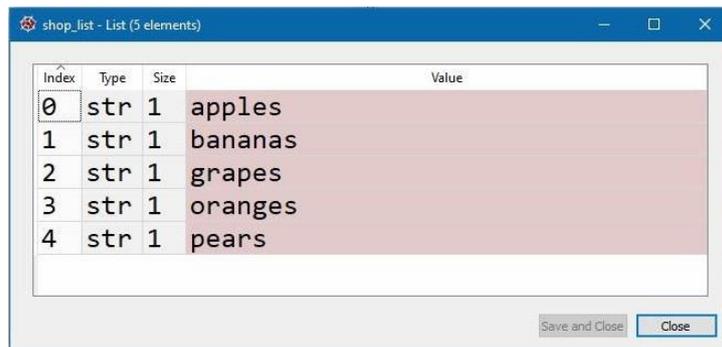
Or as individual lines:

```
1. shop_list=['apples',  
2.           'bananas',  
3.           'grapes',  
4.           'oranges',  
5.           'pears']
```

Once created and the script ran, the new `shop_list` variable will display in the variable explorer.



If it is double clicked it will open in its own window

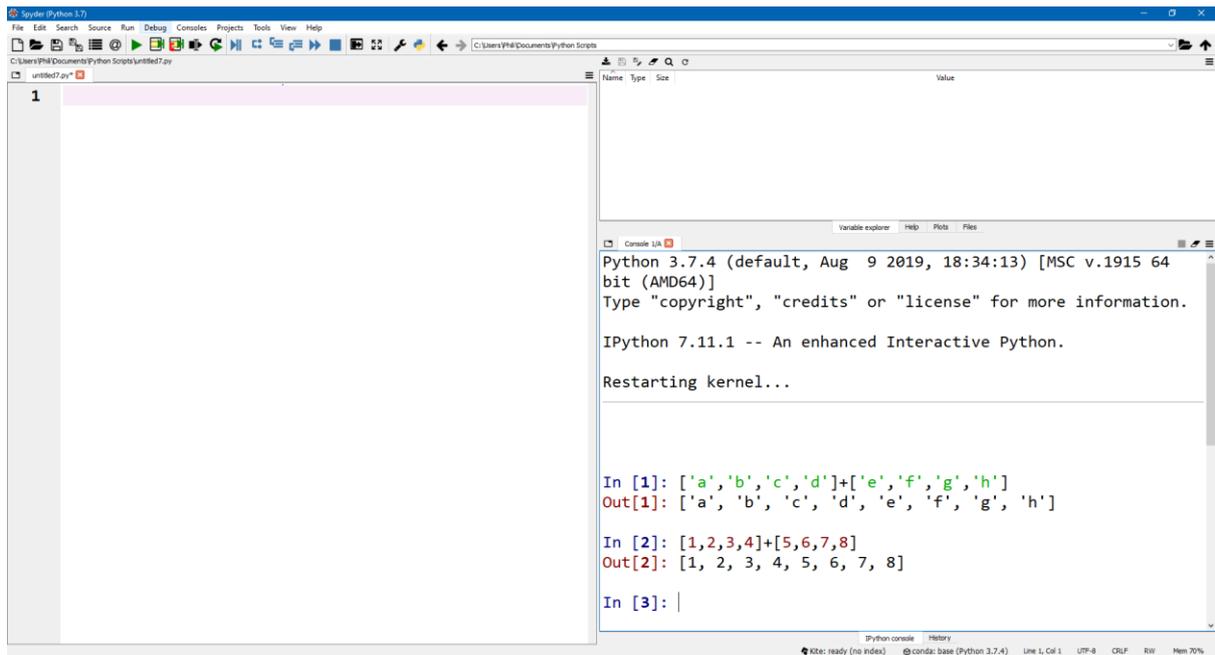


Concatenation (+) and Duplication (*)

When the + operator is used on two lists, they will be concatenated. For example:

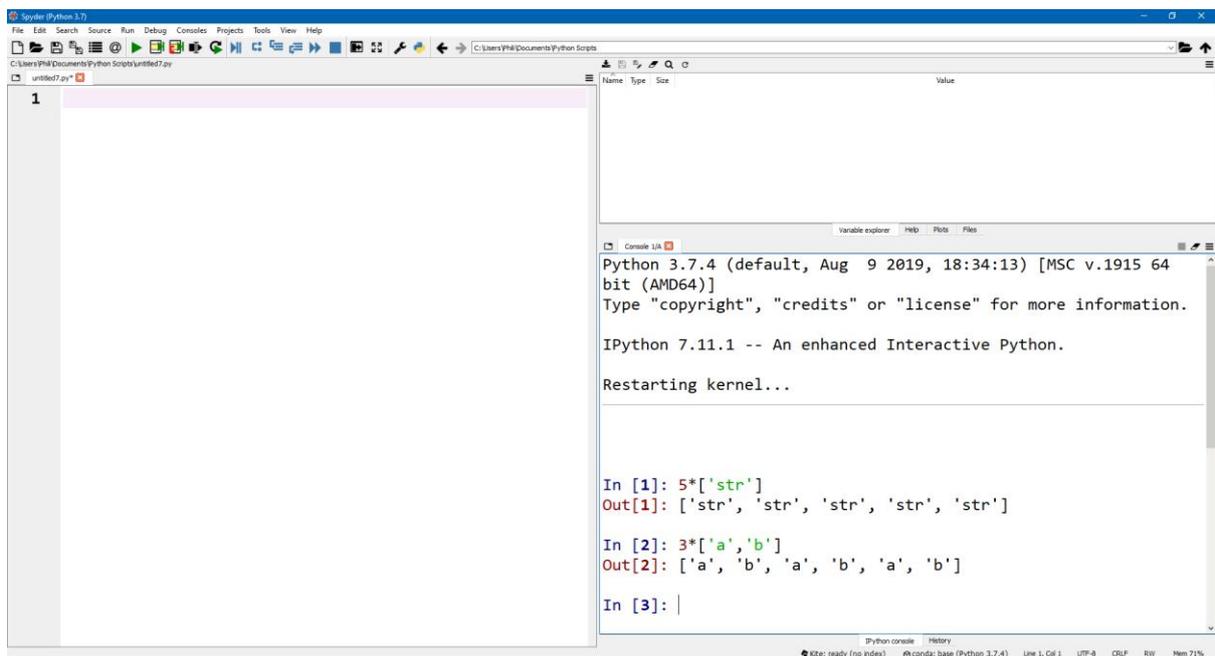
```
['a', 'b', 'c', 'd'] + ['e', 'f', 'g', 'h']
```

```
[1, 2, 3, 4] + [5, 6, 7, 8]
```



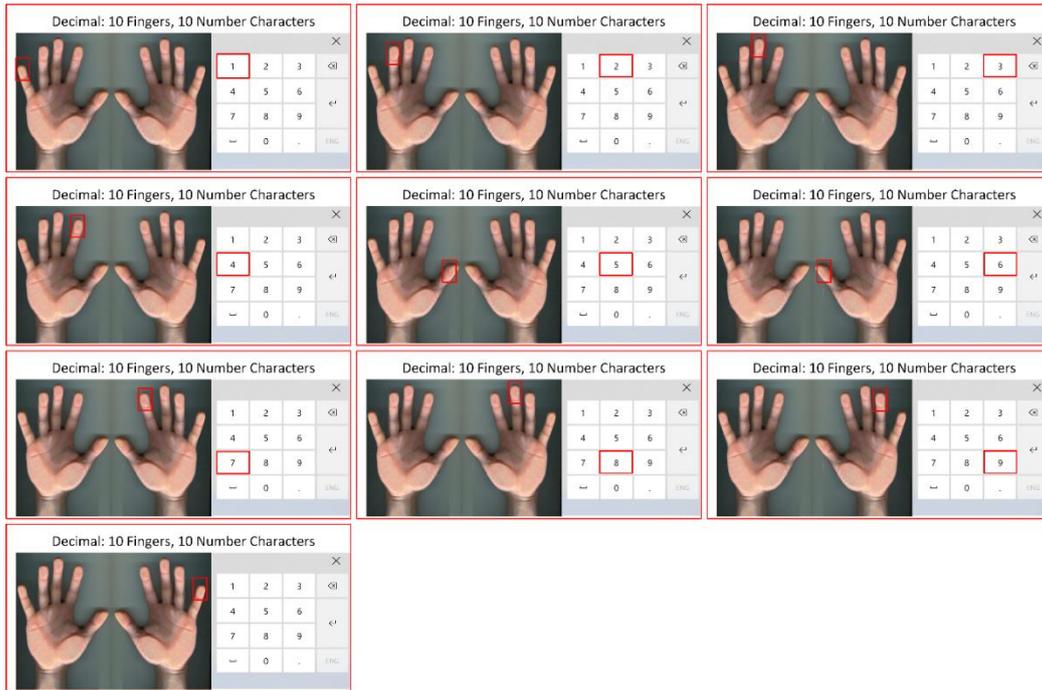
A list can also be multiplied by an integer and all the elements present in it will be replicated by the number of times it is multiplied by the integer.

```
5* ['str']
3* ['a', 'b']
```

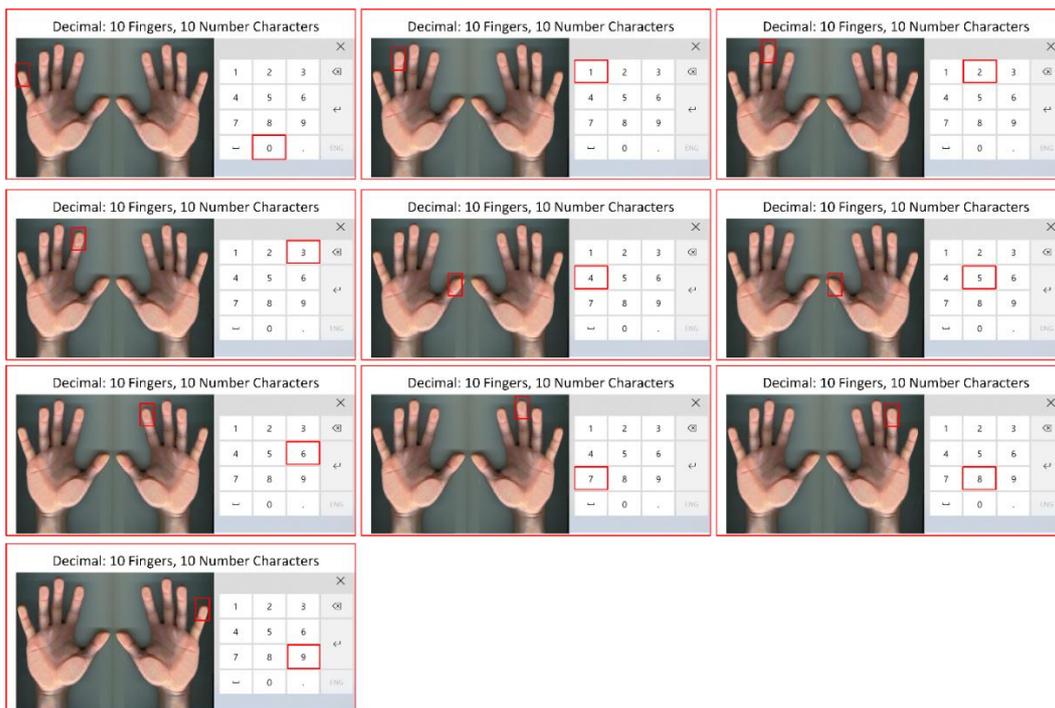


Indexing

In English, we have ten numeric digits 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 which is known as the decimal system and originates since we have 10 fingers. Typically, when we count, we assign our "first" finger to 1 and work our way across to our last finger which we assign the double digits 10. We count from 1,2,3,4,5,6,7,8,9,10.

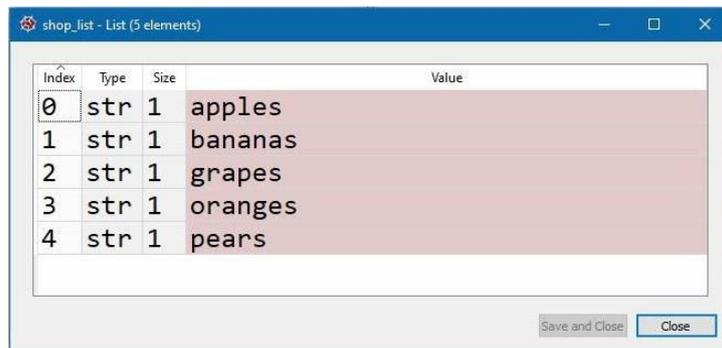


This counting mechanism, so called first order counting because it starts at 1 has the flaw that one of the numeric digits 0 is not assigned to a finger.



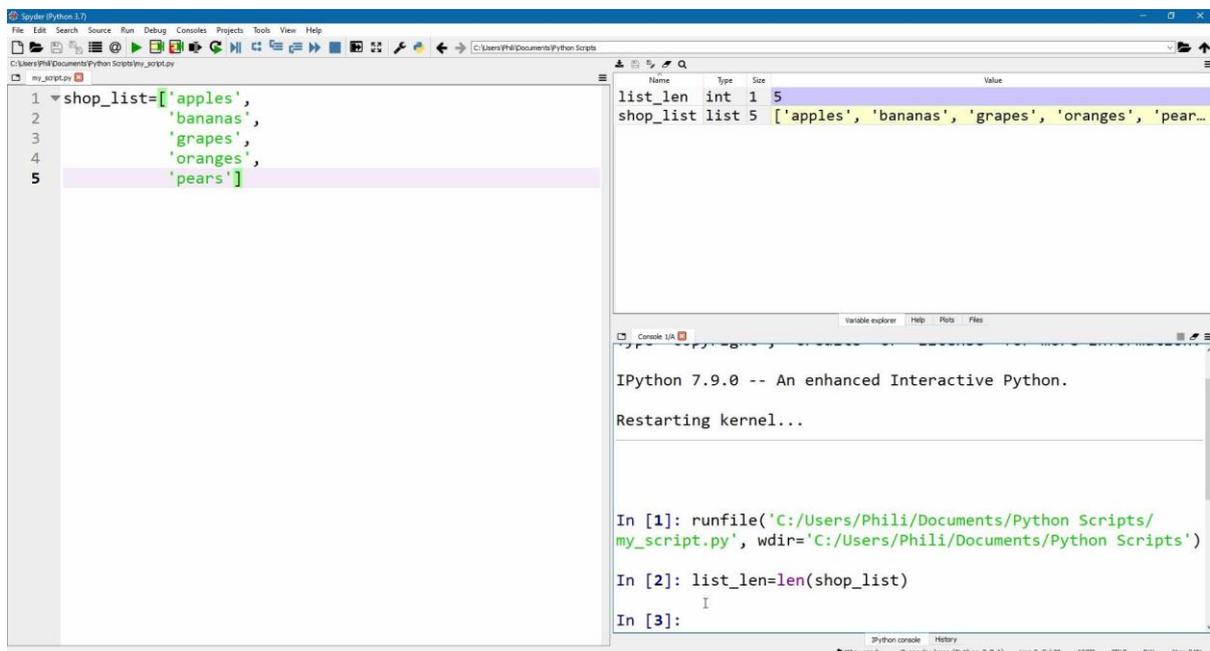
In Python and most other computer languages on the other hand, we want to assign the 10 digits to the 10 fingers so start counting at 0 and our "zereth" finger is 0 and when we count from 0 to 10 we count 0,1,2,3,4,5,6,7,8 and 9. Note that the value 10 itself is not included when we count from 0 to 10 i.e. we go up to 10 and stop 1 integer values before we get to 10 at 9. This counting mechanism is known as zero order indexing.

Returning to `shop_list` we can see that there are 5 elements, and these are indexes 0,1,2,3 and 4 i.e. the indexes start and go up to the value of the length of the list but don't include it.



We can get the length of the list using the function `len()` and inputting the list `shop_list` as the input argument.

```
len(shop_list)
```



It is also possible to select an index in the list by calling up the list followed by square brackets `[]` and typing its numeric index. For instance:

```
shop_list[0]
```

Will return the string:

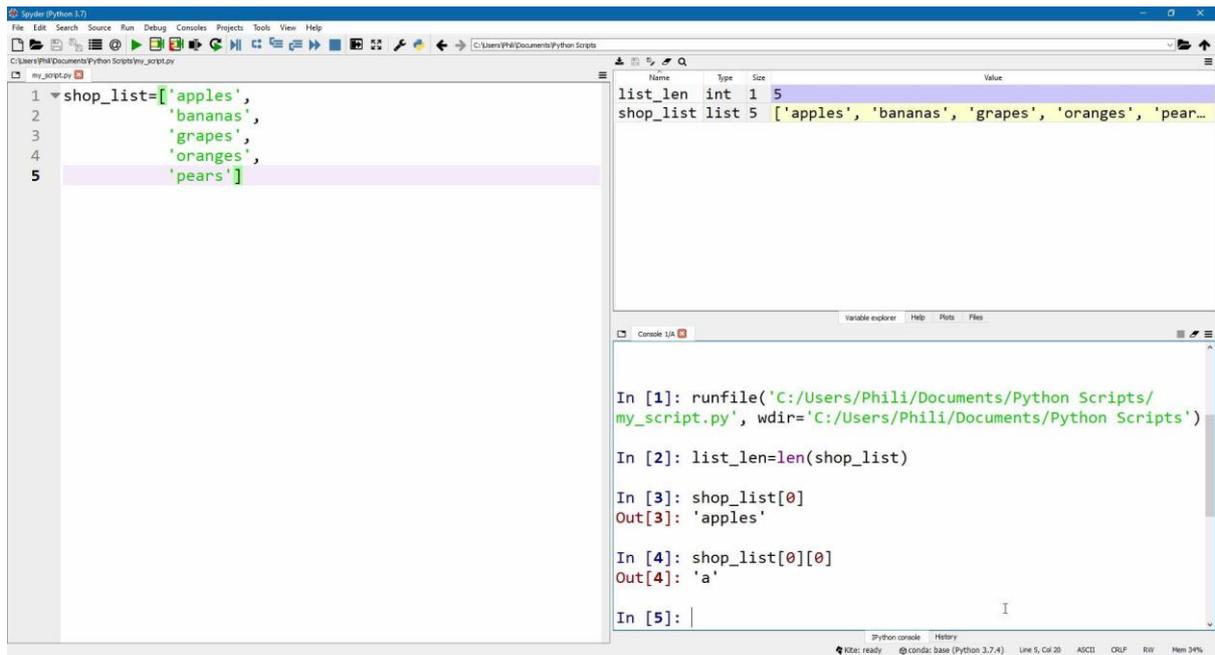
```
'apples'
```

However the string also has multiple characters and these can subsequently also be indexed into. Note the quotations don't count as an index in a string.

```
shop_list[0][0]
```

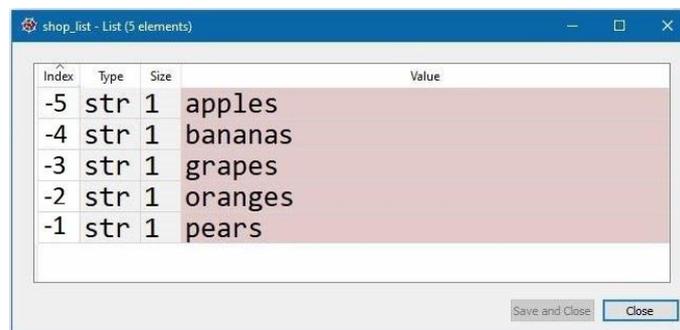
Will return the string:

```
'a'
```



Note in `shop_list[m][n]` the index `m` corresponds to the index in `shop_index` and the index `n` corresponds to the index in the word `'apples'`.

Python also allows indexing from the last value. Recall how when we zero order index from 0 to 5, we start from 0 and go up to the last value of 5 but don't include it in our list. The last value that we display in our list, which is one less than the value at the end of the list that we don't reach, in terms of negative indexing is called `-1` and we increment in integer steps until we end at negative the length of the list.



For example

```
shop_list[-4]
```

Returns:

```
'bananas'
```

A value in a list can be reassigned by indexing into the index on the left hand side and using the assignment operation to assign it to a new value on the right hand side.

```
shop_list[-4]='strawberries'
```

Index	Type	Size	Value
0	str	1	apples
1	str	1	strawberries
2	str	1	grapes
3	str	1	oranges
4	str	1	pears

This is of course equivalent to:

```
shop_list[1]='strawberries'
```

Slicing

The colon can also be used to create a slice containing multiple elements from a list.

```
1. shop_list=['apples',
2.            'bananas',
3.            'grapes',
4.            'oranges',
5.            'pears']
```

Index	Type	Size	Value
0	str	1	apples
1	str	1	bananas
2	str	1	grapes
3	str	1	oranges
4	str	1	pears

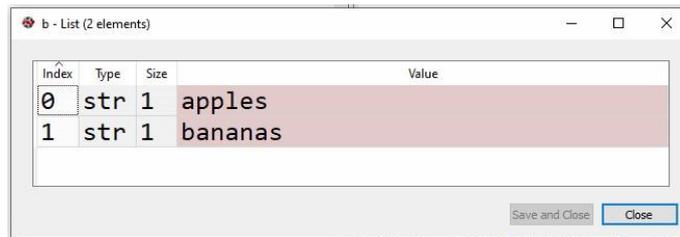
A slice of a list can be selected using a colon `:`. A lower and upper bound can be specified. Note that when we are slicing, we select up to but don't include the upper bound.

```
a=shop_list[1:3]
```

Index	Type	Size	Value
0	str	1	bananas
1	str	1	grapes

Therefore `a` only contains `'bananas'` and `'grapes'` which were index 1 and index 2 in `shop_list`. `'oranges'` which is index 3 in `shop_list` is not included.

```
b=shop_list[0:2]
```

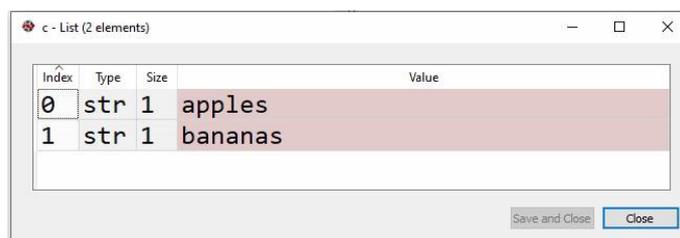


Index	Type	Size	Value
0	str	1	apples
1	str	1	bananas

Therefore `b` only contains `'apples'` and `'banana'` which were index 0 and index 1 in `shop_list`. `'grapes'` which is index 2 in `shop_list` is not included.

If no lower bound is selected, it is assumed the lower bound is `0`.

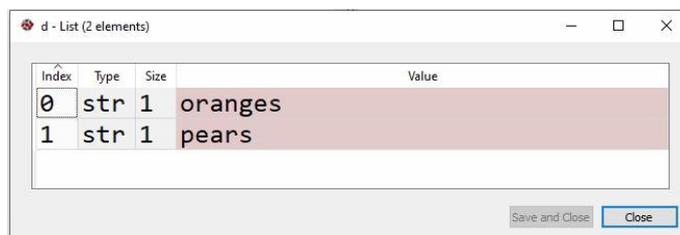
```
c=shop_list[:2]
```



Index	Type	Size	Value
0	str	1	apples
1	str	1	bananas

To select a slice which includes the last index on the original list we need to go up to one value higher than the last index. In this case we need to slice up to `5` but doesn't include the index `5` (which doesn't exist).

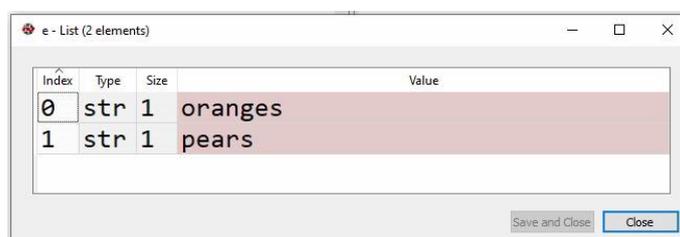
```
d=shop_list[3:5]
```



Index	Type	Size	Value
0	str	1	oranges
1	str	1	pears

If no upper bound is specified the upper bound is assumed to be the last index plus 1.

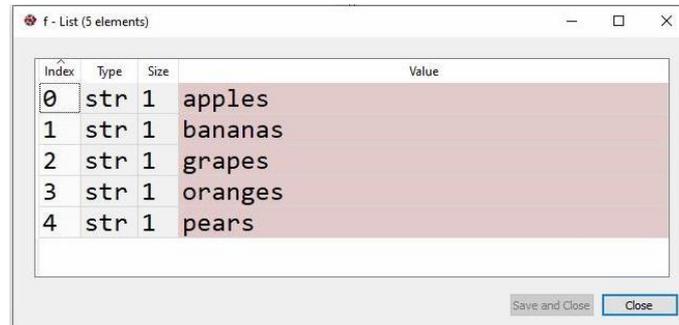
```
e=shop_list[3:]
```



Index	Type	Size	Value
0	str	1	oranges
1	str	1	pears

The colon on its own returns a copy of the complete list:

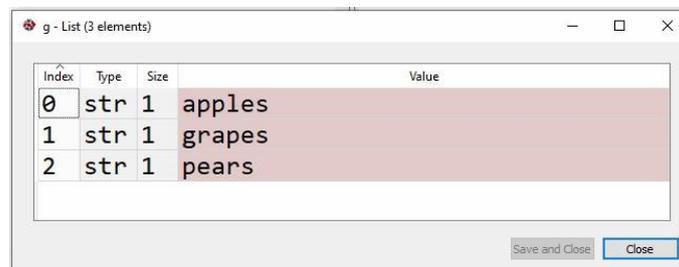
```
f=shop_list[:]
```



Index	Type	Size	Value
0	str	1	apples
1	str	1	bananas
2	str	1	grapes
3	str	1	oranges
4	str	1	pears

A double colon followed by an integer for instance `::n` will return every nth value. In this case we can use `::2` to return every second value.

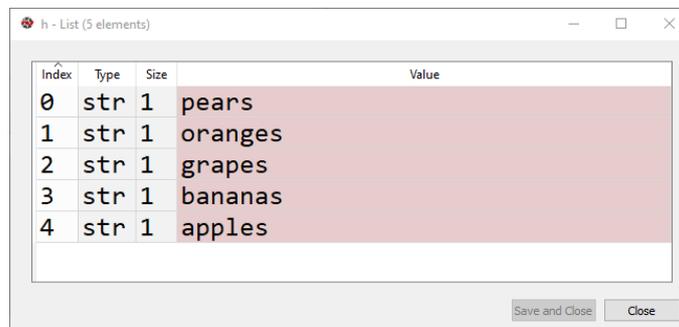
```
g=shop_list[::2]
```



Index	Type	Size	Value
0	str	1	apples
1	str	1	grapes
2	str	1	pears

To reverse the values in the list we can index using `::-1`

```
h=shop_list[::-1]
```



Index	Type	Size	Value
0	str	1	pears
1	str	1	oranges
2	str	1	grapes
3	str	1	bananas
4	str	1	apples

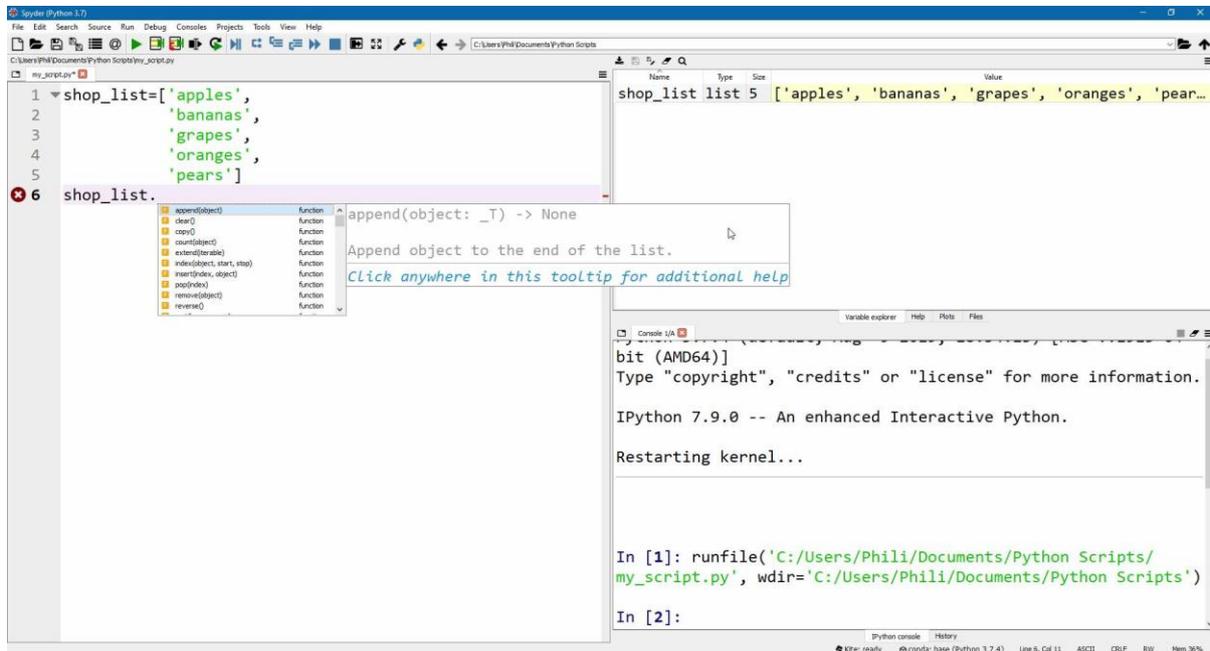
We can index into the 0th element of the list and get the word `'apples'` which we can then reverse.

```
i=shop_list[0][::-1]
```

```
'selppa'
```

List Methods

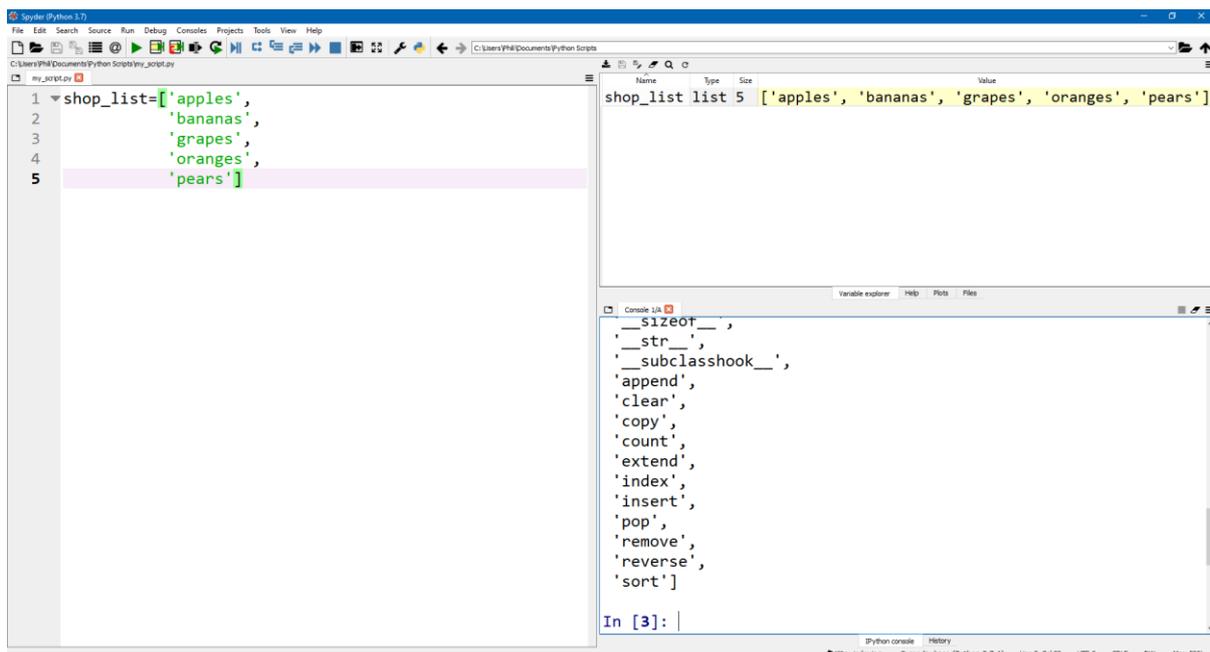
So far we've looked at a handful of functions such as `len()`, `print()` and `input()`. These were called directly and the input arguments if applicable were placed into the open parenthesis. A list itself is an object. A number of "functions" applicable to an object can be called up by dot indexing from the object. To access these typing in the objects name (variable name) of the list followed by a `.` and then `<method name>`.



When functions are called up from an object they are referred to as methods opposed to functions. Alternatively a list of methods can be accessed using the function `dir()` with the list as the input argument.

```
dir(shop_list)
```

For the time being, we can ignore the values returned that begin and end with underscores:



The list method `index` takes a single input argument which is the value of one of the indexes in the list for example the string `'apples'` and returns the index of it, in this case it will be index `0`.

```
shop_list.index('apples')
```

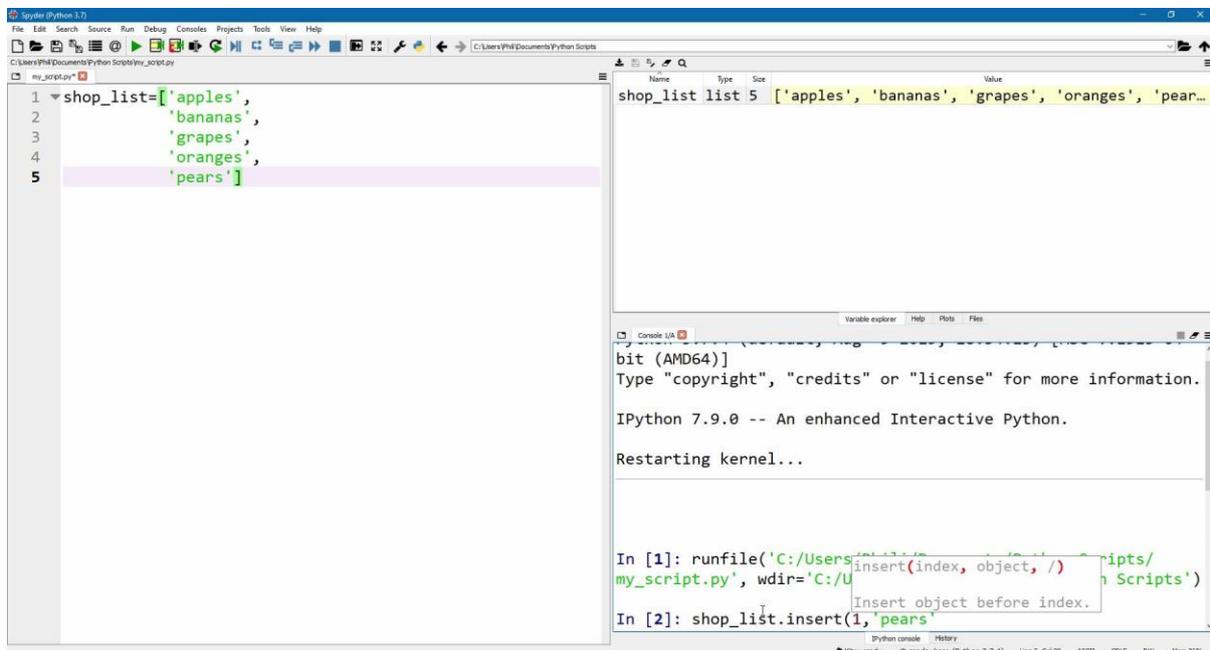
`0`

If the item is not present on the list an error would be returned `ValueError: Value is not in list`.

The list method `insert`, requires 2 input arguments, the 0th input argument is the index we wish to insert a new value and the 1st input argument is the new value. Using this list method followed with open parenthesis `(` will display information about the form of the input arguments.

If we want to insert a new string `'pears'` on `shop_list` at index `1`.

```
shop_list.insert(1, 'pears')
```



The screenshot shows a window titled "shop_list - List (6 elements)" with a table representing the list's contents:

Index	Type	Size	Value
0	str	1	apples
1	str	1	pears
2	str	1	bananas
3	str	1	grapes
4	str	1	oranges
5	str	1	pears

Note the subtle difference here, before we indexed into the first value of the list and replaced the string `'bananas'` with the string `'strawberries'`, this time we have instead inserted the string `'pears'` and all indexes of the original strings in the list at this point and later are shifted down by 1.

We can also use the list method `reverse` to reverse the list; this method has no input arguments.

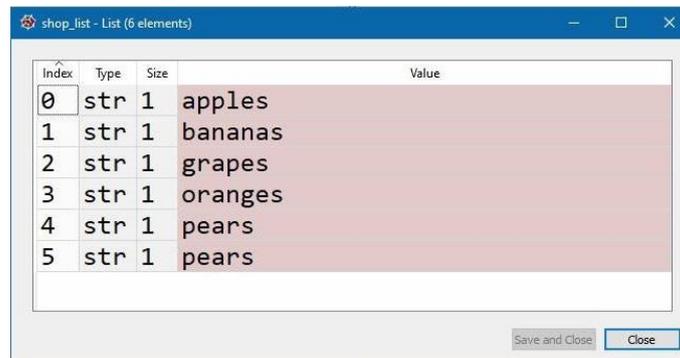
```
shop_list.reverse()
```



Index	Type	Size	Value
0	str	1	pears
1	str	1	oranges
2	str	1	grapes
3	str	1	bananas
4	str	1	pears
5	str	1	apples

It is also possible to sort the list alphabetically using the list method `sort`, this method can be called without any input arguments.

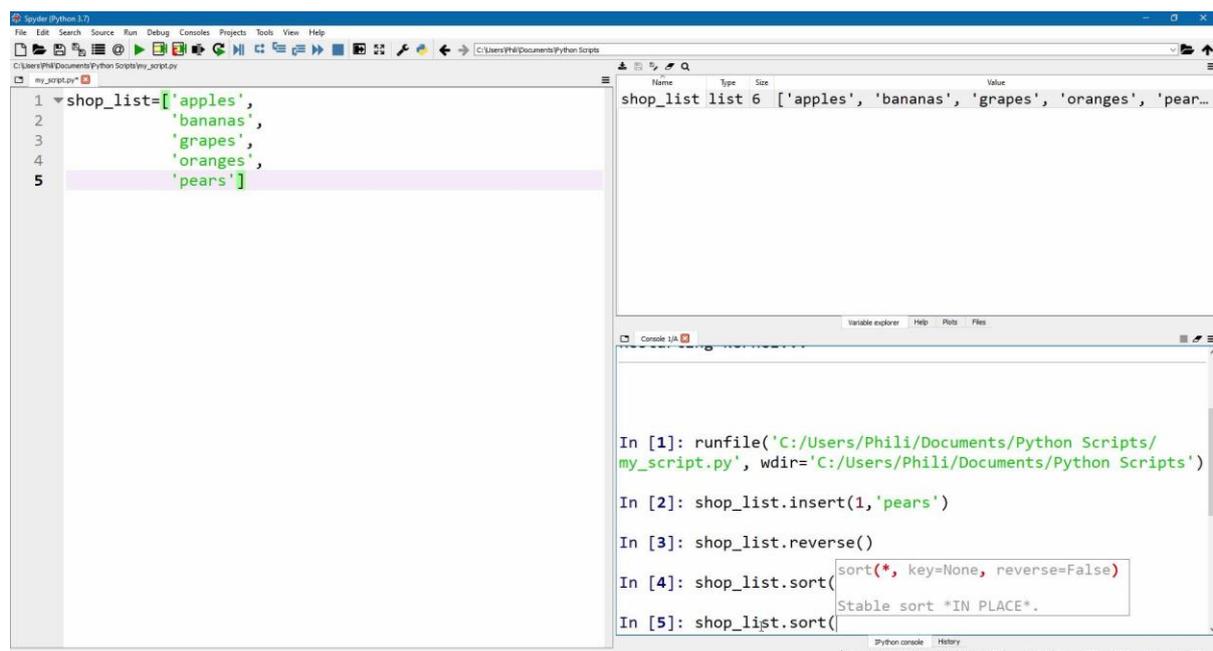
```
shop_list.sort()
```



Index	Type	Size	Value
0	str	1	apples
1	str	1	bananas
2	str	1	grapes
3	str	1	oranges
4	str	1	pears
5	str	1	pears

This method has no key input argument (often abbreviated as args) but has a keyword input argument (kwargs). More details can be found by typing the method with an open bracket:

```
shop_list.sort(
```



```
1 shop_list=['apples',
2           'bananas',
3           'grapes',
4           'oranges',
5           'pears']
```

Console output:

```
shop_list list 6 ['apples', 'bananas', 'grapes', 'oranges', 'pear...
In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/my_script.py', wdir='C:/Users/Phili/Documents/Python Scripts')
In [2]: shop_list.insert(1, 'pears')
In [3]: shop_list.reverse()
In [4]: shop_list.sort(
In [5]: shop_list.sort(
```

As seen, it displays the fact that `key=None` (meaning no key input arguments) and there is a keyword argument `reverse=False`. Key arguments when present have to input into a method, in the correct order which was demonstrated with the list method `insert(index, value)`. Keyword input arguments on the other hand are usually optional and are called after any mandatory key arguments. The keyword arguments are called by typing in their keyword and assigning it to a value. The keyword input arguments may be placed in any order, so long as they are placed after the key arguments. Keyword input arguments are usually also assigned a default value, which is used if they aren't explicitly called, when the list method `sort()` is called without this keyword input argument specified, it is by default set to `False`, so this method sorts in alphabetical order. Note that neither the keyword input argument variable name or the Boolean value are in quotations. The keyword input argument is a local method variable and like a variable name doesn't include quotations. This keyword argument can be called within the method and explicitly set to `True`.

```
shop_list.sort(reverse=True)
```



Index	Type	Size	Value
0	str	1	pears
1	str	1	pears
2	str	1	oranges
3	str	1	grapes
4	str	1	bananas
5	str	1	apples

The list method `count` counts the number a value occurs in the list. The string `'pears'` for example is a duplicate and occurs multiple times.

```
shop_list.count('pears')
```

```
2
```

We can use the list method `remove` to remove an item from the list, the input argument is the value of the item we wish to remove, for instance the string `'bananas'`.

```
shop_list.remove('bananas')
```



Index	Type	Size	Value
0	str	1	pears
1	str	1	pears
2	str	1	oranges
3	str	1	grapes
4	str	1	apples

If the input argument is instead the string `'pears'` which appears multiple times, only one value will be removed, in this case, the value at index 0. The string `'pears'` at index 1 won't be removed and will instead be shifted to index 0.

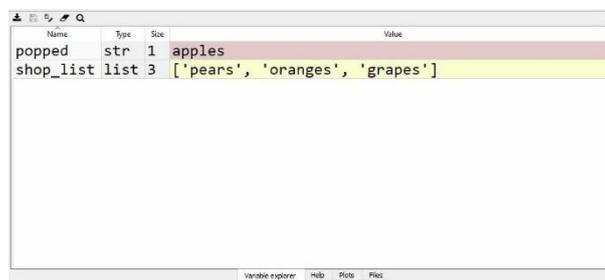
```
shop_list.remove('pears')
```



Index	Type	Size	Value
0	str	1	pears
1	str	1	oranges
2	str	1	grapes
3	str	1	apples

The list method `pop` can be used to pop off the last value in a list and optionally assign it to a separated value. `pop` has no input arguments.

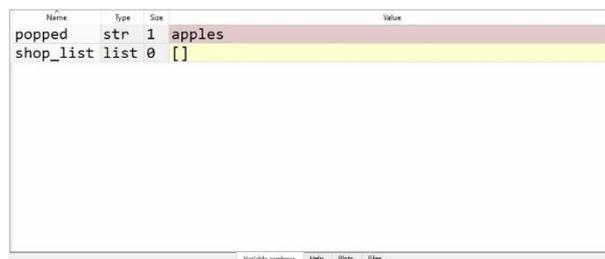
```
popped=shop_list.pop()
```



Name	Type	Size	Value
popped	str	1	apples
shop_list	list	3	['pears', 'oranges', 'grapes']

The list method `clear` will clear all items in the list, leaving an empty list. `clear` has no input arguments.

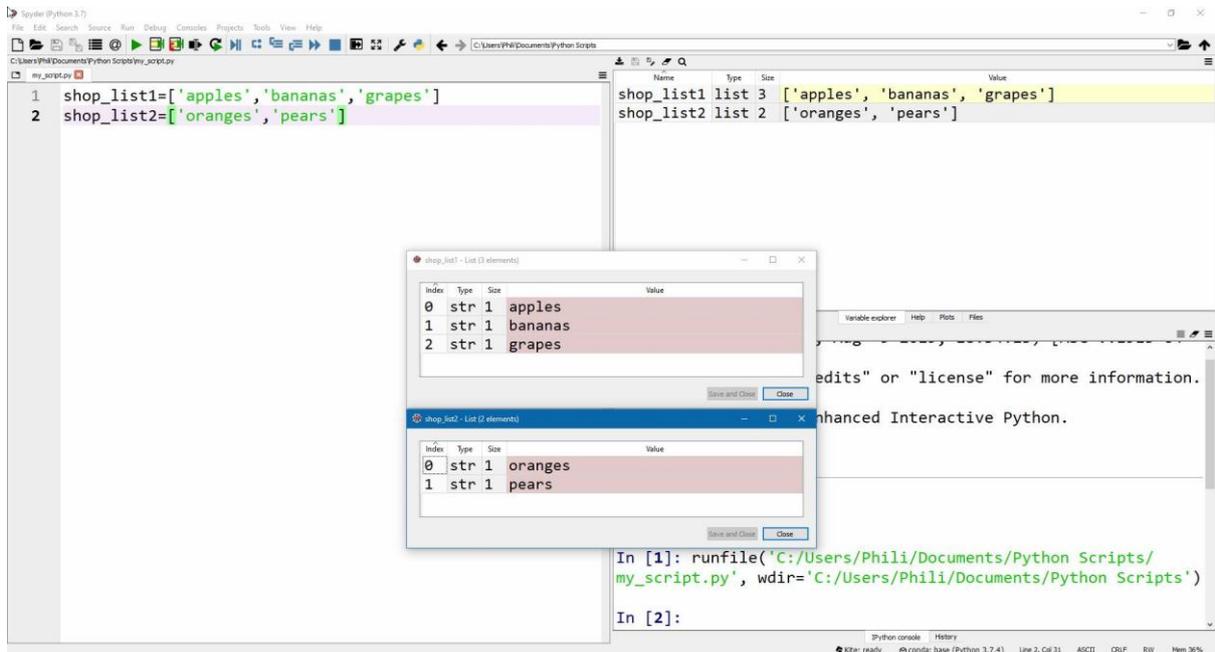
```
shop_list.clear()
```



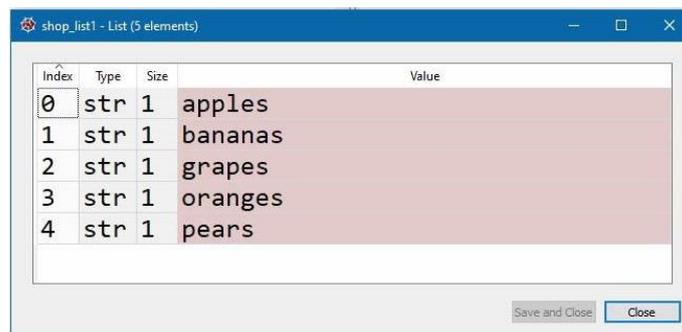
Name	Type	Size	Value
popped	str	1	apples
shop_list	list	0	[]

The list method `extend` can be used to combine two lists, the input argument is the second list which will extend from the first list.

1. `shop_list1=['apples', 'bananas', 'grapes']`
2. `shop_list2=['oranges', 'pears']`

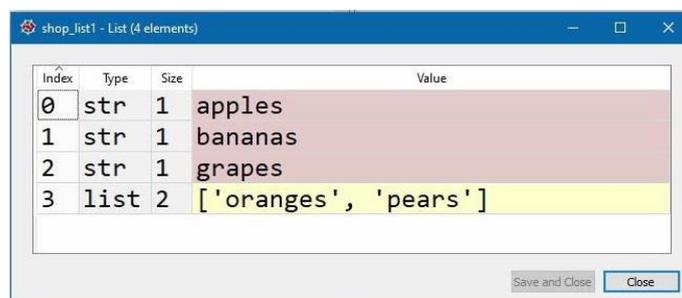


- shop_list1=['apples', 'bananas', 'grapes']
- shop_list2=['oranges', 'pears']
- shop_list1.extend(shop_list2)



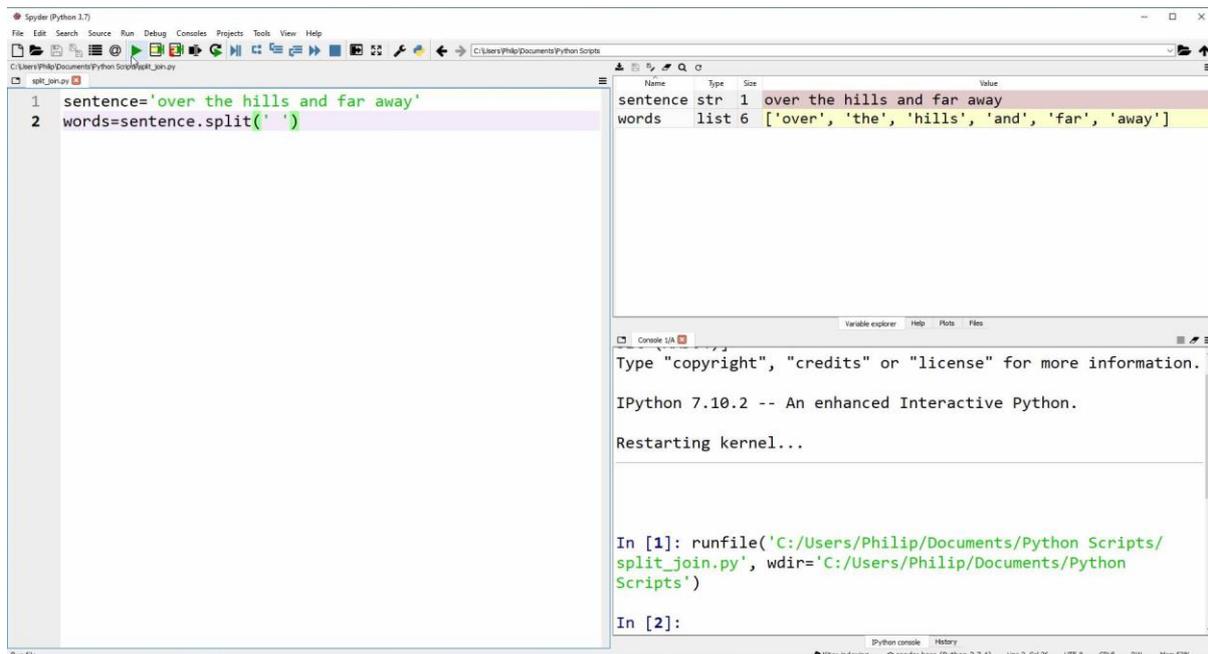
Note the list method `extend` often gets confused with `append` which will add a single index to the end of the first list and then store the second list as this last index. This is known as a nested list.

- shop_list1=['apples', 'bananas', 'grapes']
- shop_list2=['oranges', 'pears']
- shop_list1.append(shop_list2)



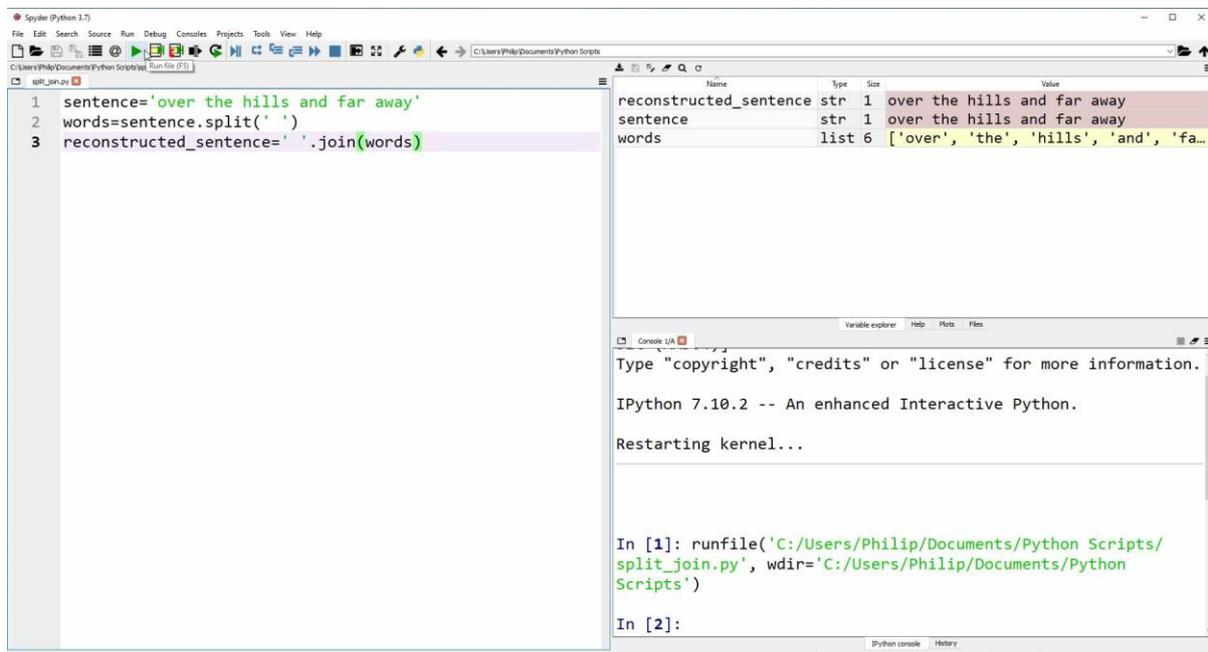
The split method can be used to split a string containing a sentence into a list of words. In this case the input argument to the method is a string of a space which indicates the use of a space as a delimiter.

1. `sentence='over the hills and far away'`
2. `words=sentence.split(' ')`



The list of words can be joined with spaces by using the method `.join` on a string of a space and having the list of words to be joined as an input argument.

1. `sentence='over the hills and far away'`
2. `words=sentence.split(' ')`
3. `reconstructed_sentence=' '.join(words)`



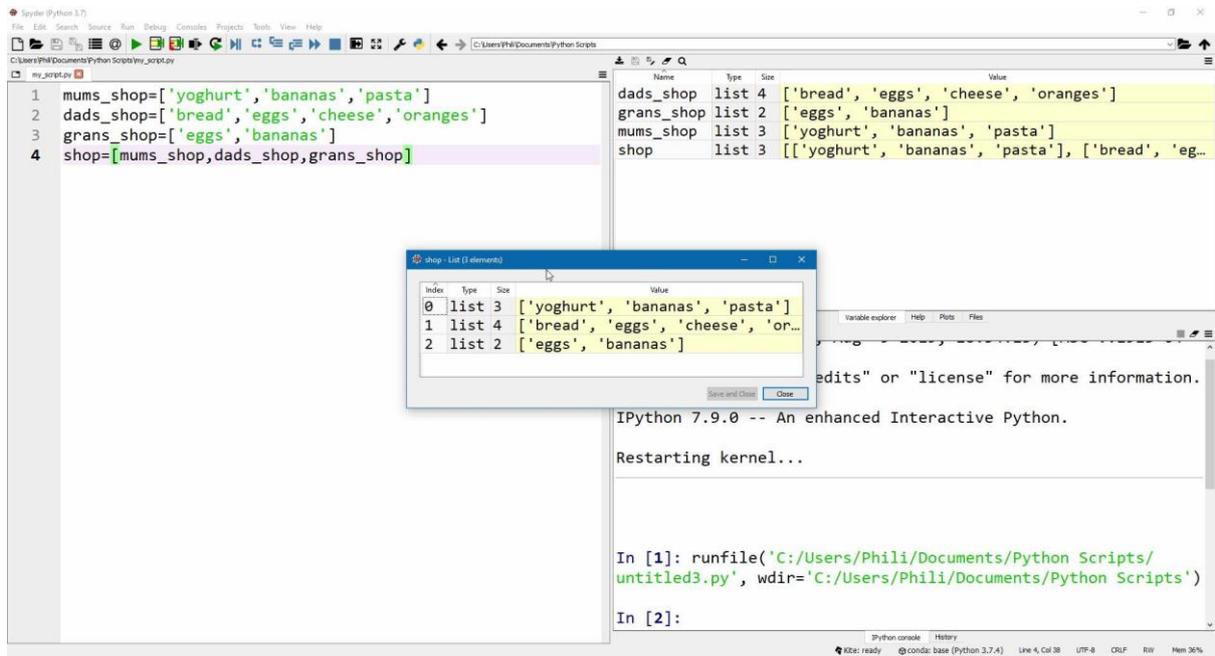
Nested Lists

A nested list can be considered as a list or set of lists embedded or nested into a list.

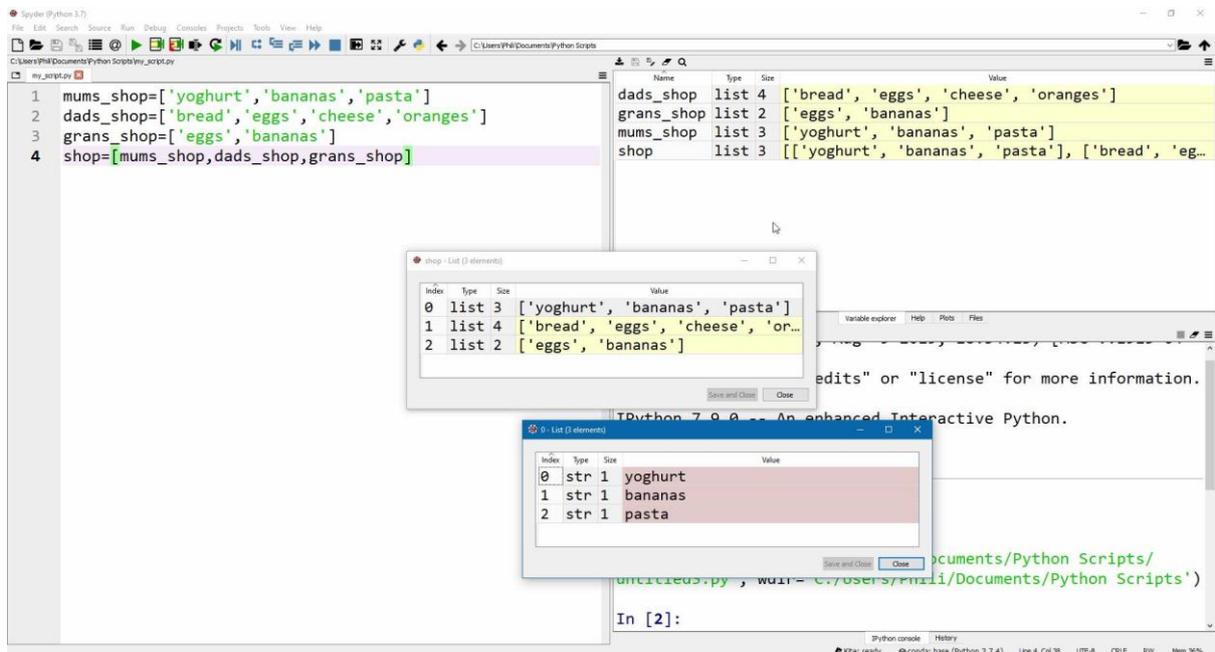
1. `mums_shop=['yoghurt', 'bananas', 'pasta']`

2. `dads_shop=['bread', 'eggs', 'cheese', 'oranges']`
3. `grans_shop=['eggs', 'bananas']`
4. `shop=[mums_shop,dads_shop,grans_shop]`

Note the values in the lists on line 1 to 3 are strings and are hence within quotation marks and highlighted green whereas the values in the list on line 4 are the variable names of other lists and hence don't have any quotation.



Note the list `shop` has index 0 which has as its value another list. If this value is double clicked, the nested list within index 0 can be viewed.



Recall to index into a list we type in the list name, followed by square brackets enclosing the index, in this case index 0.

```
shop[0]
```

```
['yoghurt', 'bananas', 'pasta']
```

Note the result is the nested list. To index within this nested list, for example to get the string 'pasta' at index 2, we need to index into this nested list:

```
shop[0][2]
```

```
'pasta'
```

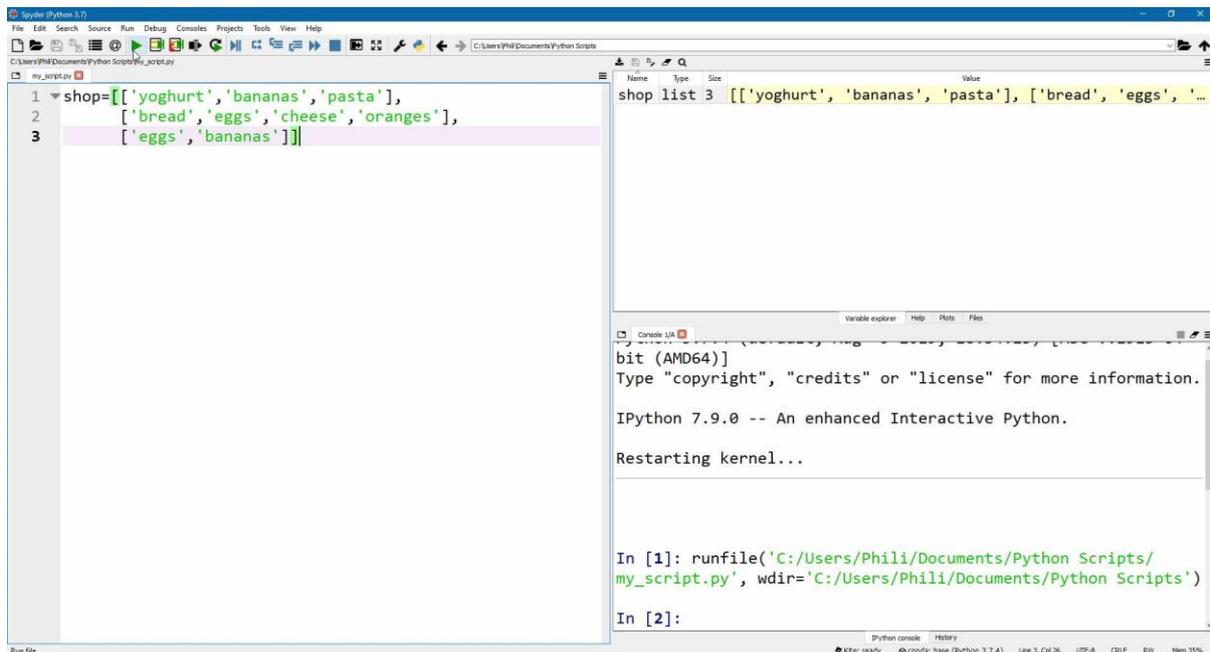
Note that the nested lists in each index of shop are all different sizes 3,4 and 2 respectively.

The above list can be created directly using the following.

```
1. shop= [ ['yoghurt', 'bananas', 'pasta'],  
2.         ['bread', 'eggs', 'cheese', 'oranges'],  
3.         ['eggs', 'bananas']]
```

This can all be placed on a single line but the convention is to have each nested list on a new line, to make the code more readable. Note if you highlight one of the square brackets in Spyder, it will highlight the other that matches with it.

```
1. shop=[ ['yoghurt', 'bananas', 'pasta'],  
2.         ['bread', 'eggs', 'cheese', 'oranges'],  
3.         ['eggs', 'bananas']]
```



Other brackets can be examined by highlighting one end of them:

```
1. shop=[ ['yoghurt', 'bananas', 'pasta'],  
2.         ['bread', 'eggs', 'cheese', 'oranges'],  
3.         ['eggs', 'bananas']]
```

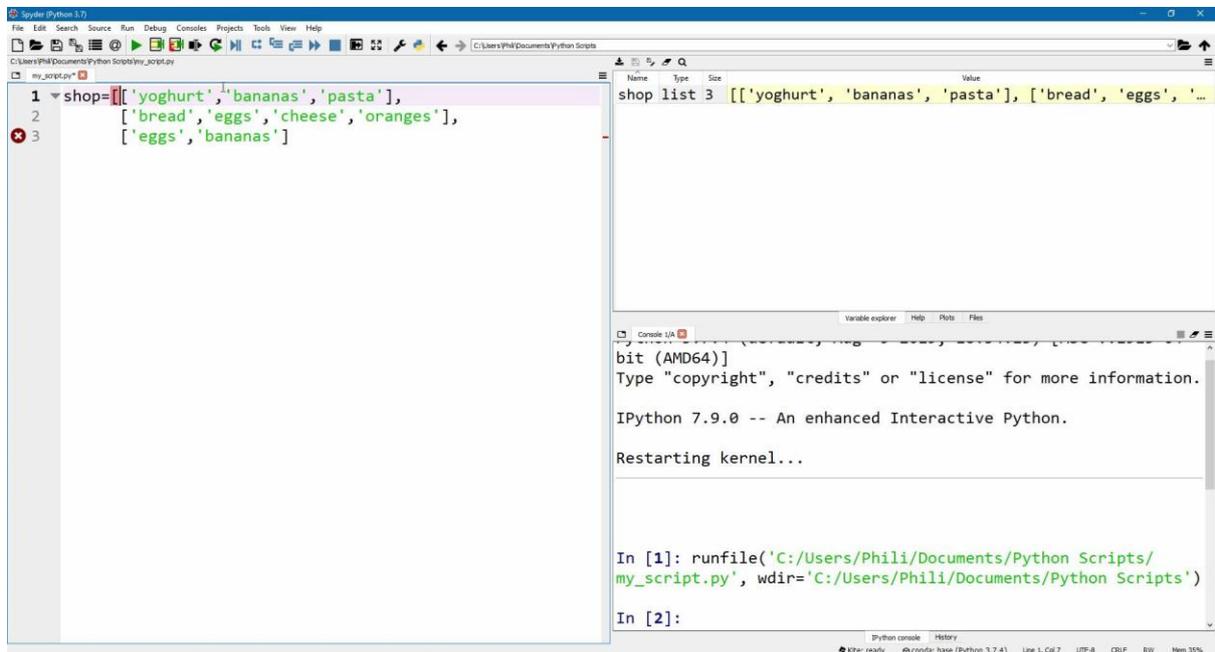
```
1. shop= [ 'yoghurt', 'bananas', 'pasta' ],
2.       [ 'bread', 'eggs', 'cheese', 'oranges' ],
3.       [ 'eggs', 'bananas' ]
```

```
1. shop= [ 'yoghurt', 'bananas', 'pasta' ],
2.       [ 'bread', 'eggs', 'cheese', 'oranges' ],
3.       [ 'eggs', 'bananas' ]
```

Orphaned square brackets will display red:

```
1. shop= [ 'yoghurt', 'bananas', 'pasta' ],
2.       [ 'bread', 'eggs', 'cheese', 'oranges' ],
3.       [ 'eggs', 'bananas' ]
```

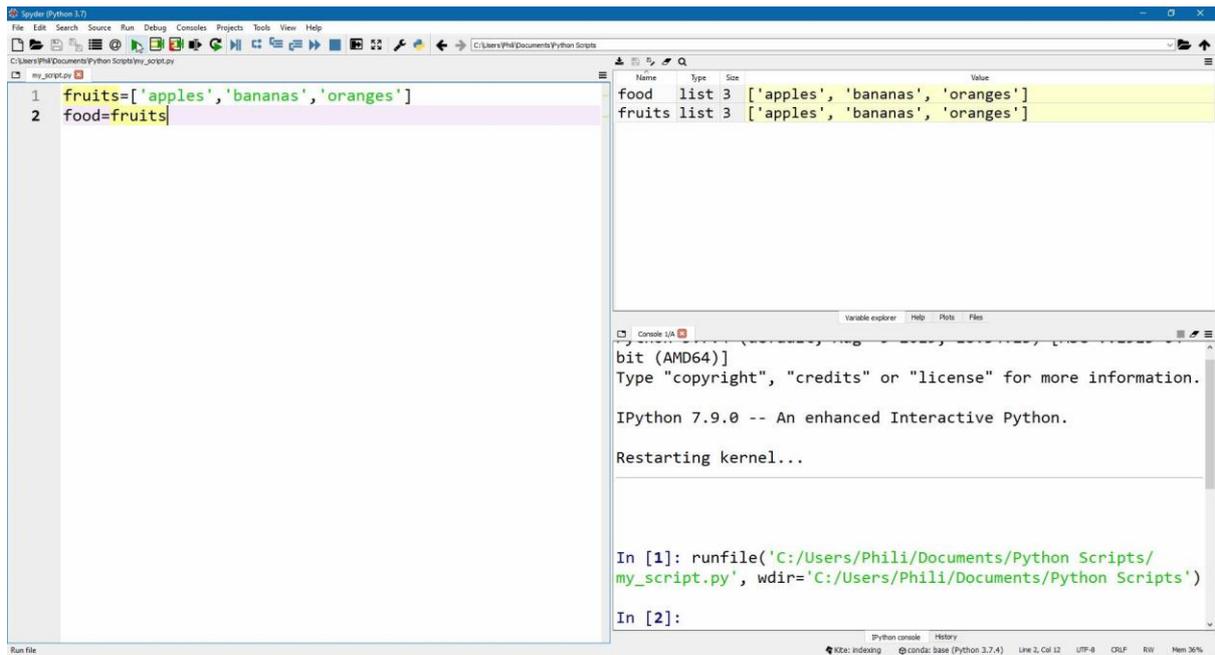
Moreover, a warning will display at the line number. In this case at line 3.



Mutability

Supposing we have the following two lines of code:

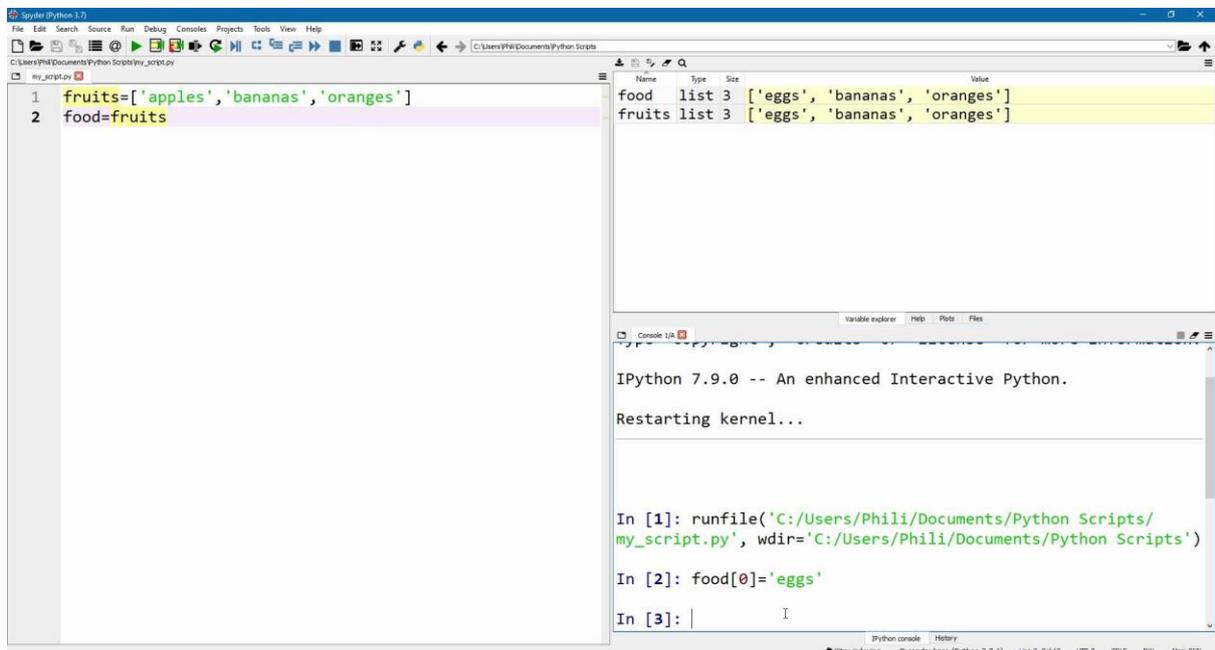
```
1. fruits= ['apple', 'banana', 'orange']
2. food=fruits
```



In line 2 we appear to have copied the first list `fruits` to a new list `food`. However, if we now index into the list `food` and replace index 0 with the string `'eggs'`.

```
1. fruits=['apples', 'bananas', 'oranges']
2. food=fruits
3. food[0]='eggs'
```

If we examine the Variable Explorer, we appear to get strange results. The string `'eggs'` has replaced the string `'apple'` in `fruits` although we only made changes to the list `food`.



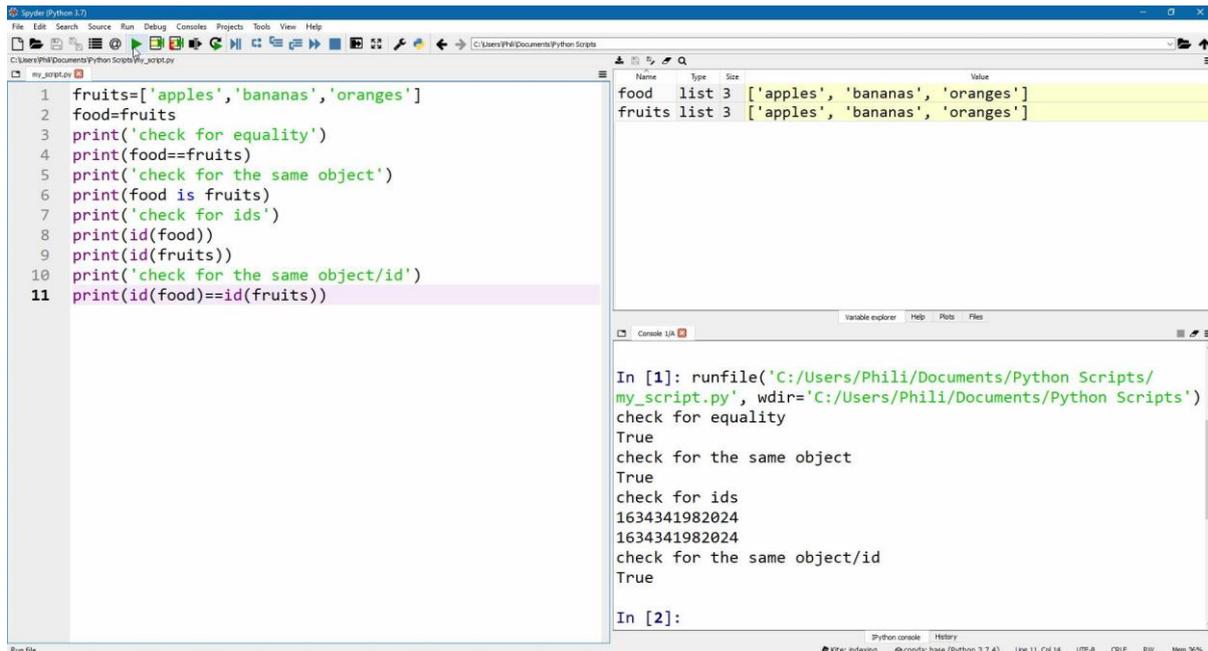
We can examine this in more detail by using:

```
1. fruits=['apples', 'bananas', 'oranges']
2. food=fruits
```

```

3. print('check for equality')
4. print(food==fruits)
5. print('check for the same object')
6. print(food is fruits)
7. print('check for ids')
8. print(id(food))
9. print(id(fruits))
10. print('check for the same object/id')
11. print(id(foods)==id(fruits))

```



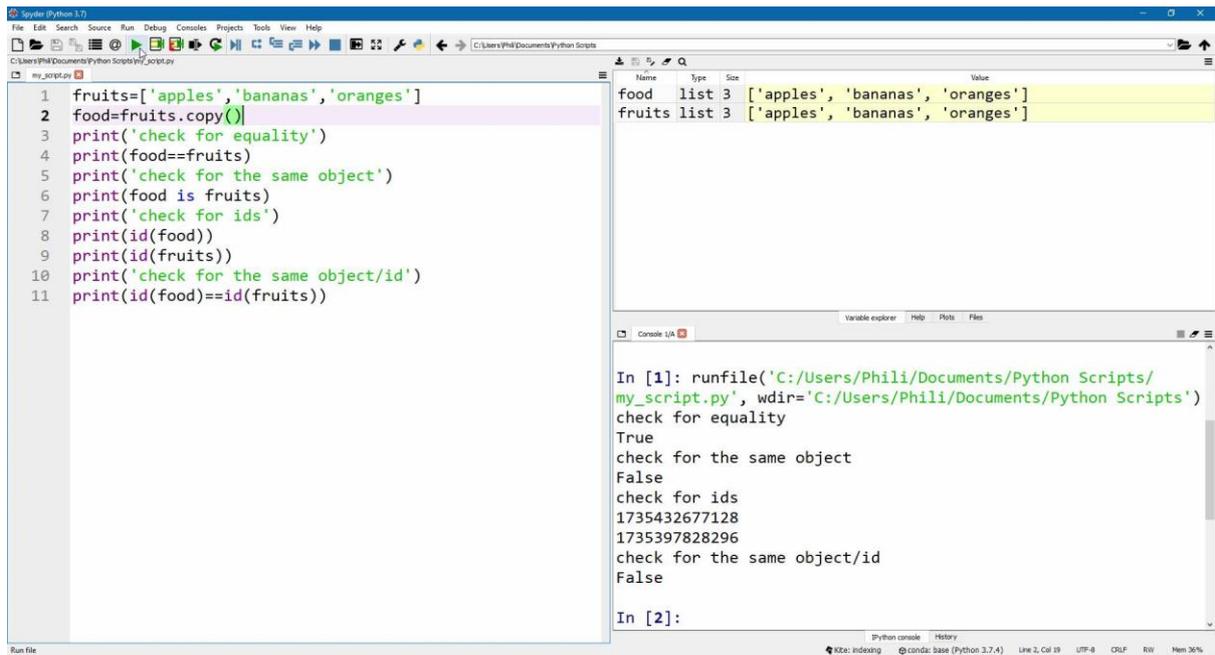
Here we see these are not only equal in the value assigned but they have matching ids and are therefore the same object in memory.

If instead we use the list method `copy` which has no input arguments, at line 2 we instead get the following:

```

1. fruits=['apple', 'banana', 'orange']
2. food=fruits.copy()
3. print('check for equality')
4. print(food==fruits)
5. print('check for the same object')
6. print(food is fruits)
7. print('check for ids')
8. print(id(food))
9. print(id(fruits))
10. print('check for the same object/id')
11. print(id(foods)==id(fruits))

```



Now we see that the lists `food` and `fruits` are equal in value but have different ids and are not the same object in memory.

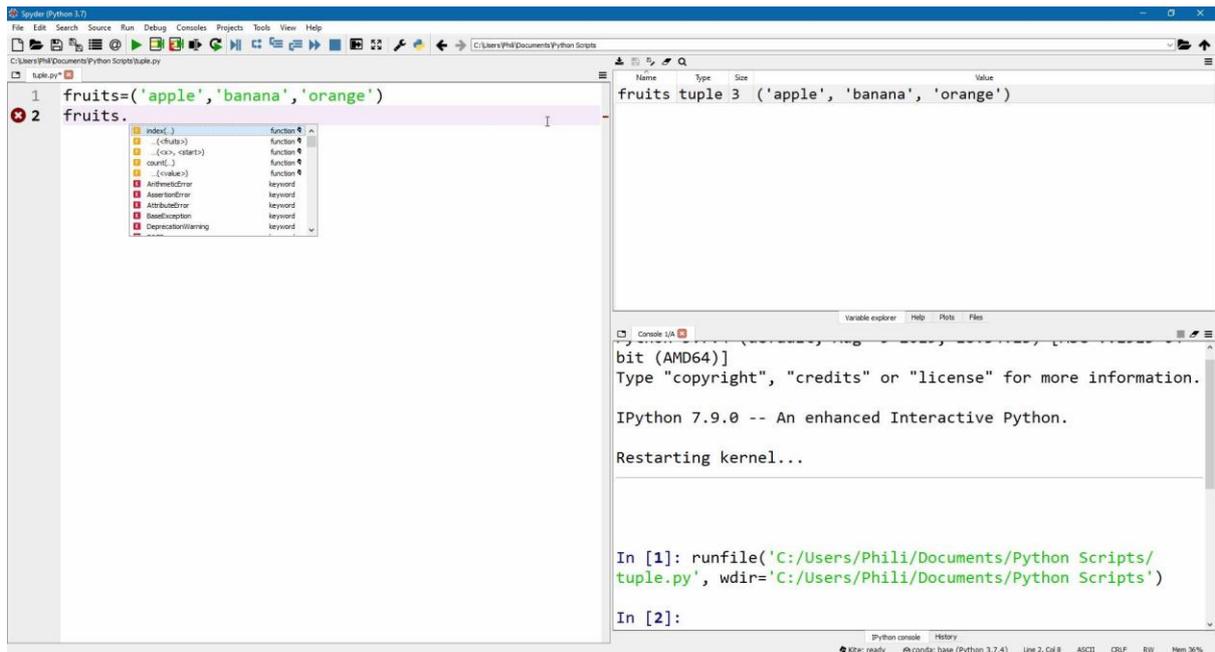
In the example above we didn't want to change the list `fruits` but accidentally mutated it. This change is an unintended consequence of a list being mutable (changeable).

Tuples

It is possible to create a tuple, which is essentially a list that cannot be mutated. Once a tuple is created it cannot be modified (unless it is deleted and recreated). A tuple is created in the same way as a list except it is enclosed in round brackets `()` opposed to square brackets `[]`.

```
fruits=('apple', 'banana', 'orange')
```

The number of tuple methods accessed by typing in the tuple's variable name followed by `.` and then `<tab>` is significantly smaller than the number of list methods due to the inability to mutate a tuple.



Note although a tuple is created using round brackets `()`, opposed to square brackets `[]`, there is no difference in the brackets, square brackets `[]` used to index into a tuple and the input arguments of list methods are enclosed in round brackets `()`.

```
fruits.counts('apple')
```

```
0
```

```
fruits[0]
```

```
'apple'
```

Note there are some subtleties when creating an empty tuple which is done with empty parenthesis, a single value enclosed in parenthesis and a tuple of only one value.

```
a=()
b='apple'
c=('apple')
d=('apple',)
```

Dictionaries

Python dictionaries work in the same manner as traditional dictionaries, in that they have a keyword or key and this is associated with a description or value. In Python dictionaries are enclosed in curly brackets `{ }` and each key and value pair are separated using a colon `:` while the comma `,` is used as a delimiter to move onto the next key and value pair. Like a list, the dictionary can be written on a single line:

```
1. dictionary={'key1': 'value1', 'key2': 'value2', 'key2': 'value3'}
```

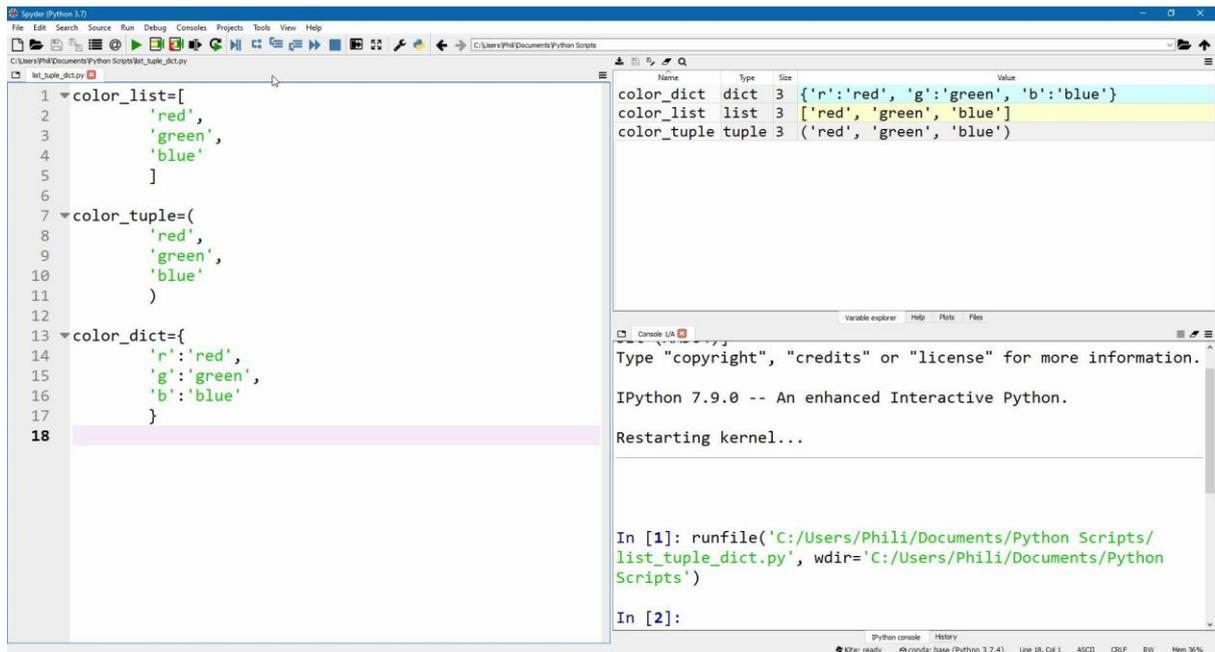
However it is usually written on separate lines so one can more readily read out the key value pairs.

```
1. dictionary={
2.     'key1': 'value1',
3.     'key2': 'value2',
4.     'key2': 'value3'
5. }
```

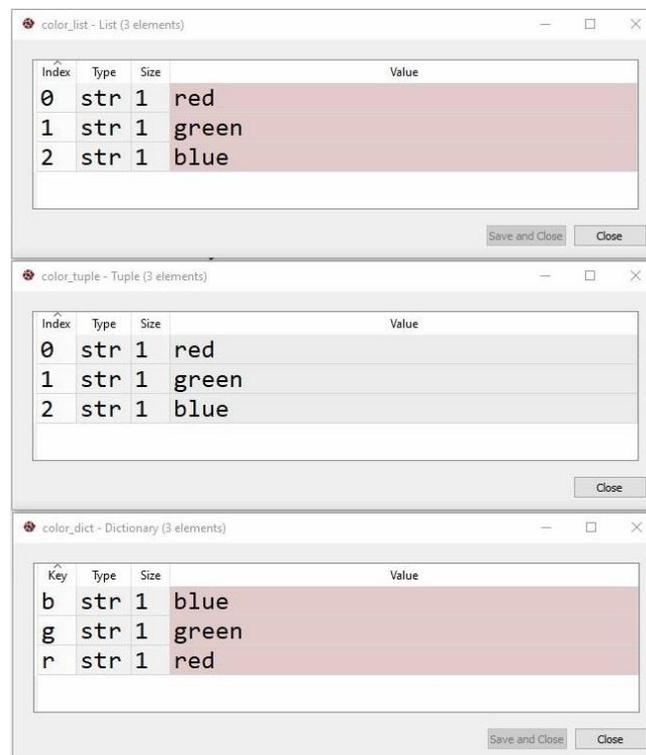
Key names may be strings or numbers, meaning there is much more versatility in a key name than a variable name. If text without quotations is used as a key, Python will try to assign the value of the variable name to the key and if not found yield a `NameError`. However in most cases, it is recommended to follow the same convention as variable names. That is to only use lower case, because dictionary keys like variable names will be case sensitive and to use the underscore `_` instead of a space.

Let's create a list of color strings, a tuple of color strings and a dictionary of color single letter key strings and full name values. In this case, each value will be placed on a separate line. We can highlight the differences in each, namely the bracket type and the use of a colon in dictionaries to separate out key and values on each line.

```
1. color_list=[
2.     'red',
3.     'green',
4.     'blue'
5. ]
6. color_tuple=(
7.     'red',
8.     'green',
9.     'blue')
10. color_dict={
11.     'r': 'red',
12.     'g': 'green',
13.     'b': 'blue'
14. }
```



The list, tuple and dictionary may be opened within the variable explorer and placed side by side. The list and the tuple look the same, both have 3 elements and therefore have indexes 0, 1 and 2. The values in the tuple are grey as they cannot be mutated. In the dictionary there is no index, instead there are the Key values.



In a list and a tuple, indexing is carried out using square brackets [] and the index. For example:

```
color_list[0]
```

```
'blue'
```

```
color_tuple[0]
```

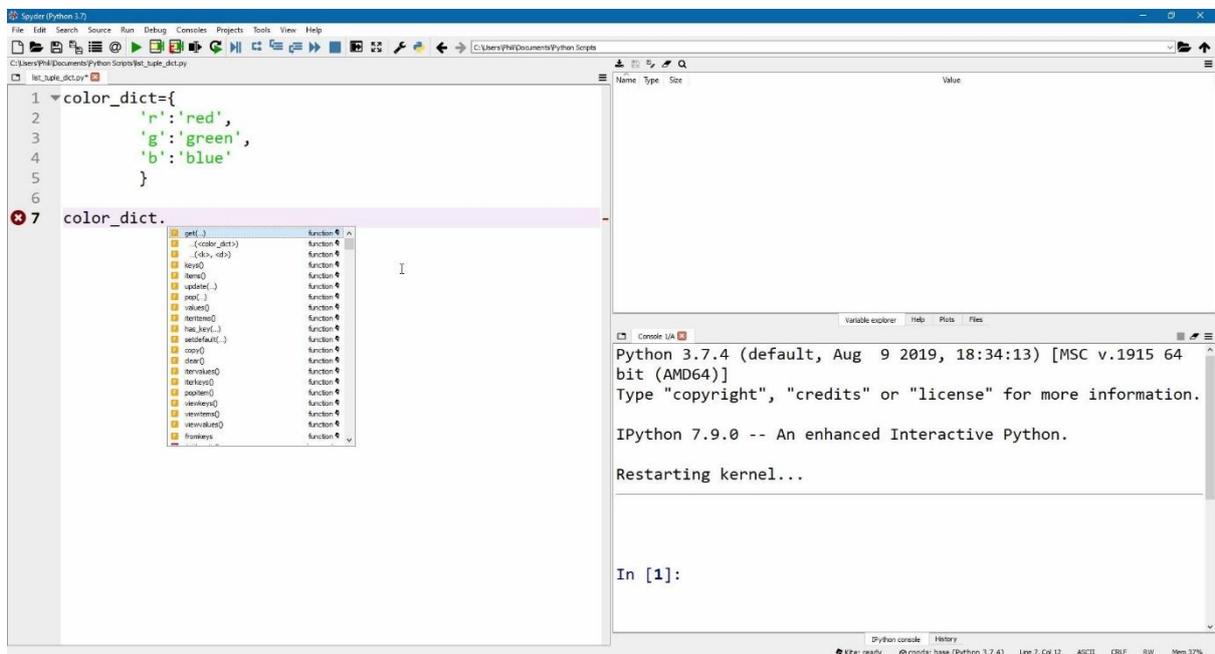
```
'blue'
```

While in a dictionary, indexing is carried out using the key. Note as the key is a string, it must be enclosed in single quotations:

```
color_dict['b']
```

```
'blue'
```

Like lists and tuples, there are a number of dictionary methods which can be accessed by typing in the dictionary name followed by a `.` and then a `<key>`.



The dictionary method `keys` may be used to list all the keys available in the dictionary, it has no input argument:

```
color_dict.keys()
```

```
dict_keys(['r', 'g', 'b'])
```

The dictionary method `values` may be used to list all the values available in the dictionary, it has no input argument:

```
color_dict.values()
```

```
dict_values(['red', 'green', 'blue'])
```

The dictionary method `items` may be used to list all the key, value pairs available in the dictionary, it has no input argument:

```
color_dict.items()
```

```
dict_items([('r', 'red'), ('g', 'green'), ('b', 'blue')])
```

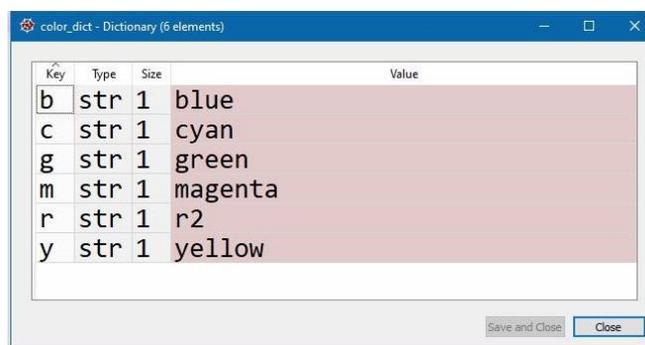
The dictionary method `get` has the key as an input argument and returns the assigned value

```
color_dict.get('r')
```

```
'red'
```

The dictionary method `update` may be used to update existing items to a dictionary or to add new values. Its input argument is itself a dictionary.

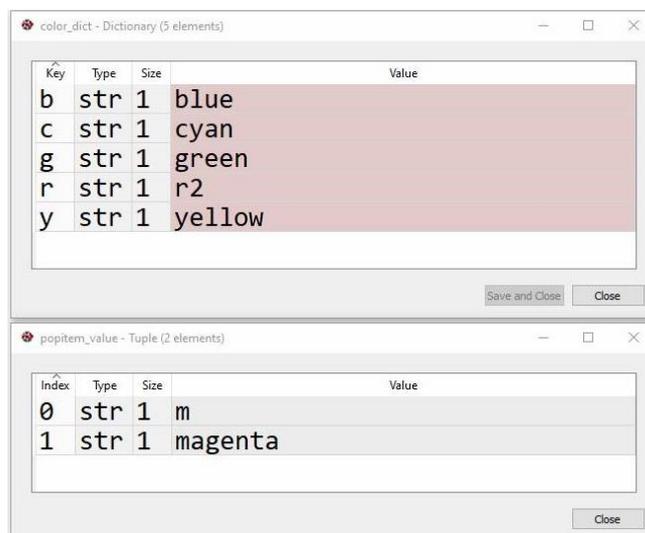
```
color_dict.update({'r':'r2','c':'cyan','y':'yellow','m':'magenta'})
```



Key	Type	Size	Value
b	str	1	blue
c	str	1	cyan
g	str	1	green
m	str	1	magenta
r	str	1	r2
y	str	1	yellow

The method `popitem` will pop the last item added and has no input arguments, it can be assigned to an output argument.

```
popitem_value=color_dict.popitem()
```

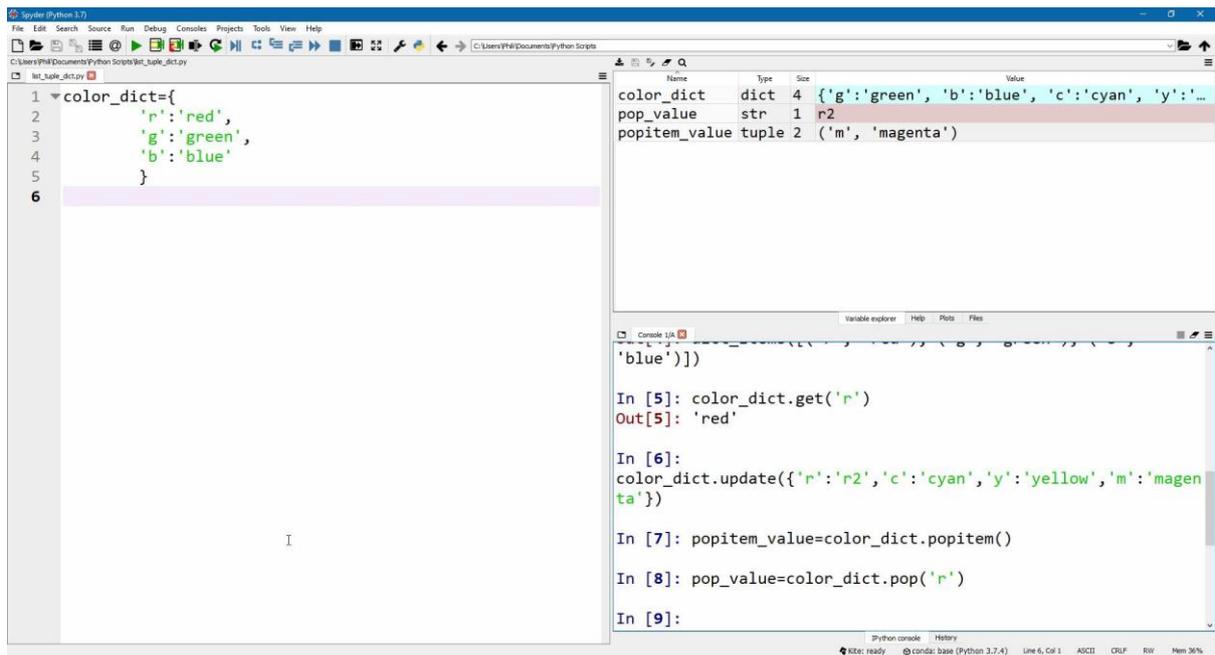


Key	Type	Size	Value
b	str	1	blue
c	str	1	cyan
g	str	1	green
r	str	1	r2
y	str	1	yellow

Index	Type	Size	Value
0	str	1	m
1	str	1	magenta

Note the output argument will be a tuple with two indexes, the original key and the original value. It is possible to use the method `pop`, using an input argument as an index. It can be assigned to an output argument but this will only save the key to a string and the original value will be lost.

```
pop_value=color_dict.pop('r')
```

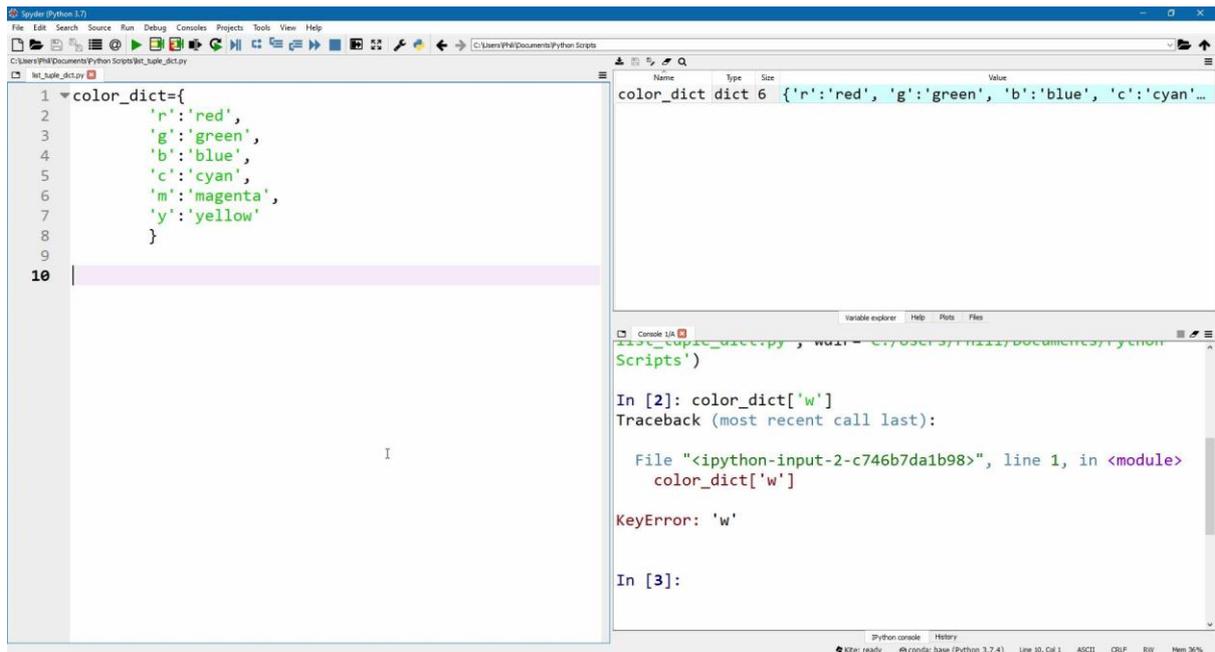


If we create the dictionary with additional 1 letter abbreviations for colors.

```
1. color_dict={
2.     'r':'red',
3.     'g':'green',
4.     'b':'blue',
5.     'c':'cyan',
6.     'm':'magenta',
7.     'y':'yellow'
8. }
```

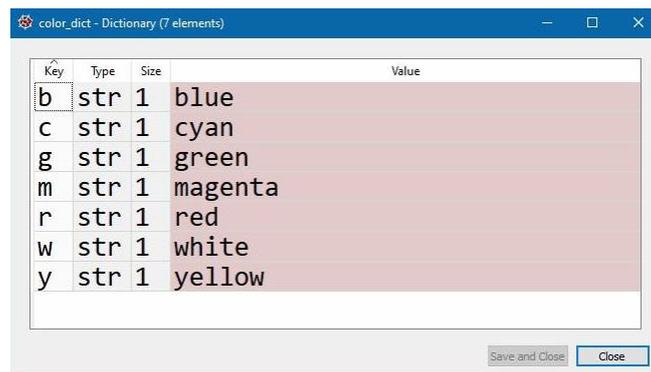
Then try to index a one-word key value that doesn't exist, we will get a `KeyError`.

```
color_dict['w']
```



This can be assigned to a new value however:

```
color_dict['w']='white'
```



In the English language there are some words that are homonyms, i.e. a key that has two different values. For example

- address: the number of the house, name of the road, and name of the town where a person lives or works, and where letters can be sent.
- address: a formal speech.

These are not allowed in Python; it simply means the key will be assigned to a new value and the old value will be forgotten. If we attempt to assign the single digit key string `'b'` to the new value `'black'`, the original value of `'blue'` will be replaced.

```
color_dict['b']='black'
```

Key	Type	Size	Value
b	str	1	black
c	str	1	cyan
g	str	1	green
m	str	1	magenta
r	str	1	red
w	str	1	white
y	str	1	yellow

For this reason, the one letter abbreviation of 'black' is usually not 'b' but 'k'. In the English language there are also synonyms, that is two words that have the same meaning:

- scared: feeling fear or worry
- frightened: feeling fear or worry

These are allowed in Python. For example.

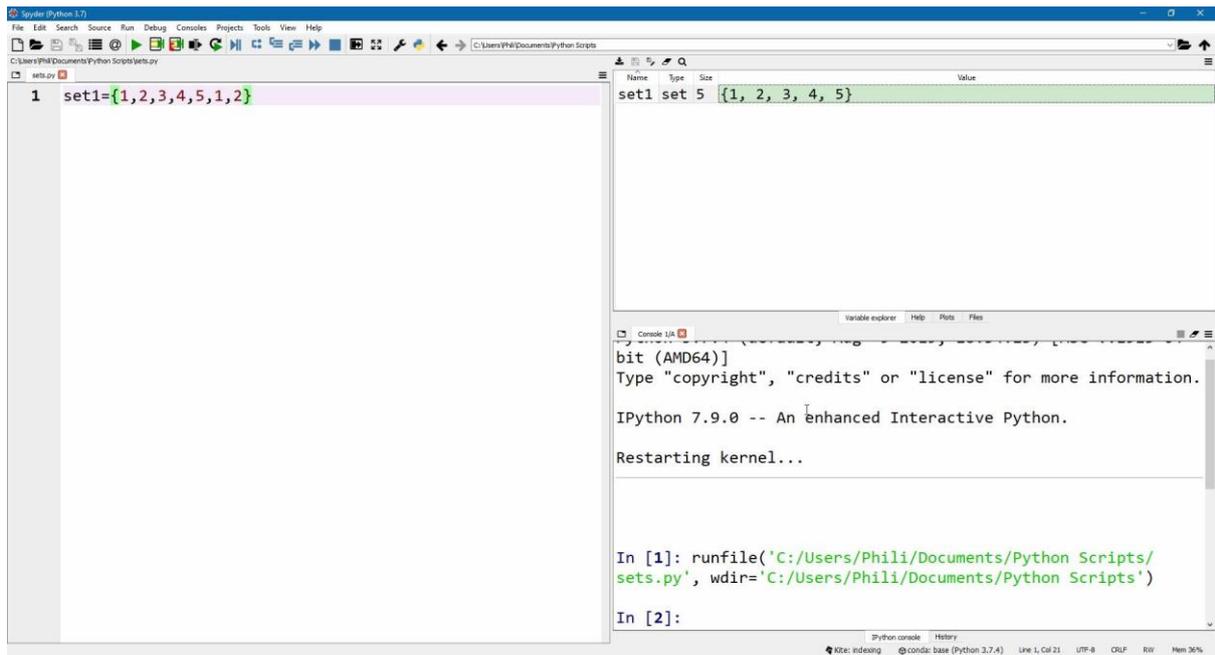
```
color_dict['w2']='white'
```

Key	Type	Size	Value
b	str	1	black
c	str	1	cyan
g	str	1	green
m	str	1	magenta
r	str	1	red
w	str	1	white
w2	str	1	white
y	str	1	yellow

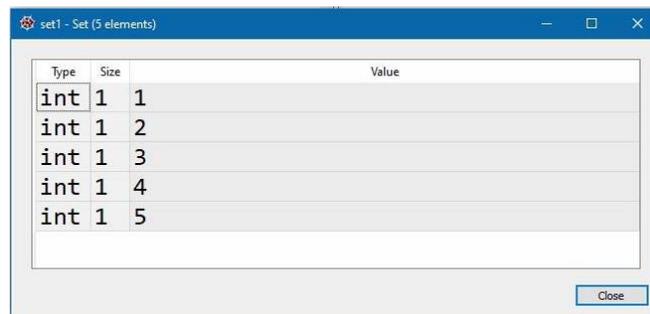
Sets

Another type of collection is a set. Like a dictionary, a set uses curly brackets `{ }` and uses the comma `,` as a delimited but does not have matched key-value pairs separated by a colon `:`. A set cannot have duplicate values, these will be removed when the set is created.

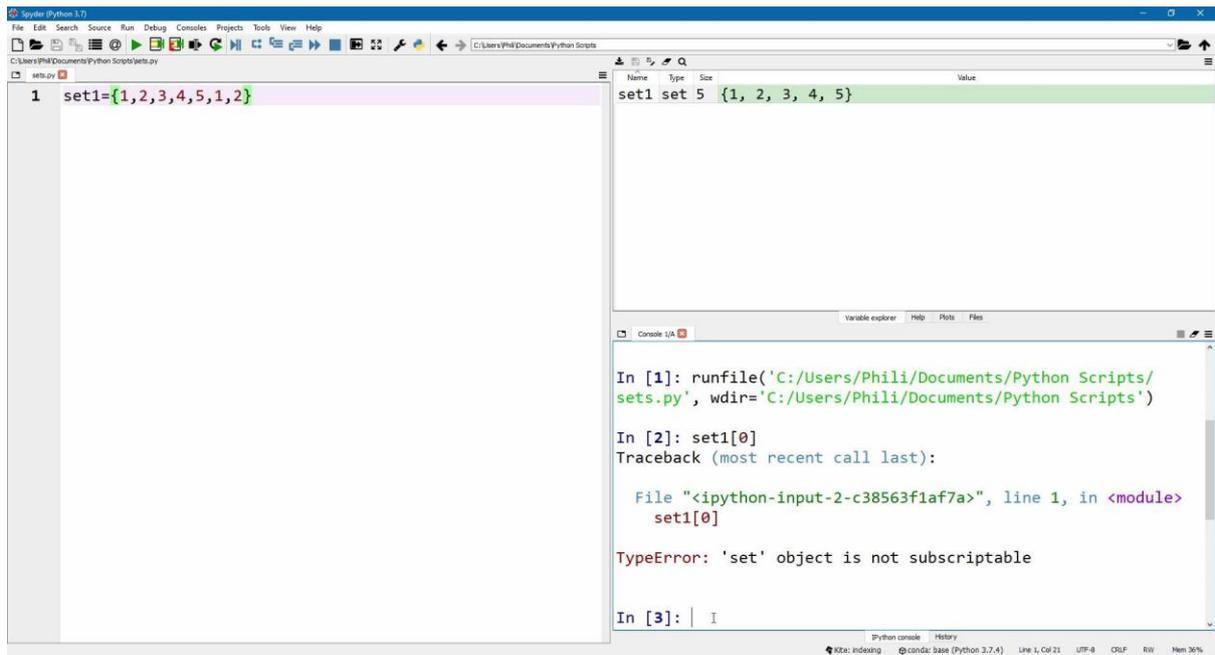
```
set1={1, 2, 3, 4, 5, 1, 2}
```



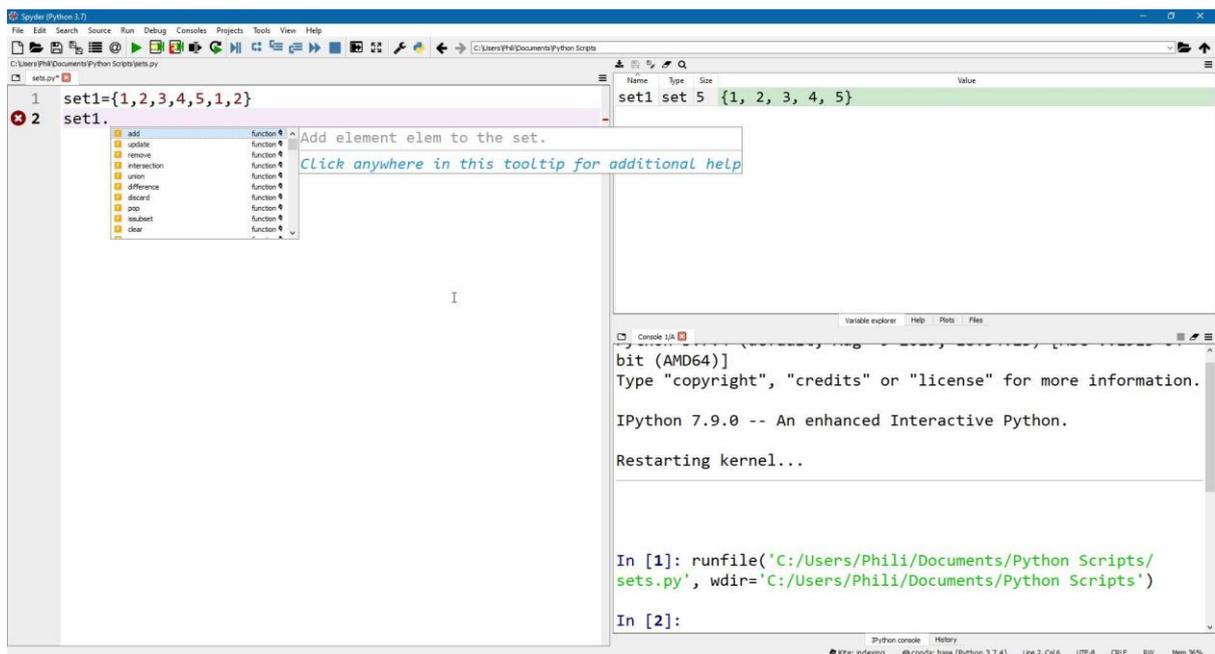
When opened in the variable explorer, one can see that only 5 values are present, although 7 values were assigned to the set. 2 were duplicates and only show once in the set.



Note also that there are no indexes for any of the values in the set. Sets are not indexable, if we try and index a numeric value, we get a `TypeError: 'set' object is not scriptable`.



Like the other collections, there are a number of methods which can be called by typing in the set name followed by a dot `.` and then the method name. The list of methods can be displayed if this is followed by a `<tab>`.



The set method `add` can be used to add an additional value to the set.

```
set1.add(6)
```

Type	Size	Value
int	1	1
int	1	2
int	1	3
int	1	4
int	1	5
int	1	6

The list method `remove` can be used to remove a value from a set.

```
set1.remove(5)
```

Type	Size	Value
int	1	1
int	1	2
int	1	3
int	1	4
int	1	6

A set is a collection of non-duplicate values. Sets are often compared with respect to one another. The set method `difference` can be used to compute the difference between the set preceding the dot (`set1`) with the set that is the input argument (`set2`) for the set method.

```
set1.difference(set2)
```

The screenshot shows the Spyder Python IDE with a script named `sets.py` containing the following code:

```
1 set1={1,2,3,4,5}
2 set2={1,2,3,7,8,9}
```

The Variable explorer on the right shows the state of the variables:

Name	Type	Size	Value
set1	set	5	{1, 2, 3, 4, 5}
set2	set	6	{1, 2, 3, 7, 8, 9}

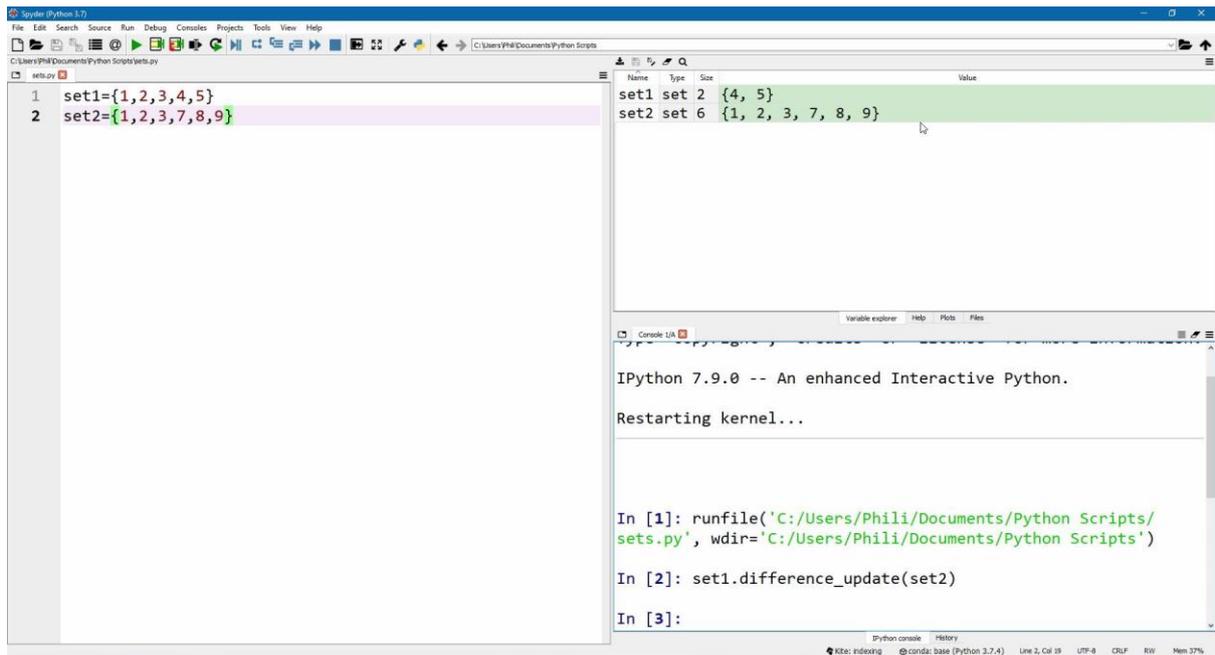
The IPython console shows the execution of the code:

```
IPython 7.9.0 -- An enhanced Interactive Python.
Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/sets.py', wdir='C:/Users/Phili/Documents/Python Scripts')
In [2]: set1.difference(set2)
Out[2]: {4, 5}
In [3]:
```

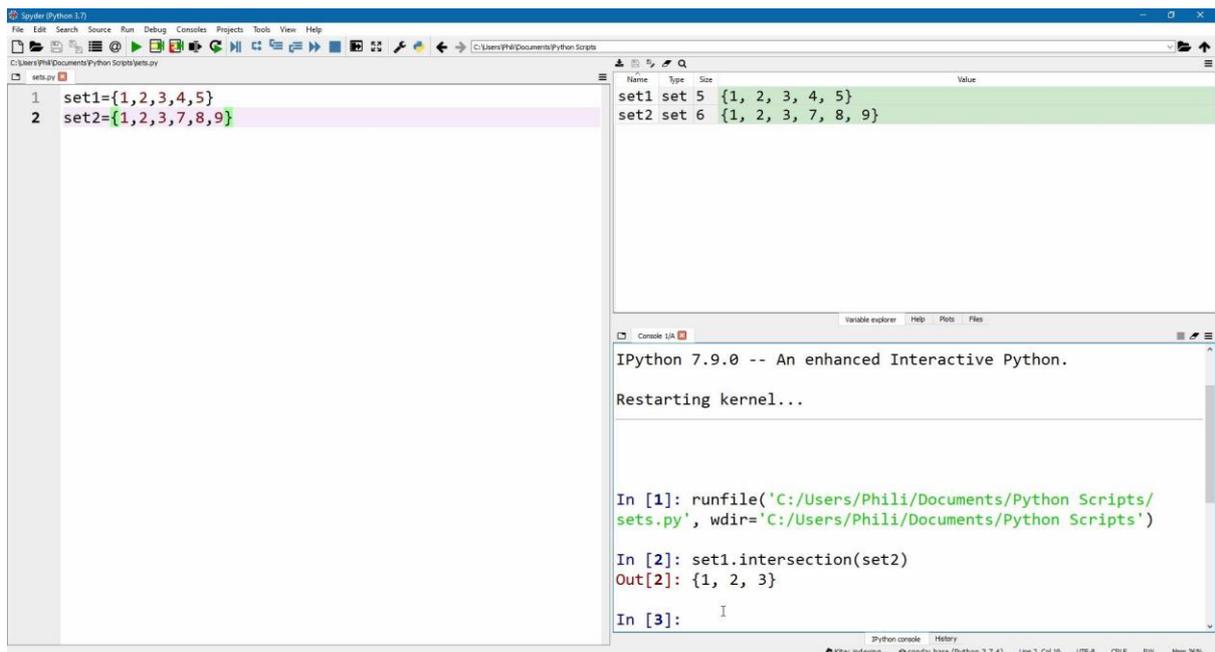
The method `difference_update` will update the first set (`set1`) when called.

```
set1.difference_update(set2)
```



The set method `intersection` will give the values which belong to both sets.

```
set1.intersection(set2)
```

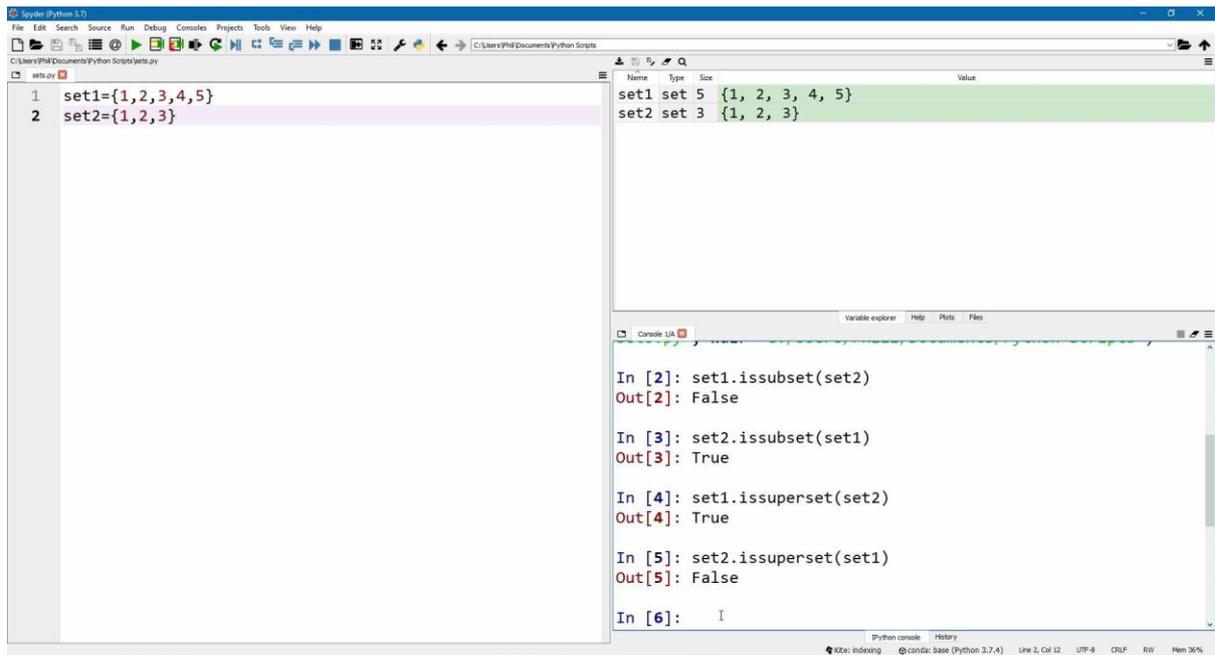


The set methods `issubset` and `issuperset`, determine whether the first set (set1) followed by the dot `.` is a subset or superset of the second set, which is the input argument.

```

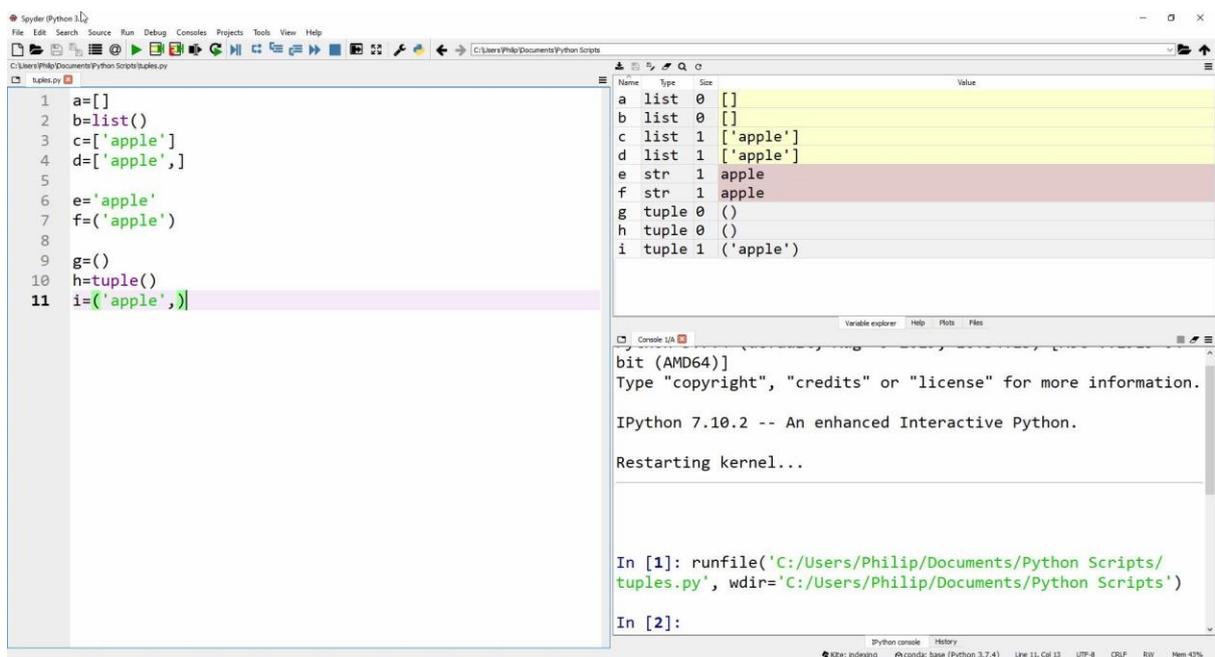
set1={1,2,3,4,5}
set2={1,2,3}
set1.issubset(set2)
set2.issubset(set1)
set1.issuperset(set2)
set2.issuperset(set1)

```



Single Value Lists, Tuples, Sets and Dictionaries

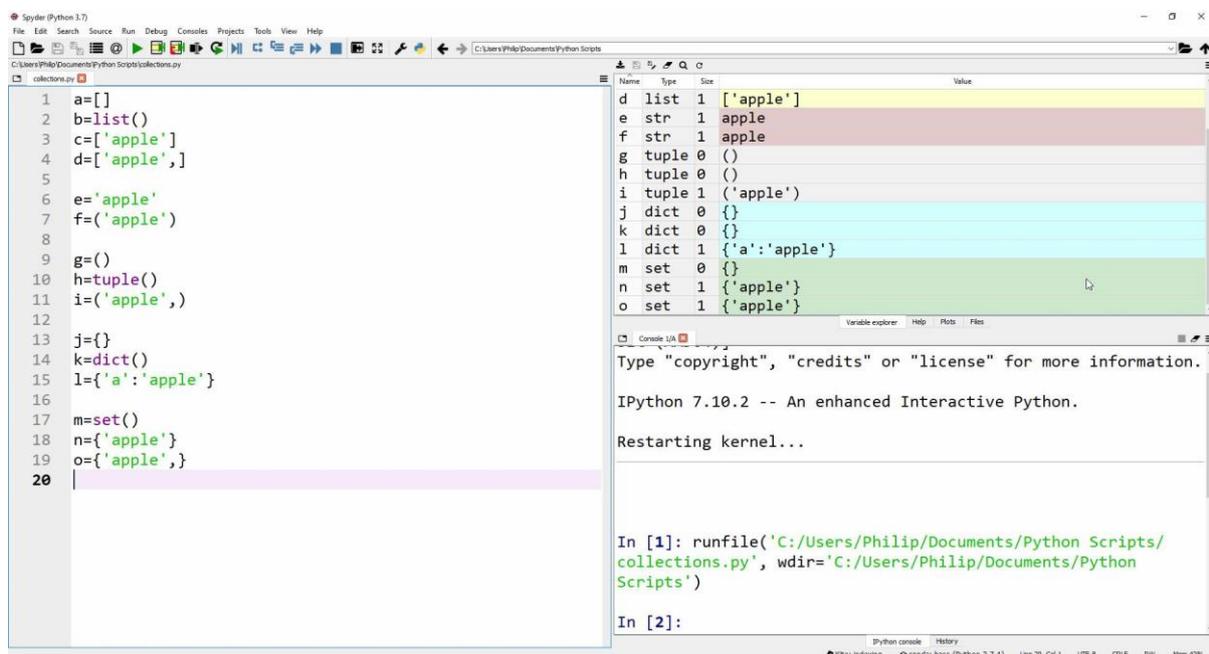
1. a=[]
2. b=list()
3. c=['apple']
4. d=['apple',]
- 5.
6. e='apple'
7. f=('apple')
- 8.
9. g=()
10. h=tuple()
11. i=('apple',)



There are some subtleties when creating an empty list, or empty tuple. A single value enclosed in `()` is taken to be a scalar value enclosed in parenthesis and not a tuple. To explicitly express it as a tuple the single value may be followed by a comma `,`. The methods `list()` and `tuple()` can be used to create an empty list or empty tuple or to create a new list from a tuple or vice versa.

Because dictionaries and sets both use `{}` and dictionaries are more commonly used than sets an empty `{}` will create a dictionary.

1. `j={}`
2. `k=dict()`
3. `l={'a':'apple'}`
- 4.
5. `m=set()`
6. `n={'apple'}`
7. `o={'apple',}`



Conditional Logic

Logical Operators

So far, the logical operators equal to `==` and `is` have been examined when strings to see if they are equal in value and if they are equal in value and the same object in memory. These have inverse operators not equal to `!=` and `is not`. When dealing with numerical values such as ints and floats, there are additional logical operators to compare their numerical value with respect to one another.

Logical Operator	Definition
<code>==</code>	equal to
<code>!=</code>	not equal to
<code><</code>	less than
<code><=</code>	less than or equal to
<code>>=</code>	greater than or equal to
<code>></code>	greater than
<code>is</code>	is (the same object in memory)
<code>is not</code>	is not (the same object in memory)

For instance we can create:

```
a=1  
b=2
```

We can check if a is greater than b:

```
a>b
```

False

We can check if a is less than b:

```
a<b
```

True

We can check if a is greater than or equal to b:

```
a>=b
```

False

We can check if a is less than or equal to b:

```
a<=b
```

True

We can check if a equal to b:

```
a==b
```

False

We can check if a is not equal to b:

```
a!=b
```

True

Combining Logical Operators

The logical operators `or` and `and` may be used to combine 2 or multiple conditions and carry the same meanings as they do in the English language.

In the case of the `and` operator, the output condition will be `True` only if both subsequent input conditions are `True`. For example:

```
True and True
```

```
True
```

```
True and False
```

```
False
```

```
False and True
```

```
False
```

```
False and False
```

```
False
```

In the case of the `or` operator, the output condition will be `True` if one or both subsequent input conditions are `True`. For example:

```
True or True
```

```
True
```

```
True or False
```

```
True
```

```
False and True
```

```
True
```

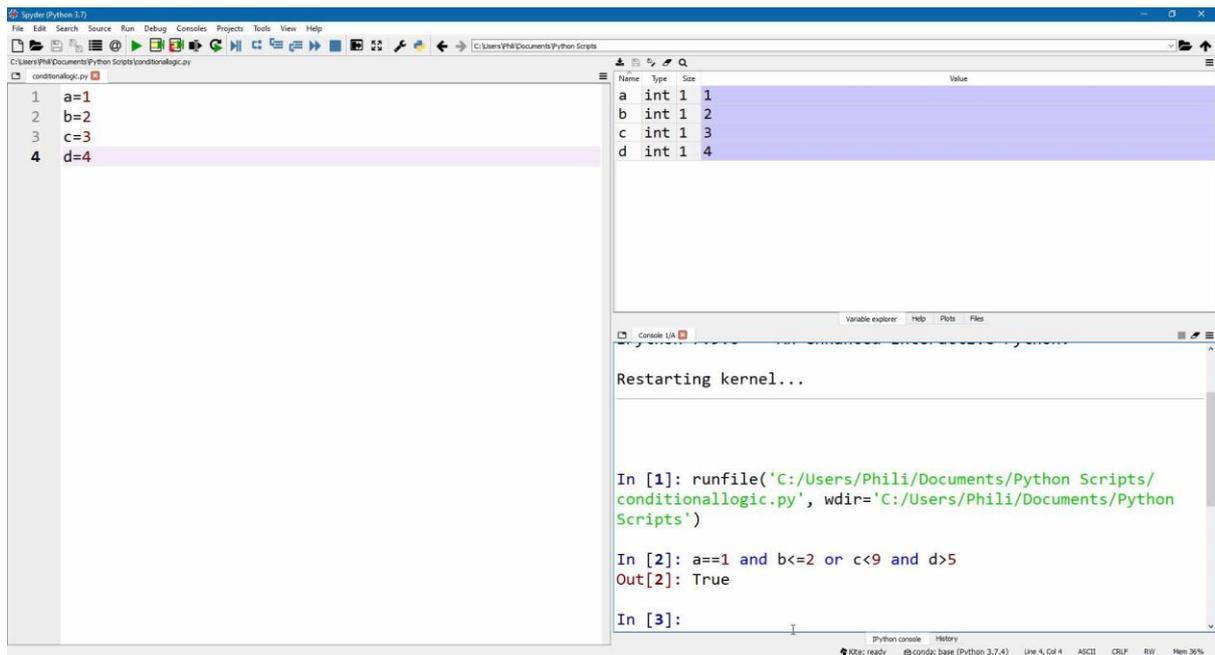
```
False and False
```

```
False
```

It is possible to use `and` or `or` to combine multiple conditions. However as more and more conditions are combined, the code can get more confusing, in which case it is recommended to use parenthesis, recalling that the operations in inner parenthesis are carried out first. Take for example:

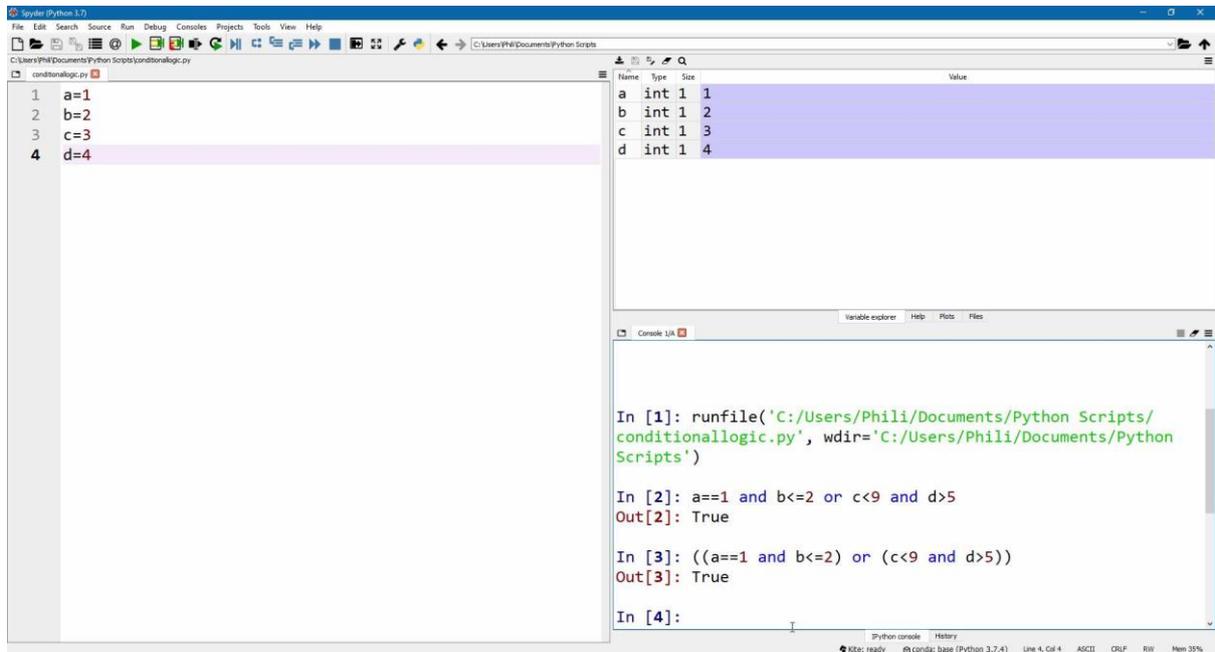
1. `a=1`
2. `b=2`
3. `c=3`
4. `d=4`
5. `a==1 and b<=2 or c<9 and d>5`

True



In this case, the two **and** statements are carried out first and the above when written out with parenthesis can be thought of as:

1. a=1
2. b=2
3. c=3
4. d=4
5. $((a==1 \text{ and } b \leq 2) \text{ or } (c < 9 \text{ and } d > 5))$



Examining the green brackets first:

$((\text{True and True}) \text{ or } (c < 9 \text{ and } d > 5))$

$((\text{True}) \text{ or } (c < 9 \text{ and } d > 5))$

Then the cyan brackets second:

```
((True) or (True and False))
```

```
((True) or (False))
```

Then the magenta brackets third:

```
(True or False)
```

True

Alternatively if parenthesis was used to carry out the **or** condition first, a different result would be yielded:

1. a=1
2. b=2
3. c=3
4. d=4
5. `((a==1 and (b<=2 or c<9)) and d>5)`

The screenshot shows a Python IDE window titled 'Spyder (Python 3.7)'. The editor on the left contains the following code:

```
1 a=1
2 b=2
3 c=3
4 d=4
```

The Variable explorer on the right shows the following variables and their values:

Name	Type	Size	Value
a	int	1	1
b	int	1	2
c	int	1	3
d	int	1	4

The Console window at the bottom shows the following execution history:

```
In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/conditionallogic.py', wdir='C:/Users/Phili/Documents/Python Scripts')
In [2]: a==1 and b<=2 or c<9 and d>5
Out[2]: True
In [3]: ((a==1 and b<=2) or (c<9 and d>5))
Out[3]: True
In [4]: ((a==1 and (b<=2 or c<9)) and d>5)
Out[4]: False
In [5]:
```

Examining the green brackets first:

```
((a==1 and (True or True)) and d>5)
```

```
((a==1 and True) and d>5)
```

Then the cyan brackets second:

```
((True and True) and d>5)
```

Then the magenta brackets third:

```
(True and d>5)
```

```
(True and False)
```

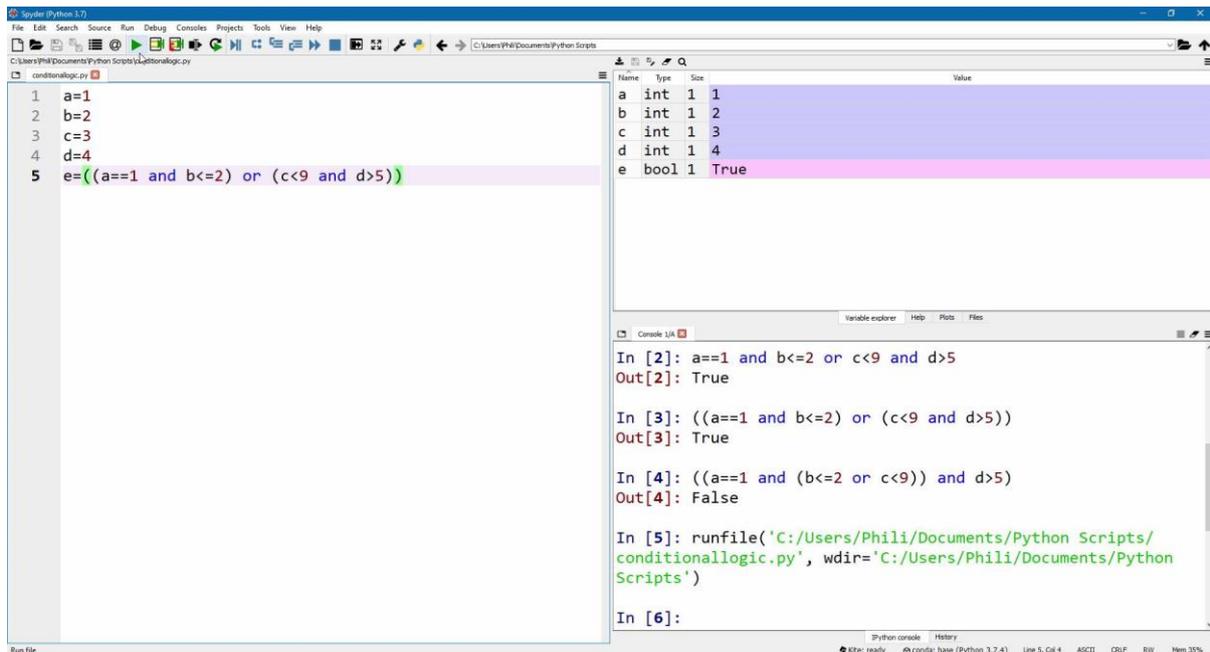
False

Logical Operators and the Assignment Operator

The logical equals operator `==` shouldn't be confused with the assignment operator `=` and they are often used together. For example:

```
e = ((a==1 and b<=2) or (c<9 and d>5))
```

Recall that the right-hand side of the assignment operator `=` is examined first. Therefore, whatever the value in the magenta brackets is calculated and then assigned to the variable `e`. From before, we seen this is the Boolean value `True`. As a result `e` will have a value of `True`.



The screenshot shows a Jupyter Notebook window with the following code in the editor:

```
1 a=1
2 b=2
3 c=3
4 d=4
5 e=((a==1 and b<=2) or (c<9 and d>5))
```

The Variable explorer on the right shows the following variables:

Name	Type	Size	Value
a	int	1	1
b	int	1	2
c	int	1	3
d	int	1	4
e	bool	1	True

The console output shows the following interactions:

```
In [2]: a==1 and b<=2 or c<9 and d>5
Out[2]: True

In [3]: ((a==1 and b<=2) or (c<9 and d>5))
Out[3]: True

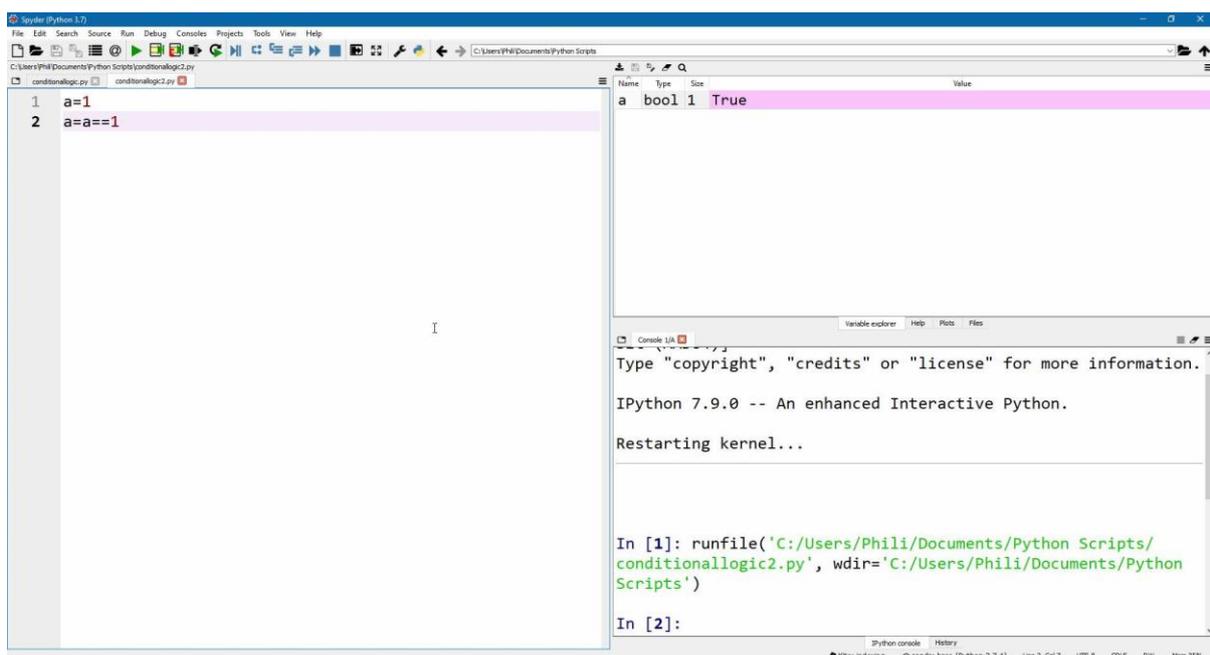
In [4]: ((a==1 and (b<=2 or c<9)) and d>5)
Out[4]: False

In [5]: runfile('C:/Users/Phili/Documents/Python Scripts/conditionallogic.py', wdir='C:/Users/Phili/Documents/Python Scripts')

In [6]:
```

The following code may look confusing when first encountered.

1. `a=1`
2. `a=a==1`



The screenshot shows a Jupyter Notebook window with the following code in the editor:

```
1 a=1
2 a=a==1
```

The Variable explorer on the right shows the following variables:

Name	Type	Size	Value
a	bool	1	True

The console output shows the following interactions:

```
Type "copyright", "credits" or "license" for more information.
IPython 7.9.0 -- An enhanced Interactive Python.
Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/conditionallogic2.py', wdir='C:/Users/Phili/Documents/Python Scripts')

In [2]:
```

In line 1 the variable `a` is initially assigned the value of `1` which is an int. In line 2, the right-hand side of the assignment operator `=` is examined first. The logical operator equals `==` checks if the variable `a` is equal to the value `1`, in this case it is, so the right-hand side of the assignment operator becomes `True`. The variable `a` is reassigned to this new value of `True` which is a Boolean.

Inverting Logical Operators

The word `not` can be used to invert logical operators. For example:

```
not True
```

```
False
```

```
not False
```

```
True
```

Adding `not` to the following:

```
e = not ((a==1 and b<=2) or (c<9 and d>5))
```

Looks at whatever the value in the magenta brackets is, in this case `True` then inverts it making it `False` and assigns this value to the variable name `e`.

```
1 a=1
2 b=2
3 c=3
4 d=4
5 e=not((a==1 and b<=2) or (c<9 and d>5))
```

Name	Type	Size	Value
a	int	1	1
b	int	1	2
c	int	1	3
d	int	1	4
e	bool	1	False

```
Out[3]: True
In [4]: ((a==1 and (b<=2 or c<9)) and d>5)
Out[4]: False
In [5]: runfile('C:/Users/Phili/Documents/Python Scripts/conditionallogic.py', wdir='C:/Users/Phili/Documents/Python Scripts')
In [6]: runfile('C:/Users/Phili/Documents/Python Scripts/conditionallogic.py', wdir='C:/Users/Phili/Documents/Python Scripts')
In [7]:
```

if, elif (else if), else

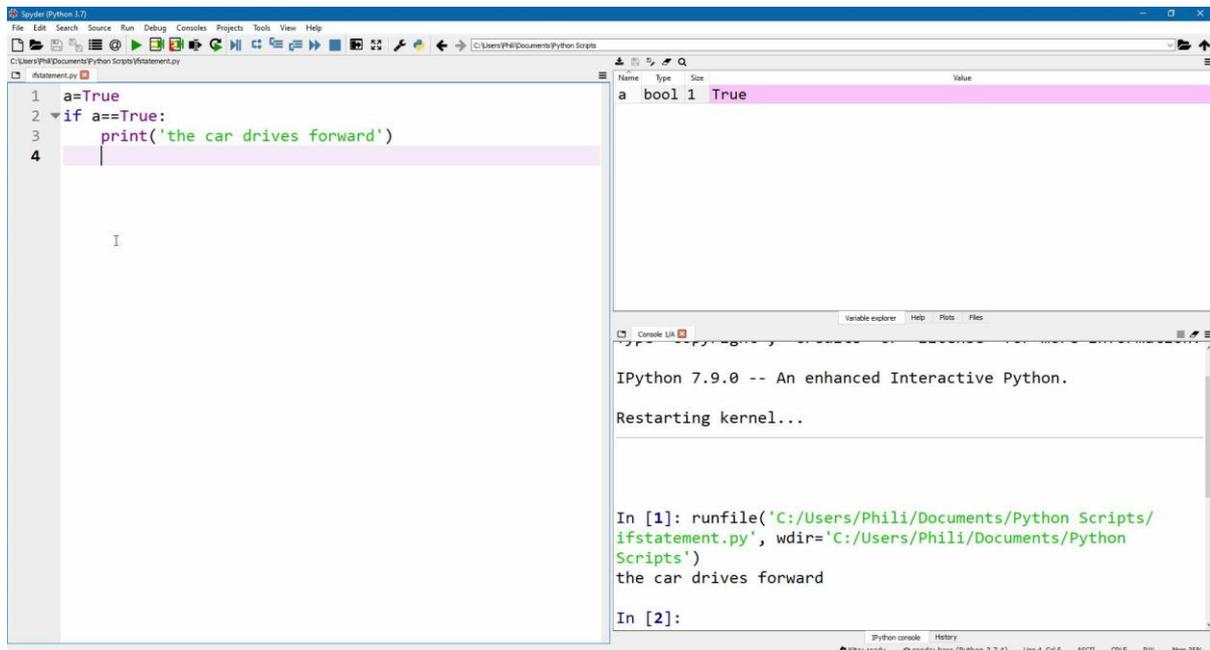
So far, all code has been executed sequentially line by line. An analogy is driving a car only on a straight road from start to finish.



Using conditional logic, it is possible to execute code only **if** a certain condition is met. To do this we use an **if** condition followed by a colon **:** and any code belonging to this **if** statement is embedded by 4 spaces. Once this indented code is written, a blank space is left behind afterwards, indicating that this is the end of the code belonging to the **if** statement.

In this case, the code within this **if** statement will be executed because the condition is **True**.

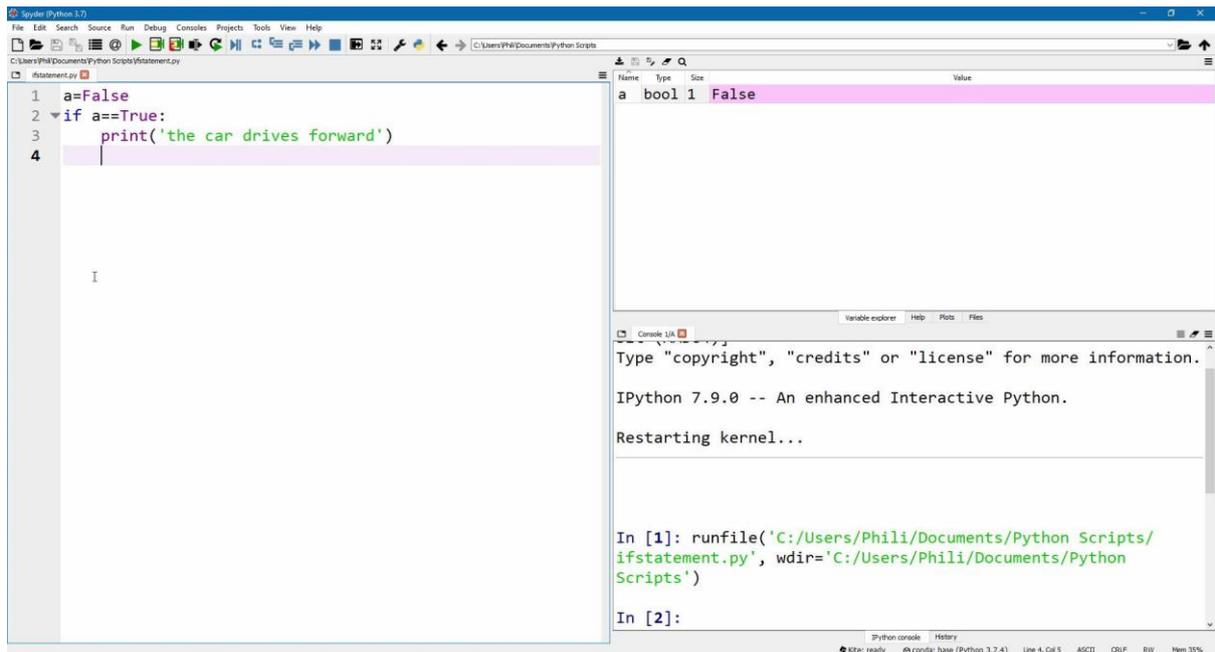
1. `a=True`
2. `if a==True:`
3. `print('the car drives forward')`
- 4.



Now in contrast, the following code will not be executed because the condition is **False**.

1. `a=False`
2. `if a==True:`
3. `print('the car drives forward')`
- 4.

In such a case, nothing is printed, and no instructions are given if the condition isn't satisfied.



The code belonging to the `if` statement, in this case the first `print` statement is indented by four spaces and then followed by a blank line. The second `print` statement after this blank line does not belong to the `if` statement and therefore is not indented. This line of code will thus always be executed regardless of the driver's decision to continue driving forward or not, in this case it would have rained independent of the driver's decision to drive on or not.

```

1. a=False
2. if a==True:
3.     print('the car drives forward')
4.
5. print('it begins to rain')

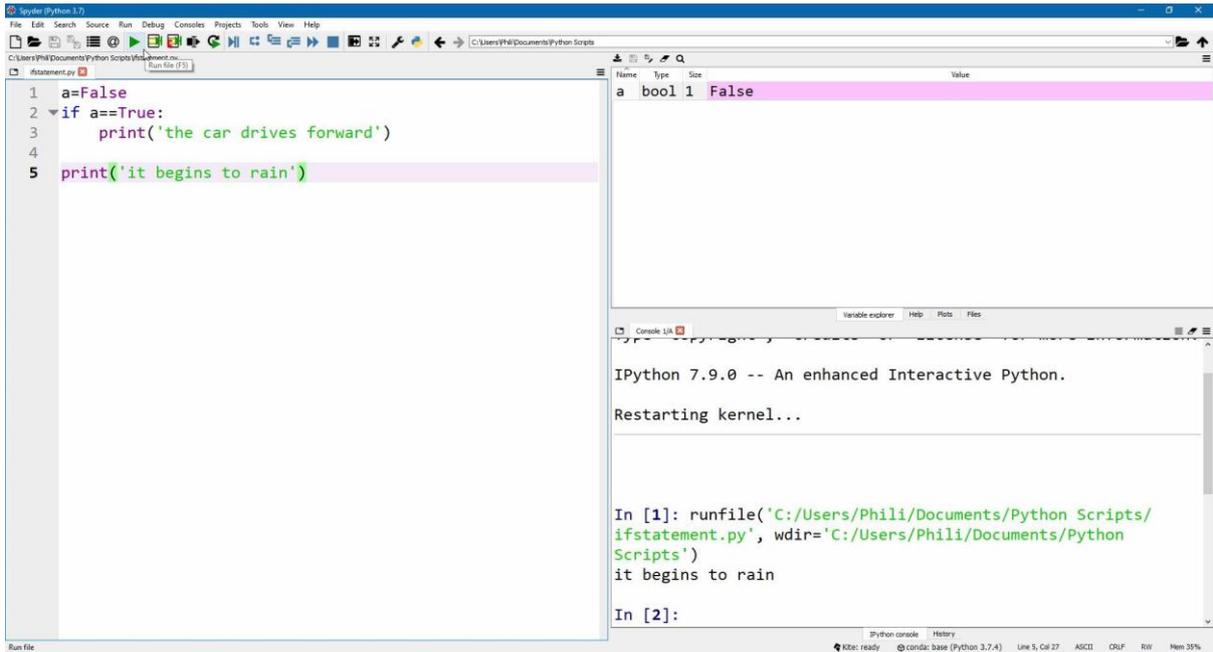
```

When using an `if` statement, highlighted in green, pay attention to indentation marked in yellow and spacing marked in red.

```

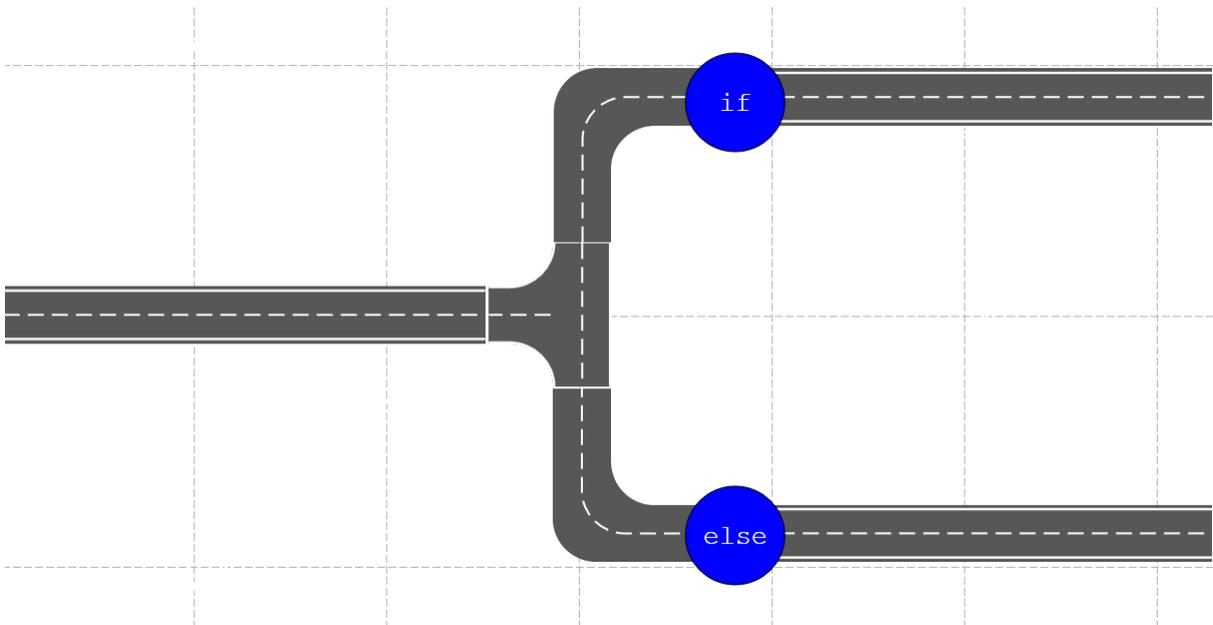
1. print('I am independent of the if statement')
2. if True:
3.     print('I belong to the if statement')
4.     print('I also belong to the if statement')
5.
6. print('I am also independent of the if statement')
7. print('I am again independent of the if statement')

```



Using conditional logic, it is possible to execute code only **if** a certain condition is met and otherwise or **else** execute code should that condition not be met.

Continuing with an analogy of a car on a road, it is the equivalent of reaching a junction.



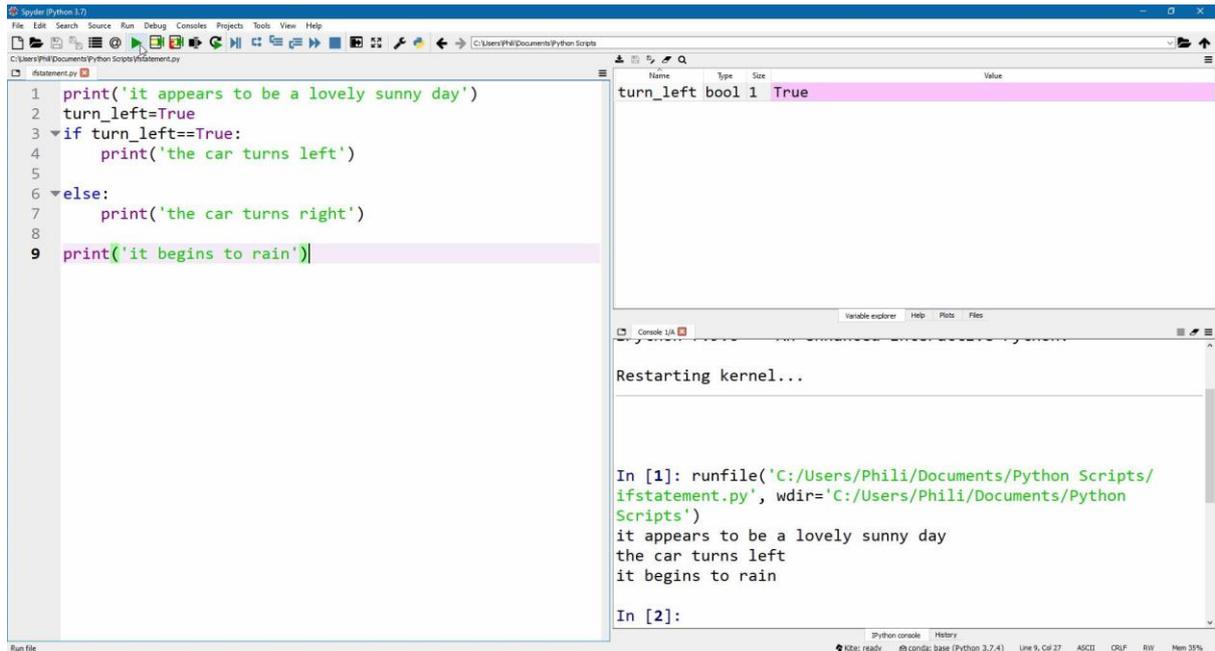
At the junction, a decision can be made by the driver, to either turn left or not. **if** the driver makes the decision to turn left, their car will turn left, and it will be observed driving up the left side of the junction following the decision. In contrast the only other option is **else**, and in this case, not driving left is assumed to indicate, driving up the right-hand side of the junction, so the driver will be observed driving up the right side of the junction. There is no circumstance when the car is observed to be simultaneously driving up both junctions (the car can only go up, one of the two junctions, not both at the same time).

In this case the driver decides to turn left.

```

1. print('it appears to be a lovely sunny day')
2. turn_left=True
3. if turn_left==True:
4.     print('the car turns left')
5.
6. else:
7.     print('the car turns right')
8.
9. print('it begins to rain')

```



Note once again note the use of colons `:`, four spaces (yellow) and blank lines (red). There is no condition attached to the `else`, as by definition `else` will only occur if the condition for `if` is not satisfied.

```

1. print('it appears to be a lovely sunny day')
2. turn_left=True
3. if turn_left==True:
4.     print('the car turns left')
5.
6. else:
7.     print('the car turns right')
8.
9. print('it begins to rain')

```

In contrast, this is what would have happened had the driver decided to not turn left.

```

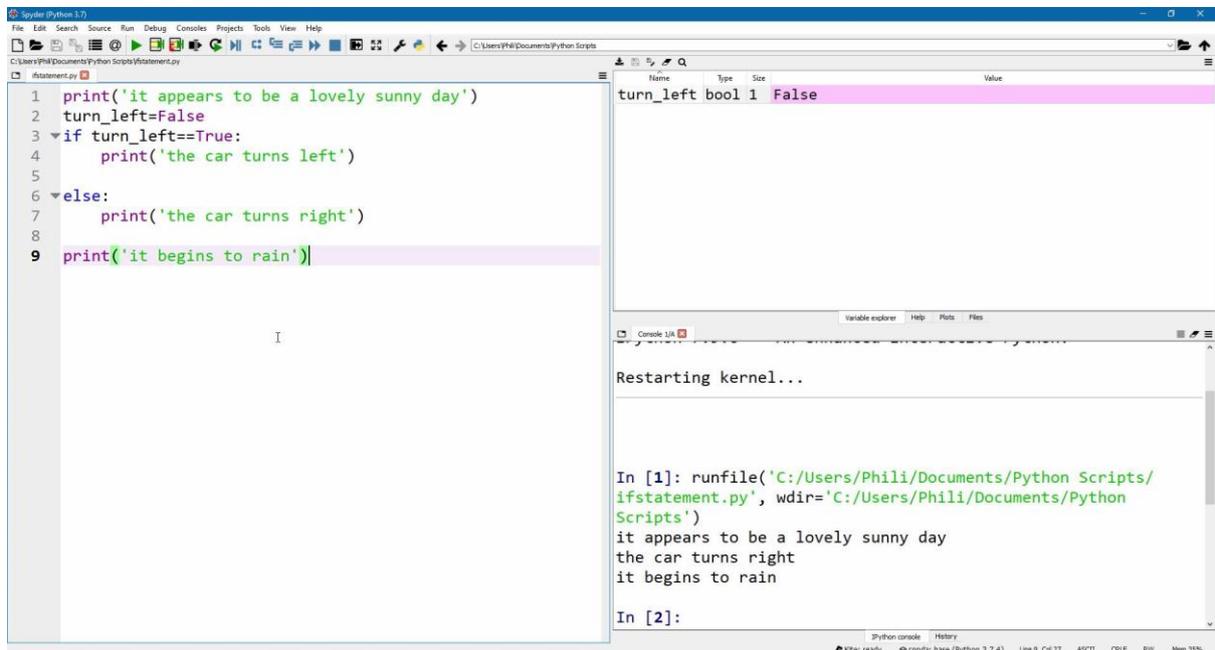
1. print('it appears to be a lovely sunny day')
2. turn_left=False
3. if turn_left==True:
4.     print('the car turns left')
5.

```

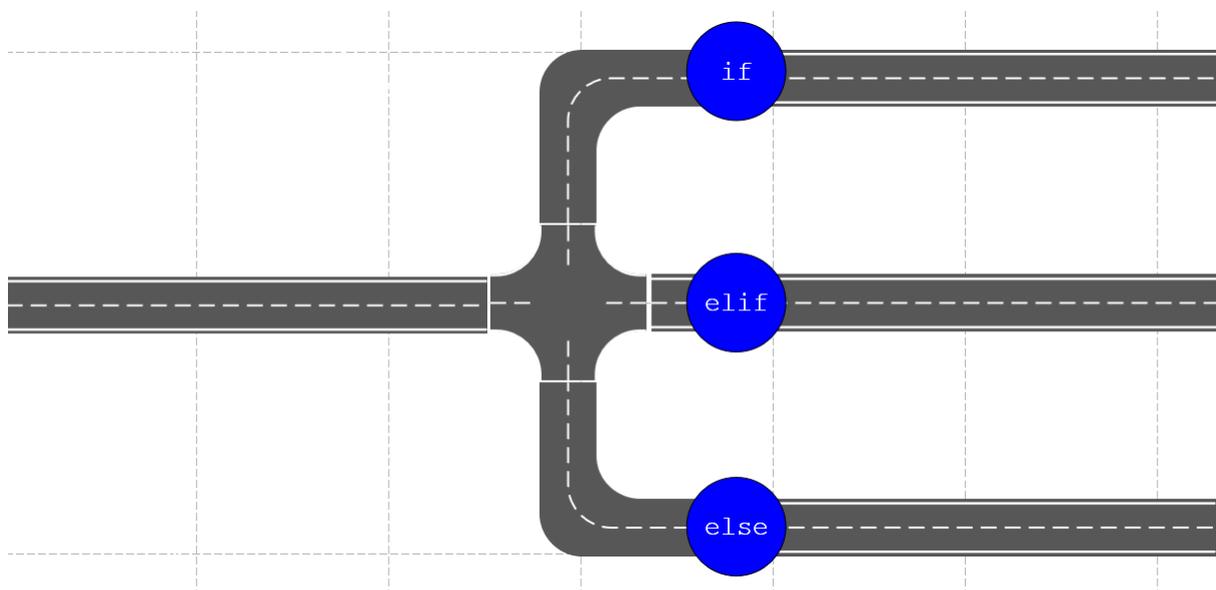
```

6. else:
7.     print('the car turns right')
8.
9. print('it begins to rain')

```



It is also possible to have a junction with more than two options. This can be thought of a car driving towards a three-way junction.



```

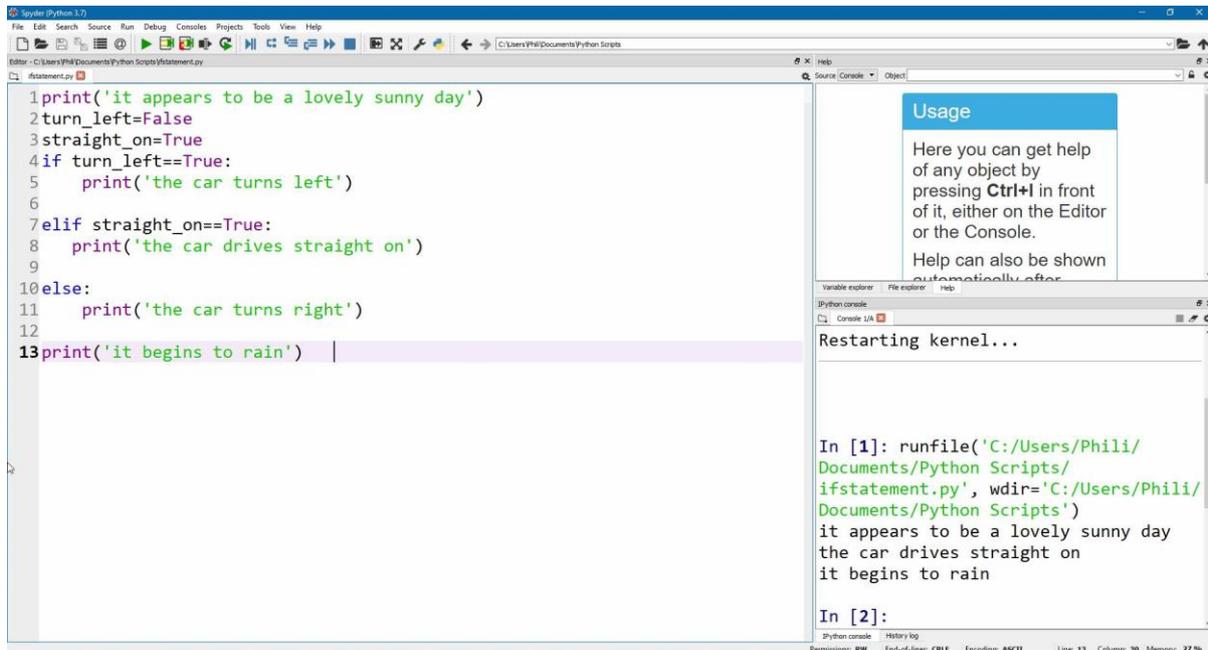
1. print('it appears to be a lovely sunny day')
2. turn_left=False
3. straight_on=True
4. if turn_left==True:
5.     print('the car turns left')
6.
7. elif straight_on==True:

```

```

8.     print('the car drives straight on')
9.
10. else:
11.     print('the car turns right')
12.
13. print('it begins to rain')

```



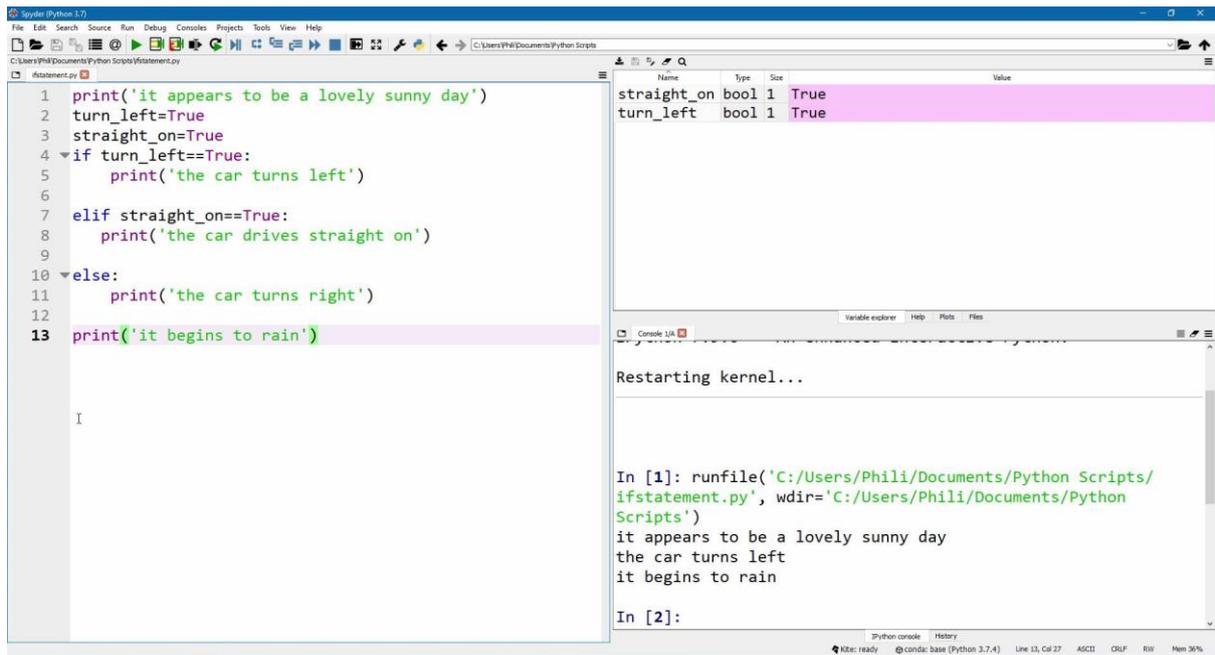
Multiple `elif` statements are possible. Continuing the analogy this leads to a junction with more and more options.

The code of the 3 way junction can be changed so that, the condition in the `if` statement and the condition in the else if `elif` statement are both `True`.

```

1. print('it appears to be a lovely sunny day')
2. turn_left=True
3. straight_on=True
4. if turn_left==True:
5.     print('the car turns left')
6.
7. elif straight_on==True:
8.     print('the car drives straight on')
9.
10. else:
11.     print('the car turns right')
12.
13. print('it begins to rain')

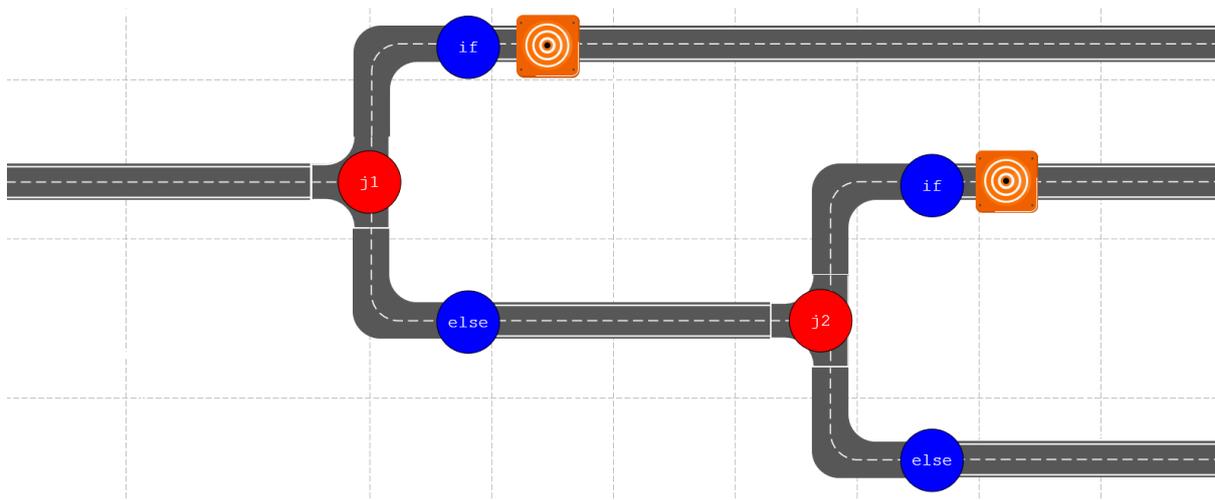
```



When this code is executed, you'll notice that only the code in the `if` statement has been executed despite the condition in the `elif` being `True`. Returning to the analogy of the car at the intersection, once the car has chosen a route to go along the intersection, it cannot choose a different route and must continue driving along the initial direction taken. Be careful when specifying the conditions of multiple else if `elif` statements.

Nested if, elif, else

Continuing with the analogy of a car on the road. It is possible to have a road that has an intersection `j1` that is later followed by another intersection `j2`. The second intersection `j1` can be thought of as being a nested intersection.



```

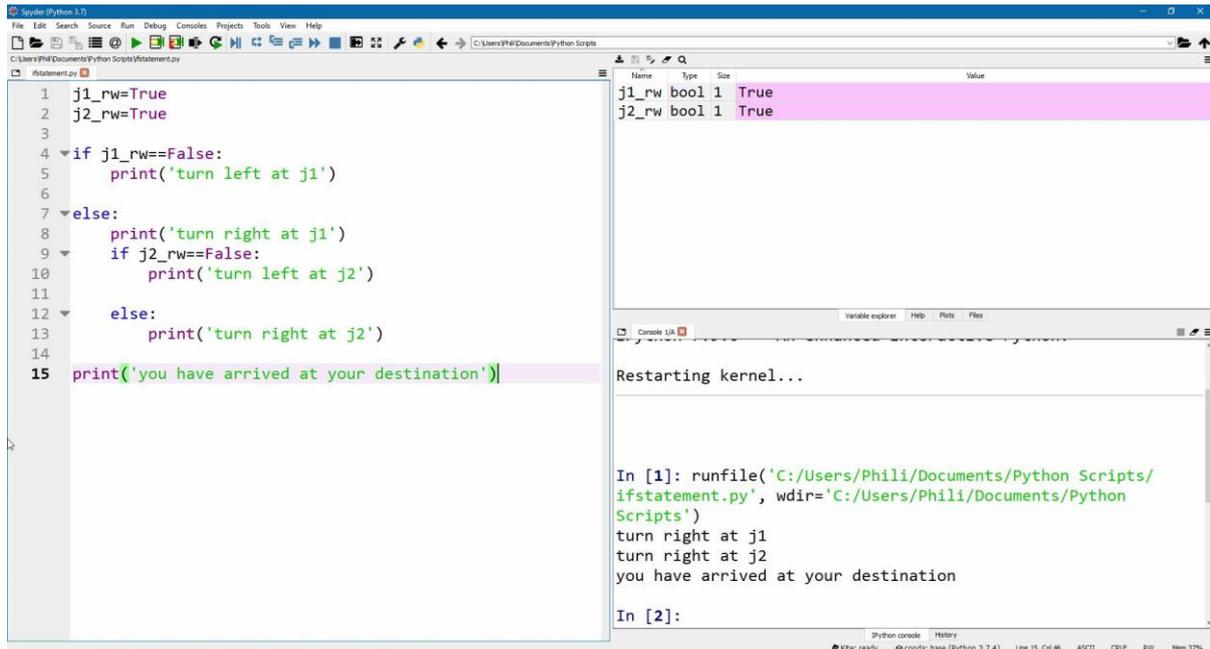
1. j1_rw=True
2. j2_rw=True
3.
4. if j1_rw==False:
5.     print('turn left at j1')
6.

```

```

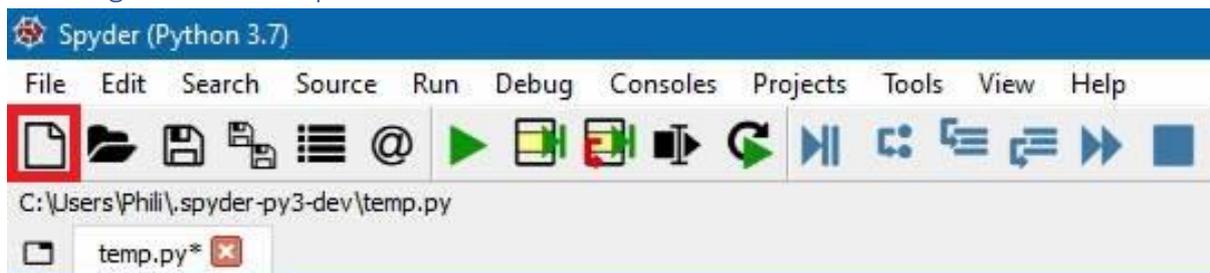
7. else:
8.     print('turn right at j1')
9.     if j2_rw==False:
10.        print('turn left at j2')
11.
12.    else:
13.        print('turn right at j2')
14.
15. print('you have arrived at your destination')

```

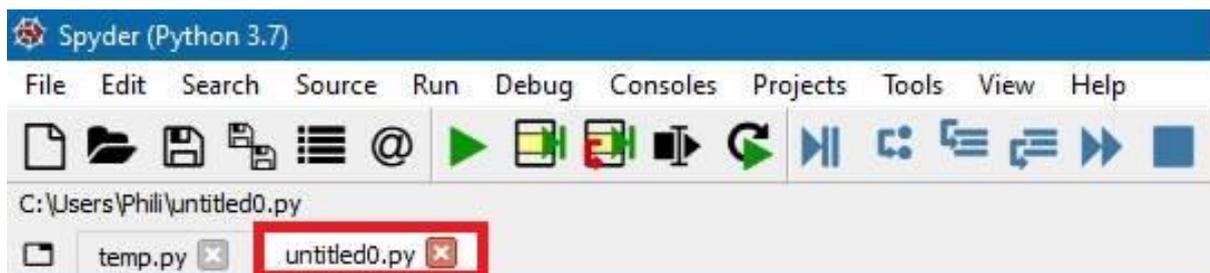


Working with Multiple Python Scripts

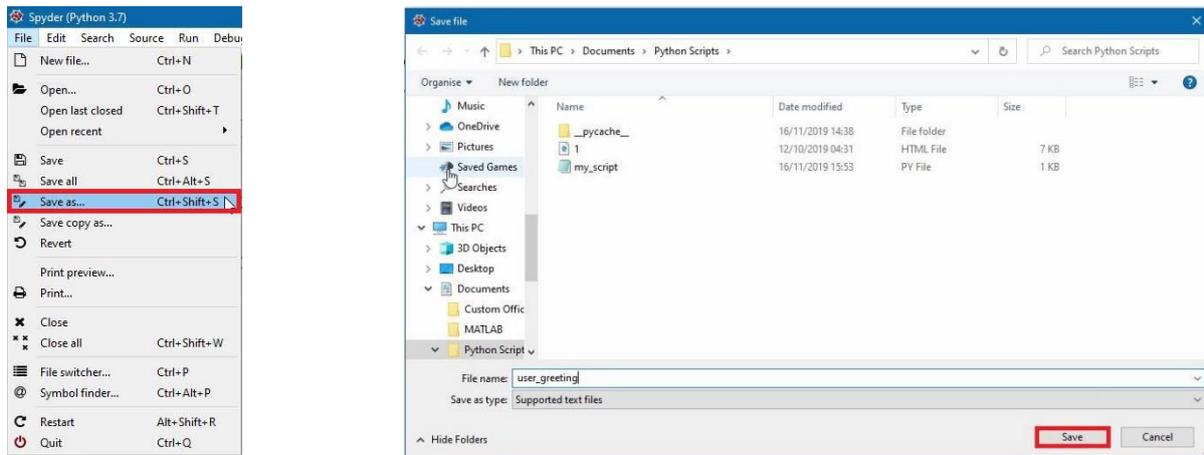
Creating a Nested Script



The new script button can be created by selecting the new script button.

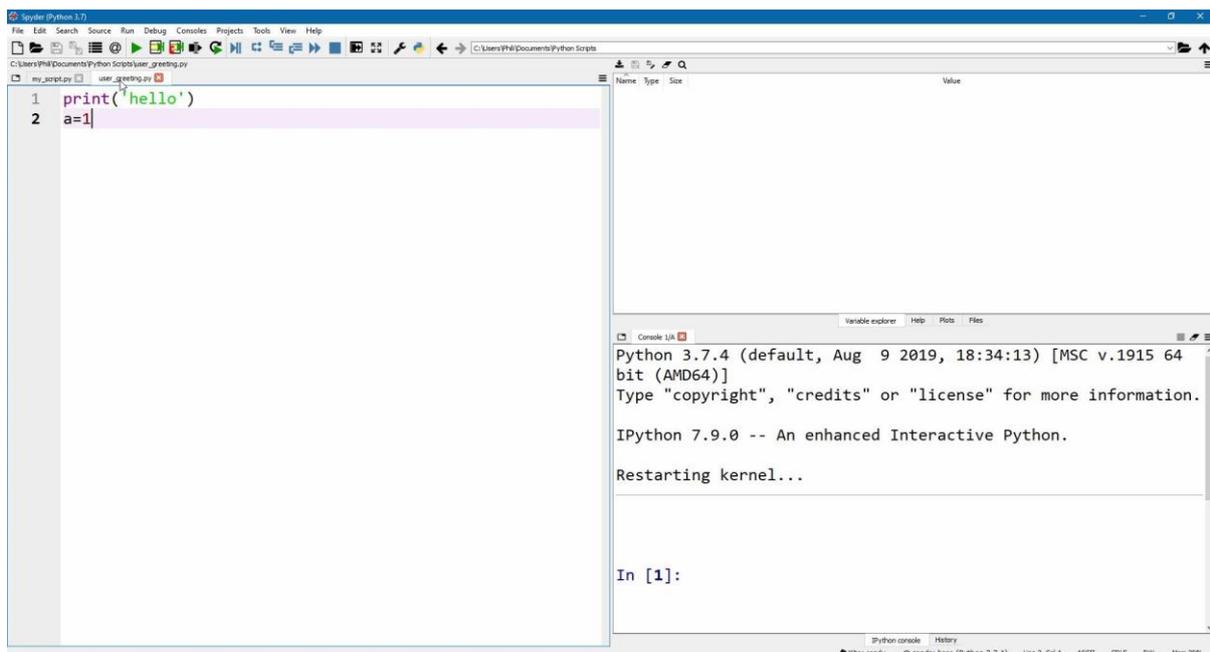


The File ↓ Save as... can be used to save the Python script files. In this case, the script files will be saved as `my_script` and `user_greeting` using the convention behind variable names.



In the script file `user_greeting`, the following code will be added.

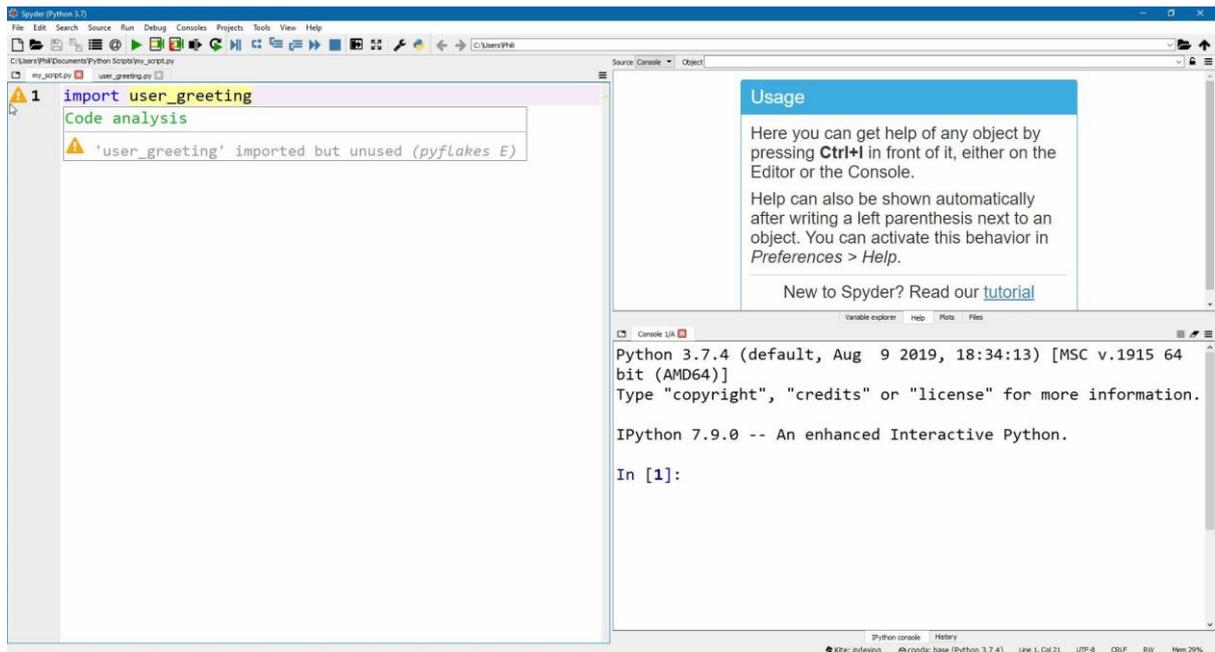
1. `print('hello')`
2. `a=1`



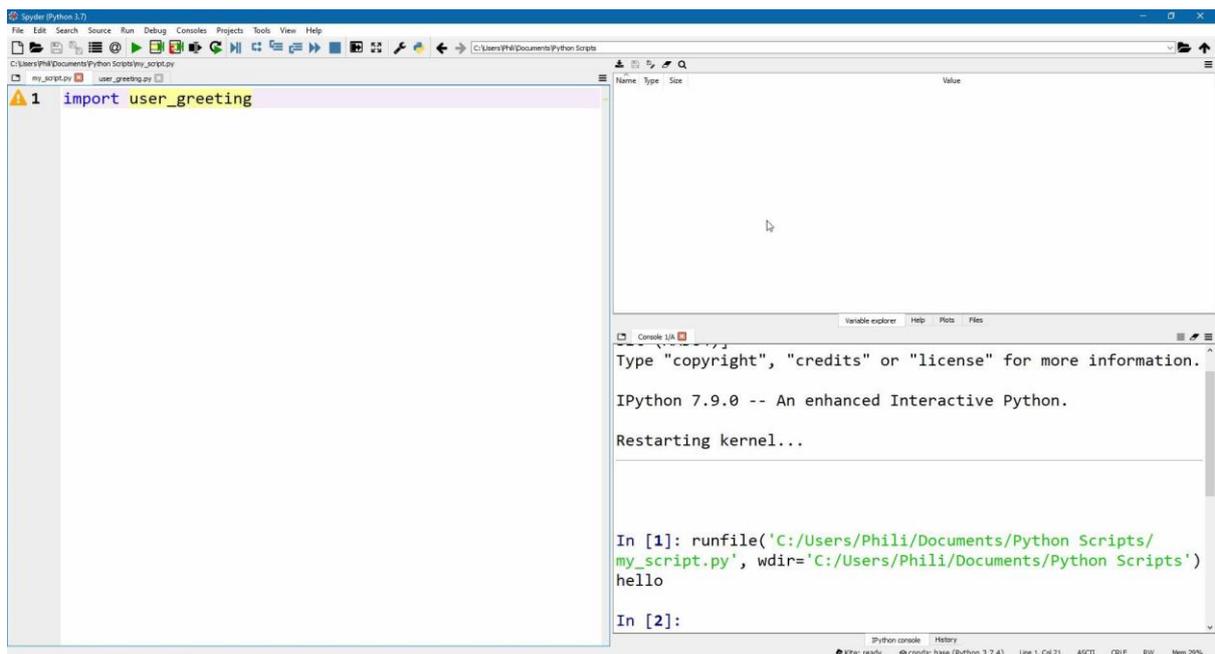
In the script file `my_script`, the script file `user_greeting` will be called by using

1. `import user_greeting`

There will be a warning exclaiming that `'user_greeting'` is imported but unused.



When `my_script` is imported, the print statement under `user_greeting` will display however the code within `user_greeting` is not executed, this can be seen from the absence of the variable `a` in the variable explorer.



Creating Custom Functions

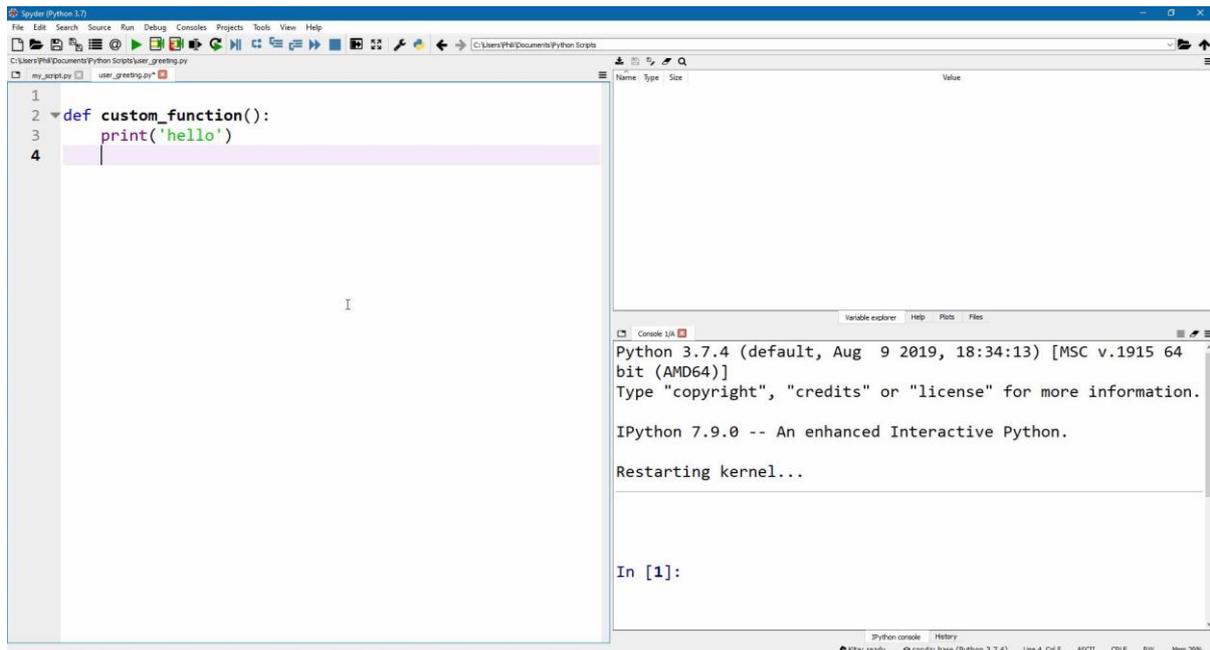
We can use the function `def` to define a custom function. This is followed by the function name, which follows the same rules as for variable names. The function name is then followed by parenthesis `()`, which will contain the input arguments, in this case there are none, so they are empty and then a colon `:`

```

1.
2. def custom_function():
3.     print('hello')
4.

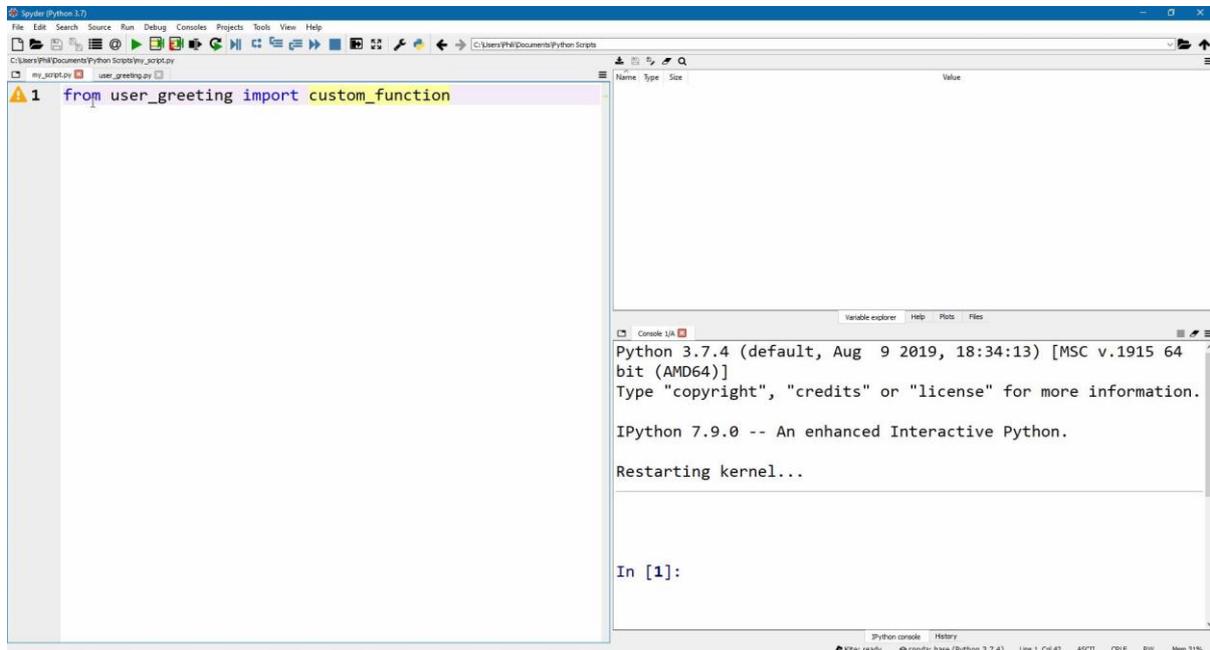
```

Like an `if`, statement, any code belonging to this function is indented by four spaces, in this case just a simple print statement. By convention a blank line is left before and after a function.



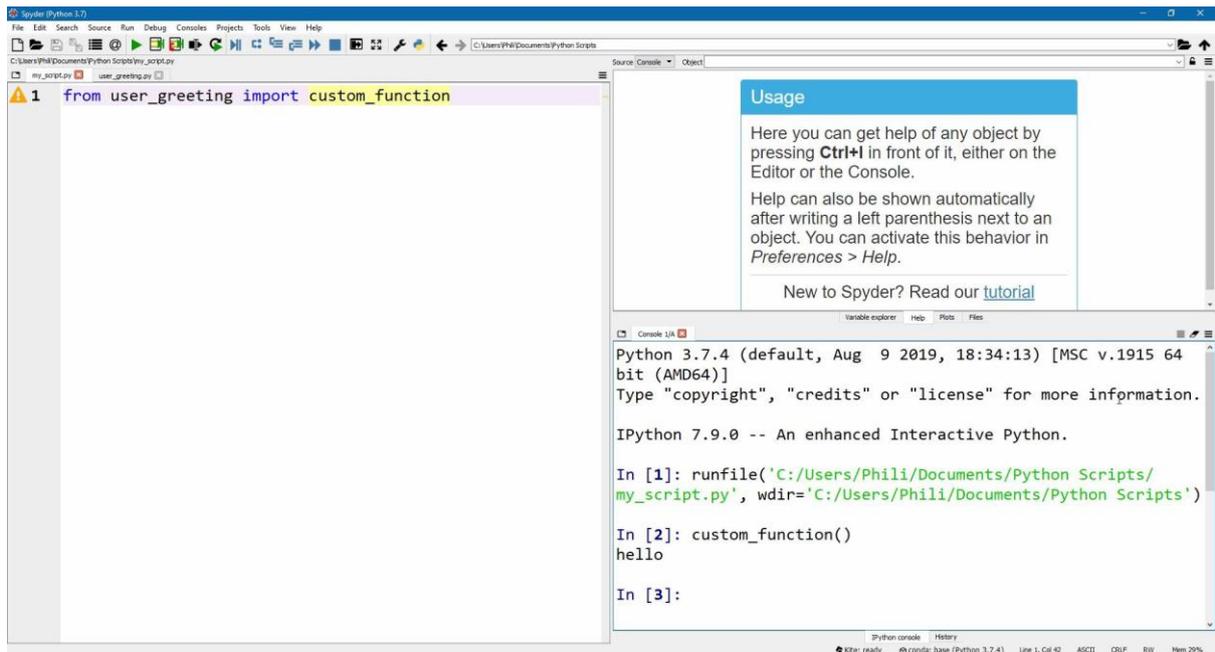
Now in `my_script`, we must `import` the function `custom_function` `from` the script file `user_greeting`. To do this we use:

1. `from user_greeting import custom_function`



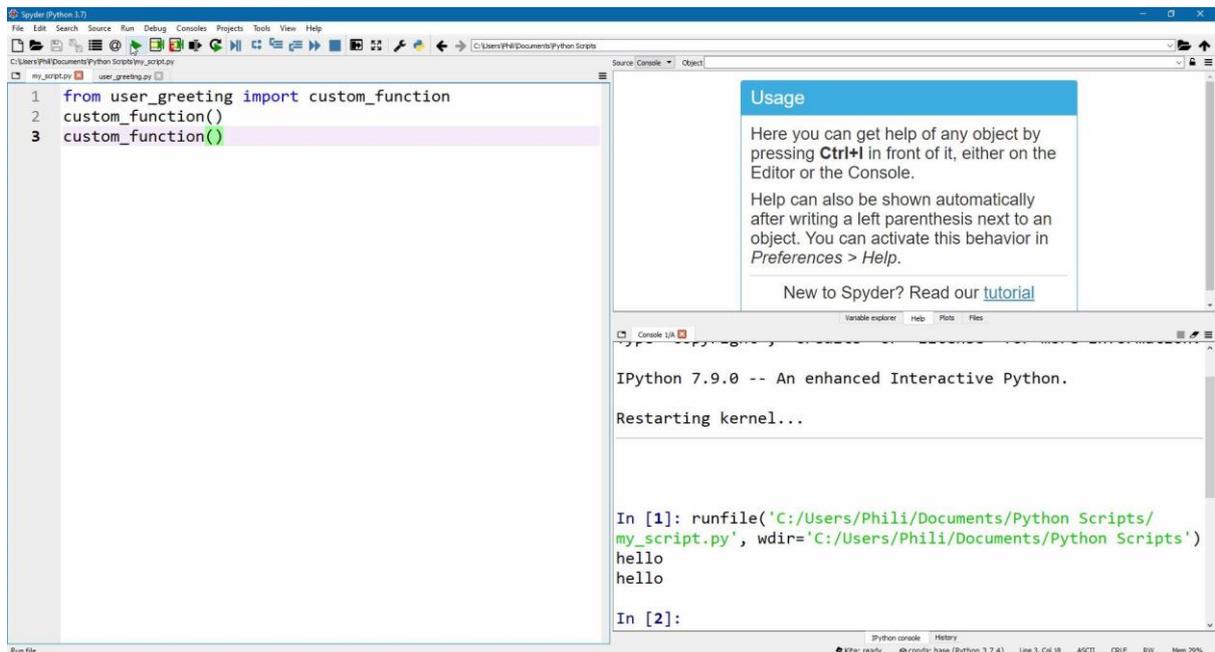
Once again, we will get the warning that the function has been imported but not used. Once imported it can be executed like any other function i.e. followed by parenthesis and in this case, no input arguments. For example, in the console.

```
custom_function()
```



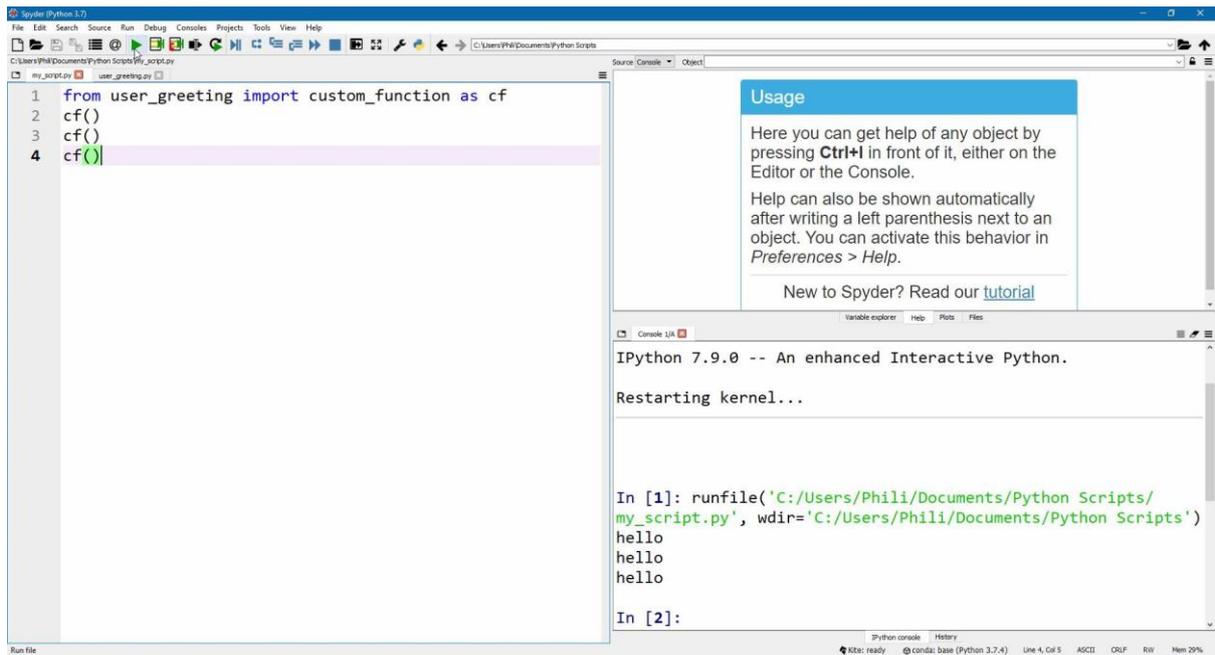
Or within the script file itself, for example in this case it will be called twice.

1. `from user_greeting import custom_function`
2. `custom_function()`
3. `custom_function()`



As a shortcut it can be imported as an abbreviation. This can then be called easily a number of times.

1. `from user_greeting import custom_function as cf`
2. `cf()`
3. `cf()`
4. `cf()`



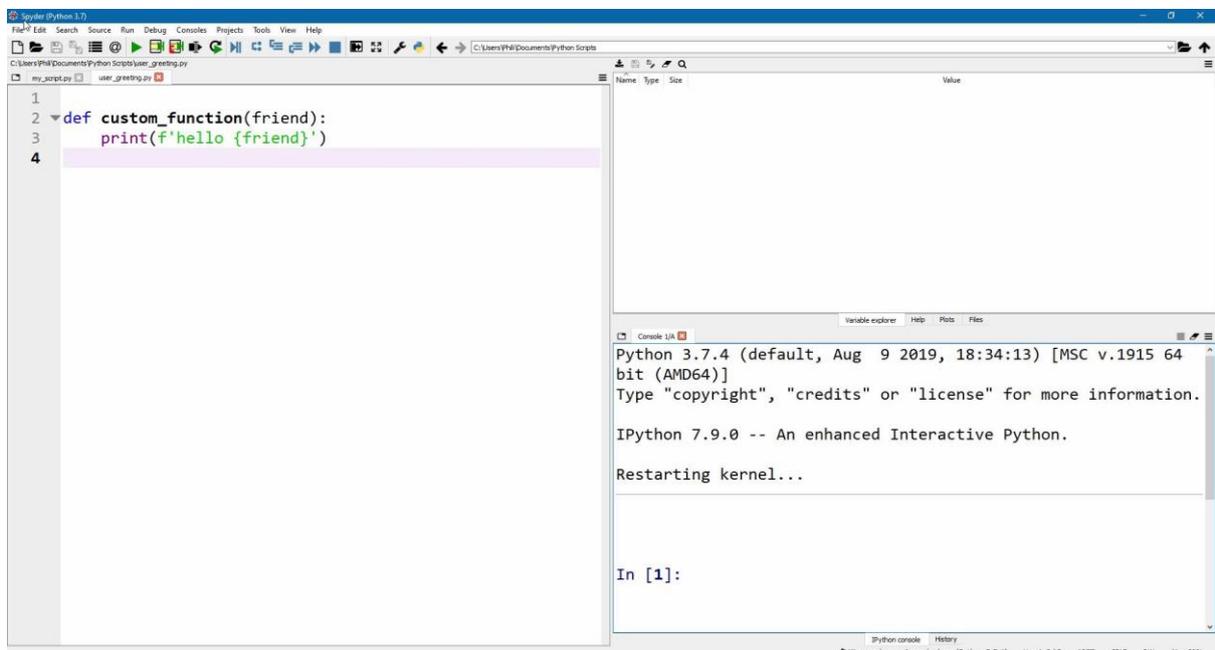
Input Argument

An input argument can be added to the `custom_function` script, for example the input argument `friend` which will be used as a variable in a formatted string.

```

1.
2. def custom_function(friend):
3.     print(f'hello {friend}')
4.

```



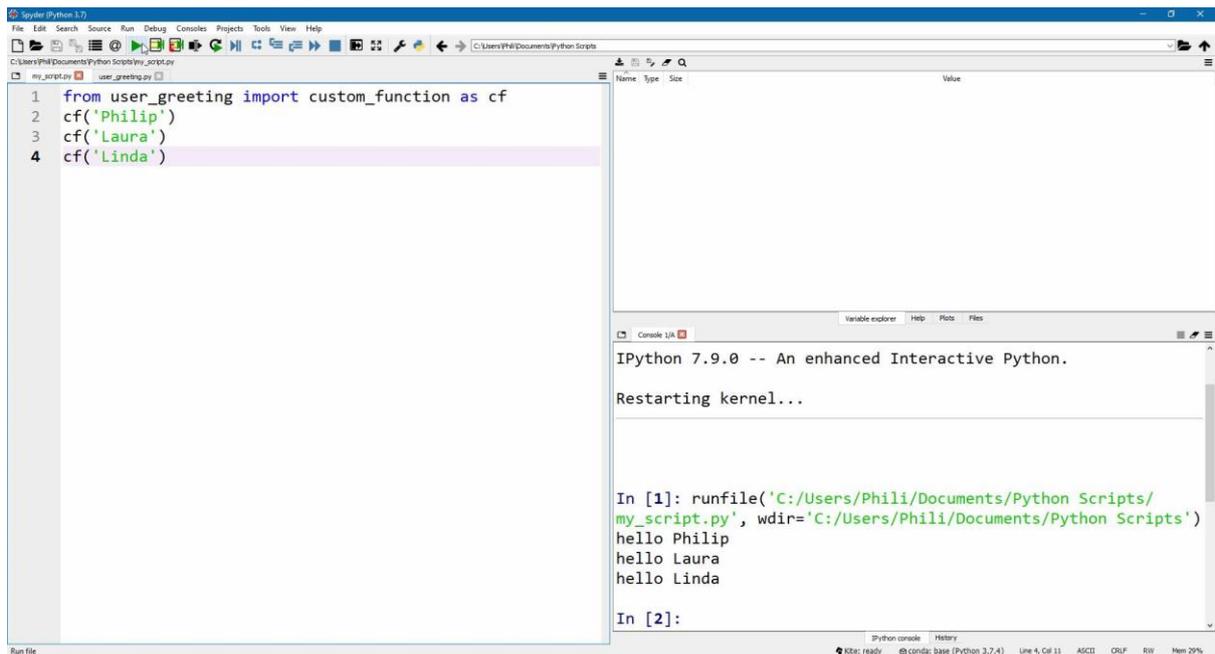
The function can be called multiple times changing the value of the input arguments each time.

```

1. from user_greeting import custom_function as cf
2. cf('Philip')
3. cf('Laura')

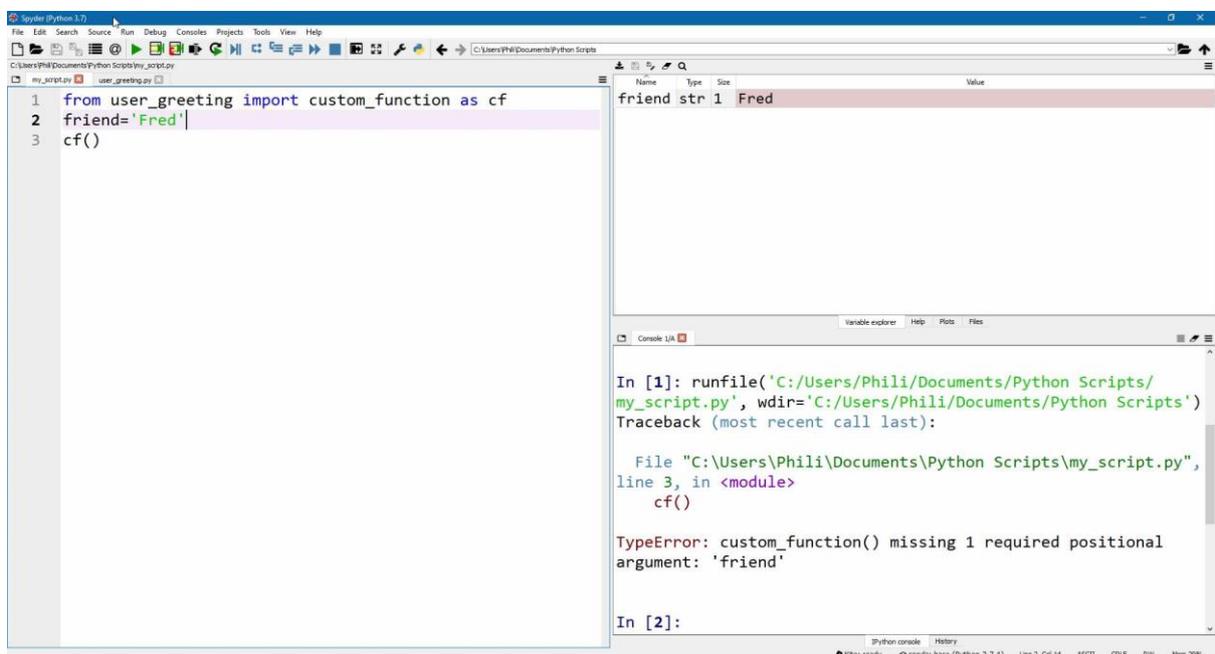
```

```
4. cf('Linda')
```



Note the input arguments to a function are local to the function and they are independent of the input in the variable explorer.

1. `from user_greeting import custom_function as cf`
2. `friend='Fred'`
3. `cf()`

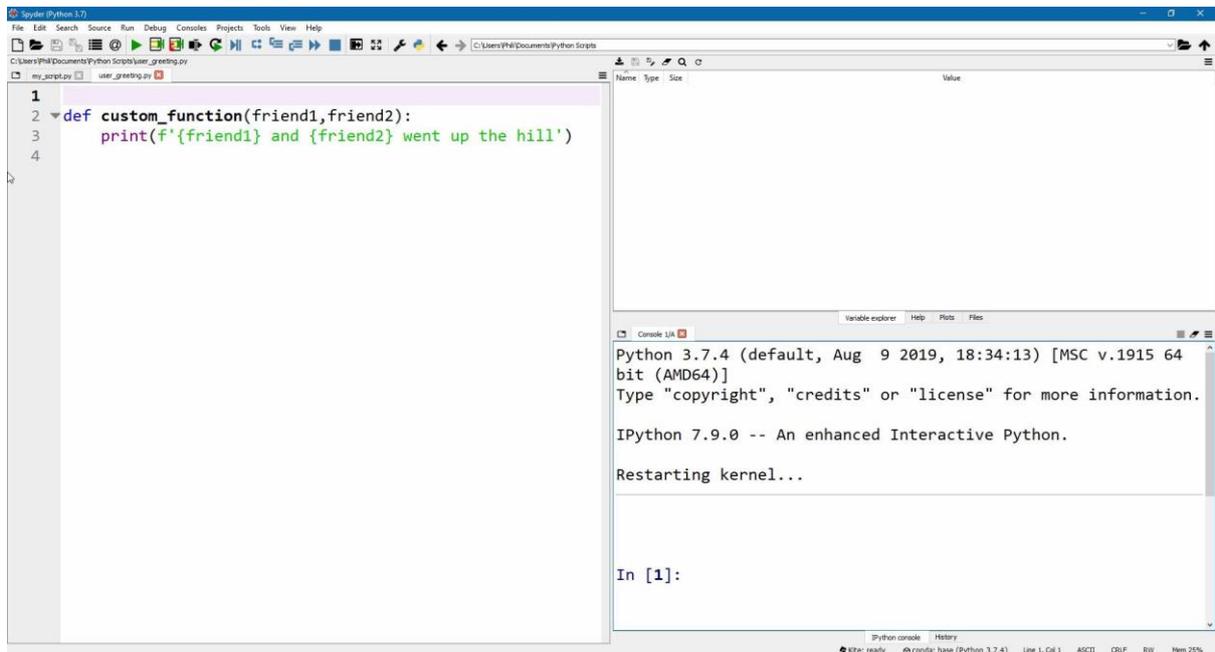


In the case above there is a `TypeError: custom_function() missing 1 required positional argument: 'friend'` and this shows in spite of the assignment of the variable `friend` in the variable explorer. The missing argument is a missing local function positional argument and is independent of the variable name `friend` within the variable explorer.

Positional Input Arguments

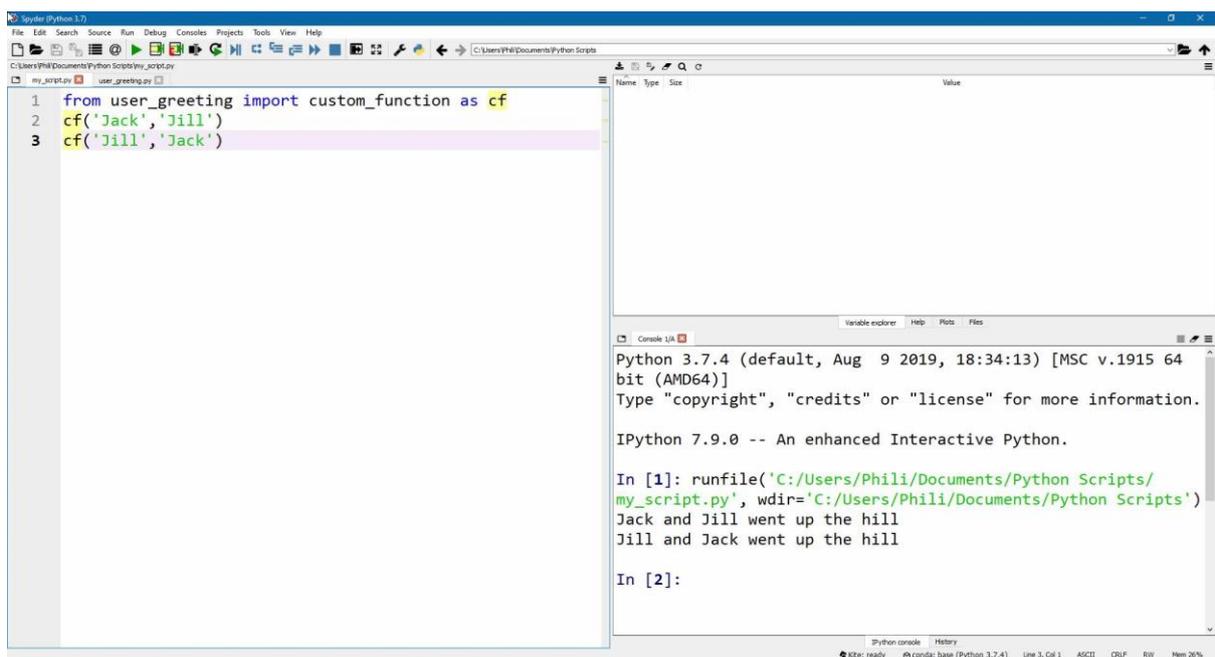
A function can have multiple positional input arguments and these need to be specified in the correct order. In the function below there are two input arguments `friend1` and `friend2`.

```
1.  
2. def custom_function(friend1, friend2):  
3.     print(f'{friend1} and {friend2} went up the hill')  
4.
```



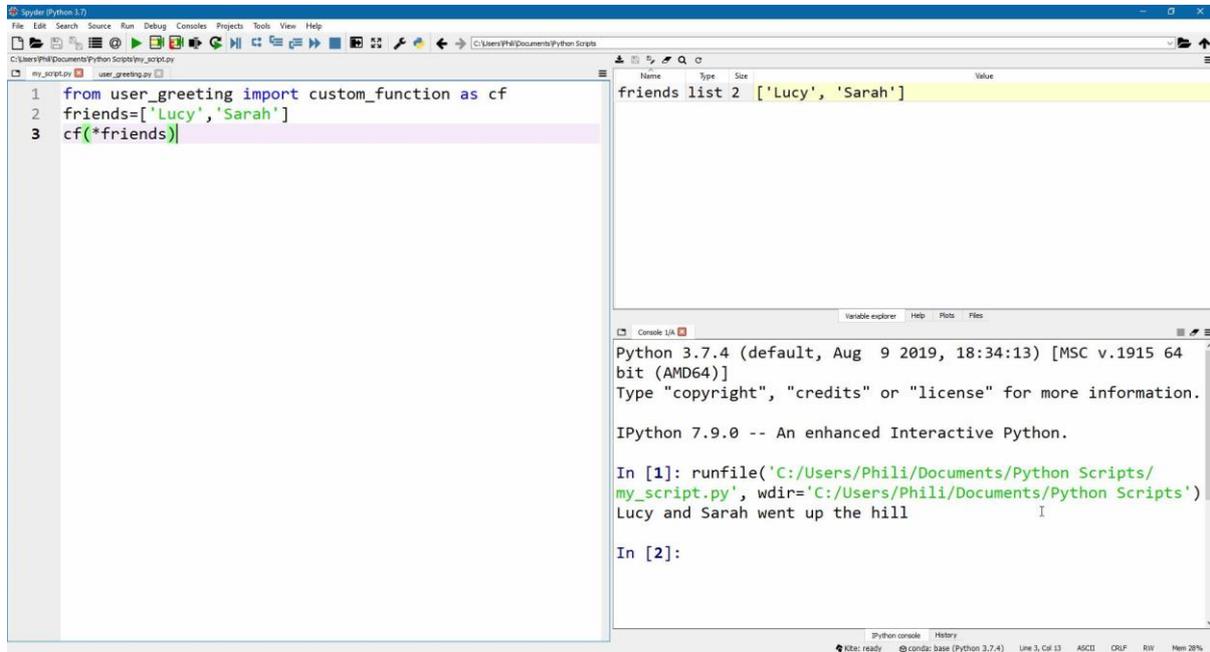
The order that they are called in will change what is printed on the console window.

```
1. from user_greeting import custom_function as cf  
2. cf('Jack', 'Jill')  
3. cf('Jill', 'Jack')
```



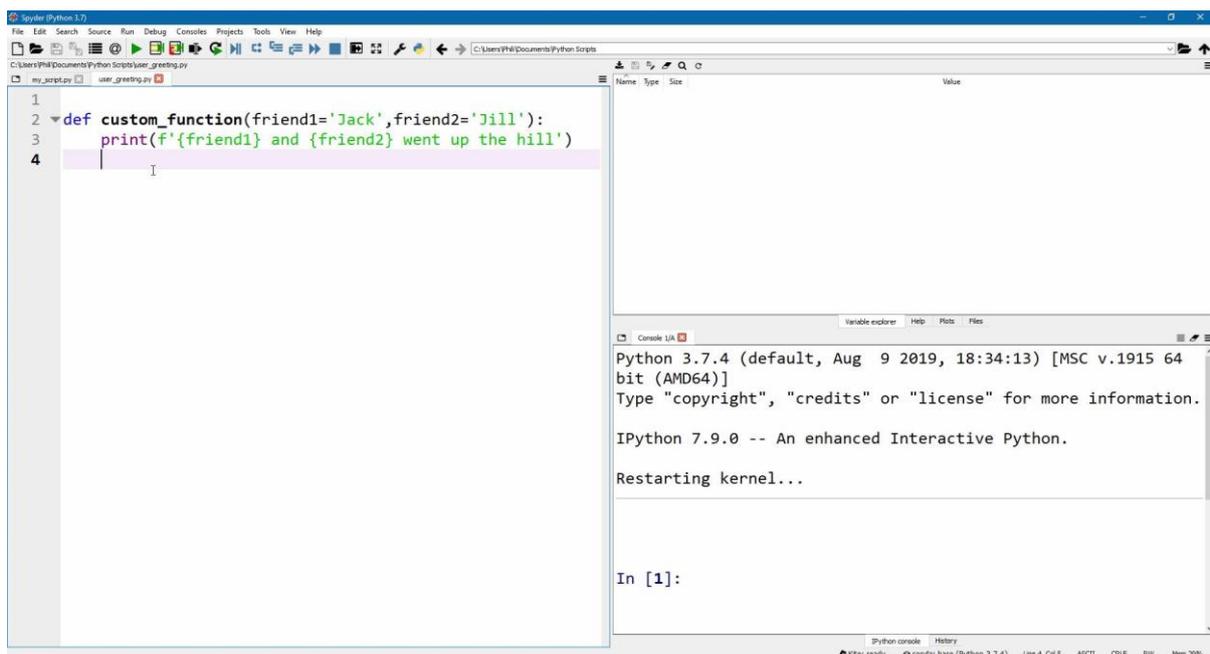
Positional arguments may be called using a list or tuple. When calling the function using a list or tuple, it needs to be prefixed with a *. This will unpack the list or tuple as the positional input arguments for example:

1. `from user_greeting import custom_function as cf`
2. `friends=['Lucy', 'Sarah']`
3. `cf(*friends)`



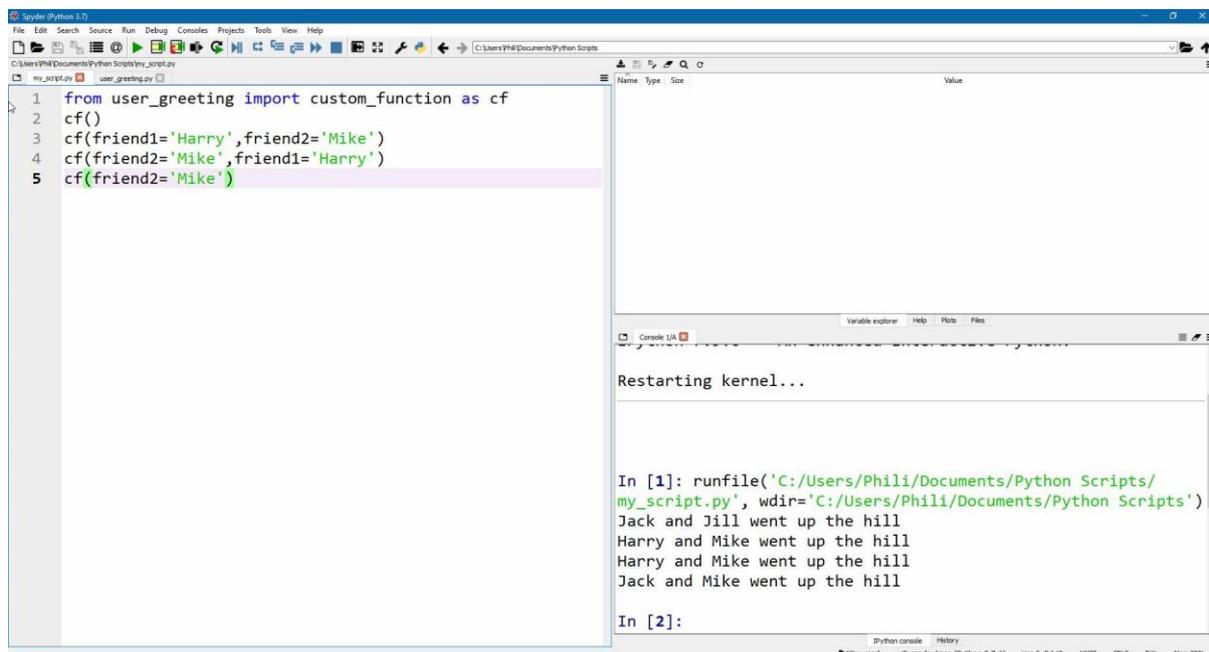
Keyword Input Arguments

- 1.
2. `def custom_function(friend1='Jack', friend2='Jill'):`
3. `print(f'{friend1} and {friend2} went up the hill')`
- 4.



When a function is created with keyword arguments opposed to positional arguments, the arguments are assigned a default value in this case `friend1='Jack'` and `friend2='Jill'`. This means the function can be called without specifying these keyword input arguments and they will instead take on the default value. The function can be changed from the default values by assigning them to new values when calling the input arguments within the function. Note because a keyword is used, the position of these keyword arguments can be swapped, and it will still work i.e. line 3 and line 4 will print out the same statement.

```
1. from user_greeting import custom_function as cf
2. cf()
3. cf(friend1='Harry', friend2='Mike')
4. cf(friend2='Mike', friend1='Harry')
5. cf(friend2='Mike')
```



The screenshot shows the Spyder Python IDE interface. The editor window on the left contains the following code:

```
1 from user_greeting import custom_function as cf
2 cf()
3 cf(friend1='Harry', friend2='Mike')
4 cf(friend2='Mike', friend1='Harry')
5 cf(friend2='Mike')
```

The console window on the right shows the output of the code execution:

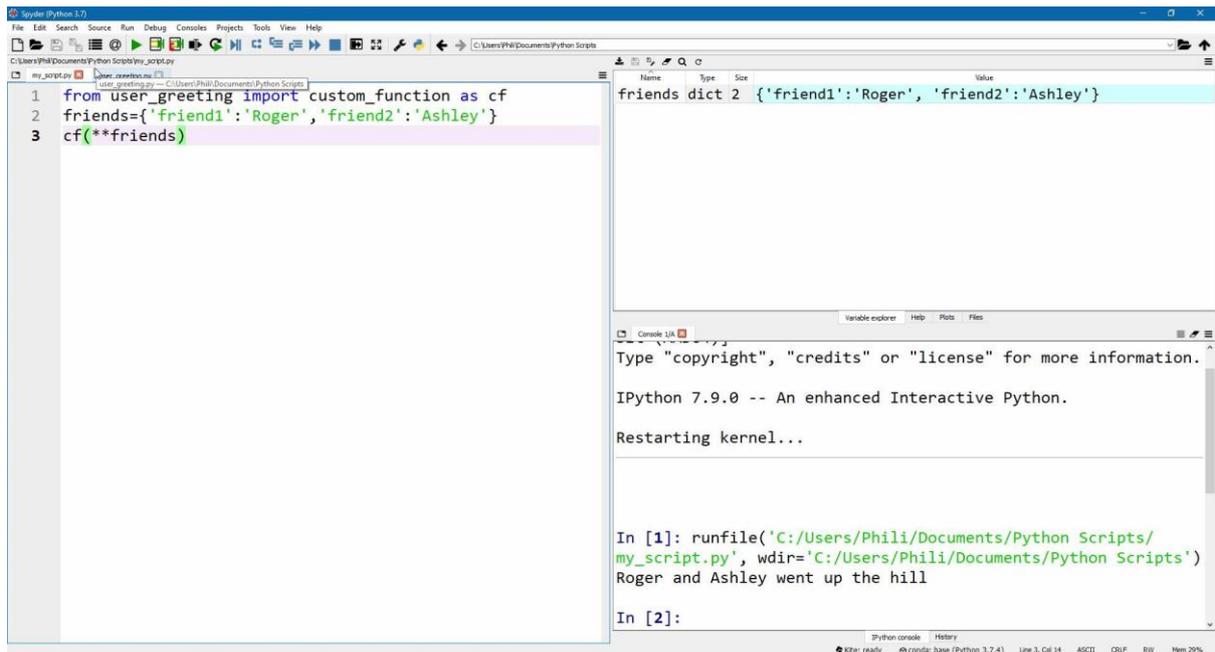
```
Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/
my_script.py', wdir='C:/Users/Phili/Documents/Python Scripts')
Jack and Jill went up the hill
Harry and Mike went up the hill
Harry and Mike went up the hill
Jack and Mike went up the hill

In [2]:
```

Keyword arguments may be called using a Dictionary. When calling the function using a Dictionary, it needs to be prefixed with a `**`. This will unpack the Dictionary key and values as the Keyword Input Arguments and their values for example:

```
1. friends={'friend1':'Roger', 'friend2':'Ashley'}
2. cf(**friends)
```



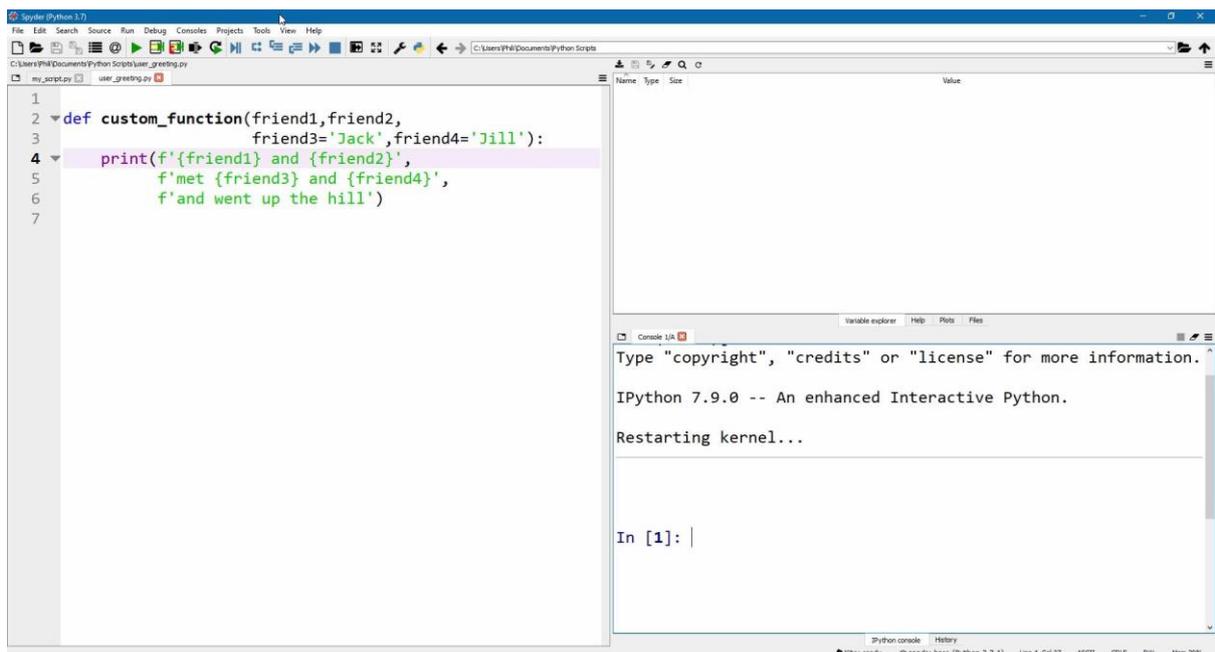
Positional Arguments and Keyword Input Arguments

Positional Arguments and Keyword Arguments can be utilised together. The positional arguments must be called first, however. In this case the custom function can be modified to contain two positional arguments and two keyword arguments

```

1.
2. def custom_function(friend1, friend2,
3.                     friend3='Jack', friend4='Jill'):
4.     print(f'{friend1} and {friend2}',
5.           f'met {friend3} and {friend4}',
6.           f'and went up the hill')
7.

```

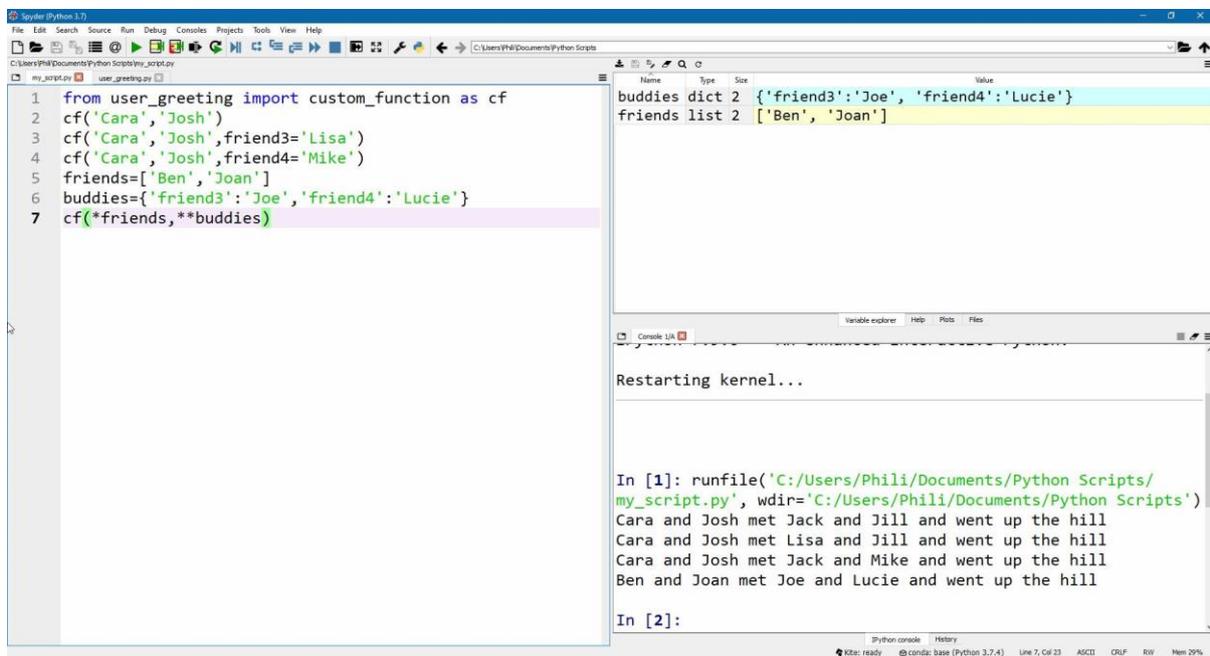


This can be called using:

```
1. from user_greeting import custom_function as cf
2. cf('Cara', 'Josh')
3. cf('Cara', 'Josh', friend3='Lisa')
4. cf('Cara', 'Josh', friend4='Mike')
5. friends=['Ben', 'Joan']
6. buddies={'friend3':'Joe', 'friend4':'Lucie'}
7. cf(*friends, **buddies)
```

Here on line 2,3 and 4 the positional input arguments are called first. In line 2 no keyword arguments are called, and the default values are hence used. In line 3 and 4 one of the two keywords arguments are used.

In line 5 and line 6 a list of positional input arguments and a dictionary of keyword input arguments are created. These are input to the function and unpacked using `*` and `**` in line 7.



The screenshot shows the Spyder Python IDE interface. On the left, a code editor displays the following Python code:

```
1 from user_greeting import custom_function as cf
2 cf('Cara', 'Josh')
3 cf('Cara', 'Josh', friend3='Lisa')
4 cf('Cara', 'Josh', friend4='Mike')
5 friends=['Ben', 'Joan']
6 buddies={'friend3':'Joe', 'friend4':'Lucie'}
7 cf(*friends, **buddies)
```

On the right, the Variable Explorer shows the state of variables:

Name	Type	Size	Value
buddies	dict	2	{'friend3': 'Joe', 'friend4': 'Lucie'}
friends	list	2	['Ben', 'Joan']

Below the Variable Explorer, the IPython console shows the output of the code execution:

```
Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/
my_script.py', wdir='C:/Users/Phili/Documents/Python Scripts')
Cara and Josh met Jack and Jill and went up the hill
Cara and Josh met Lisa and Jill and went up the hill
Cara and Josh met Jack and Mike and went up the hill
Ben and Joan met Joe and Lucie and went up the hill

In [2]:
```

Note the string in `user_greeting.py` called the two positional arguments before the two keyword arguments. This isn't a requirement however when assigning input arguments to a function the positional arguments must be present before the keyword arguments.

Document Strings

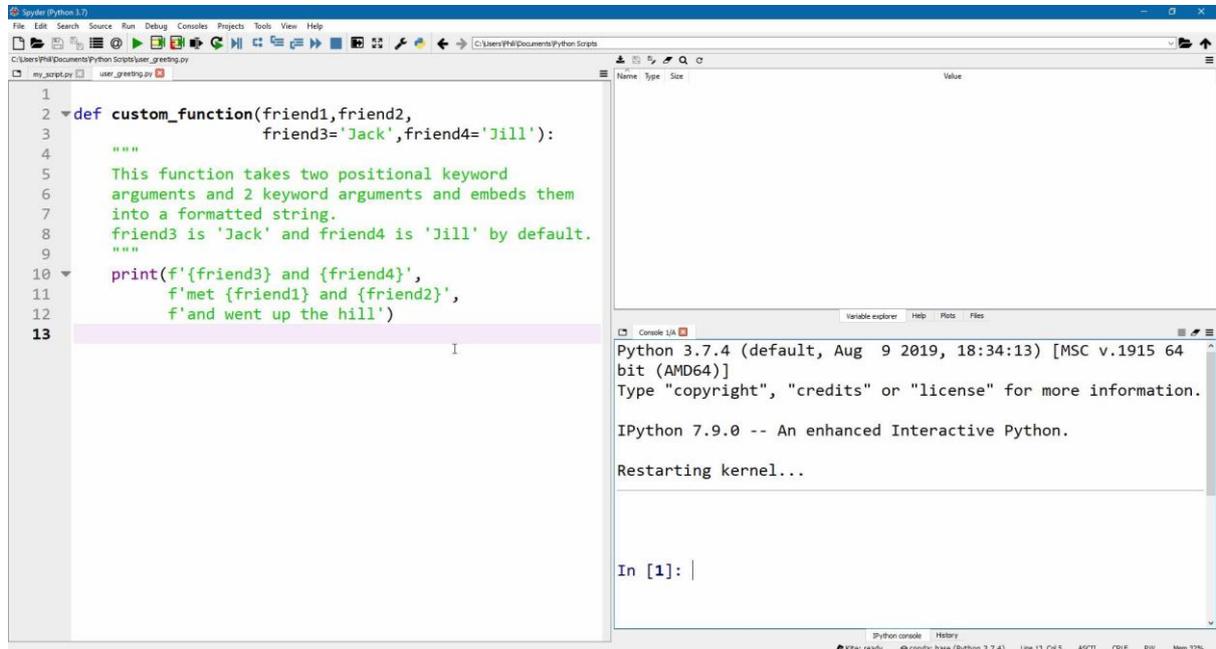
The code in functions can get quite complicated, it is recommended to use document screens to explain what a function does. This is done by enclosing in triple quotes

```
8.
9. def custom_function(friend1, friend2,
10.                    friend3='Jack', friend4='Jill'):
11.     """
12.     This function takes two positional keyword
13.     arguments and 2 keyword arguments and embeds them
14.     into a formatted string.
15.     friend3 is 'Jack' and friend4 is 'Jill' by default.
```

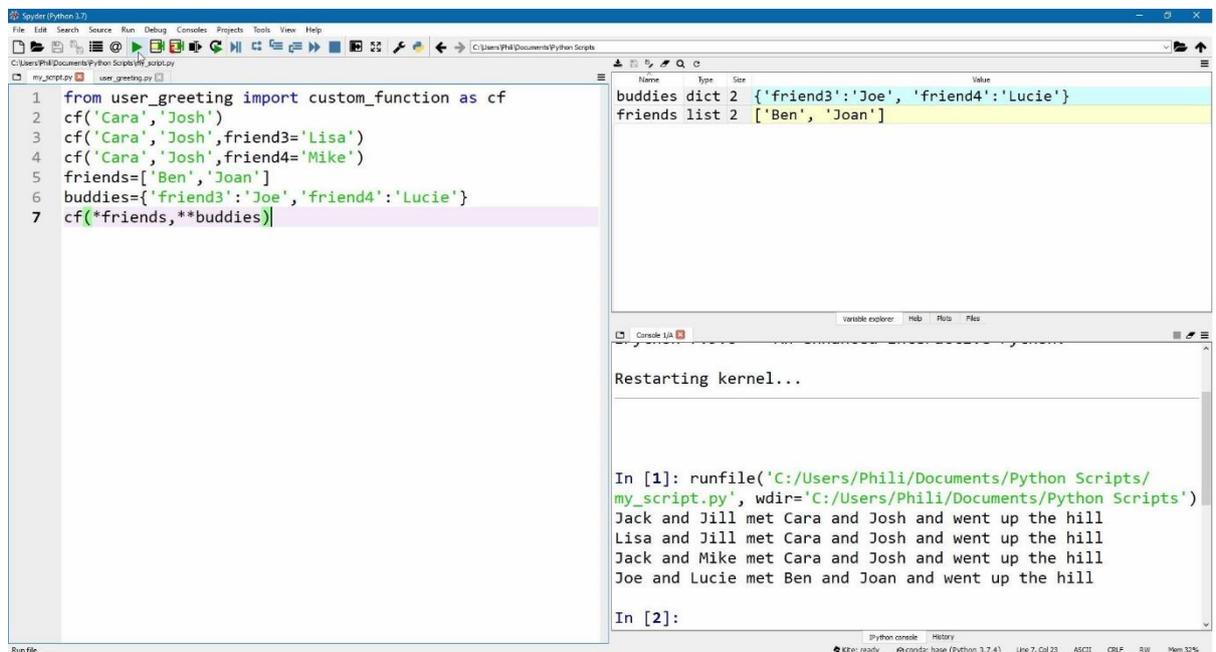
```

16. """
17.     print(f'{friend3} and {friend4}',
18.           f'met {friend1} and {friend2}',
19.           f'and went up the hill')
20. """

```

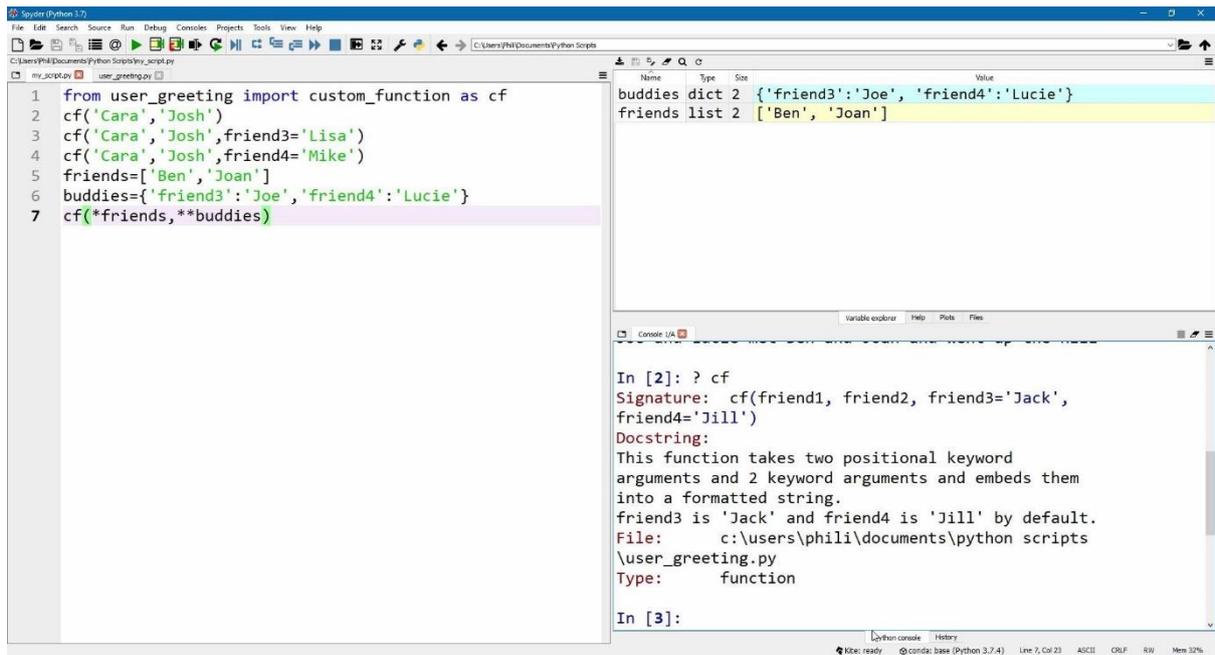


The script can be ran using the same positional input arguments and keyword input arguments as before. As seen in this case the keyword input arguments are used in the string before the positional input arguments in this string.



To get help for the imported `custom_function` as `cf`. The following can be typed in the console.

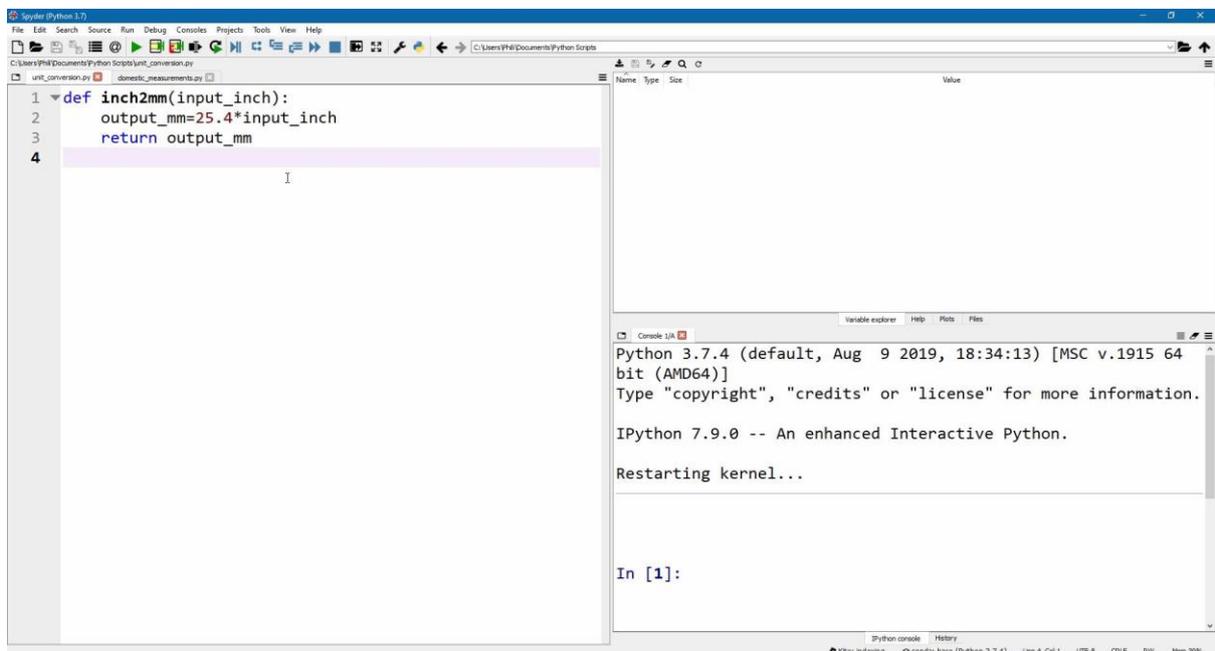
```
? cf
```



Return Statement

So far only inputs have been looked at in a function and a print statement has been used to print to the console. It is possible for a function to return a value which means when the function is called, the value of the function using the selected inputs can be assigned to a variable.

1. `def` `inch2mm(input_inch)`
2. `output_mm=25.4*input_inch`
3. `return` `output_mm`
- 4.



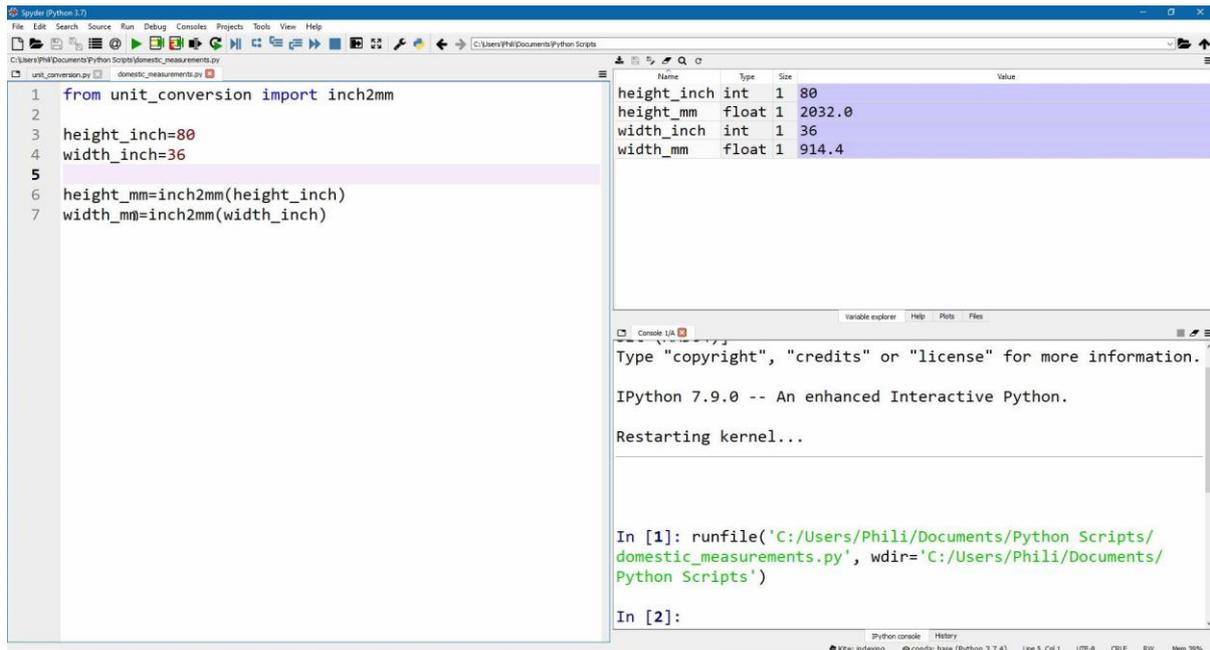
Supposing we have the dimensions of a door 80 inches in height and 36 inches wide, but we need the dimensions in mm, we can use this function to convert them using:

```

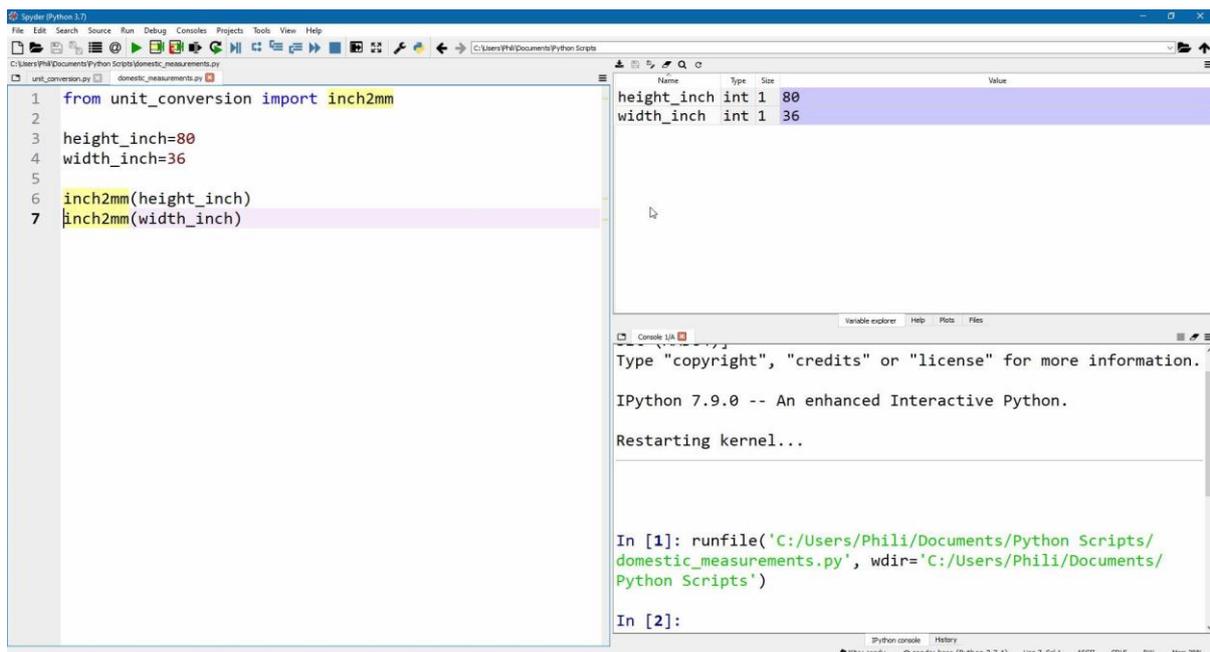
1. from unit_conversion import inch2mm
2.
3. height_inch=80
4. width_inch=36
5.
6. height_mm=inch2mm(height_inch)
7. width_mm=inch2mm(width_inch)

```

Running this will calculate the height in mm and assign it to the variable `height_mm` and the width in mm and assign it to the variable `width_mm`.

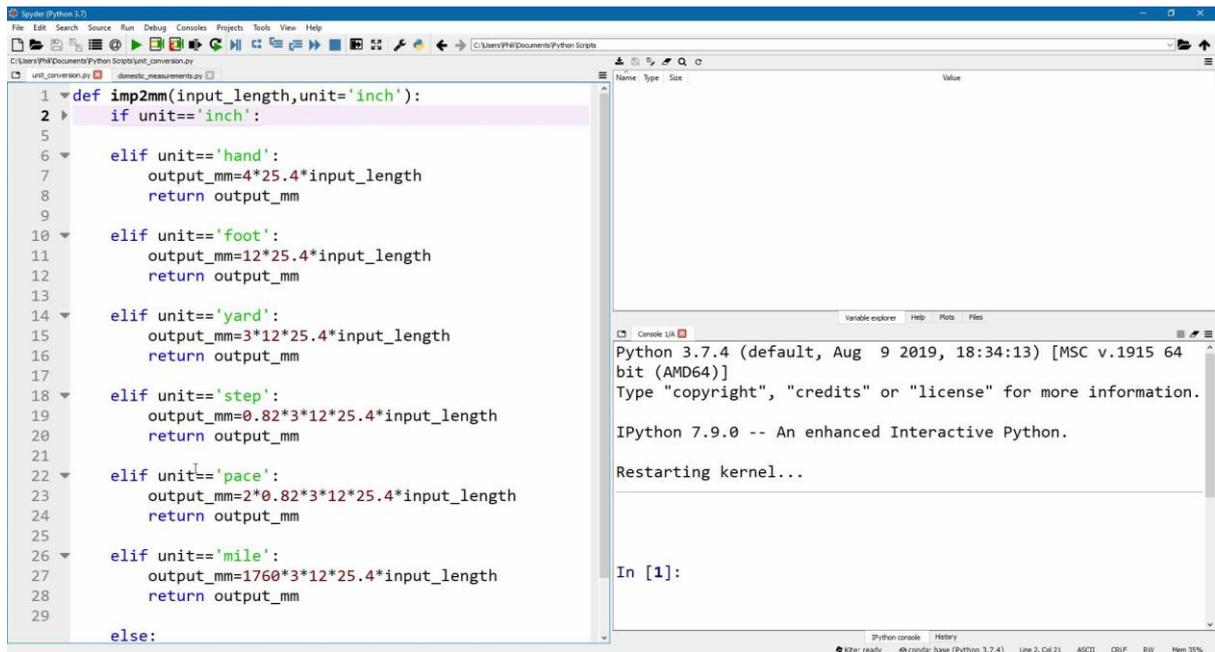


Note if the code is run without variable assignment or without embedding with a print statement the value will be calculated in the background but not displayed.

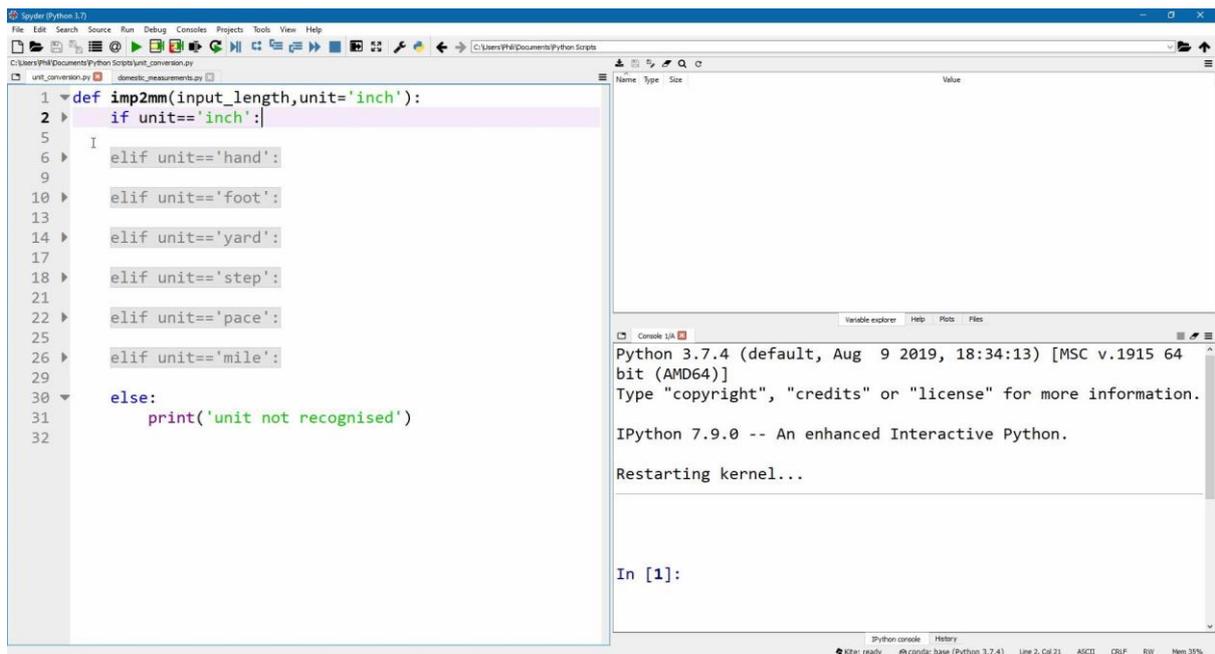


A more advanced imperial to metric function can be made. This function will have two input arguments, a positional input argument which is the `input_length` and a keyword argument `unit` which is the imperial unit which is assigned to a default string of `'inch'`. An if, multiple elif and else statement may be used to calculate the length in mm using the conversion factor to mm for each imperial unit. This can be returned as a single output, `output_mm`.

```
1. def imp2mm(input_length, unit='inch') :
2.
3.     if unit=='inch':
4.         output_mm=25.4*input_length
5.         return output_mm
6.
7.     elif unit=='hand':
8.         output_mm=4*25.4*input_length
9.         return output_mm
10.
11.    elif unit=='foot':
12.        output_mm=12*25.4*input_length
13.        return output_mm
14.
15.    elif unit=='yard':
16.        output_mm=3*12*25.4*input_length
17.        return output_mm
18.
19.    elif unit=='step':
20.        output_mm=0.82*3*12*25.4*input_length
21.        return output_mm
22.
23.    elif unit=='pace':
24.        output_mm=2*0.82*3*12*25.4*input_length
25.        return output_mm
26.
27.    elif unit=='mile':
28.        output_mm=1760*3*12*25.4*input_length
29.        return output_mm
30.
31.    else:
32.        print('unit not recognised')
```



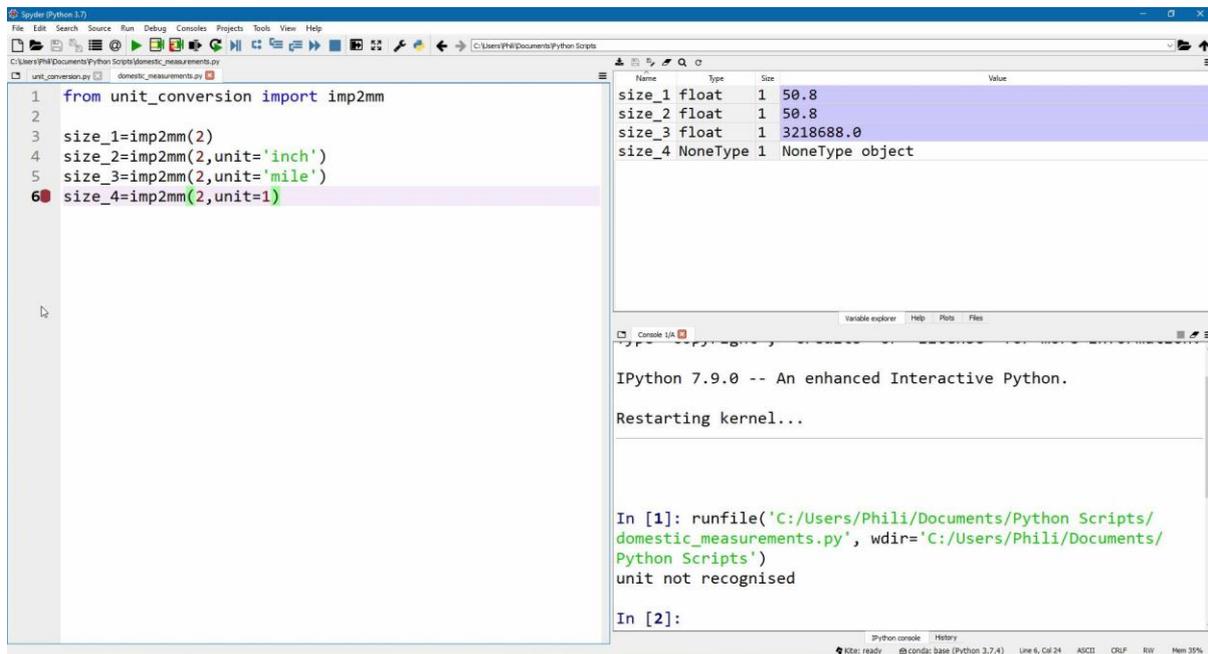
Note beside each indentation, there is a ▼ beside the line number. This can be used to collapse the code.



This can be imported into `domestic_measurements` and then used to convert a number of values from imperial to metric.

1. `from unit_conversion import imp2mm`
2. `size_1=imp2mm(2)`
3. `size_2=imp2mm(2,unit='inch')`
4. `size_3=imp2mm(2,unit='mile')`
5. `size_4=imp2mm(2,unit=1)`

All the converted values will be stored in the variable explorer. The variable `size_4` is a `NoneType` object as it was not assigned a value, instead a print statement showed in the console informing the user that the unit was not recognised.



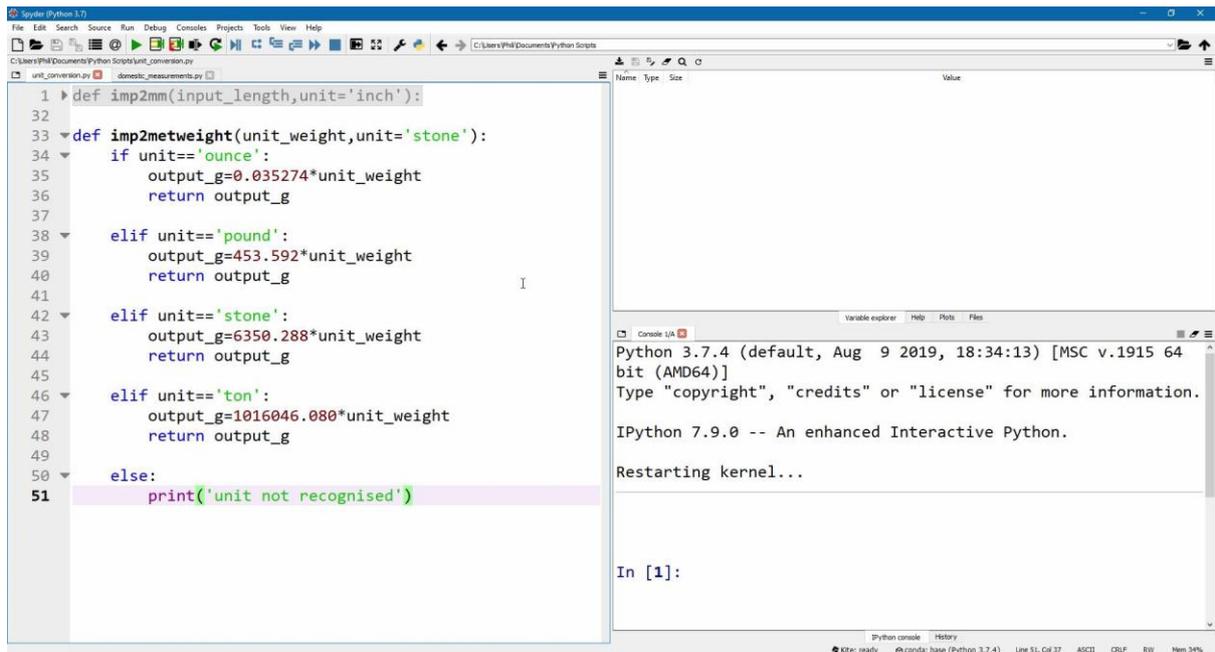
An additional function within `unit_conversion` can be made, converting imperial weights to metric values in grams.

```

1. def imp2metweight(unit_weight,unit='stone'):
2.
3.     elif unit=='ounce':
4.         output_g=0.035274*unit_weight
5.         return output_g
6.
7.     elif unit=='stone':
8.         output_g=453.592*unit_weight
9.         return output_g
10.
11.    elif unit=='pound':
12.        output_g=6350.288*unit_weight
13.        return output_g
14.
15.    elif unit=='ton':
16.        output_g=1016046.080*unit_weight
17.        return output_g
18.
19.    else:
20.        print('unit not recognised')

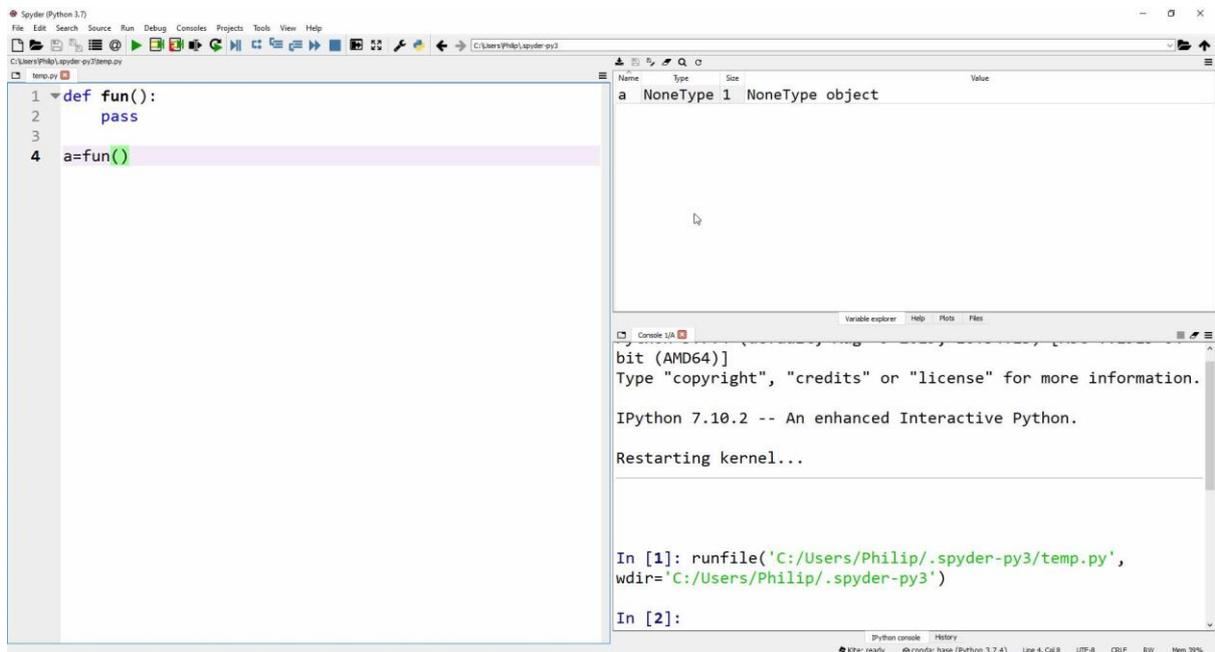
```

Note that the previous function can be collapsed to save screen space while working on the existing function.



Functions which don't have any return values can be assigned to variables when they are called. `fun` which contains `pass` and hence just does nothing is called and assigned to `a`.

1. `def fun() :`
2. `pass`
- 3.
4. `a=fun()`



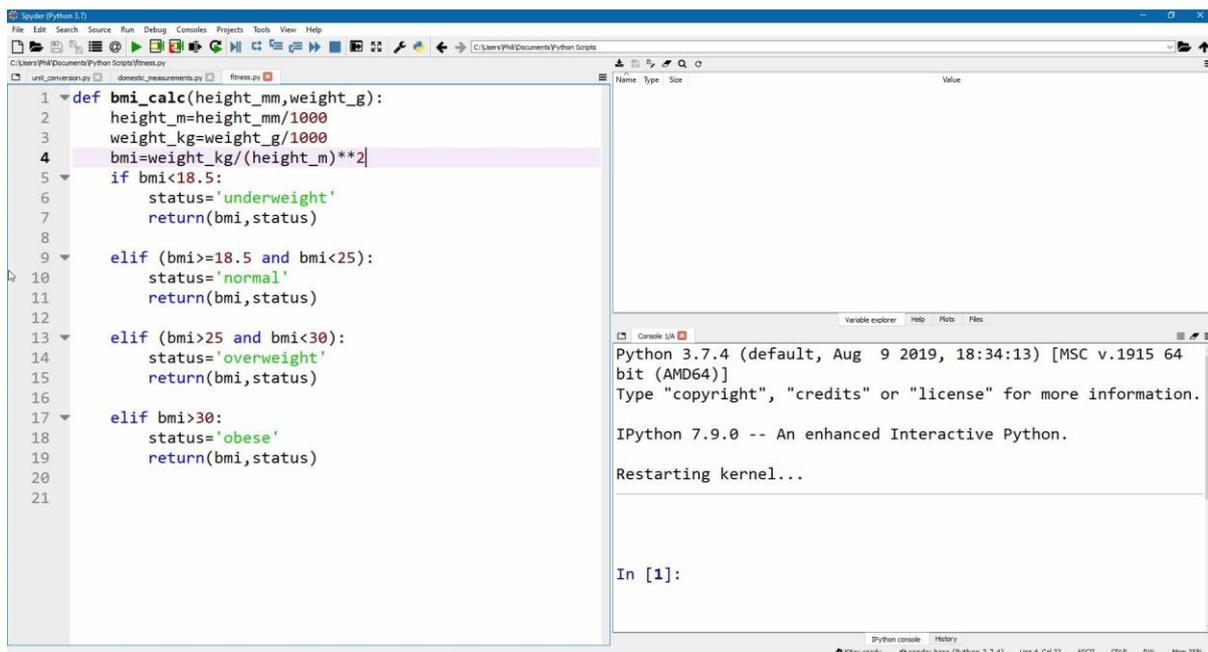
In this case `a` has a no value and is of type `NoneType`.

Nested Functions

Another function can be made in a separate Python script fitness, the function `bmi_calc` has two input arguments which are the height in mm and the weight in g and returns a list with 2 elements, the bmi as a number and the status as a string.

```
1. def bmi_calc(height_mm,weight_g):
2.     height_m= height_mm/1000
3.     weight_kg=weight_g/1000
4.     bmi=weight_kg/(height_m)**2
5.     if bmi<18.5:
6.         status='underweight'
7.         return(bmi,status)
8.
9.     elif (bmi>=18.5 and bmi<25):
10.        status='normal'
11.        return(bmi,status)
12.
13.    elif (bmi>25 and bmi<30):
14.        status='overweight'
15.        return(bmi,status)
16.
17.    elif bmi>30:
18.        status='obese'
19.        return(bmi,status)
20.
21.
```

This function can be used with the converted height and weight of a person to calculate their bmi as well as identify their health classification.



The screenshot shows the Spyder Python IDE interface. The left pane displays the code for the `bmi_calc` function, which is identical to the code block above. The right pane shows the IPython console output, which includes the Python version (3.7.4), system information (MSC v.1915 64 bit AMD64), and the IPython version (7.9.0). The console also shows the prompt `In [1]:`, indicating that the function has been executed.

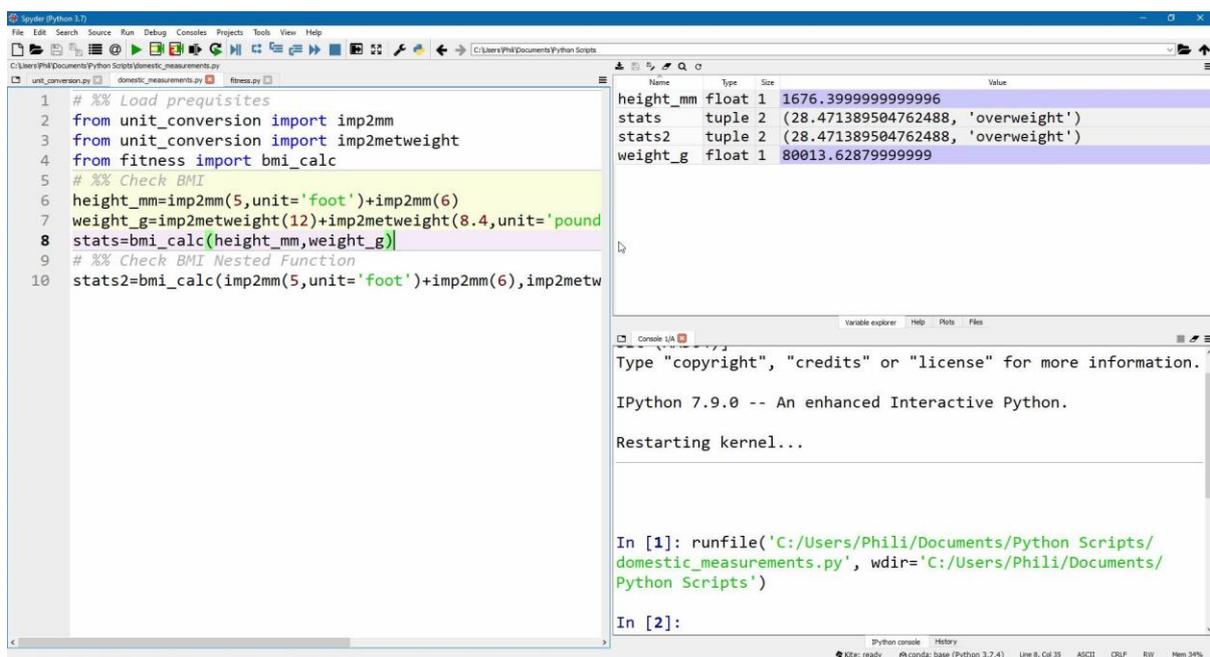
```

1. # %% Load prerequisites
2. from unit_conversion import imp2mm
3. from unit_conversion import imp2metweight
4. from fitness import bmi_calc

5. # %% Check BMI
6. height_mm=imp2mm(5,unit='foot')+imp2mm(6)
7. weight_g=imp2metweight(12)+imp2metweight(8.4,unit='pound')
8. stats=bmi_calc(height_mm,weight_g)

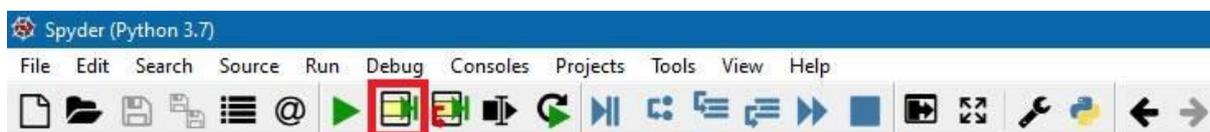
9. # %% Check BMI Nested Function
10. stats2=bmi_calc(imp2mm(5,unit='foot')+imp2mm(6),imp2metweight(
    12)+imp2metweight(8.4,unit='pound'))

```

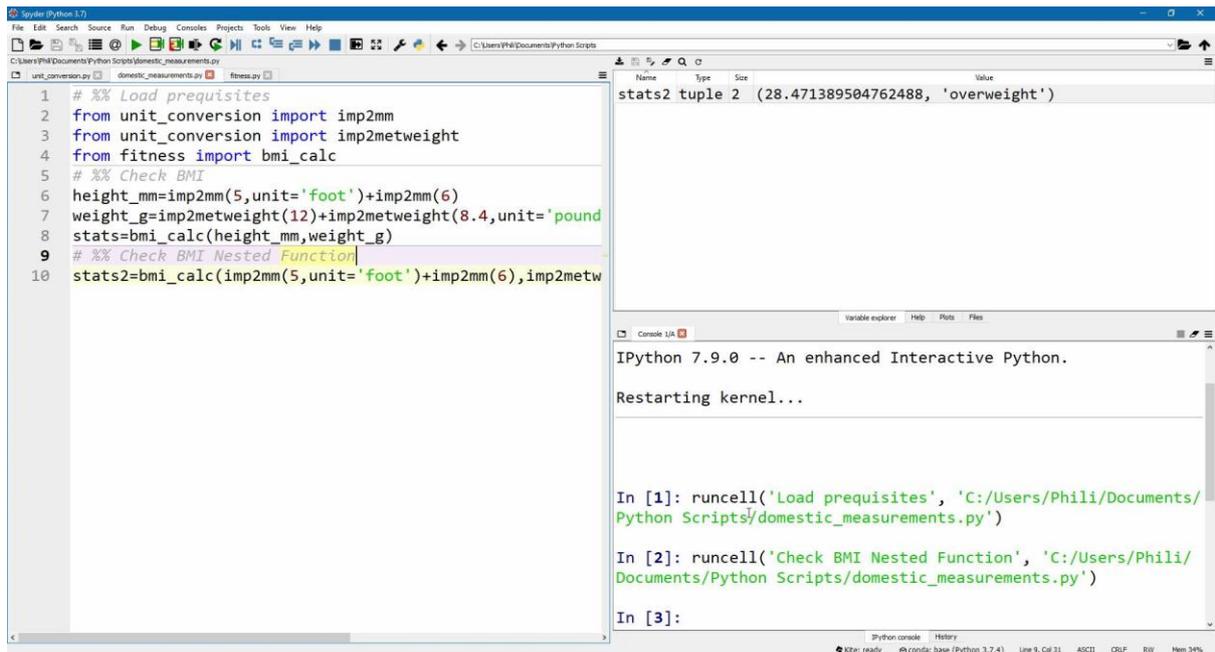


In the code above `# %% Comment` is utilized to create separate sections or cells. The first cell loads the necessary custom functions from file. The second cell converts the imperial height and weight of a person and calculates their BMI and health classification and the third cell repeats the second section on one line using a nested function.

Each cell can be ran individually, for example, the first cell can be ran which has the necessary prerequisites and then the third cell may be ran without running the second cell.



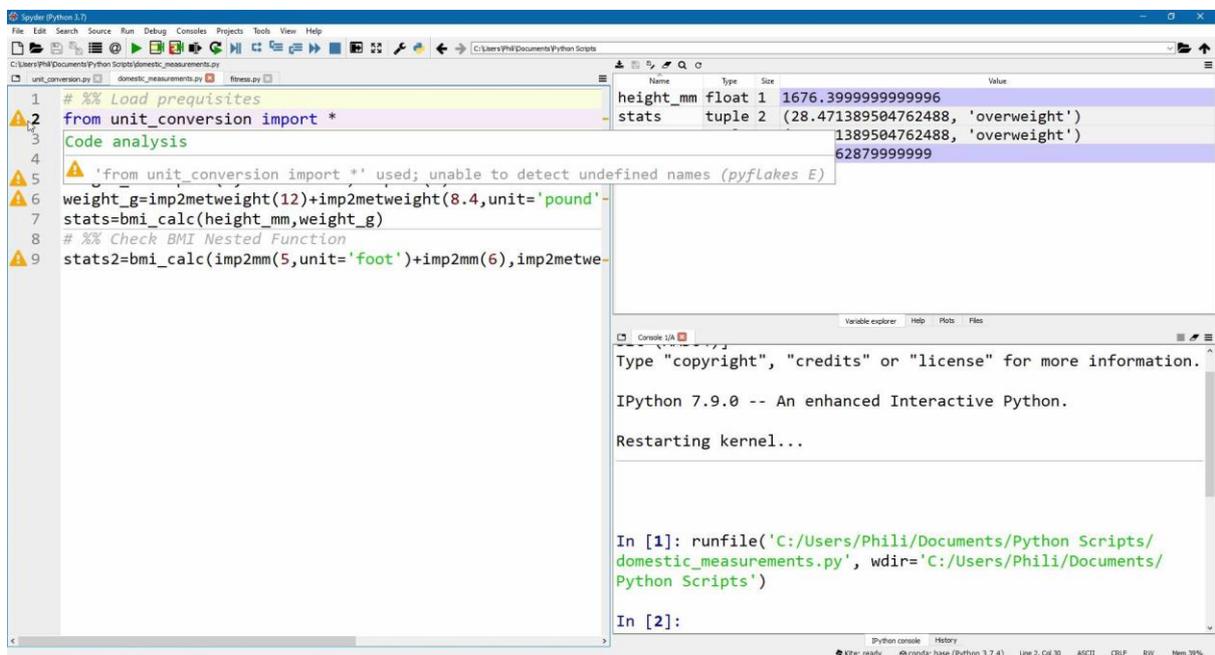
This gives only stats2 on the variable explorer:

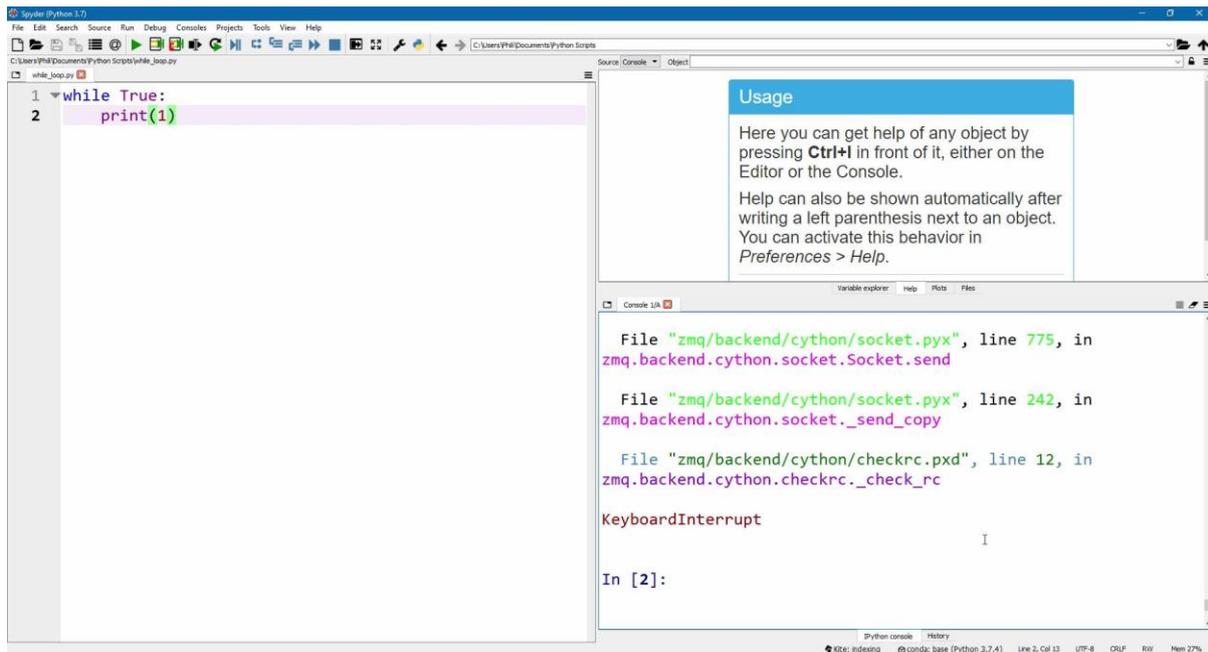


It is possible to load multiple functions using:

```
from unit_conversion import *
```

However, doing so gives several warnings in the code editor. The first warning tells us that it is unable to detect undefined names. This is because each individual function is not specified and when the functions are called up later, Python does not know if these function names are correct or exist as they weren't imported with a name. It is thus frowned upon to use the * to import all functions from a file as it becomes harder to trace the code and is hence more error prone.

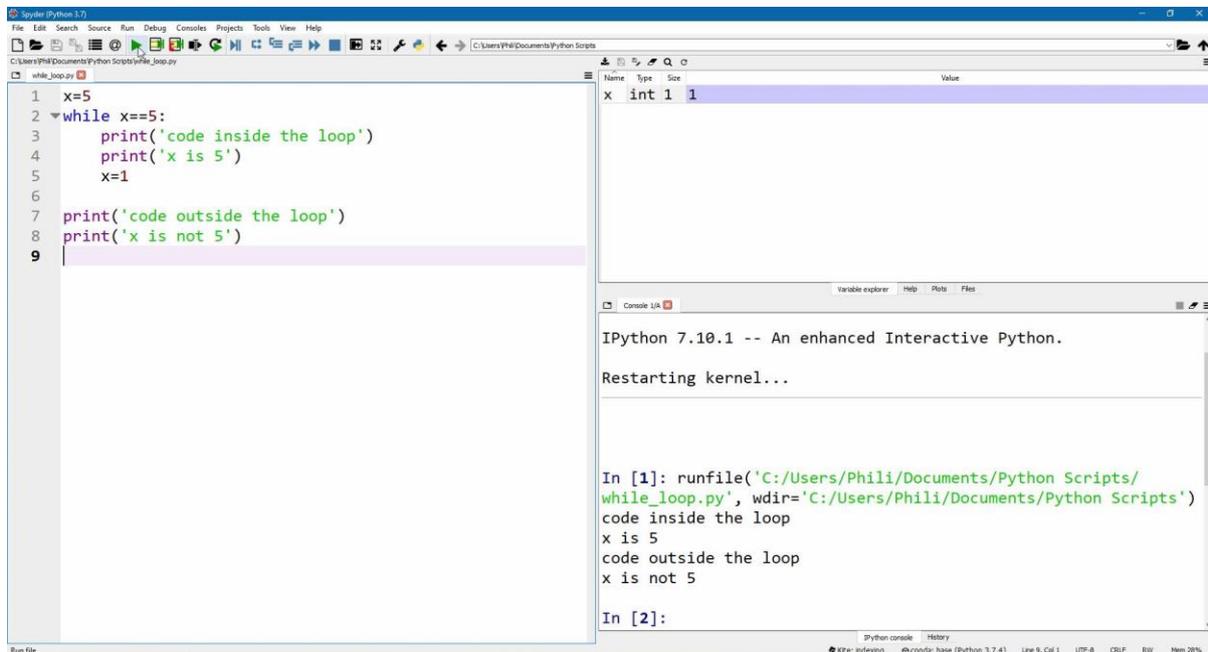




```

1. x=5
2. while x==5:
3.     print('code inside the loop')
4.     print('x is 5')
5.     x=1
6.
7. print('outside the loop')
8. print('x is not 5')
9.

```

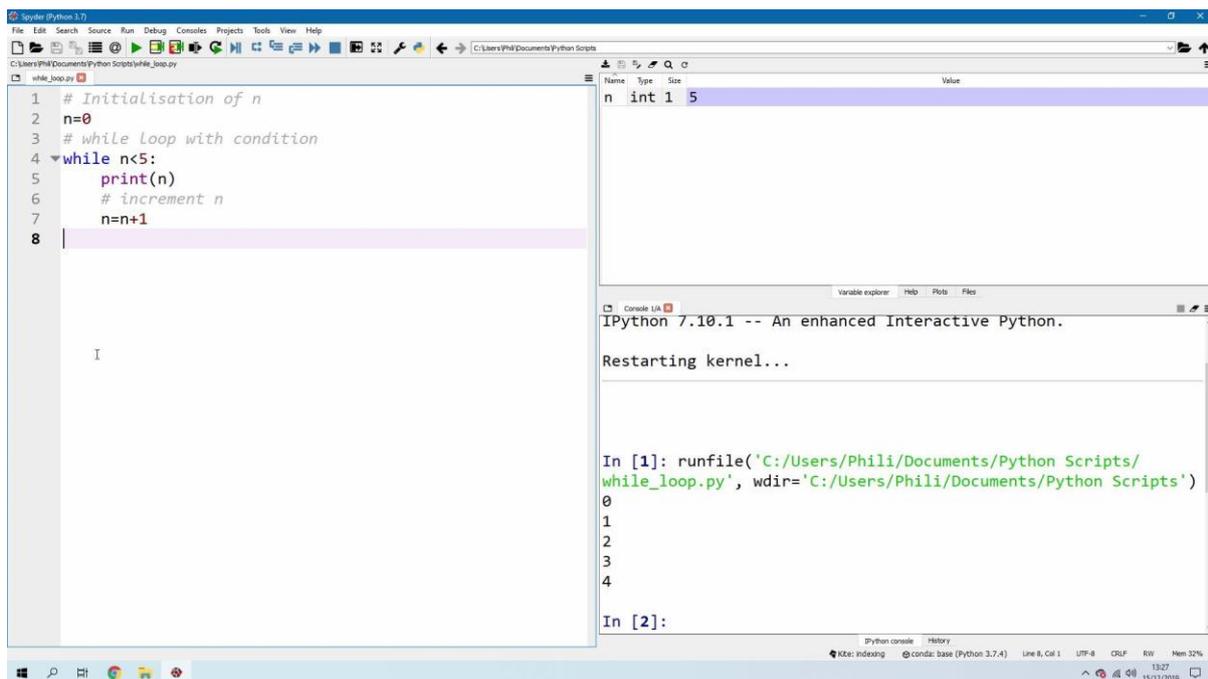


In the above loop, the variable `x` is initialised and set to a value of `5`. The loop condition is that the code within the while loop will only run while `x==5`. During the execution of the code in the loop, the value of `x` is reassigned. This makes the condition for the `while` loop `False` on its next iteration

meaning the loop is broken. The following loop can be made using a less than `<` condition opposed to a check for equality `==`.

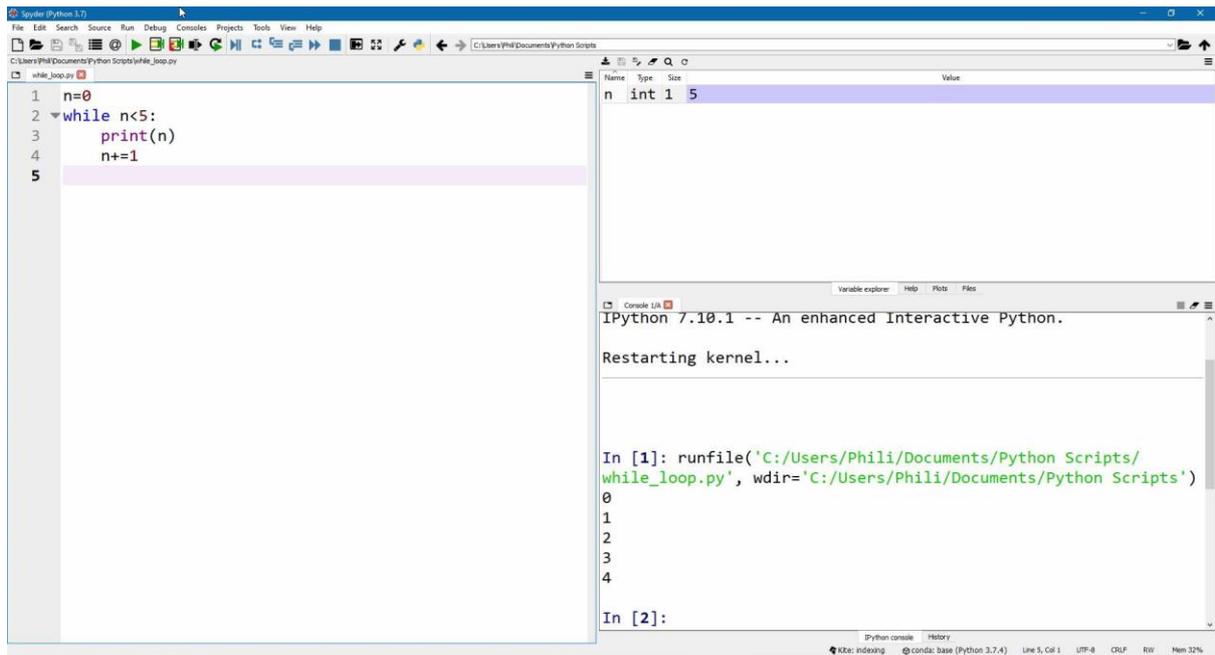
```
1. # initialisation of n
2. n=0
3. # while loop with condition
4. while n<5:
5.     print(n)
6.     # increment n
7.     n=n+1
8.
```

In this case the condition of the while loop will be satisfied when `n=0` (initialised value) and the value of `n` will be printed. `n` will be reassigned to `n=1` and the condition of the while loop will once again be satisfied so the new value of `n` will be printed. `n` will be reassigned to `n=2` and the condition of the while loop will once again be satisfied so the new value of `n` will be printed. `n` will be reassigned to `n=3` and the condition of the while loop will once again be satisfied so the new value of `n` will be printed. `n` will be reassigned to `n=4` and the condition of the while loop will once again be satisfied so the new value of `n` will be printed. Finally, `n` will be reassigned to `n=5` and the condition of the while loop will not be satisfied meaning the loop is exited and the code in the loop isn't iterated. Therefore, the value of `n=5` is not printed although one can see that `n` has a value of `5` in the variable explorer.



Because `n=n+1` is commonly used with loops. Python has a shortcut for this `n+=1`. The command `n=n-1` can likewise be abbreviated as `n-=1`.

```
1. n=0
2. while n<5:
3.     print(n)
4.     n+=1
5.
```



For Loop

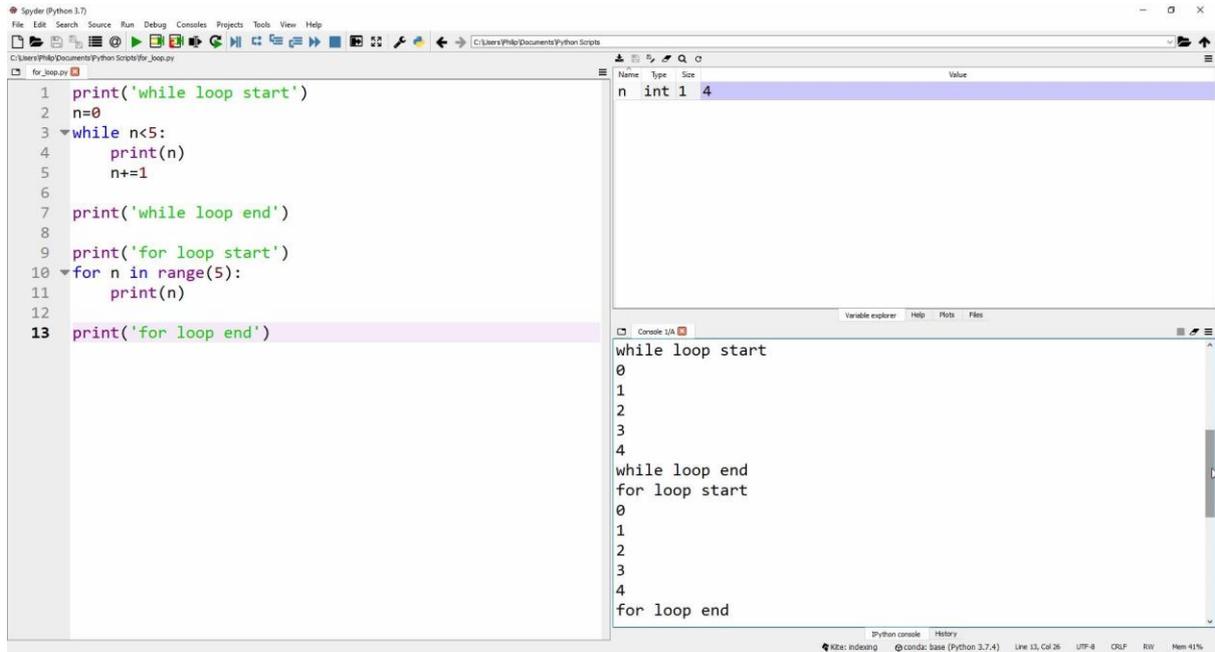
We know that the code below has a finite number of values which satisfy the **while** condition i.e. $n=0$, $n=1$, $n=2$, $n=3$, $n=4$ and $n=5$ i.e. run for 5 times. In line 1 we initialise a variable known as a counter. Line 2 begins the **while** loop and we use this variable to check whether our condition is satisfied. Line 4 increments the counter.

```
1. n=0
2. while n<5:
3.     print(n)
4.     n+=1
5.
```

This can be simplified using a **for** loop opposed to a **while** loop. In the **for** loop we must specify a counter.

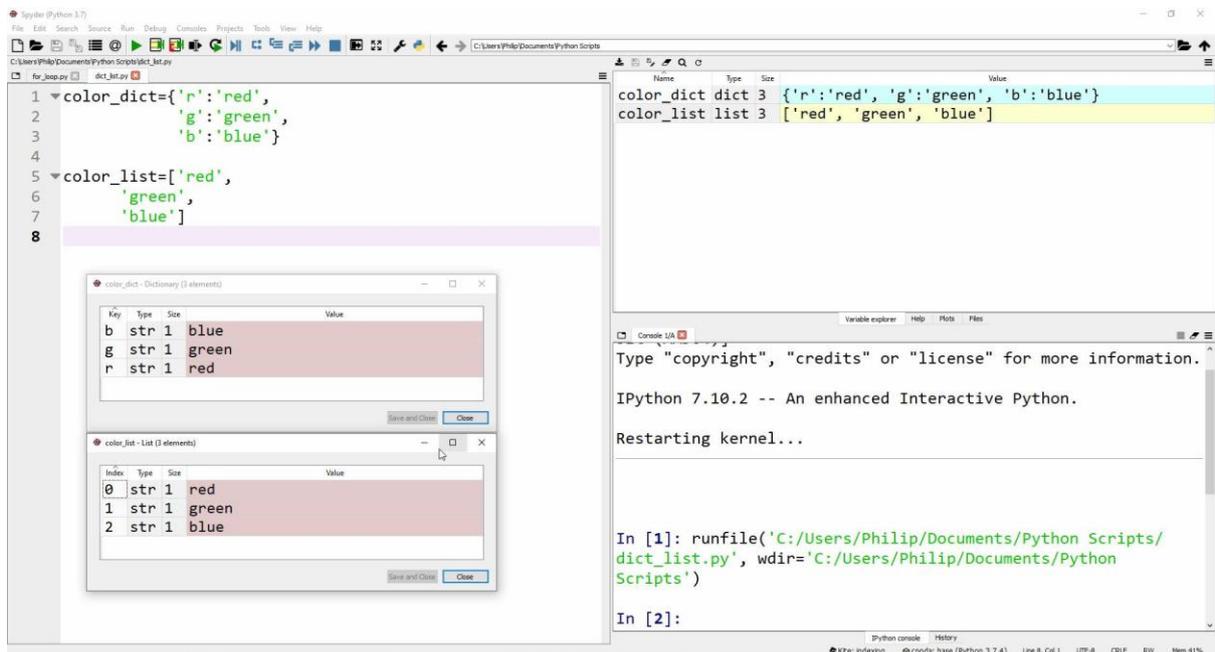
```
1. for n in range(5):
2.     print(n)
3.
```

In both cases a counter or range is explicitly expressed. When working with a list, the properties of the list such as the length of the list can be taken to obtain the upper bound.



Let's first explore the `in` command using a dictionary and a list.

1. `color_dict={'r':'red', 'g':'green', 'b':'blue'}`
2. `color_list=['red', 'green', 'blue']`

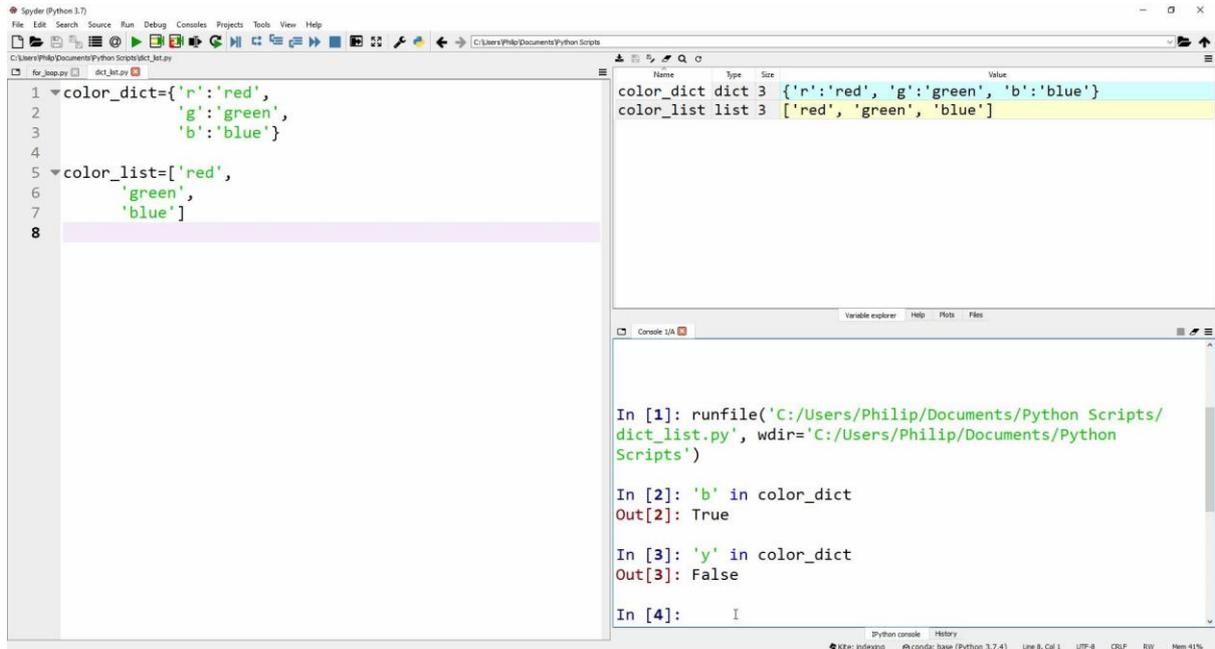


If we open both up in the variable explorer, we can see that the dictionary has the keys `'b'`, `'g'` and `'r'` while the lists has the indexes `0,1` and `2`.

We can then use the `in` command to check whether a key exists in a dictionary for instance.

```
'b' in color_dict
'y' in color_dict
```

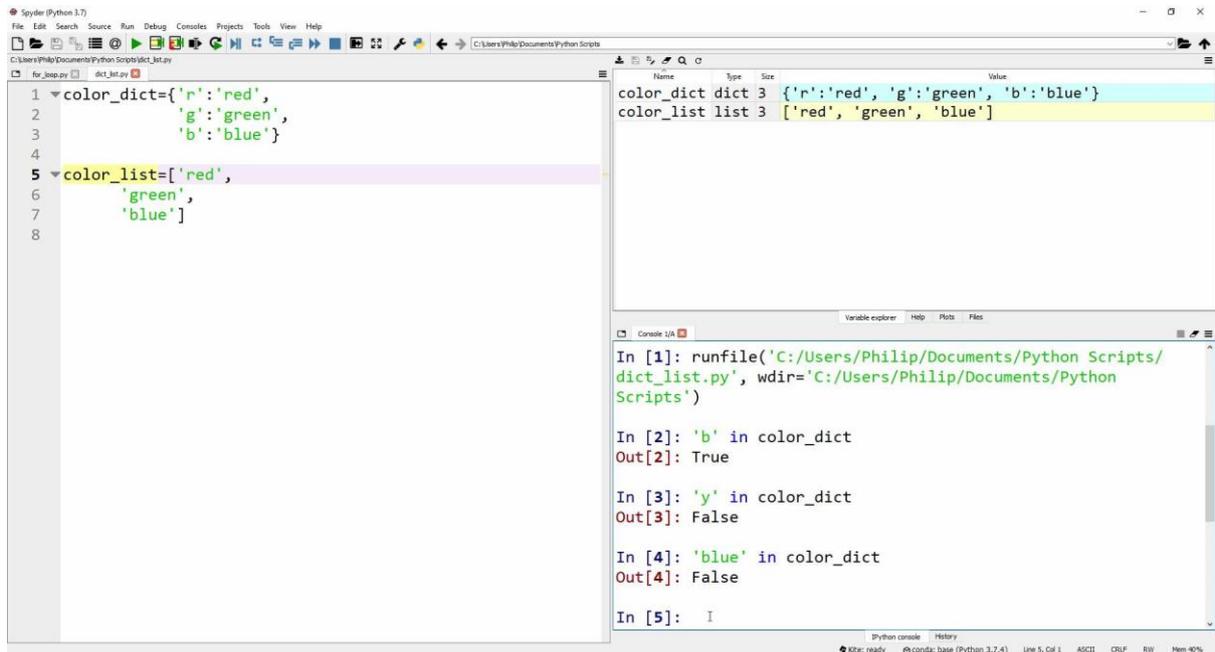
Which yield the Boolean `True` and `False` respectively.



In a dictionary the `in` command only searches through keys and it does not search through values so:

```
'blue' in color_dict
```

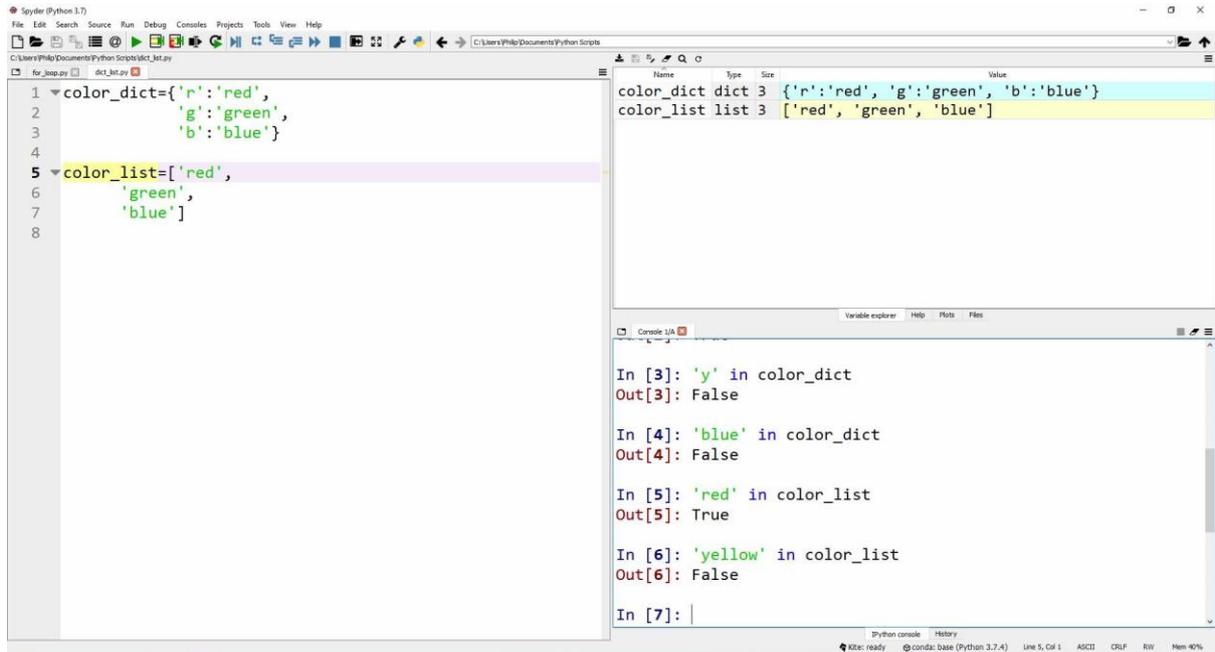
will search for the key `'blue'` which doesn't exist and return `False`.



In a list however there are no keys, so the command `in` will search for the value and not the index. For example:

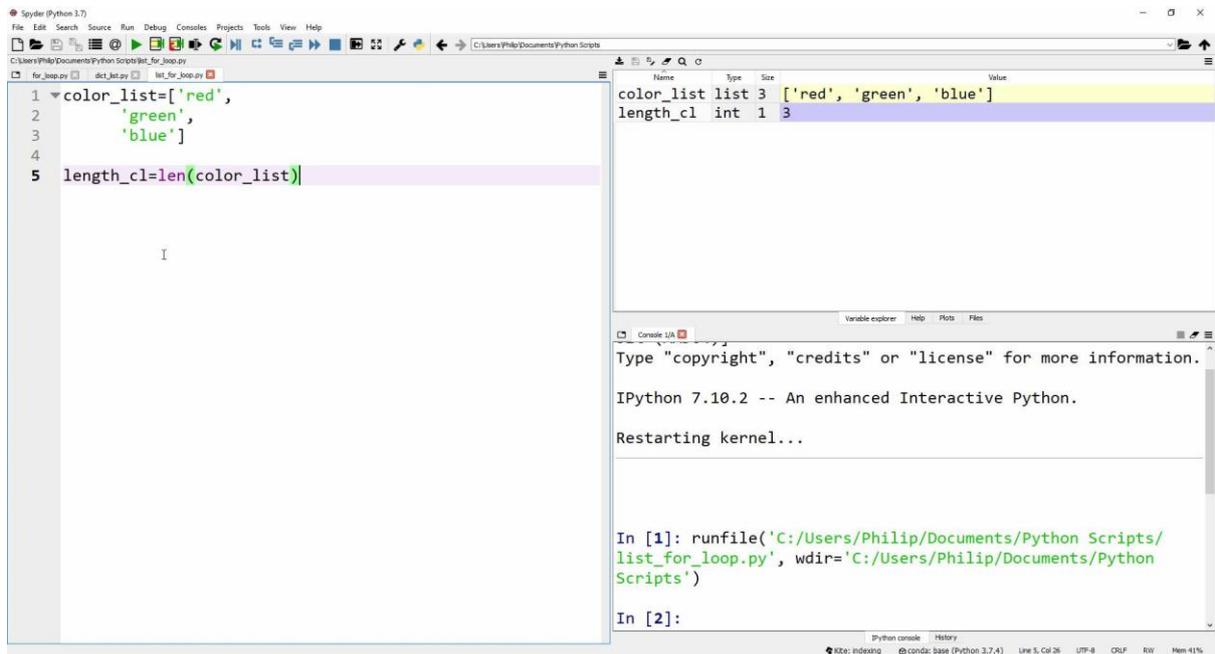
```
'red' in color_list
'yellow' in color_list
```

Will return `True` and `False` respectively.



For the list we can compute the length of the list using the `len` function.

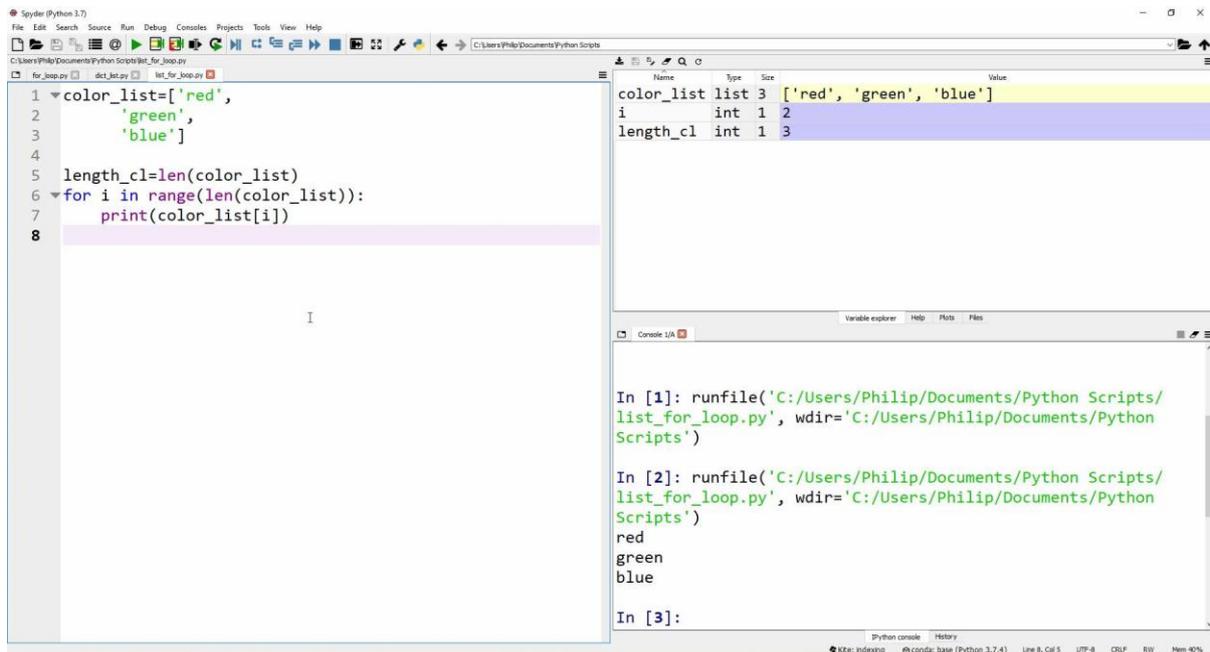
1. `color_list=['red', 'green', 'blue']`
- 2.
3. `length_cl=len(color_list)`



And we can thus use this value as the upper bound for a `for` loop.

1. `color_list=['red', 'green', 'blue']`
- 2.
3. `length_cl=len(color_list)`
4. `for i in range(len(color_list)):`
5. `print(color_list[i])`

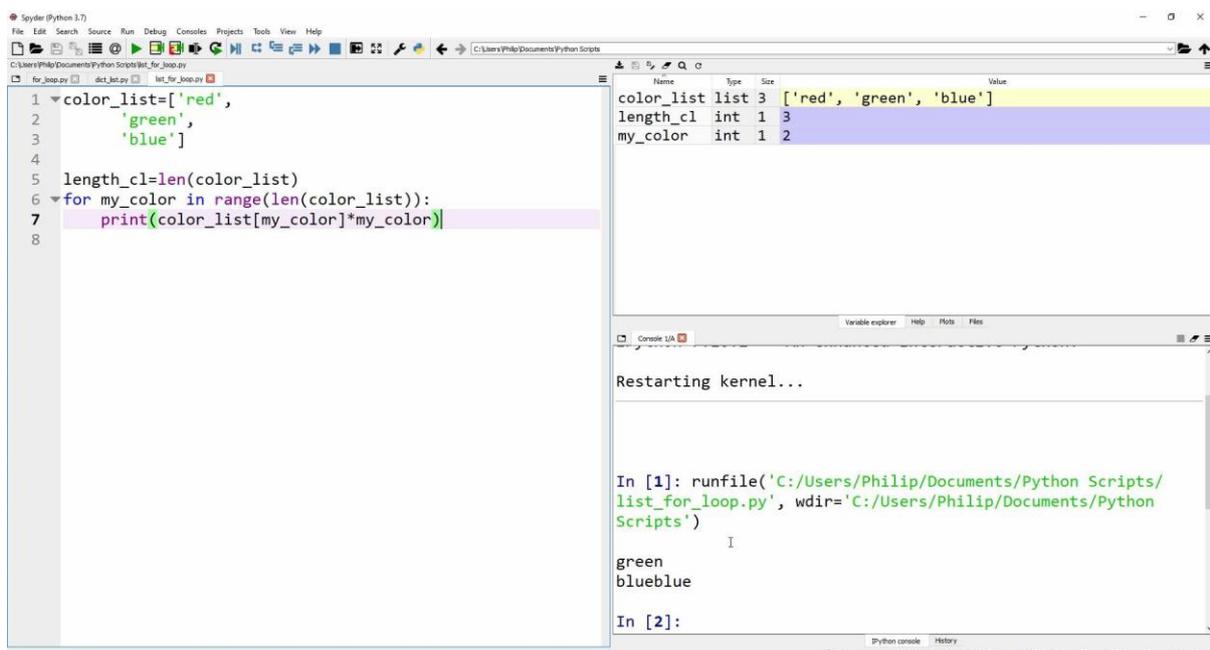
In this example we chose the letter **i** as an abbreviation for index however this is just a loop variable which we step through and it can be given any name we desire.



We can see this loop variable **i** show on the variable explorer. After the loop ends, its value is 2 which corresponds to the maximum index **2** of `colour_list`.

We can repeat this and call the loop variable **my_color** and use it as a multiplier in the **print** statement.

```
1. color_list=['red', 'green', 'blue']  
2.  
3. length_cl=len(color_list)  
4. for my_color in range(len(color_list)):  
5.     print(color_list[my_color]*my_color)
```

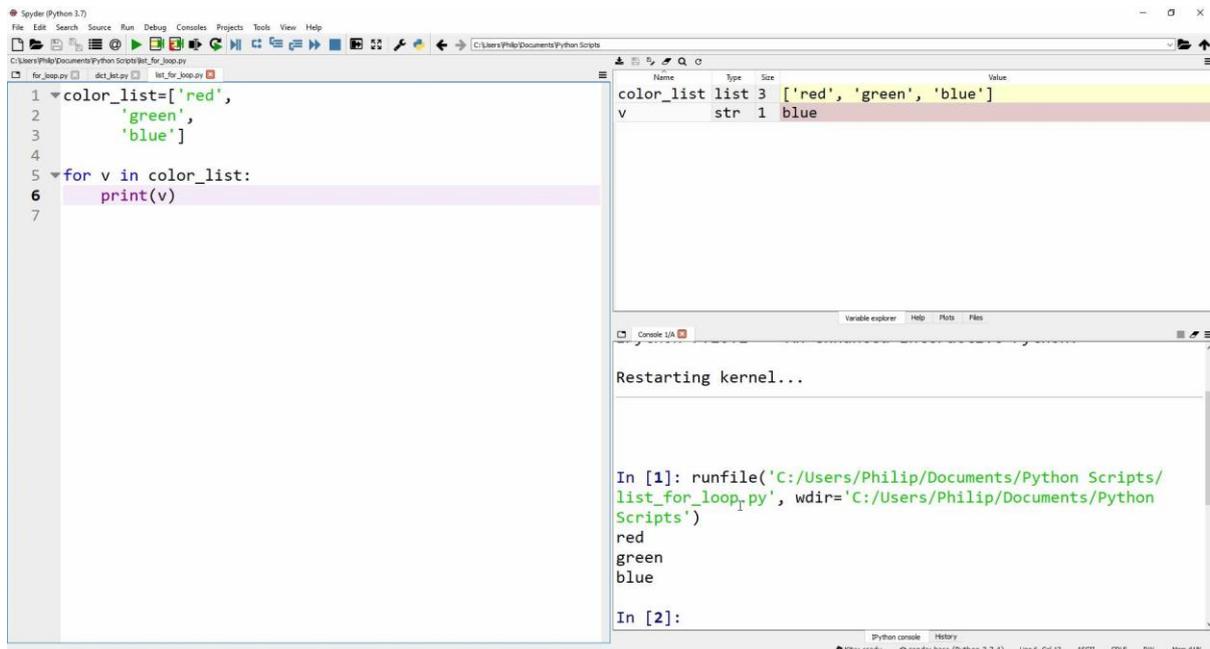


This means 'red' which is index 0 is printed out 0 times, 'green' which is index 1 is printed out 1 time and 'blue' which is index 2 is printed out 2 times.

The line `for i in range(len(color_list))` is quite cumbersome and can be abbreviated to `for v in color_list`. This has the similarity to English i.e. "for every value in color_list".

Note that `in` returns the value for item in the list opposed to the index of each item in the list as we seen when we used `in` out with a `for` loop. The code above can thus be simplified to:

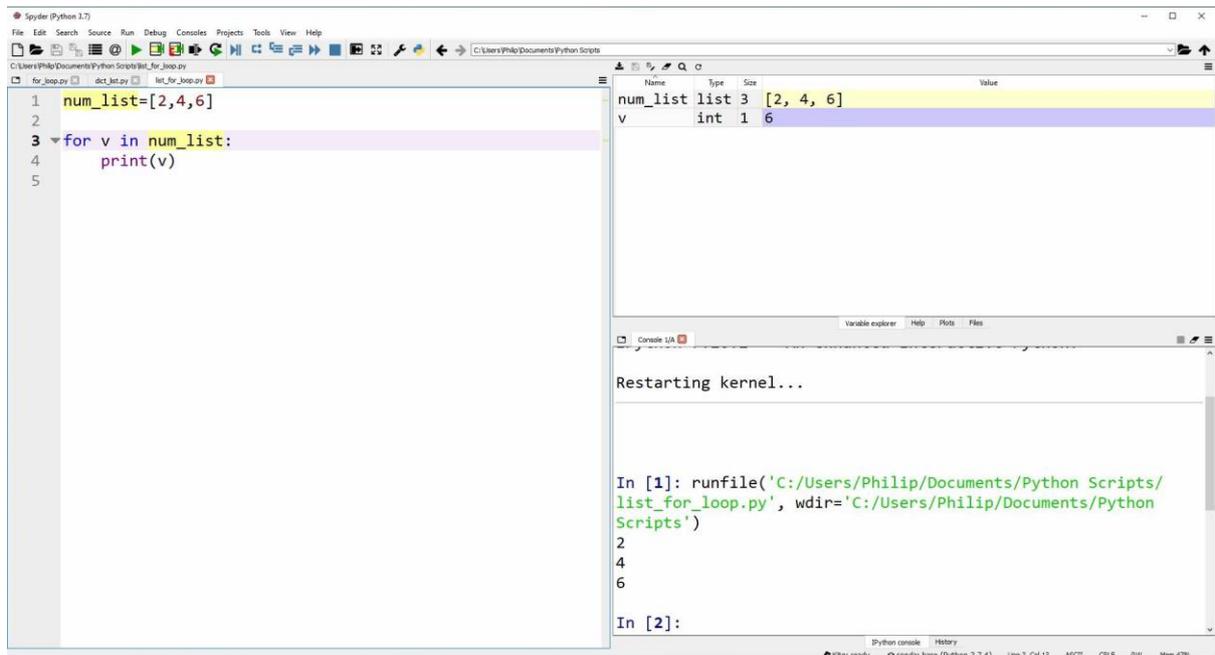
```
1. color_list=['red', 'green', 'blue']
2.
3. for v in color_list:
4.     print(v)
```



This can also be observed if we loop through a numerical list instead.

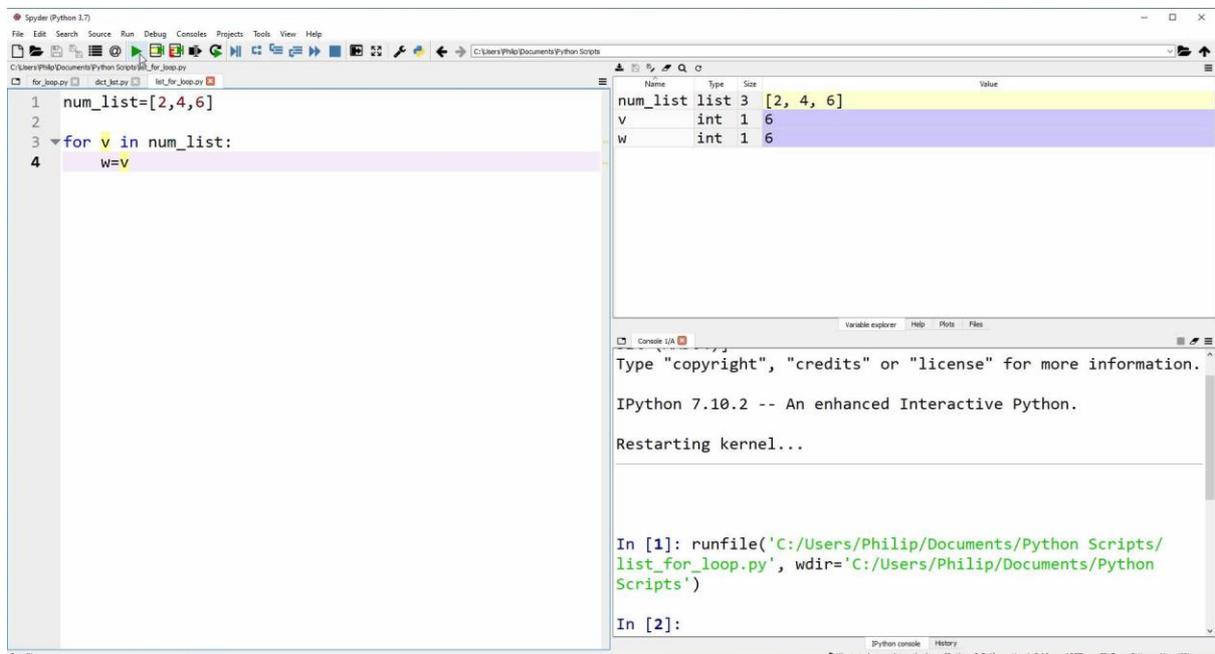
```
1. num_list=[2, 4, 6]
2. for v in num_list:
3.     print(v)
```

Here we see in the variable explorer that the value of `v` once the loop completes is `6`.



So far for demonstration we have only looked at printing values to the console. In many other cases we may want to create a new variable using code within the for loop. For instance, let's try to use the values of `num_list` within the for loop to create a new variable `w` which has a copy of the values of `num_list`.

1. `num_list=[2,4,6]`
2. `for v in num_list:`
3. `w=v`

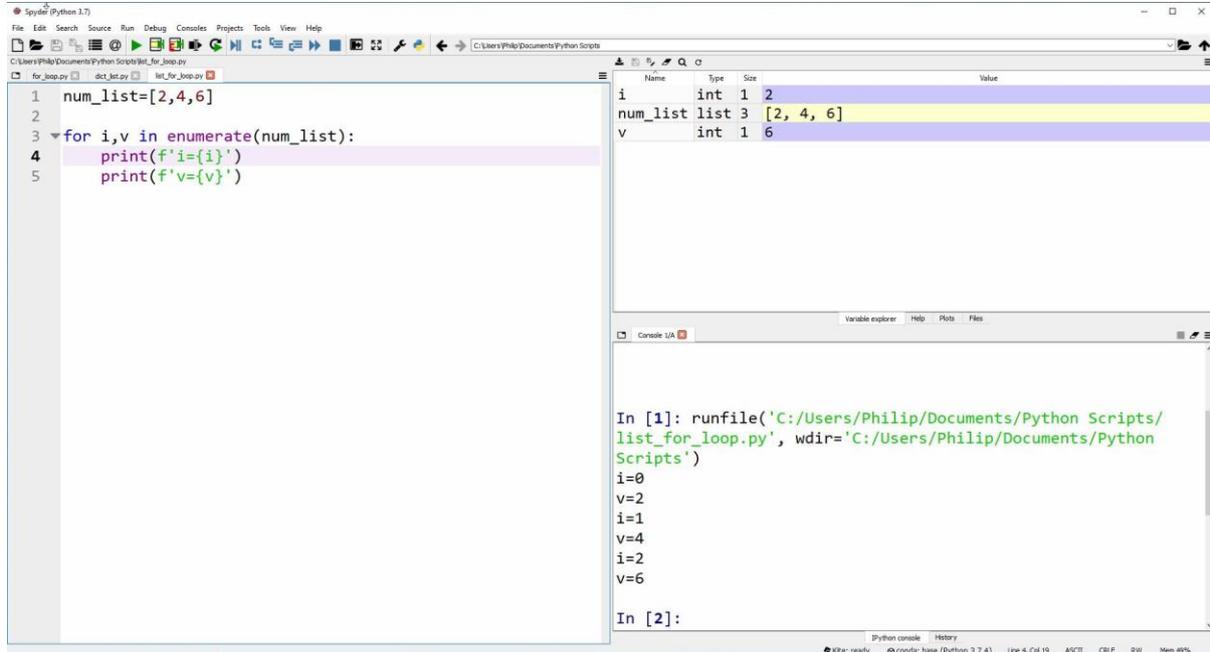


In this case the variable `w` has been assigned to the loop variable `v` and `v` is assigned the value of `num_list` and then overwritten for each iteration of the loop. In such a case it is useful to loop over both the indexes and values of a list which can be done using the `enumerate` function.

```

1. num_list=[2,4,6]
2.
3. for i,v in enumerate(num_list):
4.     print(f'i={i}')
5.     print(f'v={v}')

```

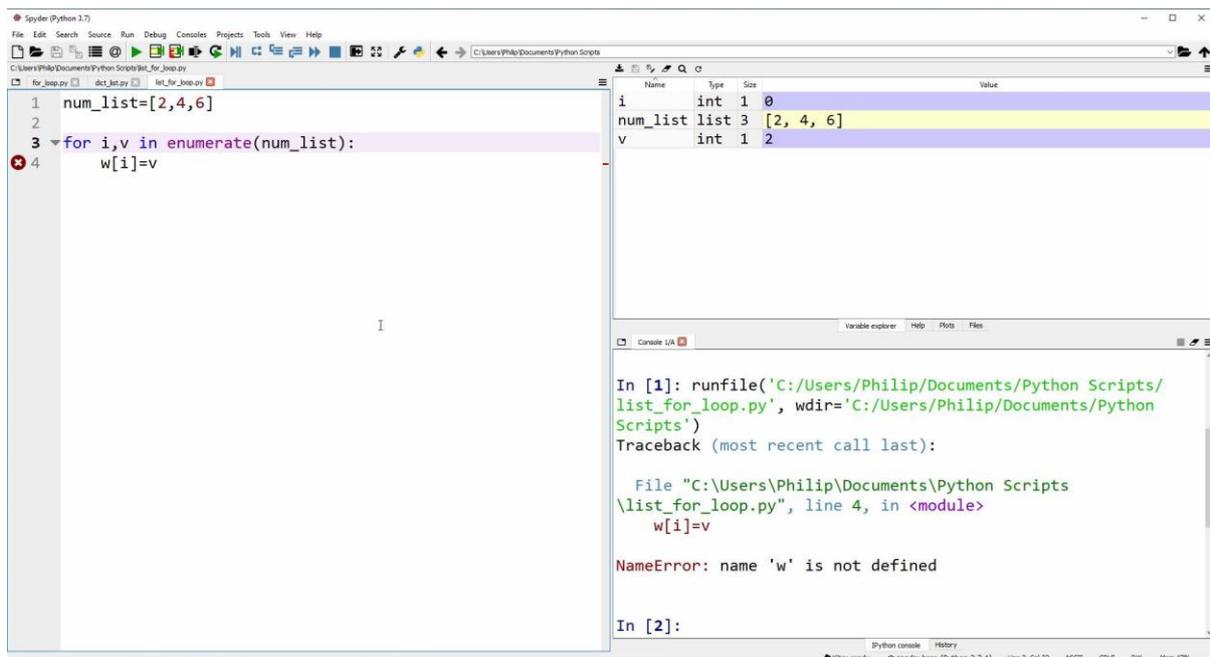


For `num_list` index `i` has a value of `v`. To create a copy of the values in `num_list` and assign them to a new variable `w`. means that `w[i]=v`.

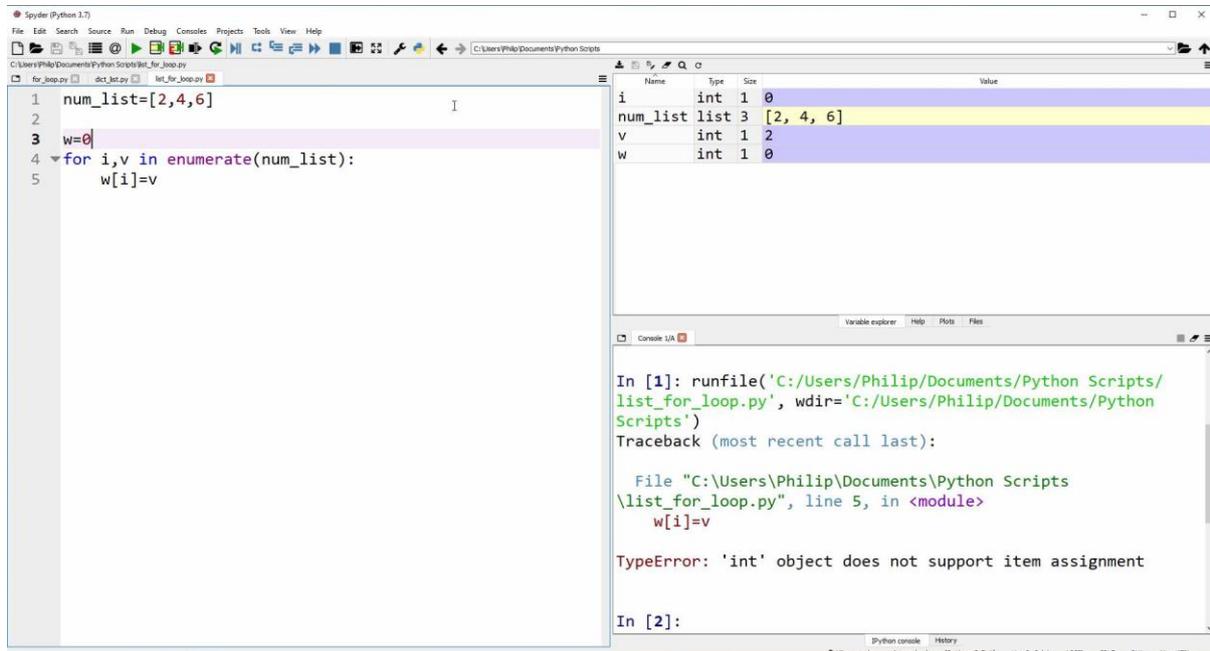
```

1. num_list=[2,4,6]
2. for i,v in enumerate(num_list):
3.     w[i]=v

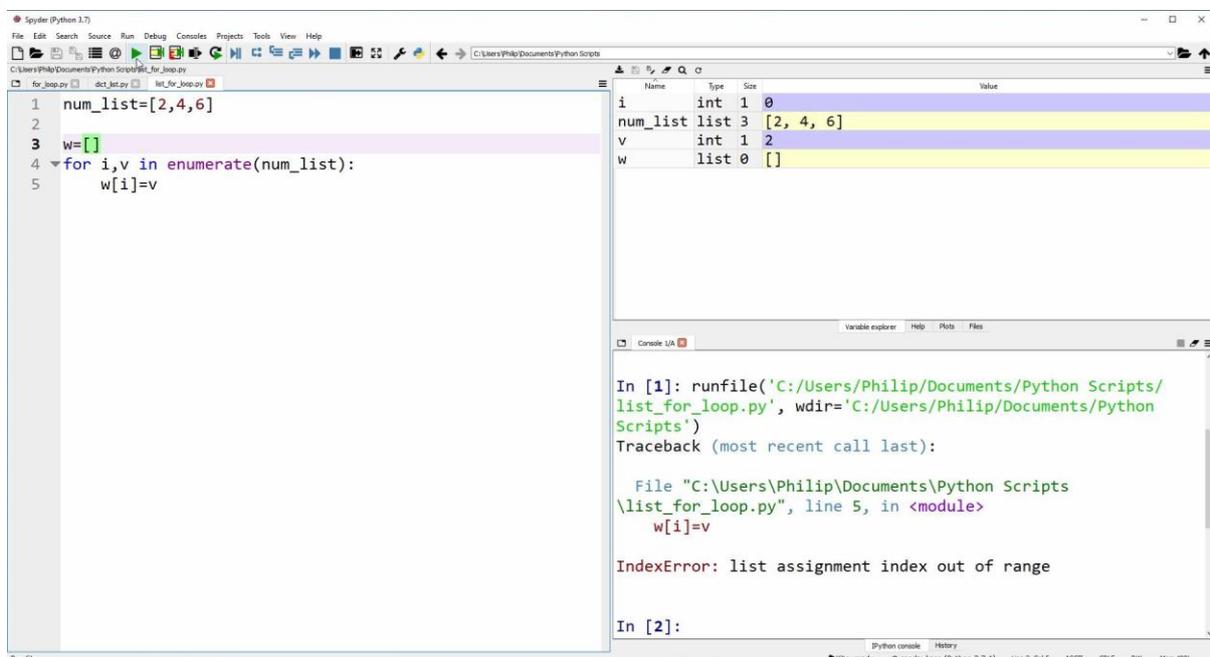
```



However when we attempt to do this we get an error `NameError: name 'w' is not defined`. This is because we are trying to index into a variable that has not been assigned and we must initialise `w` before we begin a loop. If we initialise it to 0 before beginning we will get a `TypeError: 'int' object does not support item reassignment`.



If we try and initialize it to an empty list, we get another error `IndexError: list assignment out of range`.

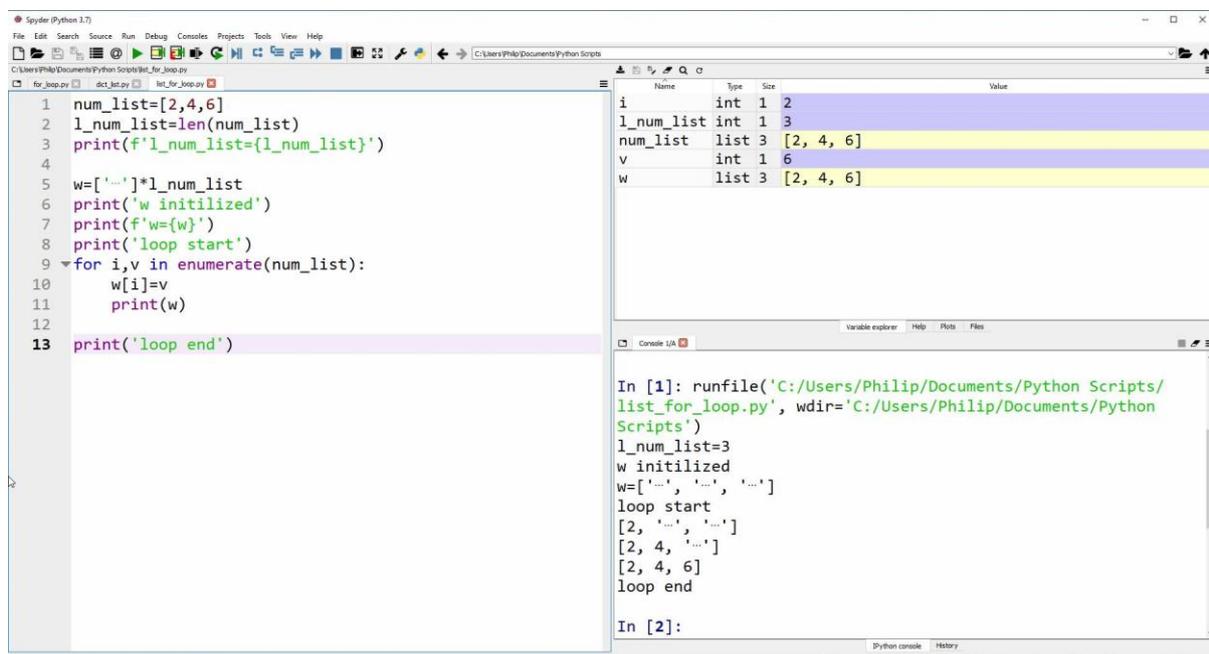


We must instead initialize it to have the final dimensions of the same list. To do this we can calculate the length of the list which will return an integer. We can then create a list using a string and multiply it by this integer. If we add `print` statements, we can see what `w` looks like before the loop and after each iteration of the loop.

```

1. num_list=[2,4,6]
2. l_num_list=len(num_list)
3. print(f'l_num_list={l_num_list}')
4.
5. w=['...']*l_num_list
6. print('w initialized')
7. print(f'w={w}')
8. print('loop start')
9. for i,v in enumerate(num_list):
10.     w[i]=v
11.     print(w)
12.
13. print('loop end')

```



Numerical Lists

So far, we have only looked at lists in the form of shopping lists which containing string values. In data science it is common to construct a collection of numbers together in a numeric list, commonly called a vector. Supposing we have a series of measurements where we measure both the velocity of a rocket and the time passed since the rocket launched.

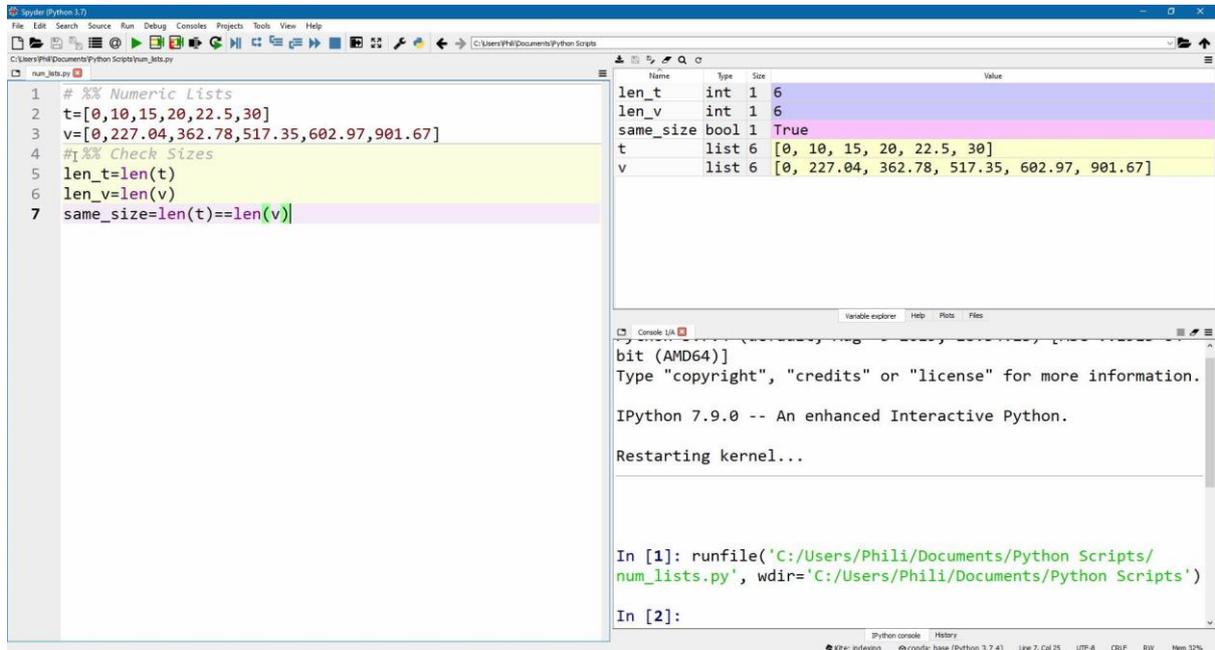
time (s)	velocity (m/s)
0	0
10	227.04
15	362.78
20	517.35
22.5	602.97
30	901.67

Two lists may be constructed from this dataset. Both lists should be the same length which can be checked using the `len` function and an equals statement comparing the length of both lists should give a Boolean value of `True`.

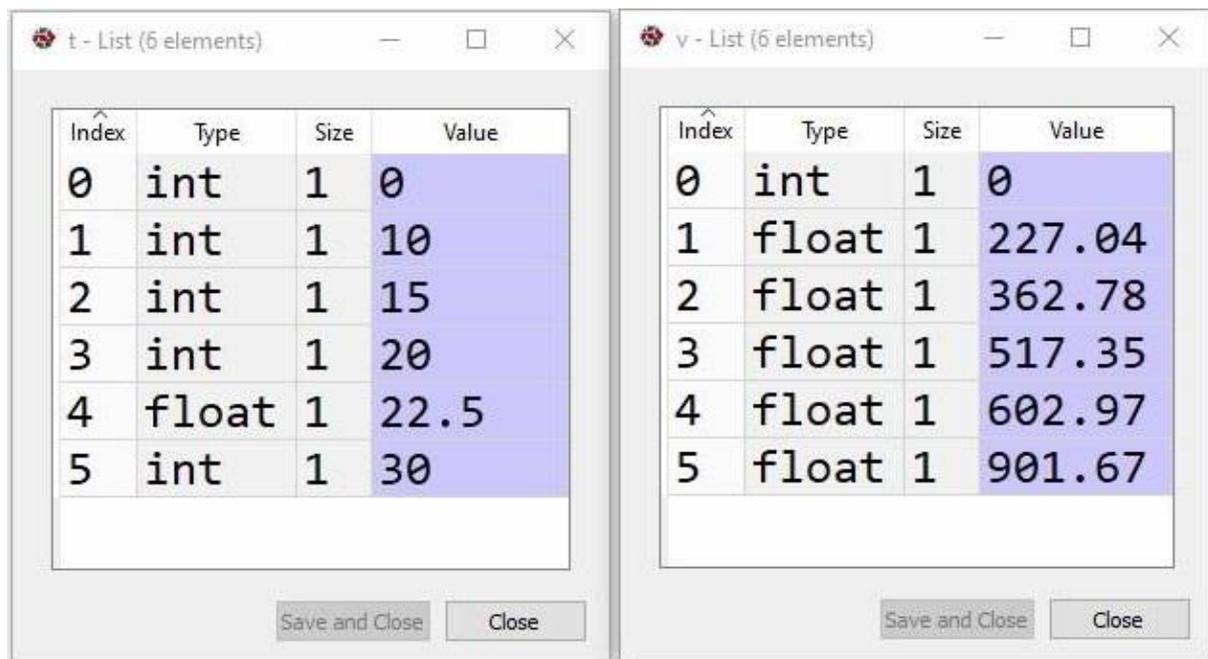
```

1. # %% Numeric Lists
2. t=[0,10,15,20,22.5,30]
3. v=[0,227.04,362.78,517.35,602.97,901.67]
4. # %% Check Size
5. len_t=len(t)
6. len_v=len(v)
7. same_size=len(v)==len(t)

```



The variables `t` and `v` can be expanded in the variable explorer. Here we see both have 6 elements and as a consequence the `same_size` Boolean is `True`. If we look at the Type we see that whole numbers have the Type `int` and decimal numbers have the Type `float`.

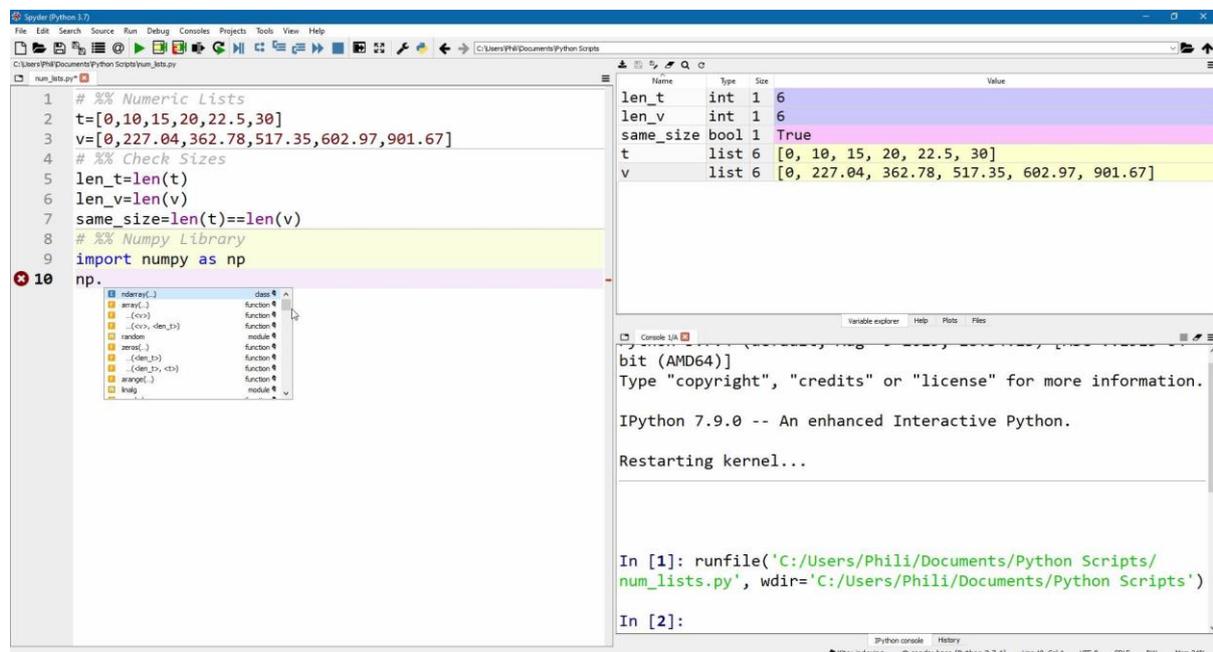


In lists each index can store a different data Types, such as numerical ints and floats shown above but they can also contain, strings, Booleans and as we have seen earlier other nested lists. This versatility is an advantage in some applications but a drawback in other applications. For example, in plotting when we want a list of numeric data, the ability to introduce a string or a nested list increases the likelihood of introducing a value, the plotting functions don't recognise and secondly increases the memory of each list which isn't a problem with such small list but can become a bottleneck for datasets with thousands or millions of datapoints.

The NUMerical PYthon Library (NUMPY)

We have seen earlier how to create and load custom functions. Python also has several standard libraries which each contain a multitude of functions. One of the most popular libraries is the numeric Python Library known as NumPy, usually written just in lower case. The Anaconda installation will contain this library by default. To use this library, we need to import it. As the numpy library is so commonly used, it is convention to import it as a two letter abbreviation `np`.

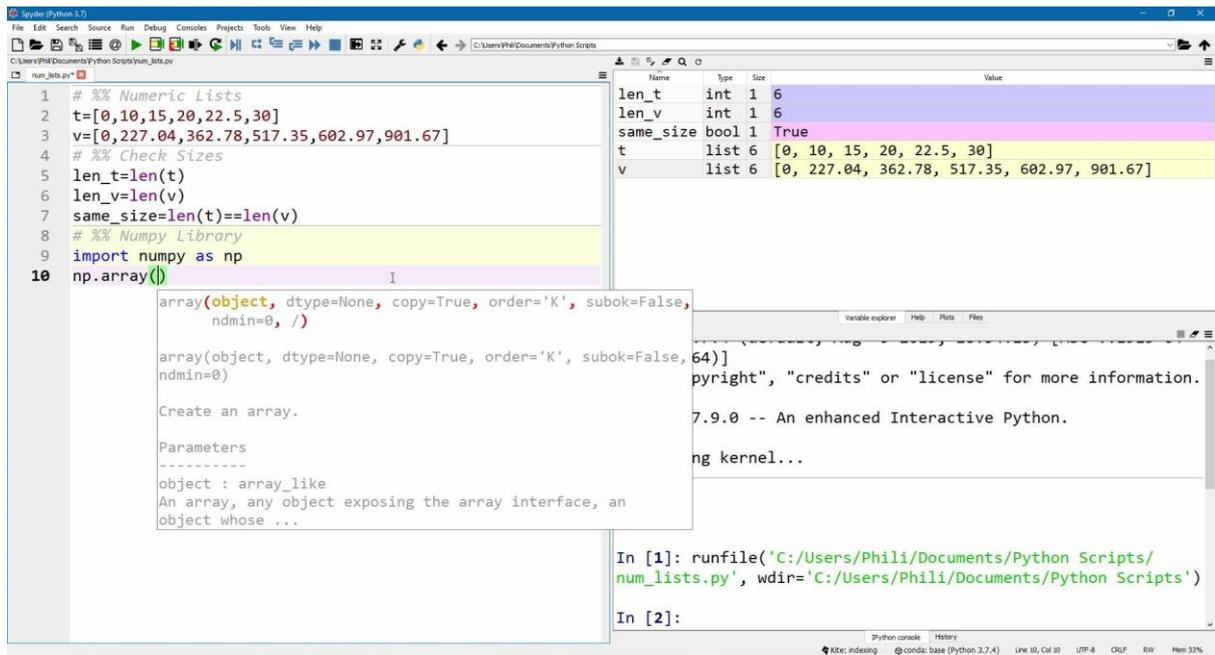
```
10. # %% Numpy Library
11. import numpy as np
```



Once the numpy library is imported as `np`, functions contained within the numpy library may be called up by typing in `np` and then dot `np.`. If `np.` is followed by a `<` then a list of available functions will display. It is worthwhile reinforcing a subtle point here. If functions are called directly from a library, they are functions from the library however if they are called directly from an object (which displays in the variable explorer) they are methods of the object.

```
library.function()
object.method()
```

It is worth scrolling through these when first loading a standard library. In this case, we will use `np.array`. Note typing in the function with empty brackets will display information about the input arguments. In this case, the only positional argument is `object`, which is the vector which we wish to make a numeric array from. There are keyword input arguments such as `dtype` which is assigned to a default value of `none`.



The `np.array` function can accept a python list as input argument, for example the lists `t` and `v` created earlier on. A list can also be directly typed in as an input argument.

```

1. # %% Numeric Lists
2. t=[0,10,15,20,22.5,30]
3. v=[0,227.04,362.78,517.35,602.97,901.67]
4. # %% Check Size
5. len_t=len(t)
6. len_v=len(v)
7. same_size=len(v)==len(t)
8. # %% Numpy Library
9. import numpy as np
10. t2=np.array(t)
11. v2=np.array(v)
12. t3_int=np.array([0,10,15,20,23,30])

```

The type shown in the variable explorer is an `array of float64` for `t2` and `v2` because the numpy array contains floats however it is `array of int32` for `t3_int` because the numpy array contains only integers.

```

1 # %% Numeric Lists
2 t=[0,10,15,20,22.5,30]
3 v=[0,227.04,362.78,517.35,602.97,901.67]
4 # %% Check Sizes
5 len_t=len(t)
6 len_v=len(v)
7 same_size=len(t)==len(v)
8 # %% Numpy Library
9 import numpy as np
10 t2=np.array(t)
11 v2=np.array(v)
12 t3_int=np.array([0,10,15,20,23,30])

```

Name	Type	Size	Value
len_t	int	1	6
len_v	int	1	6
same_size	bool	1	True
t	list	6	[0, 10, 15, 20, 22.5, 30]
t2	Array of float64 (6,)		[0. 10. 15. 20. 22.5 30.]
t3_int	Array of int32 (6,)		[0 10 15 20 23 30]
v	list	6	[0, 227.04, 362.78, 517.35, 602.97...
v2	Array of float64 (6,)		[0. 227.04 362.78 517.35 602.9...

```

bit (AMD64)
Type "copyright", "credits" or "license" for more information.

IPython 7.9.0 -- An enhanced Interactive Python.

Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/
num_lists.py', wdir='C:/Users/Phili/Documents/Python Scripts')

In [2]:

```

Numbers Decimal (Humans) vs Binary (Computers)

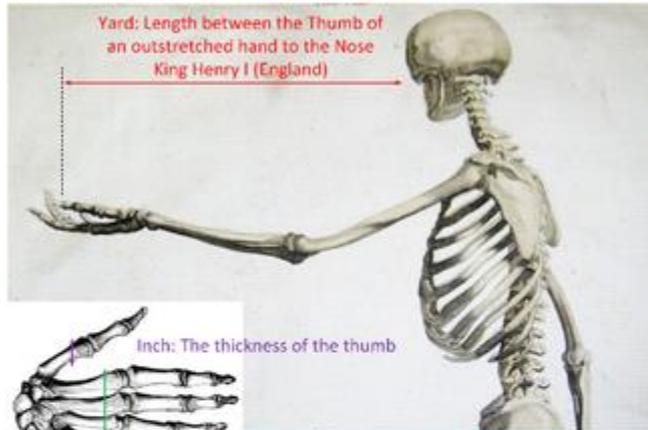
Before looking at how computers store numbers it is insightful to take a moment to take a look at how humans process objects and how a human's measurement system has developed. Take a long step back in time and assume that no numeric system, no monetary system and no languages have been invented yet.

Firstly, you only have the natural objects around yourself to work with. Then your point of reference would naturally be yourself and you would take the first thing to hand (your hand) to perform a measurement with. As your left hand and right hand are the same size you could size objects up using both your hands (this could lead to the unit the hand). If the object being measured was smaller than your hand you could instead use the thickness of each your four fingers to size it (this could lead to the unit the inch). Next you could compare the thickness of your hand to your foot (leading to the new unit the foot) which you could use for larger sizes. For a larger distance you could compare it to the length of your outstretched arm from your nose (leading to the new unit the yard) and for larger distances you could compare them to the distance you take when making a step (leading to the unit the step) and when you take a double step (leading to the unit the pace). Assuming you then went for a long walk you could then compare the distance of multiple paces (leading to the unit the mile). This gives the origins of Imperial units.

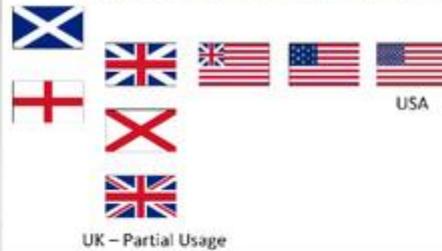
Now humans are collaborative so assume you have to work with your best friend on building a house from scratch. The problem is that your best friend is really tall and you are really short. If your best friend measures one side of the house using his hands and feet and you measure the other side of the house using your hands and feet then the house will be lopsided. Now assume you are living in a very basic village, all the humans in it had to get together and agree on a standard measurement. A single person could be taken as the standard of measurement, the king or queen who rules your village could be sized up by the village carpenter who could prescribe a series of markings on a piece of wood making a "ruler". Now although the carpenter is skilled their work isn't perfect and there are errors within the distance between each individual markings. These errors and the sizes spacing the markings only make the ruler suitable for use within a certain length scale. These concepts should be thought

about when performing any type of measurement science and are the basis of British Imperial Units (commonly used in the USA and the UK).

British Imperial Units



Countries using the Imperial Measurement System



Unit	Relation	Metric
Inch		2.54 cm
Hand	4 Inches	10.16 cm
Foot	12 Inches	30.48 cm
Yard	3 Feet	91.44 cm
Single Step	0.82 Yards	74.98 cm
Pace (Double Step)	1.64 Yards	149.96 cm
Mile	1760 Yards ~1000 Paces	1609 m

Mile:
1000 Paces from the Roman Empire



Early Standardisation:
Everyone had slightly different body dimensions.
A carpenter sized up the king or "ruler" of the day and marked up his dimensions on a piece of wood.

As a consequence to this day we call these measurement devices "rulers".



Decimal Counting System vs Binary Counting System

Note however that the imperial measurement system above has a different multiplication factor between each unit and although we can round it to make it nice whole numbers, we have to remember that there are 4 inches in a hand, 3 hands in a foot and 3 feet in a yard for example. When it comes to areas and volumes this becomes more complicated as the scaling factors get squared and cubed etc.

We must also assign characters to each number so we can understand them. Now we could create an arbitrary number of characters but as we touched upon when looking at zero-order indexing. We prescribe a character to each one of our 10 fingers. This gives rise to the decimal counting system.



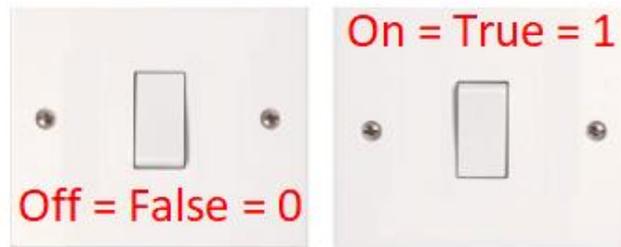
Our numeric counting system utilises 10 characters because each character corresponding to one of our fingers 0,1,2,3,4,5,6,7,8,9. To create larger numbers such as 11 we require 2 digits. On paper we can write down a number with an arbitrary number of number of digits. We also tend to use commas to separate out thousands, millions etc. in large numbers. For example 4,294,967,296.

The metric measurement system not only tries to use the 10 character decimal system but also to designate a scaling factor of 10 or 1,000 for every divisor.

Text	Symbol	Factor
tera	T	1,000,000,000,000
giga	G	1,000,000,000
mega	M	1,000,000
kilo	k	1,000
hecto	h	100
deca	da	10
		1
deci	d	0.1
centi	c	0.01
milli	m	0.001
micro	μ	0.000 001
nano	n	0.000 000 000
pico	p	0.000 000 000 001

Consider the metric units above and apply them to length. Now consider measurement devices. A ruler with metric markings for instance may work well in the decametres to millimetres range but will be too inaccurate for micrometres and certainly too inaccurate nanometres and picometres. Moreover, the dimensions of measurements used are usually application specific. For instance if you are trying to buy a wardrobe for your house, you are likely to measure its dimensions in centimetres cm and millimetres mm, attempting and failing to measure it down to an accuracy of a nanometre nm is pointless as the thickness of the material (wood) will be specified by the manufacturer to have a tolerance or error on the micrometre μm range. On the other side of the length scale, there is no need to compare it to the dimension of a kilometre km as this vastly exceeds the size of your house.

In the case of a computer, there are not 10 fingers and numbers are instead encoded using what can be thought of as a series of switches. These switches are known as binary switches as they only have 2 positions; off (0) and on (1). Recall using zero order indexing we count to 2 but do not include 2. For a single switch known as a 1 bit system there is only the combination 0_2 or 1_2 which correspond to 0_{10} and 1_{10} the subscript 2 and 10 denote binary and decimal respectively.



Bit 0

Binary Character	Decimal Value	Total
0	$2^0=1_{10}$	$0 \times 1_{10}$
0_2		0_{10}

Bit 0

Binary Character	Decimal Value	Total
1	$2^0=1_{10}$	$1 \times 1_{10}$
1_2		1_{10}

For a two switch or 2 bit system there can be the numbers 00_2 , 01_2 , 10_2 and 11_2 which correspond to 0_{10} , 1_{10} , 2_{10} and 3_{10} .



Bit 1		Bit 0		Total
Binary Character	Decimal Value	Binary Character	Decimal Value	
0	$2^1=1_{10}$	0	$2^0=1_{10}$	$0 \times 2_{10} + 0 \times 1_{10}$
00_2				0_{10}

Bit 1		Bit 0		Total
Binary Character	Decimal Value	Binary Character	Decimal Value	
0	$2^1=1_{10}$	1	$2^0=1_{10}$	$0 \times 2_{10} + 1 \times 1_{10}$
01_2				1_{10}

Bit 1		Bit 0		Total
Binary Character	Decimal Value	Binary Character	Decimal Value	
1	$2^1=1_{10}$	0	$2^0=1_{10}$	$1 \times 2_{10} + 0 \times 1_{10}$
10_2				2_{10}

Bit 1		Bit 0		Total
Binary Character	Decimal Value	Binary Character	Decimal Value	
1	$2^1=1_{10}$	1	$2^0=1_{10}$	$1 \times 2_{10} + 1 \times 1_{10}$
11_2				3_{10}

For a 3 bit system there are three switches and the combinations giving the numbers 000_2 , 001_2 , 010_2 , 011_2 , 100_2 , 101_2 , 110_2 , 111_2 which correspond to 0_{10} , 1_{10} , 2_{10} , 3_{10} , 4_{10} , 5_{10} , 6_{10} , 7_{10} .

Bit 2		Bit 1		Bit 0		Total
Binary Character	Decimal Value	Binary Character	Decimal Value	Binary Character	Decimal Value	
0	$2^2=4_{10}$	0	$2^1=1_{10}$	0	$2^0=1_{10}$	$0 \times 4_{10} + 0 \times 2_{10} + 0 \times 1_{10}$
000_2						0_{10}

Bit 2		Bit 1		Bit 0		Total
Binary Character	Decimal Value	Binary Character	Decimal Value	Binary Character	Decimal Value	
0	$2^2=4_{10}$	0	$2^1=1_{10}$	1	$2^0=1_{10}$	$0 \times 4_{10} + 0 \times 2_{10} + 1 \times 1_{10}$
001_2						1_{10}

Bit 2		Bit 1		Bit 0		Total
Binary Character	Decimal Value	Binary Character	Decimal Value	Binary Character	Decimal Value	
0	$2^2=4_{10}$	1	$2^1=1_{10}$	0	$2^0=1_{10}$	$0 \times 4_{10} + 1 \times 2_{10} + 0 \times 1_{10}$
010_2						2_{10}

Bit 2		Bit 1		Bit 0		Total
Binary Character	Decimal Value	Binary Character	Decimal Value	Binary Character	Decimal Value	
0	$2^2=4_{10}$	1	$2^1=1_{10}$	1	$2^0=1_{10}$	$0 \times 4_{10} + 1 \times 2_{10} + 1 \times 1_{10}$
011_2						3_{10}

Bit 2		Bit 1		Bit 0		Total
Binary Character	Decimal Value	Binary Character	Decimal Value	Binary Character	Decimal Value	
1	$2^2=4_{10}$	0	$2^1=1_{10}$	0	$2^0=1_{10}$	$1 \times 4_{10} + 0 \times 2_{10} + 0 \times 1_{10}$
100_2						4_{10}

Bit 2		Bit 1		Bit 0		Total
Binary Character	Decimal Value	Binary Character	Decimal Value	Binary Character	Decimal Value	
1	$2^2=4_{10}$	0	$2^1=1_{10}$	1	$2^0=1_{10}$	$1 \times 4_{10} + 0 \times 2_{10} + 1 \times 1_{10}$
101_2						5_{10}

Bit 2		Bit 1		Bit 0		Total
Binary Character	Decimal Value	Binary Character	Decimal Value	Binary Character	Decimal Value	
1	$2^2=4_{10}$	1	$2^1=1_{10}$	0	$2^0=1_{10}$	$1 \times 4_{10} + 1 \times 2_{10} + 0 \times 1_{10}$
110_2						6_{10}

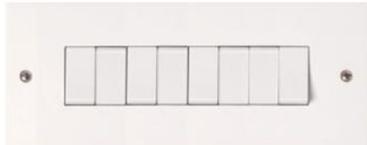
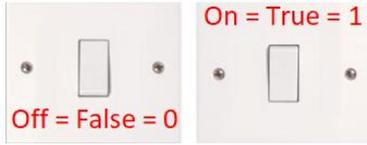
Bit 2		Bit 1		Bit 0		Total
Binary Character	Decimal Value	Binary Character	Decimal Value	Binary Character	Decimal Value	
1	$2^2=4_{10}$	1	$2^1=1_{10}$	1	$2^0=1_{10}$	$1 \times 4_{10} + 1 \times 2_{10} + 1 \times 1_{10}$

A *n* bit binary system allows us to count to a value of 2^{*n*}. We count to the value 2^{*n*} in steps of 1 but don't reach it (zero bit). Thus, the last value is 2^{*n*}-1.

Decimal (10 Characters)
0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Binary 1 Bit (2¹=2 combinations)
0, 1

8 Bit (2⁸=256 combinations)
1 Byte
1 Octet



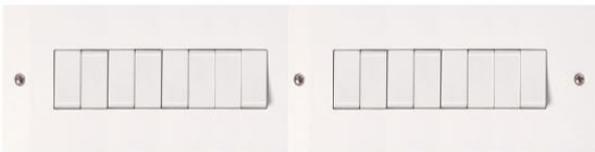
Comma Acts as a Separator for Every 3 Characters
4294967296
4,294,967,296

2 Bit (2²=4 combinations)
00, 01,
10, 11

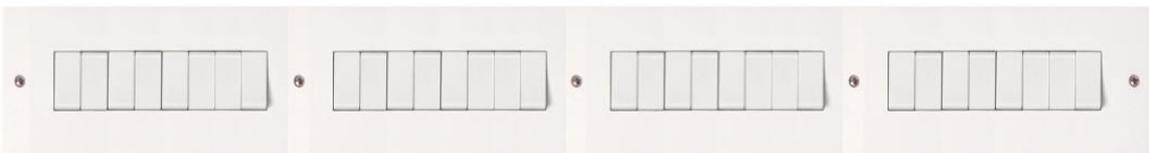


3 Bit (2³=8 combinations)
4 Bit (2⁴=16 combinations)
5 Bit (2⁵=32 combinations)
6 Bit (2⁶=64 combinations)
7 Bit (2⁷=128 combinations)

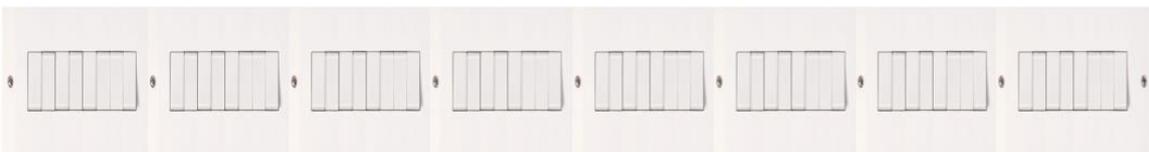
16 Bit (2¹⁶=65,536 combinations)
2 Bytes
2 Octets



32 Bit (2³²=4,294,967,296 combinations)
4 Bytes
4 Octets



64 Bit (2⁶⁴=18,446,744,073,709,551,616 combinations)
8 Bytes
8 Octets



8 Bit Integer Unsigned

In the case of an Unsigned (no negative values) 8 Bit Integer (whole numbers only) Binary (0 or 1 for each Bit) Counting System. We would count to a maximum value of $2^n - 1_{10} = 2^{(8_{10})} - 1_{10} = 256_{10} - 1_{10} = 255_{10}$ and the starting value would be 0_{10} . The 8 Bit Integer Unsigned therefore counts from 0_{10} to 255_{10} using.

Binary (unsigned integer 8)	Decimal
00000000 ₂	0 ₁₀
00000001 ₂	1 ₁₀
00000010 ₂	2 ₁₀
00000011 ₂	3 ₁₀
00000100 ₂	4 ₁₀
⋮	⋮
11111111 ₂	255 ₁₀

The unsigned int8 counting system has various uses such as for encoding characters on a keyboard and is important for color selection (which will be discussed in more detail later).

American Standard Code for Information Interchange (ASCII)

The ASCII character maps each of the unsigned 8 Bit integers to a command or character. The first 32 are mainly unprintable control codes used to control peripherals such as printers. The next values correspond to commonly used keys on English Keyboards. They can also be brought up by holding the alt button and typing the corresponding number on the number square of the keyboard.

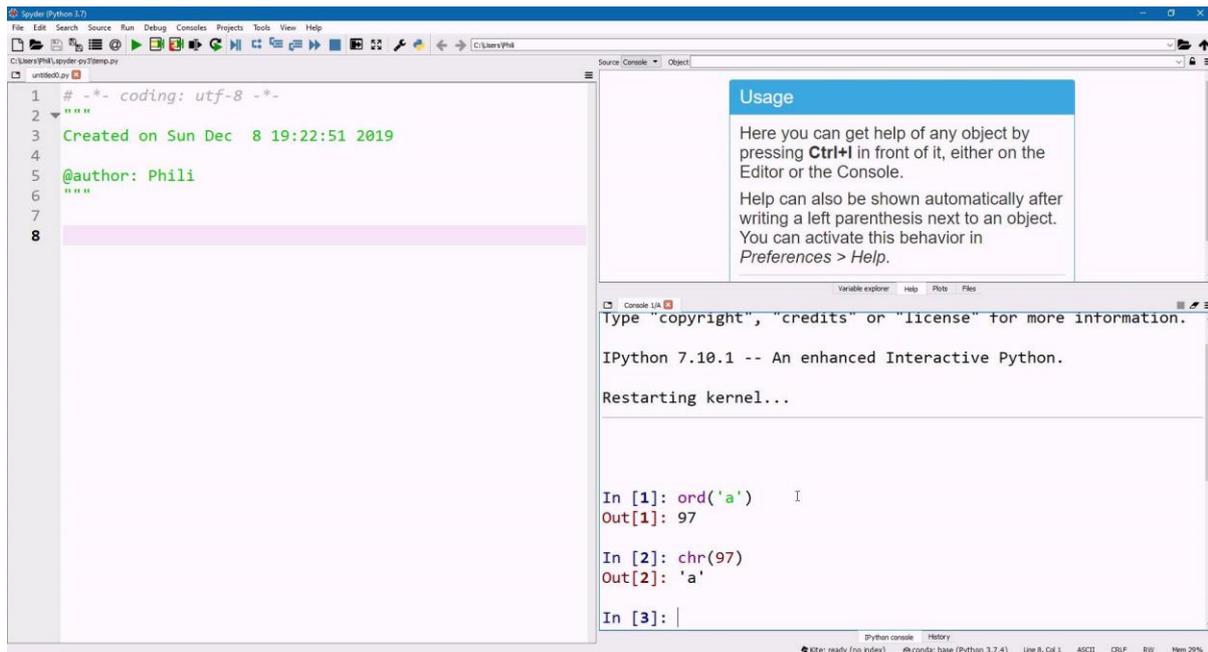
0	NULL	8	BS	16	DLE	24	CAN
1	SOH	9	HT	17	DC1	25	EM
2	STX	10	LF	18	DC2	26	SUB
3	ETX	11	VT	19	DC3	27	ESC
4	EOT	12	FF	20	DC4	28	FS
5	ENQ	13	CR	21	NAK	29	GS
6	ACK	14	SO	22	SYN	30	RS
7	BEL	15	SI	23	ETB	31	US
32		40	(48	0	56	8
33	!	41)	49	1	57	9
34	"	42	*	50	2	58	:
35	#	43	+	51	3	59	;

36	\$	44	,	52	4	60	<
37	%	45	-	53	5	61	=
38	&	46	.	54	6	62	>
39	'	47	/	55	7	63	?
64	@	72	H	80	P	88	X
65	A	73	I	81	Q	89	Y
66	B	74	J	82	R	90	Z
67	C	75	K	83	S	91	[
68	D	76	L	84	T	92	\
69	E	77	M	85	U	93]
70	F	78	N	86	V	94	^
71	G	79	O	87	W	95	_
96	`	104	h	112	p	120	x
97	a	105	i	113	q	121	y
98	b	106	j	114	r	122	z
99	c	107	k	115	s	123	{
100	d	108	l	116	t	124	
101	e	109	m	117	u	125	}
102	f	110	n	118	v	126	~
103	g	111	o	119	w	127	DEL

Characters from 128-255 consist of the extended ASCII characters.

The function `ord` may be used to look up a character and return its integer value and the function `chr` may be able to look up an integer value and return a character. For example:

```
ord('a')
chr(97)
```



In other words, the computer knows that if character 97_{10} or $0110\ 0001_2$ is selected from ASCII then the character `'a'` should be printed on screen.

Hexadecimal Counting System

As well as the Binary System and the Decimal System it is common to use the Hexadecimal system which has 16_{10} or 2^4 characters. We first take the ten numbers from the decimal system and then use the first 6 capital letters in the alphabet to give us the desired characters.

Binary: 0,1

Decimal: 0,1,2,3,4,5,6,7,8,9

Hexadecimal 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Because 16_{10} is 2^4 , a single character can be used to represent 4 bits.

Binary (unsigned integer 4)	Hexadecimal	Decimal
0000 ₂	0 ₁₆	00 ₁₀
0001 ₂	1 ₁₆	01 ₁₀
0010 ₂	2 ₁₆	02 ₁₀
0011 ₂	3 ₁₆	03 ₁₀
0100 ₂	4 ₁₆	04 ₁₀
0101 ₂	5 ₁₆	05 ₁₀
0110 ₂	6 ₁₆	06 ₁₀
0111 ₂	7 ₁₆	07 ₁₀
1000 ₂	8 ₁₆	08 ₁₀

1001 ₂	9 ₁₆	09 ₁₀
1010 ₂	A ₁₆	10 ₁₀
1011 ₂	B ₁₆	11 ₁₀
1100 ₂	C ₁₆	12 ₁₀
1101 ₂	D ₁₆	13 ₁₀
1110 ₂	E ₁₆	14 ₁₀
1111 ₂	F ₁₆	15 ₁₀

This means an unsigned integer 8 counting system can be represented by two characters opposed to 8.

Binary (unsigned integer 8)	Hexadecimal	Decimal
0000 0000 ₂	00 ₁₆	0 ₁₀
⋮		
0000 1010 ₂	0A ₁₆	10 ₁₀
⋮		
1010 1010 ₂	AA ₁₆	170 ₁₀
⋮		⋮
1111 1111 ₂	FF ₁₆	255 ₁₀

This 8 Bit system is commonly used for encoding colors which will be discussed in more detail in a later section.

8 Bit Integer Signed

In the case of an Unsigned (Negative and Positive Values) 8 Bit Integer (Whole Numbers Only) Binary (0 or 1 for each Bit) Counting System half of the values would be negative and half the values would be positive. Therefore, the lowest number we can count to would equal:

$$\frac{-2^n}{2} = \frac{-2^{\binom{8}{10}}}{2} = \frac{-256_{10}}{2_{10}} = -128_{10}$$

We need to take a value for 0, so the highest number which we can count to would be

$$\frac{2^n}{2} - 1 = \frac{2^{\binom{8}{10}}}{2} - 1 = \frac{256_{10}}{2_{10}} - 1_{10} = 127_{10}$$

This means we can count from -128_{10} to 127_{10} .

Binary (signed integer 8)	Decimal
00000000 ₂	-128 ₁₀
00000001 ₂	-127 ₁₀
00000010 ₂	-126 ₁₀
00000011 ₂	-125 ₁₀
00000100 ₂	-124 ₁₀
⋮	⋮
11111111 ₂	127 ₁₀

Bits and Bytes

To store higher values of integer numbers, more and more bits will need to be utilised which can get confusing quite quickly. Recall that when we write a large number in decimal, we frequently write it with commas to denote every 1,000 or 1,000,000 or 1,000,000,000. In binary it is common to group bits into bytes and a byte corresponds to 8 bits. If using the Hexadecimal notation, two Hexadecimal characters are placed side by side to represent a byte.

16 Bit Integer Unsigned

For a 16 Bit Integer Unsigned Binary Counting System we would count up to a maximum value of $2^n - 1_{10} = 2^{(16)} - 1_{10} = 65536_{10} - 1_{10} = 65535_{10}$ and the starting value would be 0_{10} .

Binary (unsigned integer 16)	Hexadecimal	Decimal
00000000 00000000 ₂	00 00 ₁₆	0 ₁₀
00000000 00000001 ₂	00 01 ₁₆	1 ₁₀
00000000 00000010 ₂	00 02 ₁₆	2 ₁₀
00000000 00000011 ₂	00 03 ₁₆	3 ₁₀
00000000 00000100 ₂	00 04 ₁₆	4 ₁₀
⋮	⋮	⋮
11111111 11111111 ₂	FF FF ₁₆	65535 ₁₀

16 Bit Integer Signed

For a 16 Bit Integer Binary Counting System half of the values would be negative and half the values would be positive. Therefore, the lowest number we can count to would equal:

$$\frac{-2^n}{2} = \frac{-2^{(16)}}{2} = \frac{-65536_{10}}{2_{10}} = -32768_{10}$$

We need to take a value for 0, so the highest number which we can count to would be

$$\frac{2^n}{2} - 1 = \frac{2^{(16)}}{2} - 1 = \frac{65536_{10}}{2_{10}} - 1_{10} = 32767_{10}$$

This means we can count from -128_{10} to 127_{10} .

Binary (signed integer 16)	Decimal
00000000 00000000 ₂	-32768_{10}
00000000 00000001 ₂	-32767_{10}
00000000 00000010 ₂	-32766_{10}
00000000 00000011 ₂	-32765_{10}
00000000 00000100 ₂	-32764_{10}
⋮	⋮
11111111 11111111 ₂	32767_{10}

Scientific Notation

So far, we have looked at encoding both positive and negative integers and we have deduced that in order to store large numbers we require a sufficient number of bits. Unfortunately however we do not have an infinite number of bits. Moreover we also have not considered numbers with a fractional or decimal component. Such numbers can range from extremely small to extremely large. Let's take for example the approximate length of a picometer and the distance covered in a light year. To specify these in decimal form on a piece of paper we would write down:

0.000 000 000 001₁₀ m

9 000 000 000 000 000₁₀ m

Adding these two together would give:

9 000 000 000 000 000.000 000 000 001₁₀ m

This number has 27 digits in decimal notation (10 characters), meaning storing this number to the precision above physically in binary (2 characters) would require substantial memory and we would thus struggle to perform any simple operations on the number due to the heavy memory usage.

Let's look at these numbers using scientific notation.

0.000 000 000 001₁₀

This number is smaller than 1 so we will need to multiple it by 10 and then divide it by 10.

00.000 000 000 01₁₀ × 10^{(-1)₁₀}

Let's repeat this...

000.000 000 000 1₁₀ × 10^{(-2)₁₀}

And again...

0000.000 000 001₁₀ × 10^{(-3)₁₀}

00000.000 000 01₁₀ × 10^{(-4)₁₀}

000000.000 000 1₁₀ × 10^{(-5)₁₀}

0000000.000 001₁₀ × 10^{(-6)₁₀}

00000000.000 01₁₀ × 10^{(-7)₁₀}

000000000.000 1₁₀ × 10^{(-8)₁₀}

$$0000000000.001_{10} \times 10^{(-9)}_{10}$$

$$0000000000.01_{10} \times 10^{(-10)}_{10}$$

$$0000000000.1_{10} \times 10^{(-11)}_{10}$$

We keep on going until the highest non-zero number is in front of the decimal point. In this case:

$$00000000001.0_{10} \times 10^{(-12)}_{10}$$

The zeros in the front can be removed giving:

$$1.0_{10} \times 10^{(-12)}_{10}$$

To a precision of 2 significant figures.

For the light year in meters, the number is larger than 0 so we divide by 10 and multiply by 10 instead.

$$9\ 000\ 000\ 000\ 000\ 000\ 000_{10}$$

$$9\ 000\ 000\ 000\ 000\ 000\ 000.0_{10}$$

$$900\ 000\ 000\ 000\ 000\ 000.00_{10} \times 10^{(1)}_{10}$$

$$90\ 000\ 000\ 000\ 000\ 000.000_{10} \times 10^{(2)}_{10}$$

$$9\ 000\ 000\ 000\ 000.0000_{10} \times 10^{(3)}_{10}$$

$$900\ 000\ 000\ 000.0000_{10} \times 10^{(4)}_{10}$$

$$90\ 000\ 000\ 000.00000_{10} \times 10^{(5)}_{10}$$

$$9\ 000\ 000\ 000.000000_{10} \times 10^{(6)}_{10}$$

$$900\ 000\ 000.0000000_{10} \times 10^{(7)}_{10}$$

$$90\ 000\ 000.00000000_{10} \times 10^{(8)}_{10}$$

$$9\ 000\ 000.000000000_{10} \times 10^{(9)}_{10}$$

$$900\ 000.000000000_{10} \times 10^{(10)}_{10}$$

$$90\ 000.000000000_{10} \times 10^{(11)}_{10}$$

$$9\ 000.000000000_{10} \times 10^{(12)}_{10}$$

$$900.000000000_{10} \times 10^{(13)}_{10}$$

$$90.000000000_{10} \times 10^{(14)}_{10}$$

We keep on going until the highest non-zero number is in front of the decimal point. In this case:

$$9.000000000000_{10} \times 10^{(15)}_{10}$$

We can remove the trailing zeros to get the desired precision (lets take it to 2 significant figures):

$$9.0_{10} \times 10^{(15)}_{10}$$

Now we have the two numbers in scientific notation:

$$1.0_{10} \times 10^{(-12)}_{10}$$

$$9.0_{10} \times 10^{(15)}_{10}$$

The value at the front is called the mantissa and is a decimal number that can be stored to a desired or a maximum precision applicable for a problem.

$$1.0_{10} \times 10^{(-12)}_{10}$$

The exponent is always an integer (whole number) and reflects the number of zeros behind or in front of the decimal point:

$$1.0_{10} \times 10^{(-12)}_{10}$$

Now we need to consider the operations that can be carried out between these two numbers. Lets have a look at addition and subtraction. The smaller number (s) is much smaller than the error or precision of the bigger number (b). Thus b is unchanged when s is added or subtracted from it.

$$b+s=b \text{ where } s \ll b$$

$$b-s=b \text{ where } s \ll b$$

For multiplication of these two numbers it is far easier to do in exponent form:

$$1.0_{10} \times 10^{(-12}_{10})} \times 9.0_{10} \times 10^{(15}_{10})}$$

We simply multiply the mantissa and add the exponents:

$$1.0_{10} \times 9.0_{10} \times 10^{(-12}_{10} + 15}_{10})}$$

$$9.0_{10} \times 10^{(3}_{10})}$$

For division we divide the mantissa and subtract the exponent

$$1.0_{10} \times 10^{(-12}_{10})} \div 9.0_{10} \times 10^{(15}_{10})}$$

$$1.0_{10} \div 9.0_{10} \times 10^{(-12}_{10} - 15}_{10})}$$

$$0.111111111_{10} \times 10^{(-27}_{10})}$$

$$1.11111111_{10} \times 10^{(-28}_{10})}$$

Significant Figures

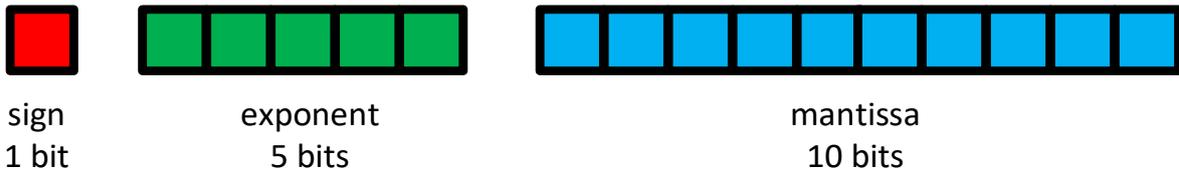
Note how when we divide 1 by 9, there is not an exact value. We instead get a recurring number. Let's look at the number 1.00000 divided by 9 using long division. For convenience 1.00000 can be taken as 100000×10^{-5} and 6 digits will be used. The 10^{-5} component can be ignored and applied to the final result after the long division has taken place allowing us to work with an integer number.

$$\begin{array}{r}
9 \overline{) 100000} \\
\underline{10} \\
0r1 \\
0 \\
9 \overline{) 100000} \\
\underline{10} \\
1r1 \\
01 \\
9 \overline{) 100000} \\
\underline{10} \\
1r1 \\
011 \\
9 \overline{) 100000} \\
\underline{10} \\
1r1 \\
0111 \\
9 \overline{) 100000} \\
\underline{10} \\
1r1 \\
01111 \\
9 \overline{) 100000} \\
\underline{10} \\
1r1 \\
01111r1 \\
9 \overline{) 100000}
\end{array}$$

The result of the long division is 011111 and applying the 1×10^{-5} brings this to 0.11111. The point to note regarding the long division is that each step of the long division leads to a remainder of 1 which is the source of recursion. At the last bit there is also a remainder of 1 but there are no more bits to store this remainder, so it is discarded. This means that no matter what precision we use (in the case above we have used 9 significant figures) there will always be an additional 1 out with the number of bits we select. This is a limitation when it comes to representing number with the arbitrary decimal system consisting of only 10 characters. Multiplying the result 9 times will give $9 \times 0.11111 = 0.99999$ which falls short of the starting value 1.00000. The error will become less and less significant when larger numbers of digits are used.

Float 16 (Half Precision) Float

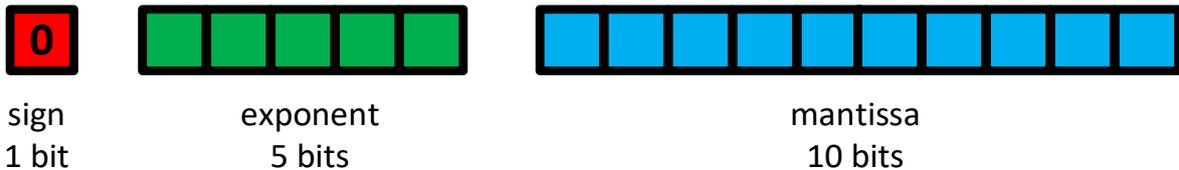
At first glance a Float may seem rather abstract. NumPy specifies a "Float16 Half precision float: sign bit, 5 bits exponent, 10 bits mantissa". Let's sketch this out. The Float16 is encoded using a binary number in scientific notation.



To see this, let's take a number with a decimal and examine it:

$$+11.5_{10}$$

The number is positive therefore the sign bit is 0. It would be 1 if the number was negative.



Let's now split this number into an integer and fraction:

$$+11.5_{10} = 11_{10} + 0.5_{10}$$

Let's examine the integer part. To do this let's have a look at the decimal values of 2^n .

Binary	Decimal
2^5	32_{10}
2^4	16_{10}
2^3	8_{10}
2^2	4_{10}
2^1	2_{10}
2^0	1_{10}

We can divide through by these to yield a whole number and remainder. This remainder can be divided through by the next power.

$$11_{10} / 32_{10} = 0_{10} r 11_{10}$$

$$11_{10} / 16_{10} = 0_{10} r 11_{10}$$

$$11_{10} / 8_{10} = 1_{10} r 3_{10}$$

$$3_{10} / 4_{10} = 0_{10} r 3_{10}$$

$$3_{10} / 2_{10} = 1_{10} r 1_{10}$$

$$1_{10} / 1_{10} = 1_{10} r 0_{10}$$

The value of the powers gives the number in binary:

$$11_{10} = 001011_2$$

Now let's do the same for the decimal part. To do this let's have a look at the decimal values of 2^{-n} :

Binary	Decimal
2^{-1}	0.5_{10}
2^{-2}	0.25_{10}
2^{-3}	0.125_{10}
2^{-4}	0.0625_{10}
2^{-5}	0.03125_{10}

We can divide through by these to yield a whole number and remainder. This remainder can be divided through by the next power.

$$0.5_{10}/0.5_{10}=1_{10}r0_{10}$$

$$0_{10}/0.25_{10}=0_{10}r0_{10}$$

$$0_{10}/0.125_{10}=0_{10}r0_{10}$$

$$0_{10}/0.0625_{10}=0_{10}r0_{10}$$

$$0_{10}/0.03125_{10}=0_{10}r0_{10}$$

The value of the powers gives the number in binary:

$$0.5_{10}=0.10000_2$$

Combining these gives:

$$11.5_{10}=001011.10000_2$$

This number is not in scientific notation and needs to be adjusted so the highest non-zero value, is in front of the decimal point. In the case of binary this is always going to be the largest 1 as 1 is the only non-zero value:

$$001011.10000_2=001011.10000_2 \times 2^0$$

$$001011.10000_2=00101.110000_2 \times 2^{(1_{10})}$$

$$001011.10000_2=0010.1110000_2 \times 2^{(2_{10})}$$

$$001011.10000_2=001.01110000_2 \times 2^{(3_{10})}=1.01110000_2 \times 2^{(3_{10})}$$

This gives the number in binary scientific notation (the exponent is still in decimal notation):

$$1.01110000_2 \times 2^{(3_{10})}$$

Where 1.01110000_2 is the mantissa and $2^{(3_{10})}$ is the exponent. Let's only look at the mantissa at this stage. When we put a number into scientific notation, we place the highest non-zero value in front of the decimal point. In decimal this can be 1,2,3,4,5,6,7,8 or 9 however in binary the only non-zero number is 1. Thus any binary number put into scientific notation will begin with $1.xxxxx$ and as all numbers have this, there is no need to encode this as it wastes memory in a computer. Therefore we take only the remaining digits 1.01110000_2 and add them to the mantissa. As this mantissa is written with only 8 bits but the numbering system being examined utilises 10 bits, two additional zeros will need to be added to the end to give 1.0111000000_2 . We can update our number to give:



sign
1 bit



exponent
5 bits



mantissa
10 bits

Now we can focus on the exponent $2^{(3_{10})}$ which in base 10 has the value of +3. The exponent is always going to be an integer and unlike the mantissa, the exponent does not have a separate value for the sign. Therefore, negative values must be considered. We have 2^n values for n bits and the first half of these will correspond to negative exponents and the second half of these will correspond to positive exponents. We must also compensate a value for zero. This means we will have to add an adjustment value of:

$$\left(\frac{2^n}{2} - 1\right)$$

In our case we have 5 bits so:

$$\left(\frac{2^5}{2} - 1\right) = 15_{10}$$

The lower bound -15_{10} will be added to the adjustment value 15_{10} and correspond to the binary value of 0 in the exponent. This compensation factor must be added to all the values of exponents when encoding as a 16 Bit float.

Our value of 3_{10} will therefore be adjusted to become $3_{10} + 15_{10} = 18_{10}$. For 5 bit let's once again have a look at the decimal values of 2^n .

Binary	Decimal
2^4	16_{10}
2^3	8_{10}
2^2	4_{10}
2^1	2_{10}
2^0	1_{10}

$$18_{10} / 16_{10} = 1_{10} r 2_{10}$$

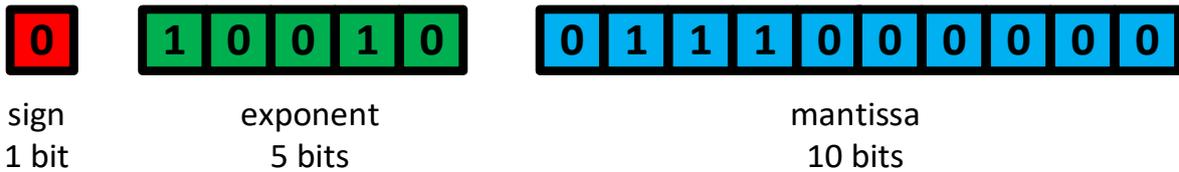
$$2_{10} / 8_{10} = 0_{10} r 2_{10}$$

$$2_{10} / 4_{10} = 0_{10} r 2_{10}$$

$$2_{10} / 2_{10} = 1_{10} r 0_{10}$$

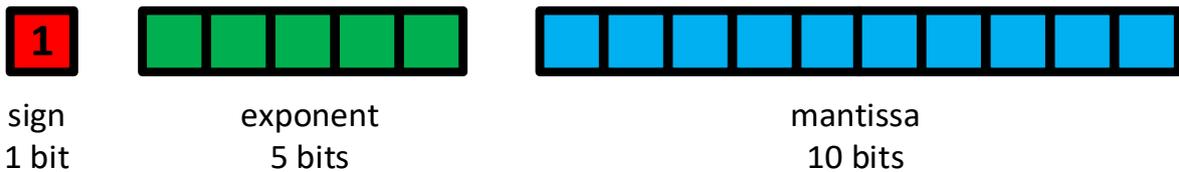
$$0_{10} / 1_{10} = 0_{10} r 0_{10}$$

This gives an adjusted exponent of **10010**. And thus, the number 11.5_{10} is encoded in the following form when using Float16 (Half precision float):



The Binary numbering system has only 2 characters and thus suffers from similar drawbacks to the decimal system which only has 10 characters. This can be seen if we try to encode the number -0.1_{10} as a binary number.

The number is negative therefore the sign bit is 1.



We have no integer part, so in this case we will just look at the decimal part. We can ignore the sign as it is already accounted for. To work out the value in the mantissa we can divide the decimal component by 2 to negative powers.

$$0.1_{10}/0.5_{10}=0_{10}r0.1_{10}$$

$$0.1_{10}/0.25_{10}=0_{10}r0.1_{10}$$

$$0.1_{10}/0.125_{10}=0_{10}r0.1_{10}$$

$$0.1_{10}/0.0625_{10}=1_{10}r0.0375_{10}$$

$$0.0375_{10}/0.03125_{10}=1_{10}r0.00625_{10}$$

$$0.00625_{10}/0.015625_{10}=0_{10}r0.00625_{10}$$

$$0.00625_{10}/0.0078125_{10}=0_{10}r0.00625_{10}$$

$$0.00625_{10}/0.00390625_{10}=1_{10}r0.00234375_{10}$$

$$0.00234375_{10}/0.001953125_{10}=1_{10}r0.000390625_{10}$$

$$0.000390625_{10}/0.0009765625_{10}=0_{10}r0.000390625_{10}$$

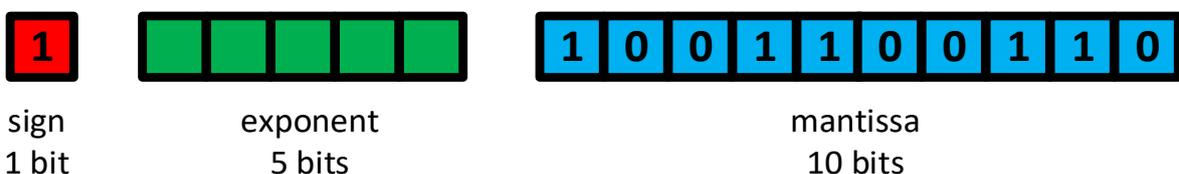
$$0.000390625_{10}/0.00048828125_{10}=0_{10}r0.000390625_{10}$$

$$0.000390625_{10}/0.000244140625_{10}=1_{10}r0.000146484375_{10}$$

$$0.000146484375_{10}/0.0001220703125_{10}=1_{10}r0.0000244140625_{10}$$

$$0.0000244140625_{10}/0.00006103515625_{10}=0_{10}r0.0000244140625_{10}$$

This gives 0.00011001100110_2 and we still have a remainder. This number is once again recurring but for Float 16 we only have 10 bits which means we take, $1.1001100110 \times 10^{(-4)_{10}}$ to give 1001100110.



The unadjusted power in this case is -4_{10} , recall that we need to add 15_{10} to this number giving 11_{10} .

$$11_{10}/16_{10}=0_{10}r2_{10}$$

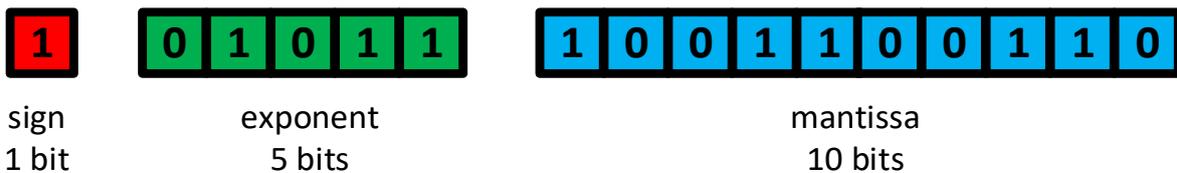
$$11_{10}/8_{10}=1_{10}r3_{10}$$

$$3_{10}/4_{10}=0_{10}r3_{10}$$

$$3_{10}/2_{10}=1_{10}r1_{10}$$

$$1_{10}/1_{10}=1_{10}r0_{10}$$

Thus -0.1_{10} is encoded as the following when using float16.



Comparison

The way a computer stores 8 bit (1 byte) integers and 16 bit (2 bytes) floats have been described above as they are amongst the simplest numbering schemes to explain. NumPy however will use 32 bit integer and 64 bit floats by default which have a higher dynamic range but in some applications using these may be a disadvantage as they have a heavier memory load. An analogy is to think of the ASCII example which maps out English keyboards. There are only 128 characters commonly used. Having a keyboard with thousands of characters could be useful in some cases but for the vast majority of cases it would just give a large number of keys which would make it harder for the user from finding the 128 keys they use and slow down their productivity. This can also happen in computer if more bits are used than necessary for large calculations. It is worth considering the dynamic range of the number systems. For integers we have the following:

Binary Integer System	Lower Bound	Upper Bound	Levels
8 Bit (1 Byte) Unsigned	0	$2^8 - 1 = 255$	$2^8 = 256$
8 Bit (1 Byte) Signed	$-\frac{2^8}{2} = -128$	$\frac{2^8}{2} - 1 = 127$	$2^8 = 256$
16 Bit (2 Bytes) Unsigned	0	$2^{16} - 1 = 65535$	$2^{16} = 65536$
16 Bit (2 Bytes) Signed	$-\frac{2^{16}}{2} = -32768$	$\frac{2^{16}}{2} - 1 = 32767$	$2^{16} = 65536$
32 Bit (4 Bytes) Unsigned	0	$2^{32} - 1 = 4294967295$	$2^{32} = 4294967296$
32 Bit (4 Bytes) Signed	$-\frac{2^{32}}{2} = -2147483648$	$\frac{2^{32}}{2} - 1 = 2147483647$	$2^{32} = 4294967296$

64 Bit (4 Bytes)		$2^{64} - 1 =$	$2^{64} =$
Unsigned	0	1844674407	1844674407
		3709551615	3709551616
64 Bit (4 Bytes)		$\frac{2^{64}}{2} - 1 =$	$2^{64} =$
Signed	$-\frac{2^{64}}{2} =$	$\frac{2^{64}}{2} - 1 =$	1844674407
	-9223372036854775808	9223372036854774	3709551616

For floats we have the so called Half, Single and Double precision float.

Binary System	Sign	Exponent	Mantissa
Float16	1 Bit	5 Bit	10 Bit
Half Precision	$+= 0, -= 1$	Range:	10 digits in the mantissa
16 Bit (2 Bytes)		$-\left(\frac{2^5}{2}\right) : \left(\frac{2^5}{2} - 1\right)$ -16: 15	
Float32	1 Bit	8 Bit	23 Bit
Single Precision	$+= 0, -= 1$	$-\left(\frac{2^8}{2}\right) : \left(\frac{2^8}{2} - 1\right)$	23 digit mantissa
32 Bit (4 Bytes)		-256: 255	
Float64 Double Precision 64 Bit (8 Bytes)	1 Bit $+= 0, -= 1$	11 Bit $-\left(\frac{2^{11}}{2}\right) : \left(\frac{2^{11}}{2} - 1\right)$ -1024: 1023	52 Bit 52 digit mantissa

Datatype and Number of Bytes

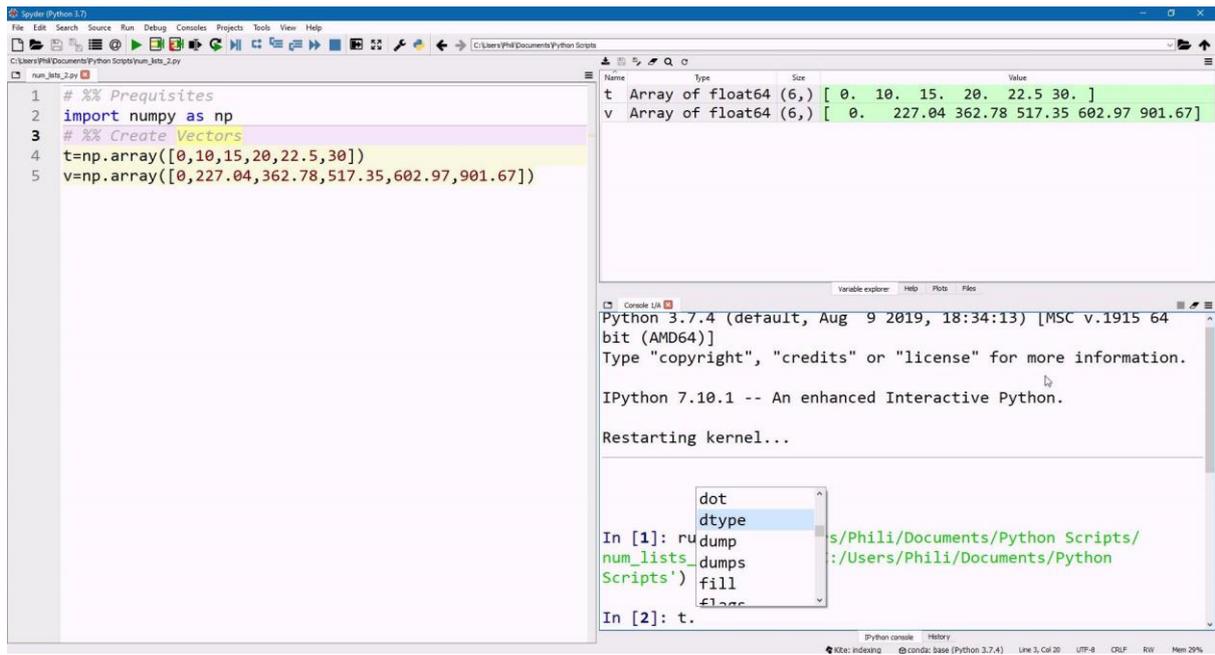
Each object created by the numpy library has the methods; `dtype` to lookup the datatype as well as `nbytes` to look up the number of bytes. Let's recreate the vectors `t` and `v` as numpy arrays creating and running the following script.

```

1. # %% Prerequisites
2. import numpy as np
3. # %% Create Vectors
4. t=np.array([0,10,15,20,22.5,30])
5. v=np.array([0,227.04,362.78,517.35,602.97,901.67])

```

The vector `t` can be called up in the console by typing in `t` followed by a `.` and then a `↵`.



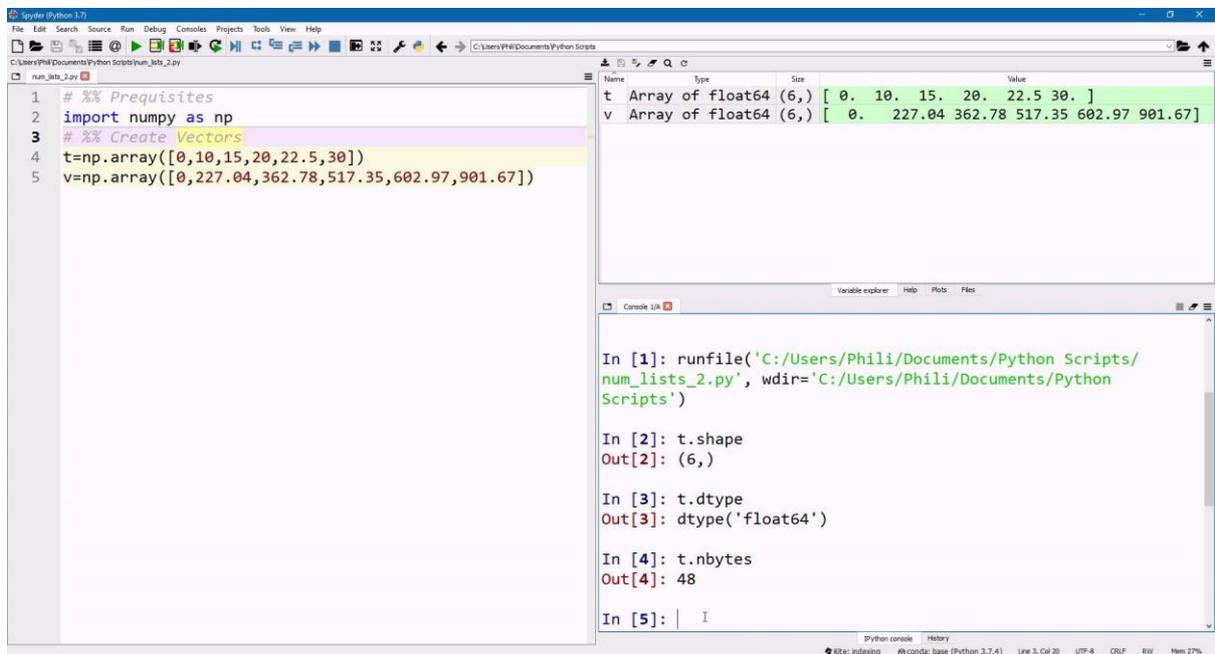
This will show a number of attributes that can be applied to the numpy vector `t`. We can look at the data type, shape of the numpy array and number of bytes using:

```

t.dtype
t.shape
t.nbytes

```

This gives a `dtype` of `float64`, `shape` of `(6,)` and `nbytes` of `48`. i.e. that there are 6 float64 values in the list which each have an 8 bytes (64 bits) taking 48 bytes of data.



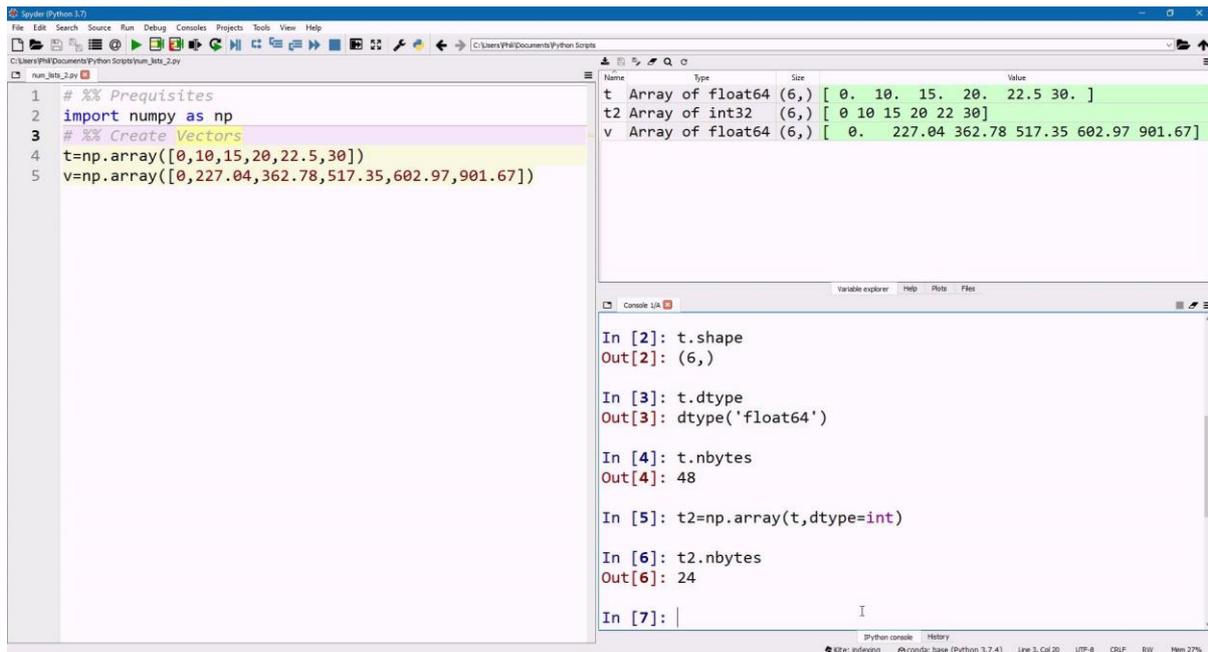
The vector `t2` can be made by copying vector `t1` and changing its `dtype` to an `int`.

```

t2=np.array(t, dtype=int)
t2.nbytes

```

Now we can see that the number of bytes is 24 because this time we have 6x4 byte (32 bit) integers.



2D Plots

Matplotlib – The Plotting Library

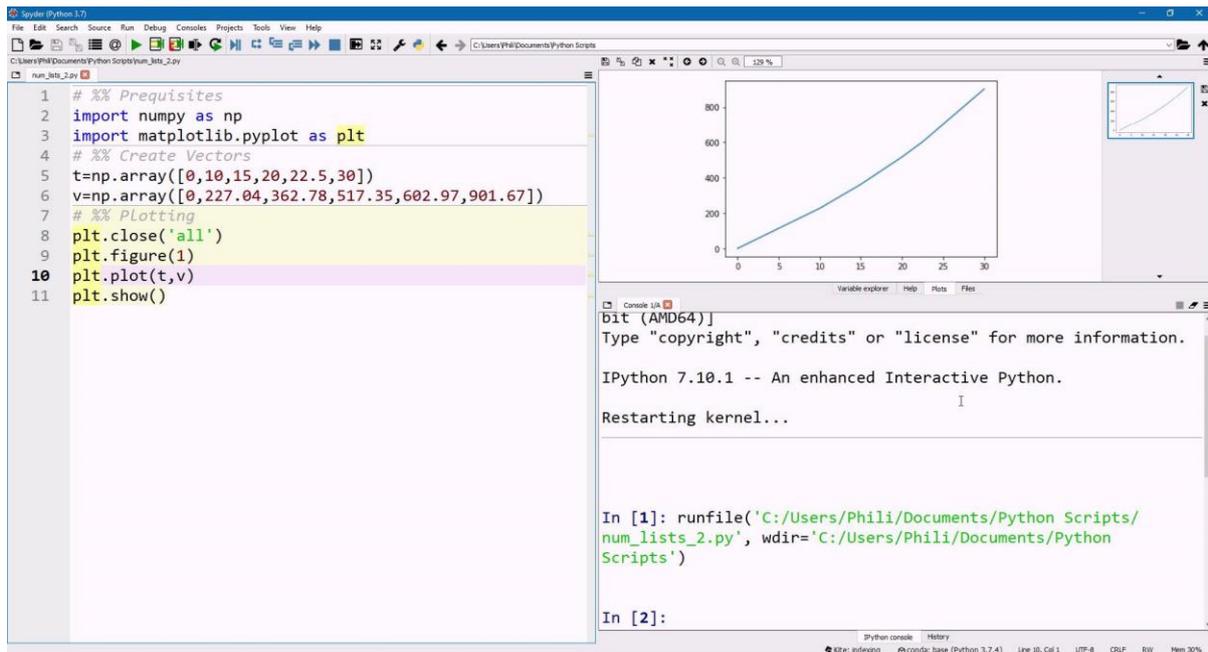
The most common Python plotting library is the matplotlib library which is highly based on Matlab. The `matplotlib.pyplot` module is used for 2D plots and is commonly imported as `plt`.

```

1. # %% Prerequisites
2. import numpy as np
3. import matplotlib.pyplot as plt
4. # %% Create Vectors
5. t=np.array([0,10,15,20,22.5,30])
6. v=np.array([0,227.04,362.78,517.35,602.97,901.67])
7. # %% Plotting
8. plt.close('all')
9. plt.figure()
10. plt.plot(t,v)
11. plt.show()

```

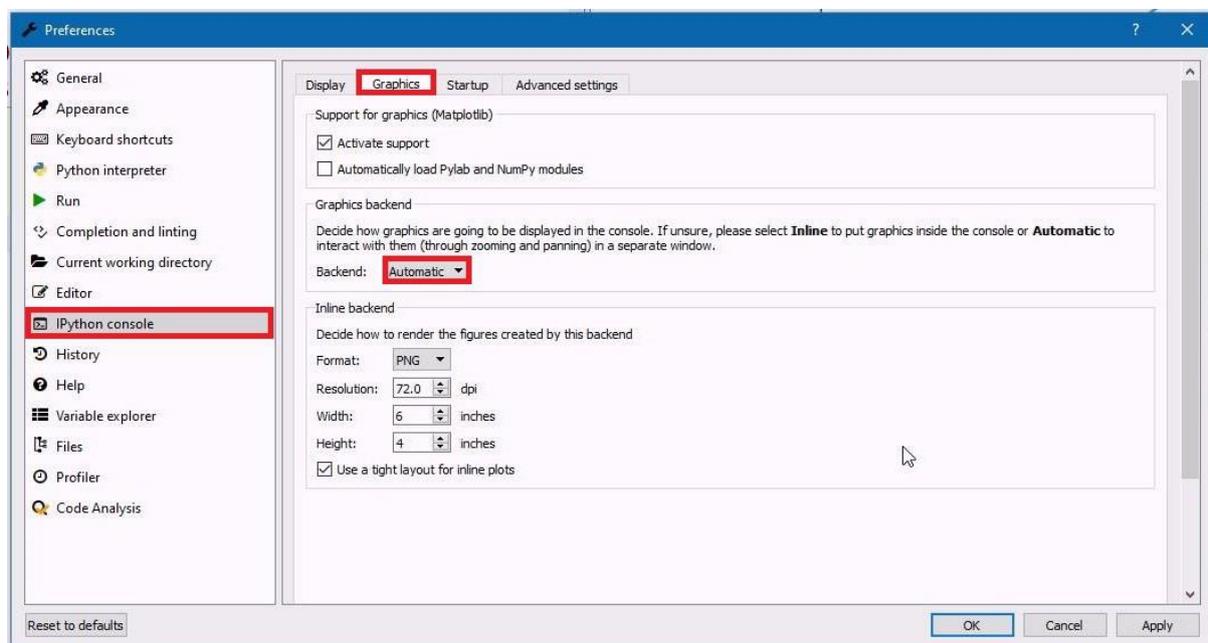
The function `figure` will create a new figure. If its input argument is left empty it will create a figure 1 if no figure is currently open or increment 1 past the last figure. Alternatively, a figure can be selected by typing in an input argument as an integer. The function `close` will close the selected figure if no input argument is selected, otherwise an input argument of an integer can be used to close a certain figure. It is common to use the command `close('all')` near the top of the script to close any previously open figures when relaunching a script. The command `plt.show()` is used to show the figure once all the lines of code for the figure have been created. By default, the view for Spyder 4 will be inline.



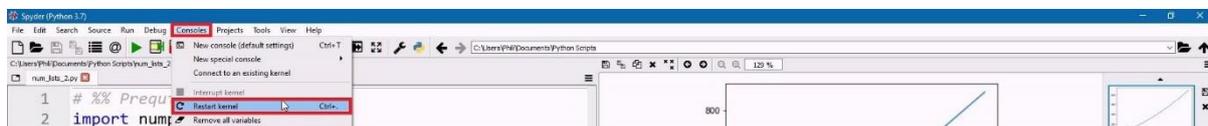
This means they will populate in the plots pane. To view the figure as a separate window, go to Tools and then Preferences:



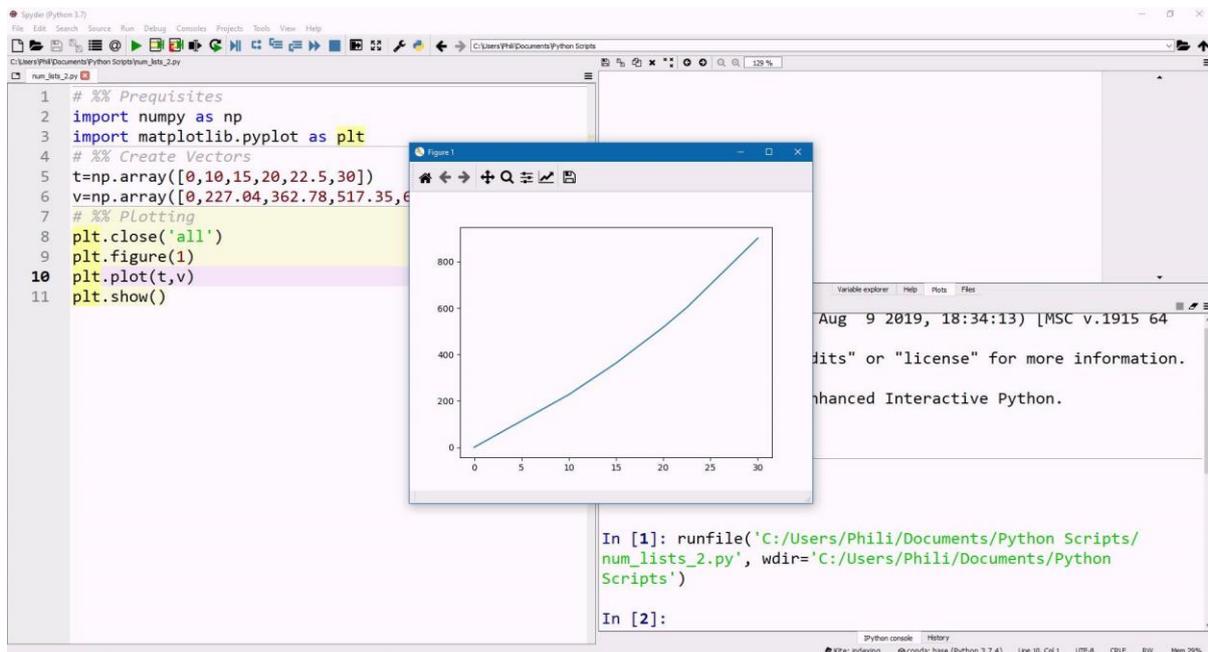
To the left-hand side select iPython Console, then select the Graphics Tab at the top and change the Graphics Backend from Inline to Automatic. Select Apply then OK.



Once this is done, restart your kernel.



Relaunching the script above will create the figure its own window.



It is also possible to toggle automatic and inline plotting without restarting the kernel using the following commands:

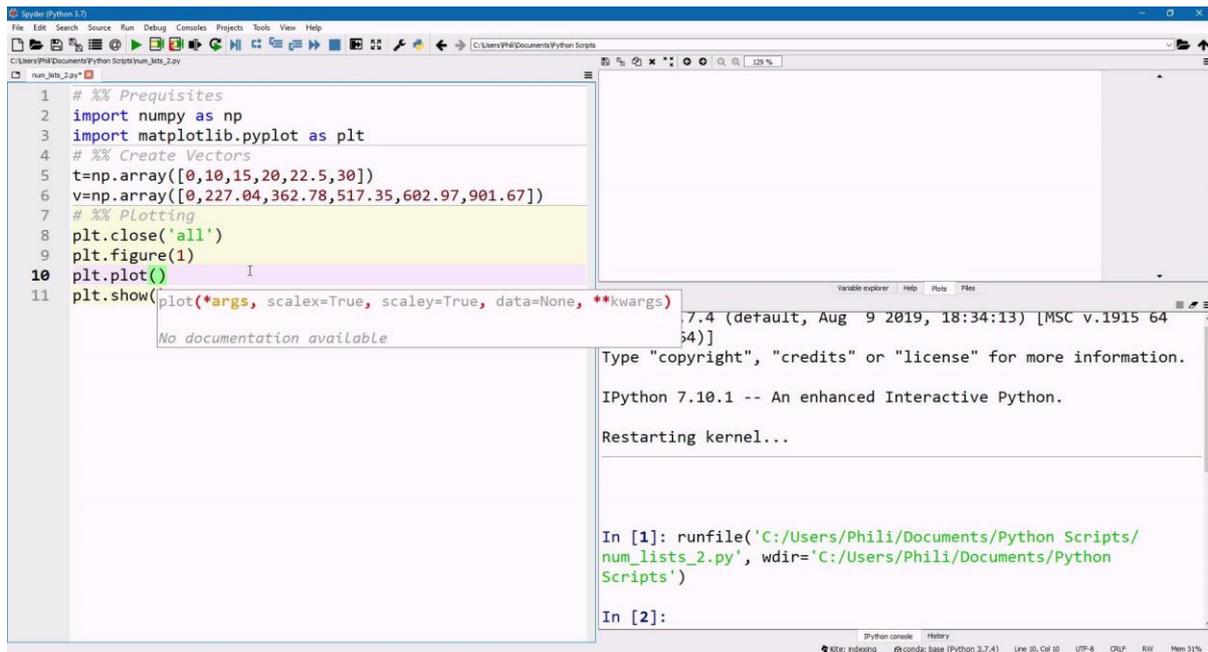
```
%matplotlib auto
%matplotlib inline
```

In the rest of this section we will plot figures out using automatic opposed to inline and screenshots will be shown of the automatic plot in a separate window.

Line Plot

We used the line plot above on line 10 and input the two positional arguments which are the x and y values respectively.

```
1. # %% Prerequisites
2. import numpy as np
3. import matplotlib.pyplot as plt
4. # %% Create Vectors
5. t=np.array([0,10,15,20,22.5,30])
6. v=np.array([0,227.04,362.78,517.35,602.97,901.67])
7. # %% Plotting
8. plt.close('all')
9. plt.figure()
10. plt.plot(t,v)
11. plt.show()
```



There are also several keyword arguments which if not assigned are default values. For instance, the line color which is a shade of blue, the line width which is by default a value of 1 and the line style which is by default solid. These can be set using the keyword arguments, `color`, `linewidth` and `linestyle` respectively.

Note the keyword argument `color` uses the old English spelling (current USA spelling) without the u. Select colors have 1 letter strings and strings (more colors will be examined in detail later)

1 Letter String

'r'
'b'
'g'
'c'
'y'
'm'
'k' ('b' is taken)
'w'

String

'red'
'blue'
'green'
'cyan'
'yellow'
'magenta'
'black'
'white'



For the `linestyle` we have:

1-2 Letter String

'solid'
'dashed'

String

'-'
'--'

'dashdot'

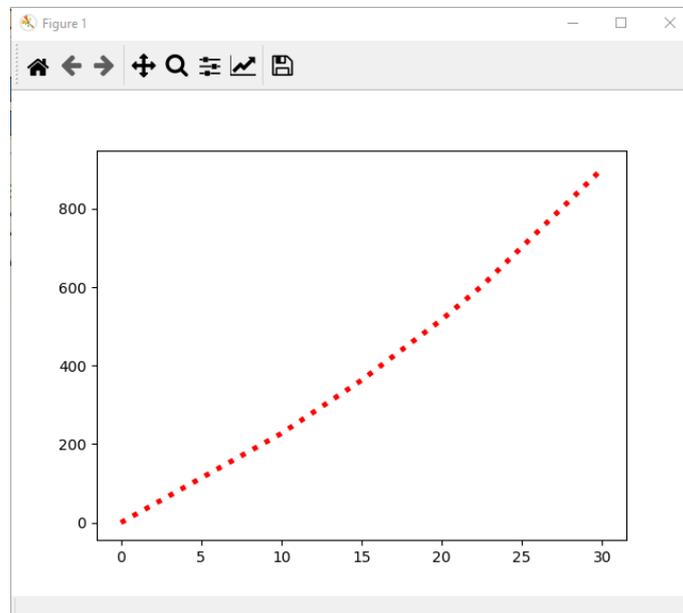
'-.'

'dotted'

':'

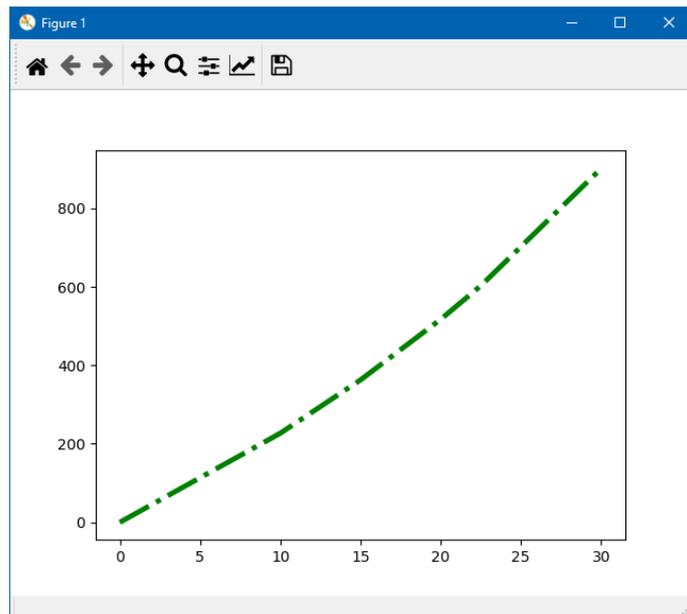
The keyword argument `linewidth` is assigned to an integer or float.

```
1. # %% Perquisites
2. import numpy as np
3. import matplotlib.pyplot as plt
4. # %% Create Vectors
5. t=np.array([0,10,15,20,22.5,30])
6. v=np.array([0,227.04,362.78,517.35,602.97,901.67])
7. # %% Plotting
8. plt.close('all')
9. plt.figure(1)
10. plt.plot(t,v,color='r',linewidth=3.5,linestyle=':')
11. plt.show()
```



If line 10 is instead changed to:

```
10. plt.plot(t,v,color='g',linewidth=3.5,linestyle='-.')
```



By default, no markers are shown for a line plot i.e. the keyword default for `marker=None`. This can be changed to:

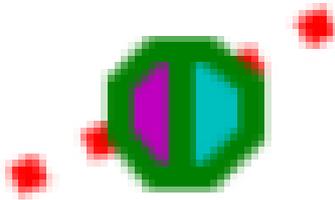
marker	Description
<code>None</code>	None (default)
<code>'.'</code>	point
<code>'/'</code>	pixel
<code>'o'</code>	circle
<code>'v'</code>	triangle_down
<code>'^'</code>	triangle_up
<code>'<'</code>	triangle_left
<code>'>'</code>	triangle_right
<code>'1'</code>	tri_down
<code>'2'</code>	tri_up
<code>'3'</code>	tri_left
<code>'4'</code>	tri_right
<code>'8'</code>	octagon
<code>'s'</code>	square
<code>'p'</code>	pentagon
<code>'P'</code>	plus
<code>'*'</code>	star

'h'	hexagon1
'H'	hexagon2
'+'	plus
'x'	x
'X'	X
'D'	diamond
'd'	thin_diamond
' '	vline
'_'	hline
0	tickleft
1	tickright
2	tickup
3	tickdown
4	caretleft
5	caretright
6	caretup
7	caretdown
8	caretleft
9	caretright
10	caretup
11	caretdown

The marker has additional keyword arguments such as `markersize` and `markeredgewidth` which take floats. One can also select the `fillstyle`.

<code>fillstyle</code>	
'full'	default
None	
'top'	has a <code>markerfacealt</code>
'bottom'	has a <code>markerfacealt</code>
'left'	has a <code>markerfacealt</code>
'right'	has a <code>markerfacealt</code>

There are three properties of the marker that can have color, these are the `markeredgecolor`, `markerfacecolor` and `markerfacecoloralt`. Let's look at an example:



```

color='r'
linewidth=3.5
linestyle=':'
marker='8'
markersize=15
markeredgewidth=3
fillstyle='left'
markerfacecolor='m' (left)
markerfacecoloralt='c' (right)
markeredgecolor='g'

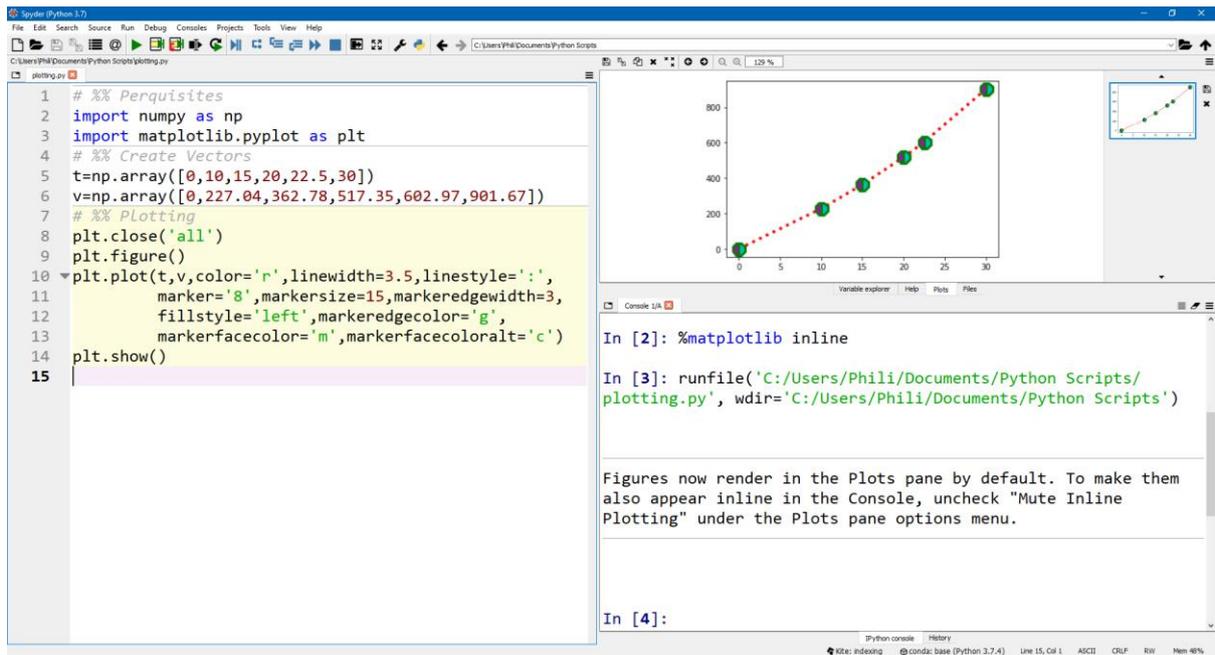
```

```

1. # %% Perquisites
2. import numpy as np
3. import matplotlib.pyplot as plt
4. # %% Create Vectors
5. t=np.array([0,10,15,20,22.5,30])
6. v=np.array([0,227.04,362.78,517.35,602.97,901.67])
7. # %% Plotting
8. plt.close('all')
9. plt.figure()
10. plt.plot(t, v, color='r', linewidth=3.5, linestyle=':',
11.          marker='8', markersize=15, markeredgewidth=3,
12.          fillstyle='left', markeredgecolor='g',
13.          markerfacecolor='m', markerfacecoloralt='c')
14. plt.show()

```

The two positional input arguments are presented first and then followed by the optional keyword arguments in line 10-13 can be listed in any order. These four lines could all be written in one very long line however this becomes hard to read so the parenthesis encasing the input arguments are split over multiple lines.

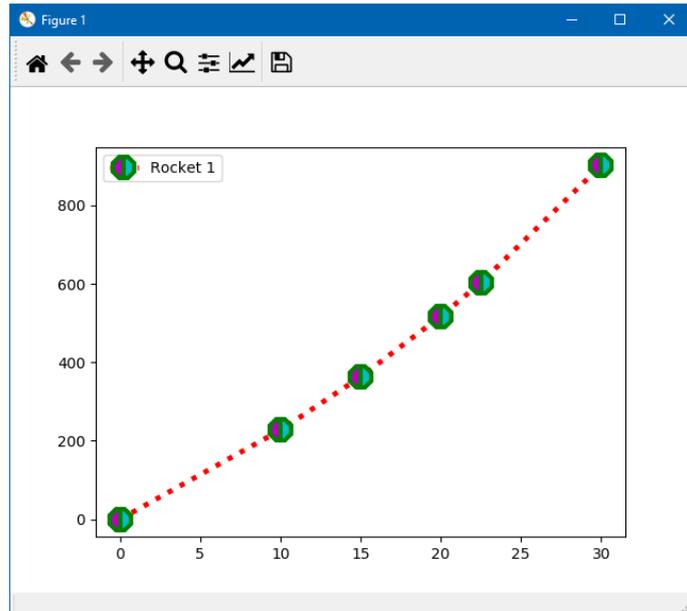


A label can also be added using a custom string and the plot can be updated to include a legend.

```

1. # %% Perquisites
2. import numpy as np
3. import matplotlib.pyplot as plt
4. # %% Create Vectors
5. t=np.array([0,10,15,20,22.5,30])
6. v=np.array([0,227.04,362.78,517.35,602.97,901.67])
7. # %% Plotting
8. plt.close('all')
9. plt.figure()
10. plt.plot(t,v,color='r',linewidth=3.5,linestyle=':',
11.          marker='8',markersize=15,markeredgewidth=3,
12.          fillstyle='left',markeredgecolor='g',
13.          markerfacecolor='m',markerfacecoloralt='c',
14.          label='Rocket 1')
15. plt.legend()
16. plt.show()

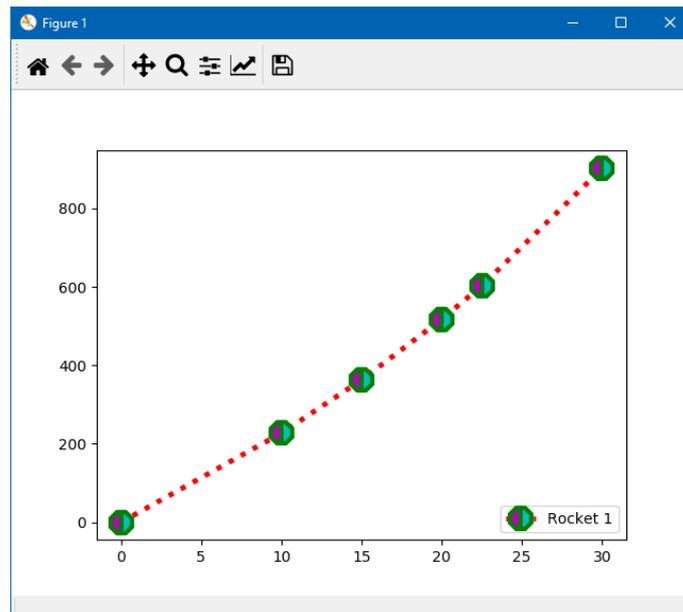
```



Input arguments can be used in line 15 such as `loc` to customise the location of the legend.

loc	loc
'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower centre'	8
'upper center'	9

If this is updated to `plt.legend(loc='lower right')`. The plot will look as follows

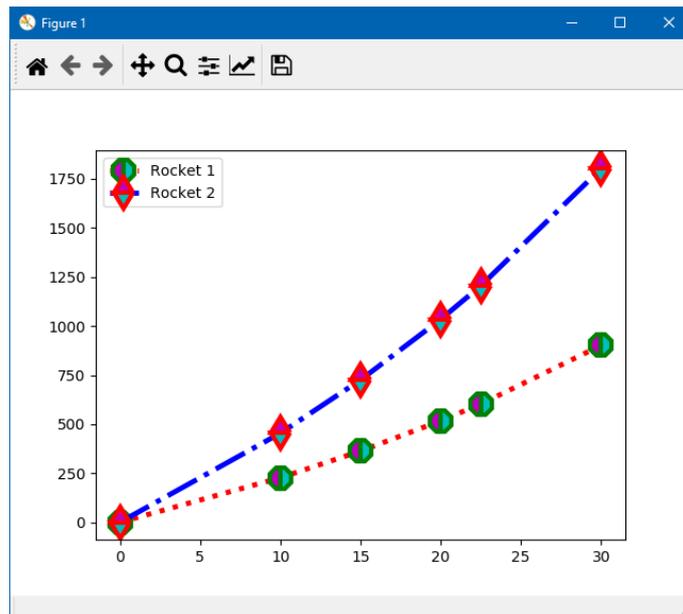


It is possible to add multiple lines to the plot for instance we can create a rocket which has velocity values $2v$.

```

1. # %% Perquisites
2. import numpy as np
3. import matplotlib.pyplot as plt
4. # %% Create Vectors
5. t=np.array([0,10,15,20,22.5,30])
6. v=np.array([0,227.04,362.78,517.35,602.97,901.67])
7. # %% Plotting
8. plt.close('all')
9. plt.figure()
10. plt.plot(t,v,color='r',linewidth=3.5,linestyle=':',
11.          marker='8',markersize=15,markeredgewidth=3,
12.          fillstyle='left',markeredgewidth=3,
13.          markerfacecolor='m',markerfacecoloralt='c',
14.          label='Rocket 1')
15. plt.plot(t,2*v,color='b',linewidth=3.5,linestyle='-.',
16.          marker='d',markersize=15,markeredgewidth=3,
17.          fillstyle='top',markeredgewidth=3,
18.          markerfacecolor='m',markerfacecoloralt='c',
19.          label='Rocket 2')
20. plt.legend(loc='upper left')
21. plt.show()

```

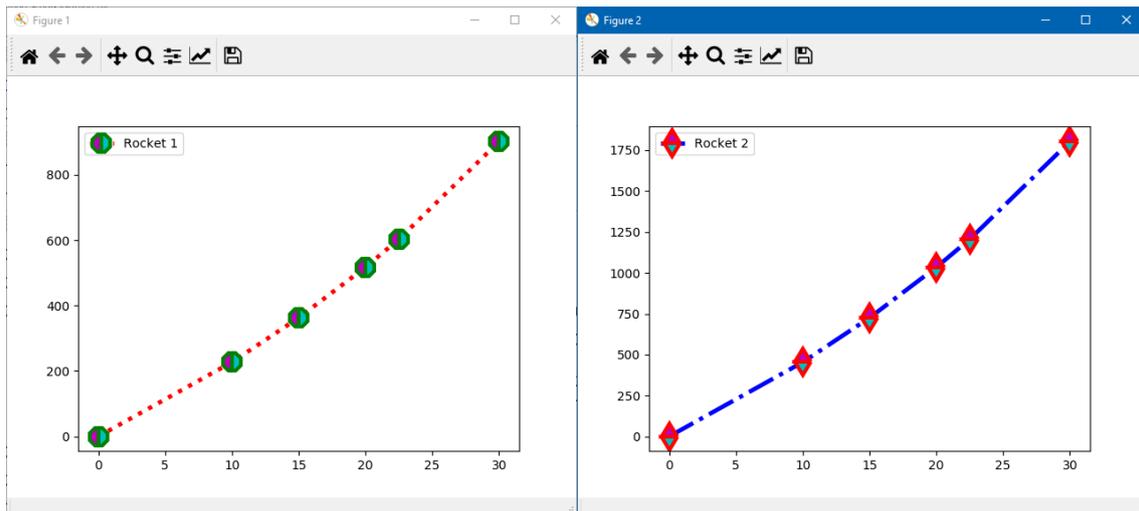


By default, plots will display on the same figure, unless a new figure is specified.

```

1. # %% Perquisites
2. import numpy as np
3. import matplotlib.pyplot as plt
4. # %% Create Vectors
5. t=np.array([0,10,15,20,22.5,30])
6. v=np.array([0,227.04,362.78,517.35,602.97,901.67])
7. # %% Plotting
8. plt.close('all')
9. plt.figure(1)
10. plt.plot(t,v,color='r',linewidth=3.5,linestyle=':',
11.         marker='8',markersize=15,markeredgewidth=3,
12.         fillstyle='left',markeredgewidth=3,
13.         markerfacecolor='m',markerfacecoloralt='c',
14.         label='Rocket 1')
15. plt.legend(loc='upper left')
16. plt.show()
17. plt.figure(2)
18. plt.plot(t,2*v,color='b',linewidth=3.5,linestyle='-.',
19.         marker='d',markersize=15,markeredgewidth=3,
20.         fillstyle='top',markeredgewidth=3,
21.         markerfacecolor='m',markerfacecoloralt='c',
22.         label='Rocket 2')
23. plt.legend(loc='upper left')
24. plt.show()

```



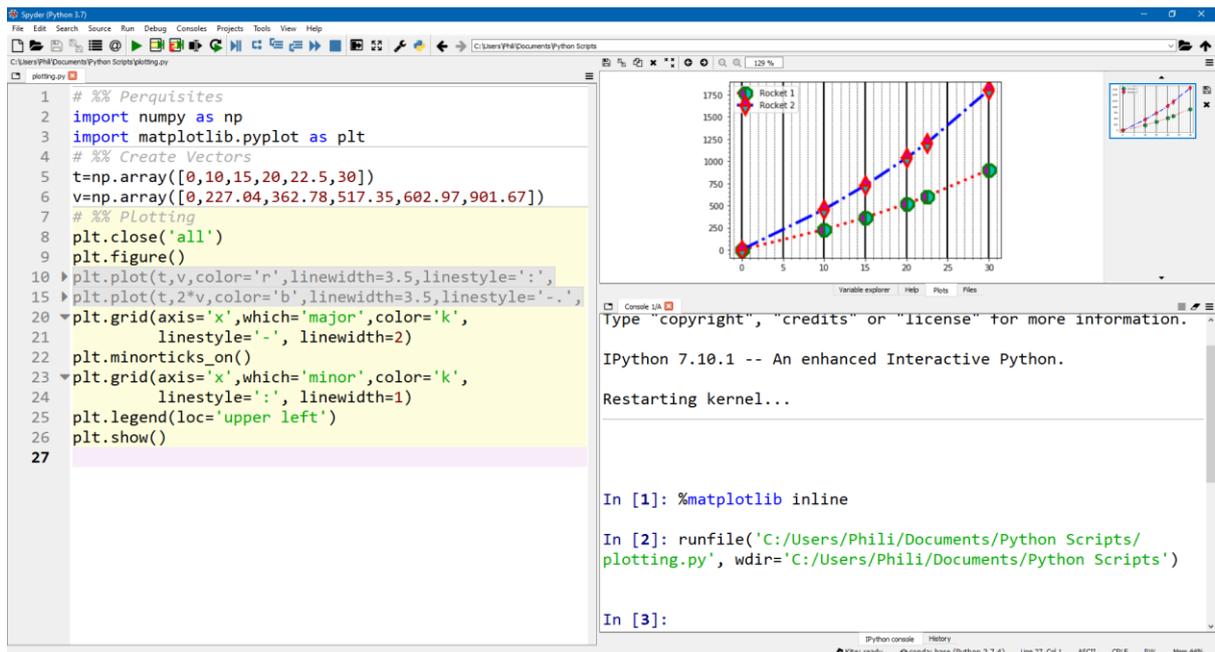
To turn grid lines on, the command `plt.grid()` needs to be used. It has multiple keyword input arguments such as `axis` which can be `'x'`, `'y'` or `'both'` and `which`, which can be `'major'`, `'minor'` or `'both'`. It also has the keyword input arguments `color`, `linestyle` and `linewidth` which act in identical manner to the same keyword arguments for `plt.plot()`. By default the minorticks aren't on, so these will need to be enabled by using `plt.minorticks_on()` which has no input arguments.

```

1. # %% Perquisites
2. import numpy as np
3. import matplotlib.pyplot as plt
4. # %% Create Vectors
5. t=np.array([0,10,15,20,22.5,30])
6. v=np.array([0,227.04,362.78,517.35,602.97,901.67])
7. # %% Plotting
8. plt.close('all')
9. plt.figure()
10. plt.plot(t,v,color='r',linewidth=3.5,linestyle=':',
11.          marker='8',markersize=15,markeredgewidth=3,
12.          fillstyle='left',markeredgecolor='g',
13.          markerfacecolor='m',markerfacecoloralt='c',
14.          label='Rocket 1')
15. plt.plot(t,2*v,color='b',linewidth=3.5,linestyle='-.',
16.          marker='d',markersize=15,markeredgewidth=3,
17.          fillstyle='top',markeredgecolor='r',
18.          markerfacecolor='m',markerfacecoloralt='c',
19.          label='Rocket 2')
20. plt.grid(axis='x',which='major',color='k',
21.          linestyle='-', linewidth=2)
22. plt.minorticks_on()
23. plt.grid(axis='x',which='minor',color='k',
24.          linestyle=':', linewidth=1)
25. plt.legend(loc='upper left')
26. plt.show()

```

The plot commands have been collapsed in the screenshot below.



The axes and title can be labelled using `plt.xlabel()`, `plt.ylabel()` and `plt.title()` with custom strings as input arguments.

```

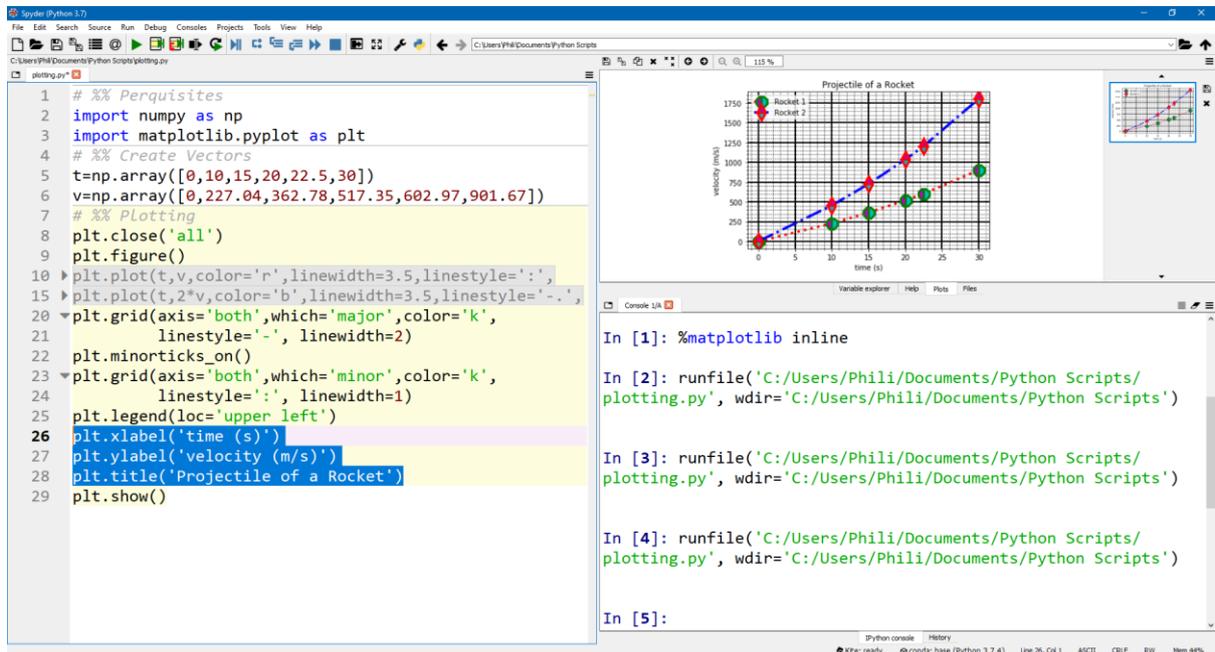
1. # %% Perquisites
2. import numpy as np
3. import matplotlib.pyplot as plt
4. # %% Create Vectors
5. t=np.array([0,10,15,20,22.5,30])
6. v=np.array([0,227.04,362.78,517.35,602.97,901.67])
7. # %% Plotting
8. plt.close('all')
9. plt.figure()
10. plt.plot(t,v,color='r',linewidth=3.5,linestyle=':',
11.          marker='8',markersize=15,markeredgewidth=3,
12.          fillstyle='left',markeredgecolor='g',
13.          markerfacecolor='m',markerfacecoloralt='c',
14.          label='Rocket 1')
15. plt.plot(t,2*v,color='b',linewidth=3.5,linestyle='-.',
16.          marker='d',markersize=15,markeredgewidth=3,
17.          fillstyle='top',markeredgecolor='r',
18.          markerfacecolor='m',markerfacecoloralt='c',
19.          label='Rocket 2')
20. plt.grid(axis='x',which='major',color='k',
21.          linestyle='-', linewidth=2)
22. plt.minorticks_on()
23. plt.grid(axis='x',which='minor',color='k',
24.          linestyle=':', linewidth=1)
25. plt.legend(loc='upper left')
26. plt.xlabel('time (s)')

```

```

27. plt.ylabel('velocity (m/s)')
28. plt.title('Projectile of a Rocket')
29. plt.show()

```

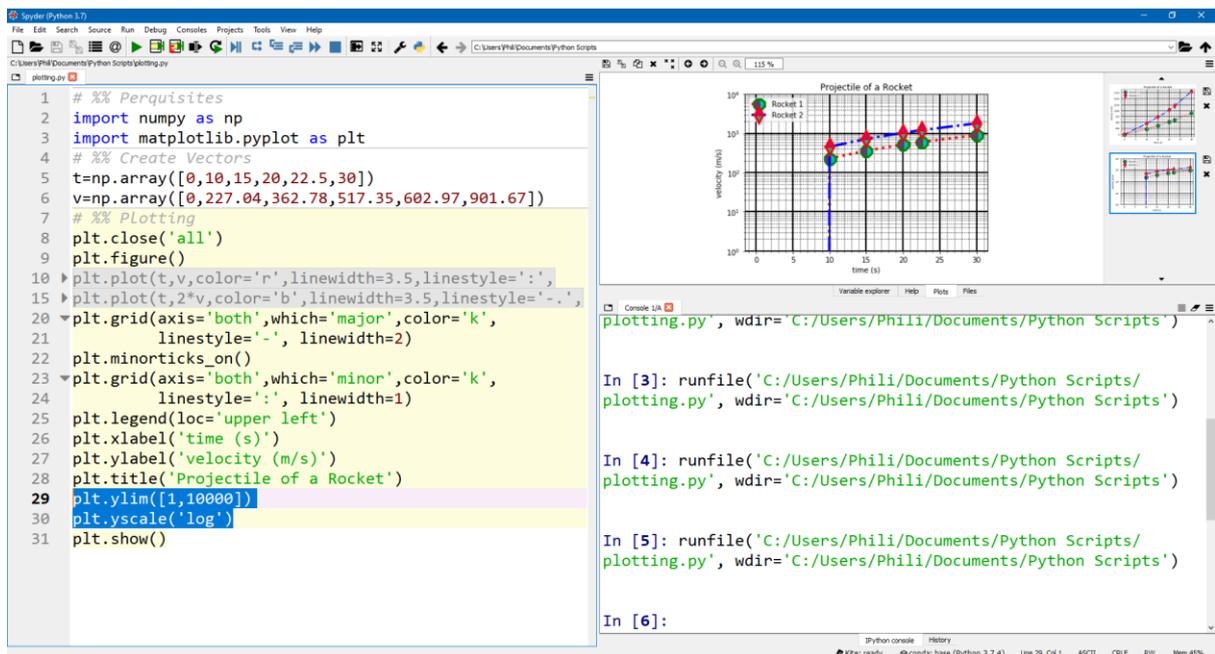


Axes limits can be set using the functions `plt.xlim()` and `plt.ylim()` respectively. These have a list with 2 elements as an input argument consisting of a lower and upper bound. The axes scales can be set using the functions `plt.xscale()` and `plt.yscale()` with string input arguments of 'linear' or 'log'. For example the following lines of code can be added:

```

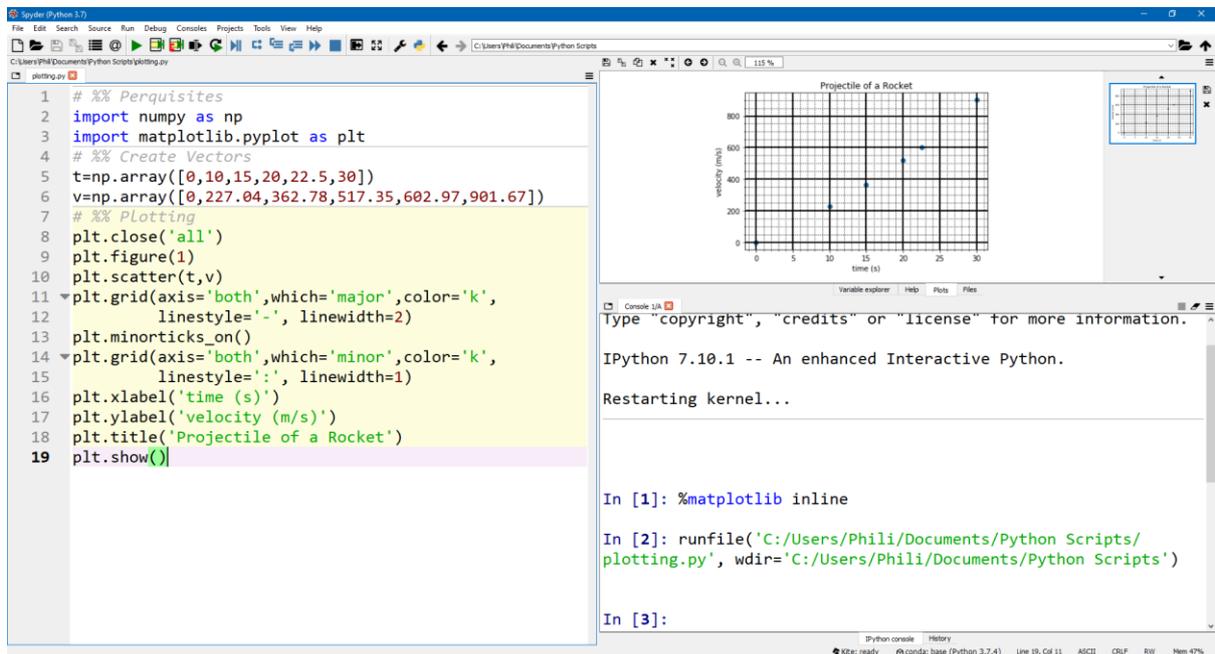
29. plt.ylim([1,10000])
30. plt.yscale('log')

```



Scatter Plot

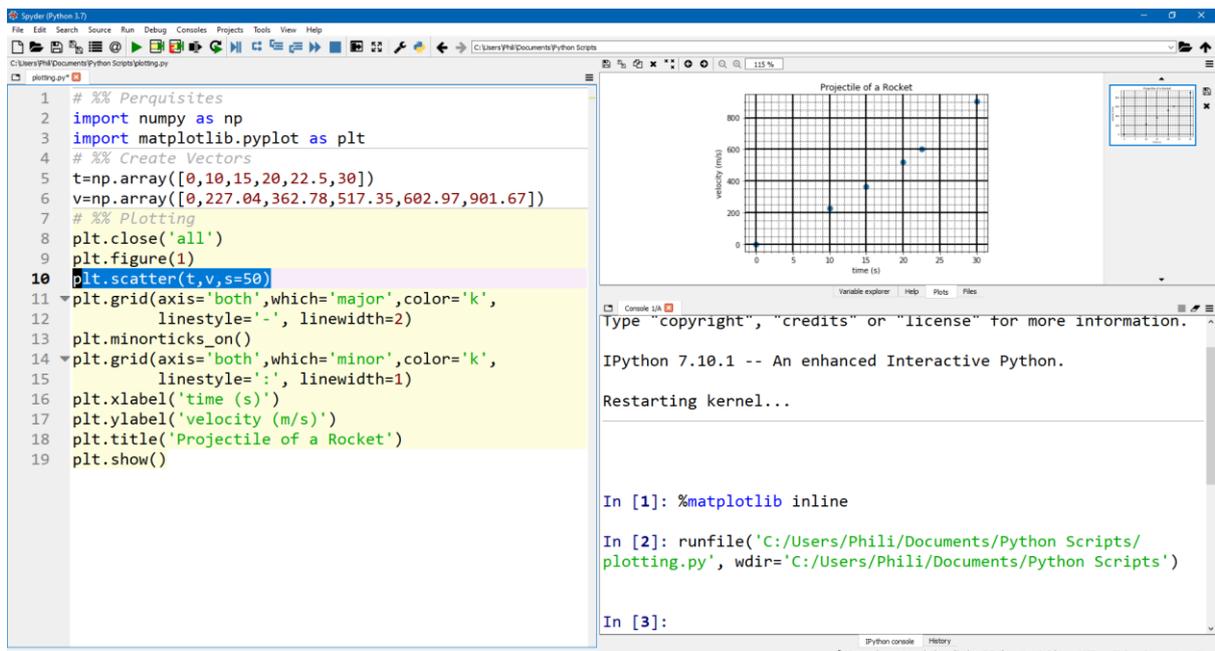
The scatter plot function `plt.scatter()` is very similar to the line plot function `plot()` sharing the two positional input arguments for the x and y data.



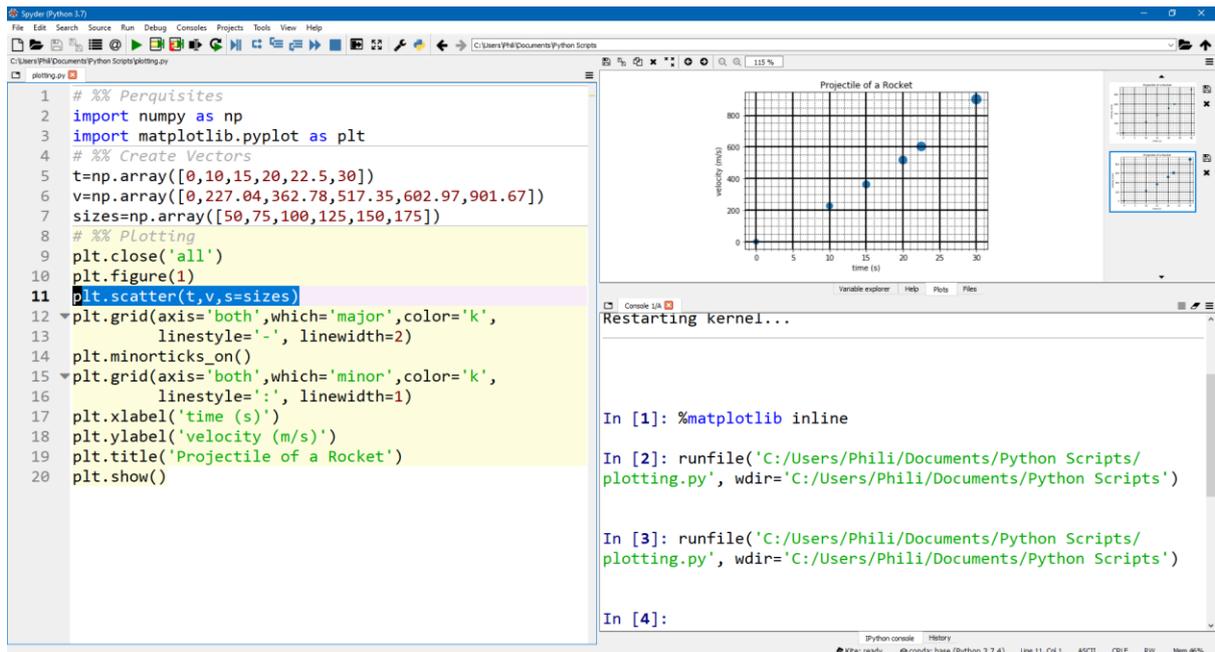
Line 10 is modified to `plt.scatter(t, v)` and all other lines were used for the line plot.

It has a positional argument `s` denoting the size of the data points. If a scalar is input, all data points will be made the same size, however if a vector is input, each point will have the size specified in the vector.

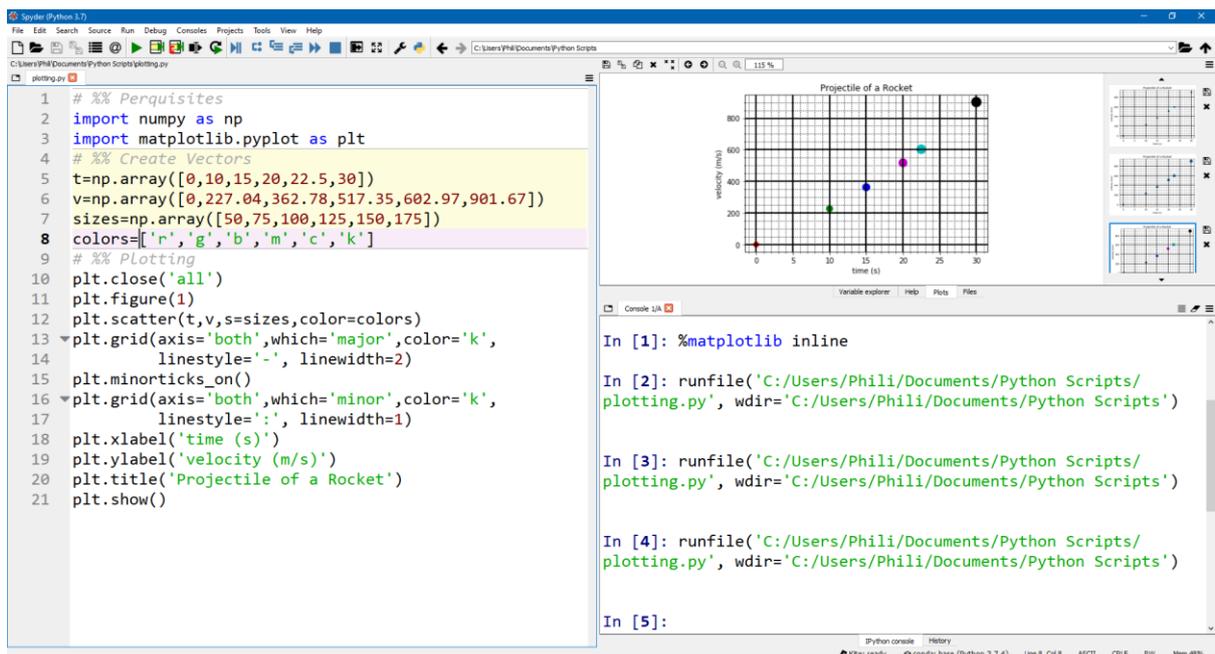
Line 10 is modified to `plt.scatter(t, v, s=50)`



The vector `sizes=np.array([50, 75, 100, 125, 150, 175])` was added in line 7 and line 11 is modified to `plt.scatter(t, v, s=sizes)`



For a scatter plot, the keyboard input argument `color` can also accept a single value or a list. Line 8 was modified to create the list of strings `colors=['r','g','b','m','c','k']` and line 12 was updated to `plt.scatter(t,v,s=sizes,color=colors)`.



It is also possible to specify a `marker` (which recognises the same strings and numeric values as the keyword `marker` in `plot()`). It is also possible to select the `linewidth` of the marker as a single float or vector of floats and individually select the `edgecolor` and `facecolor` use a single string if every point is to have the same color or using vectors if each point is to be individually specified.

```

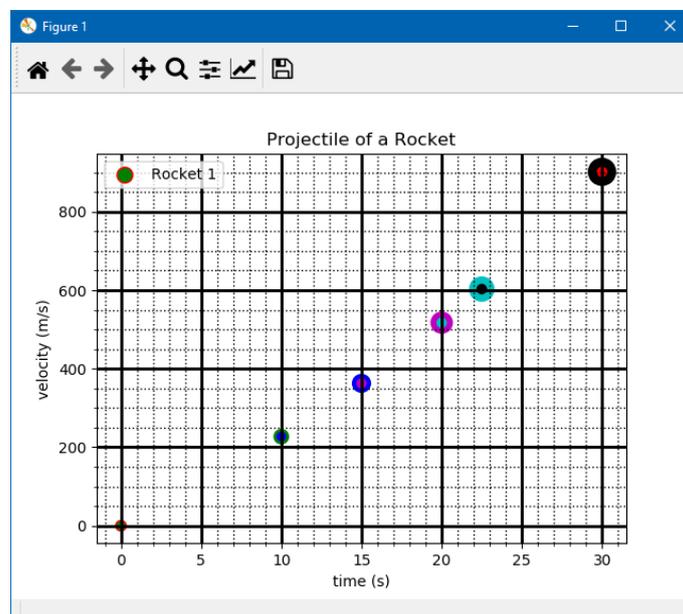
1. # %% Perquisites
2. import numpy as np
3. import matplotlib.pyplot as plt
4. # %% Create Vectors

```

```

5. t=np.array([0,10,15,20,22.5,30])
6. v=np.array([0,227.04,362.78,517.35,602.97,901.67])
7. sizes=np.array([50,75,100,125,150,175])
8. linewidths=[1,2,3,4,5,6]
9. edgecolors=['r','g','b','m','c','k']
10. facecolors=['g','b','m','c','k','r']
11. # %% Plotting
12. plt.close('all')
13. plt.figure(1)
14. plt.scatter(t,v,s=sizes,linewidth=linewidths,
15.             marker='o',
16.             edgecolor=edgecolors,
17.             facecolor=facecolors,
18.             label='Rocket 1')
19. plt.grid(axis='both',which='major',color='k',
20.          linestyle='-', linewidth=2)
21. plt.minorticks_on()
22. plt.grid(axis='both',which='minor',color='k',
23.          linestyle=':', linewidth=1)
24. plt.legend(loc='upper left')
25. plt.xlabel('time (s)')
26. plt.ylabel('velocity (m/s)')
27. plt.title('Projectile of a Rocket')

```



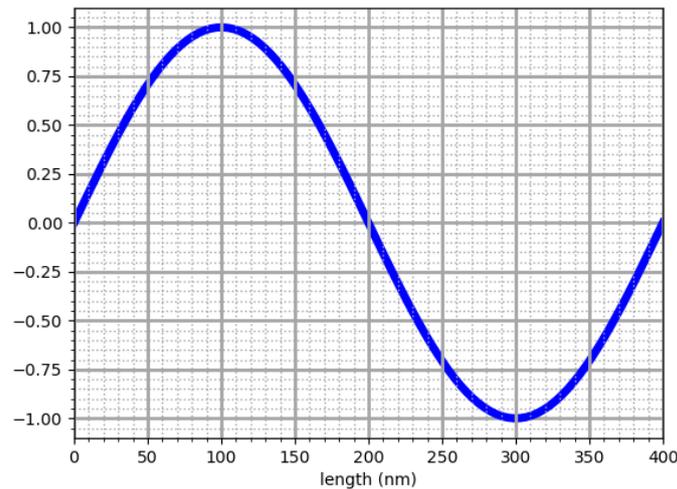
How we Encode Color

Spelling of Color

There are two different spellings to the word color. Color is actually the original spelling and was formerly used in the UK post-1776 but was updated in the UK to incorporate a u as colour, probably around 1824, when a number of spellings, weights and measures were standardised. As a result of

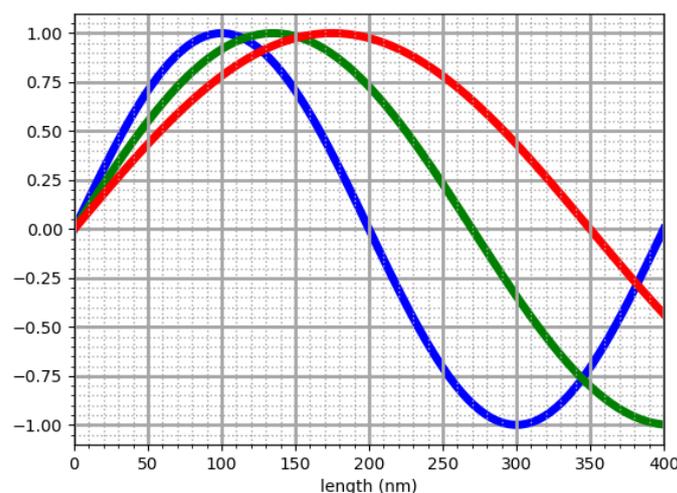
declaring independence, the USA did not collaborate with Britain and her Commonwealth and refused to standardize with the UK and instead kept the old spellings, weights and measures, resulting in a subtle variation between UK and Commonwealth English spellings with USA English spellings. In Python, US spellings are used, meaning us poor Brits will be continually frustrated when our code throws up warnings due to differences in spellings.

Theory of Light

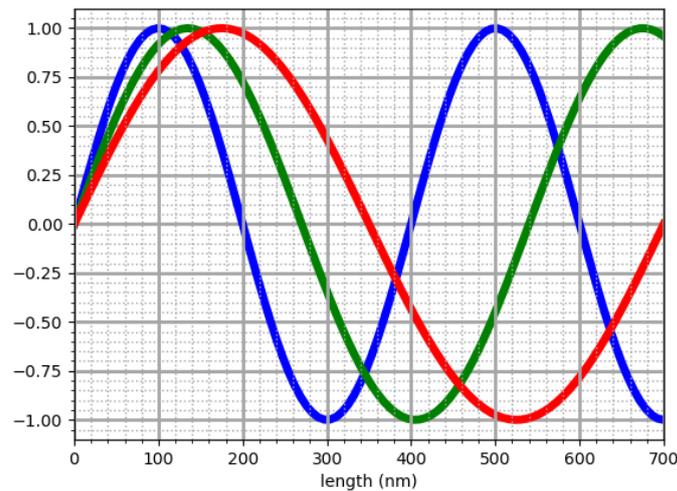


Electromagnetic radiation is known to exhibit both wave and particle like properties. Looking at light as a wave, the wavelength is defined as the time it takes for the wave to repeat itself and ranges from 10^{-16} to 10^6 m. Blue light for instance has the wavelength of 400×10^{-9} m or $0.4 \mu\text{m}$ or 400 nm (which is commonly used to describe light).

Green and red light have a wavelength of 525 nm and 700 nm respectively. These can be shown plotted on the period of 1 wavelength of blue light.

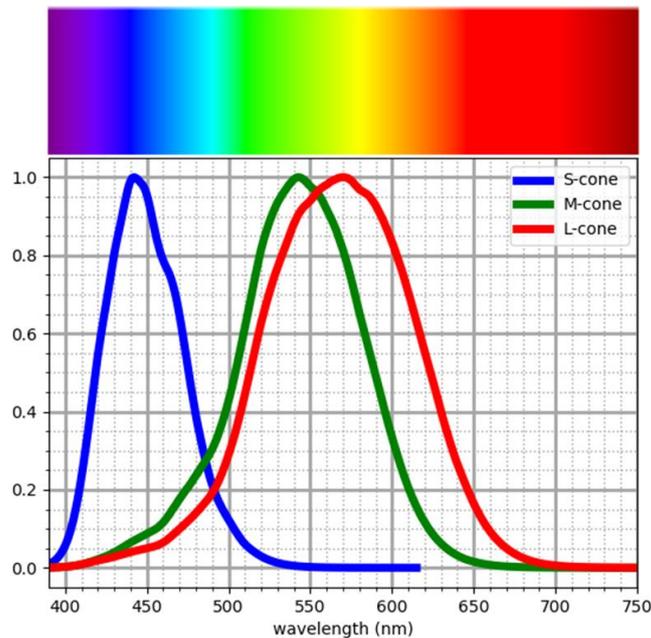


Or alternatively plotted on the period of 1 wavelength of red light. Note how the blue wave goes up and down almost 1.75 times as much as the red wave, therefore it carries more energy. Think of carrying a set of weights as analogous, the more frequently you lift your set of weights up and down, the more energy you will use up.



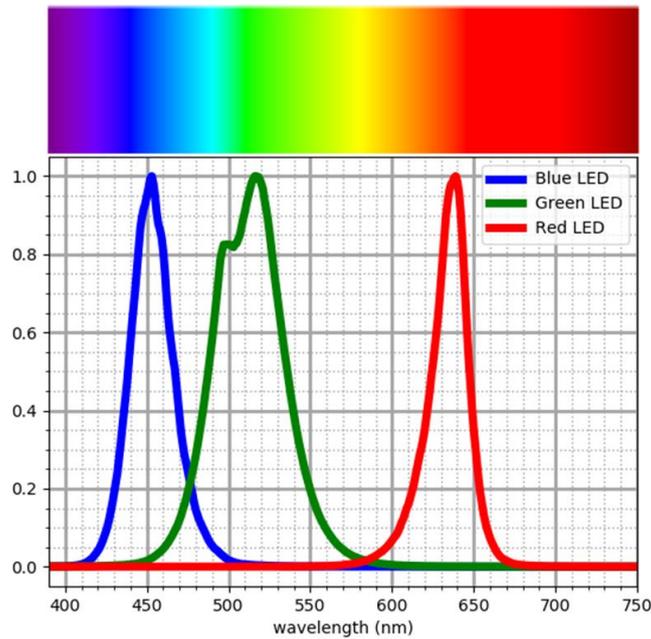
Additive Addition of Light

Visible light is defined as electromagnetic radiation that can be detected by the human eye and the human eye can only see within a very small window which is $0.39\text{-}0.75 \times 10^{-6}$ m out of the much larger 10^{-16} to 10^6 m scope. The human eye has three types of detectors, S-cones, M-cones and L-cones for Short, Medium and Long wavelength detection respectively. The average response from the three detectors in a human eye is as follows and this is of course the origin of the three “primary colors”.

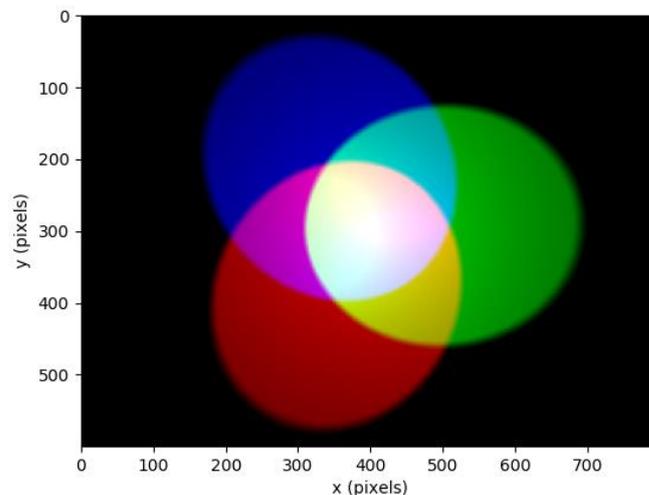


When we think of light as particles, we call the “particles of light” photons. The eye has three types of detectors known as S-cones, M-cones and L-cones which give the physiological reason behind the three primary colors. A single photon of light can be detected at 400 nm or 525 nm by an S-cone and the eye won’t know the difference between these two photons. However, for a large enough distribution of photons the brain will look at the ratio collected at the S-cones, M-cones and L-cones respectively and assign this intensity ratio to a "color". For instance, light at 400 nm has a high probability of detection at the S-cones and a comparatively low ratio of detection at the M-cones and L-cones and hence appears to the brain as the color "blue". On the other hand, light at 700 nm will give a low ratio of detection at the S-cones and M-cones and a high ratio of detection at the L-cones

and so appear the color "red". Light at 500 nm will have a small ratio at the S-cones, have a high ratio of detection at the M-cones and a medium ratio of detection at the L-cones and appear to the brain as the color "green".



Three LEDs may be selected that are blue, green and red respectively, typical emission spectra of these LEDs are shown. These LEDs can be designed so that their output overlaps spatially.



If we examine the overlapping regime between the red and the green LEDs, both the M-cones and L-cones of the eye will detect light while the S-cones won't. Using this ratio, the brain translates this as the color "yellow". However, it is important to note however that no yellow light is actually generated, and this is seen if one looks at the spectra of the LEDs. Using the spectra above when both the green and red LEDs are on, there is no output at 575 nm, there is actually a dip in the emission here.

[r,g,b] integers or [r,g,b,a] floats

The image above shows the effect of the three LEDs when they are each binary, on or off. LEDs are usually set to have a variable brightness. The convention is to use 8 Bit encoding to give $2^8=256$ levels

which is 0-255 (using zero order indexing). With this convention it is possible to select $256^3=16,581,375$ `[r, g, b]`, individual colors which satisfies all the different color ratios the brain can interpret. This means that black, which as seen above is the absence of color is encoded as `[0, 0, 0]`. The primary colors are when only one of the LEDs is on at full brightness; red `[255, 0, 0]`, green `[0, 255, 0]`, and blue `[0, 0, 255]`. The secondary colors are when two of the LEDs are on at full brightness cyan `[0, 255, 255]`, magenta `[255, 0, 255]`, yellow `[255, 255, 0]`. When the three LEDs are on at full brightness we get white `[255, 255, 255]`.

Some of the Python libraries such as the Matplotlib plotting library will normalise these `[r, g, b]`, values as floats i.e. divide through by 255 and also add a fourth channel known as alpha to indicate transparency `[r, g, b, a]` where `a=0` is totally transparent (invisible) and `a=1` is non-transparent. This means that black, which as seen above is the absence of color is encoded as `[0, 0, 0, 1]`. The primary colors are when only one of the LEDs is on at full brightness; red `[1, 0, 0, 1]`, green `[0, 1, 0, 1]`, and blue `[0, 0, 1, 1]`. The secondary colors are when two of the LEDs are on at full brightness cyan `[0, 1, 1, 1]`, magenta `[1, 0, 1, 1]`, yellow `[1, 1, 0, 1]`. When the three LEDs are on at full brightness we get white `[1, 1, 1, 1]`.

#rrggbb Hex system

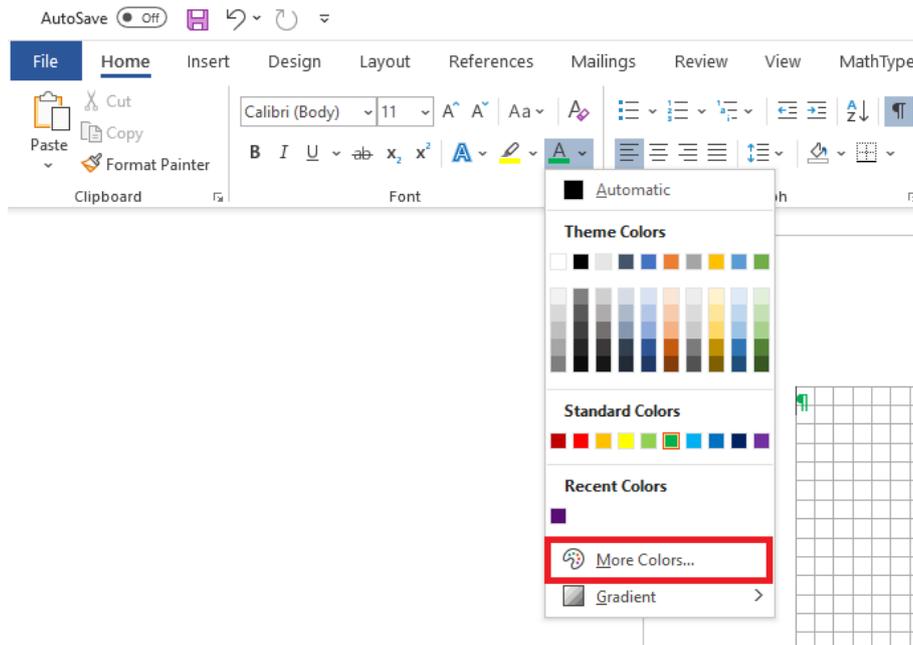
Another encoding mechanism for color is the hexadecimal system (16 characters). The 16 characters are the 10 numeric digits 0,1,2,3,4,5,6,7,8,9 and when we run out of numeric digits, we take the first 6 letters of the alphabet a,b,c,d,e,f.

In order to get to the decimal value 255, 2 digits are required for each color. This is similar to normal counting when we only have 10 characters. When we get to the 11 value, we have used all 10 single characters so therefore need to use two characters, in this case `11`. In Hex, `0f=015=15` in decimal and `1f=115=16` in decimal.

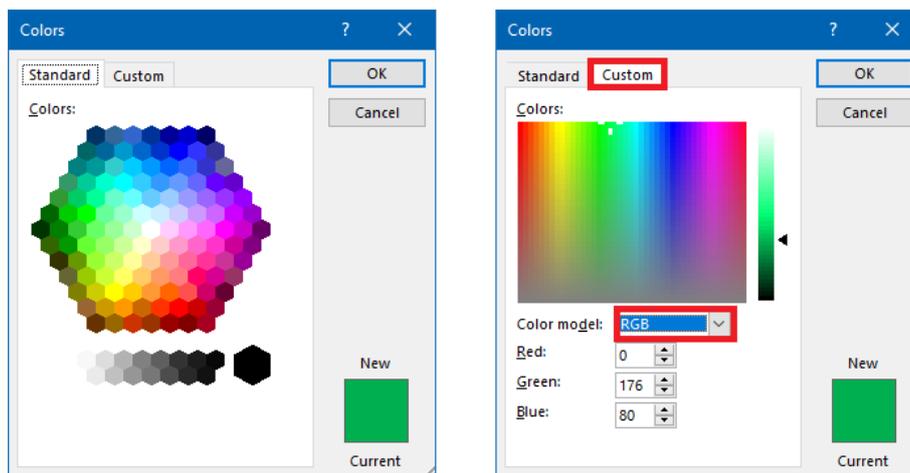
In hex form we start with a `#` and then have 2 digits for red, 2 digits for green and 2 digits for blue i.e. have the form `#rrggbb`. For instance, black is `#000000`. The primary colours are red `#ff0000`, green `#00ff00`, blue `#0000ff`. The secondary colors are when two of the LEDs are on at full brightness cyan `#00ffff`, magenta `#ff00ff`, and yellow `#ffff00`. When the three LEDs are on at full brightness, we get white `#ffffff`. Hex values are usually input as strings so enclosed in single quotation marks.

Microsoft Office Standard Colors

Many programs have a color picking for example in Microsoft Word if one selects more colors...



Then selects the Custom Tab, a [r,g,b] color model displays.



In this case, the "standard color" that Microsoft calls "green" has the values:

```
ms_green=[0/255,176/255,80/255,1]
```

There is much debate about the precise naming of colors, so we will just call its variable name `ms_green`. The [r,g,b,a] float of this color is hard to remember however we can create a dictionary called `colors` where we specify the [r,g,b,a] of the line of standard colors above.

```

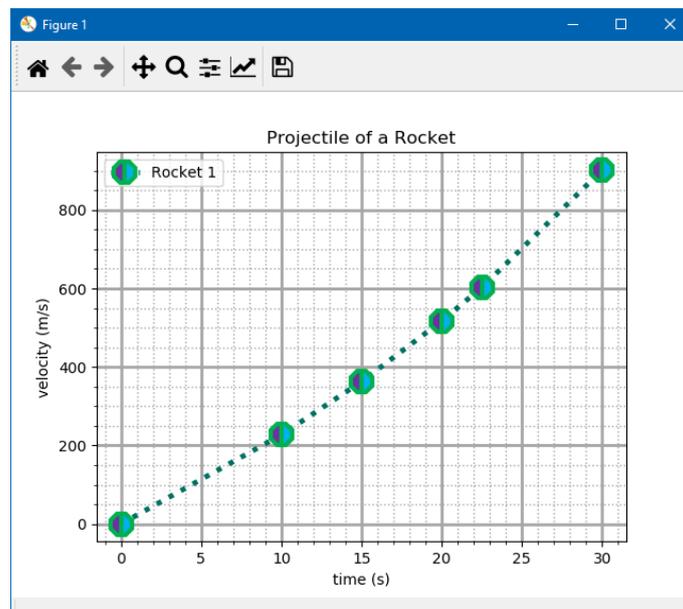
1. # %% Perquisites
2. import numpy as np
3. import matplotlib.pyplot as plt
4. # %% Dictionary
5. colors={'darkred': [192/255,0/255,0/255,1],
6.         'red': [255/255,0/255,0/255,1],
7.         'orange': [255/255,192/255,0/255,1],
8.         'yellow': [255/255,255/255,0/255,1],
9.         'lightgreen': [146/255,208/255,80,1],

```

```

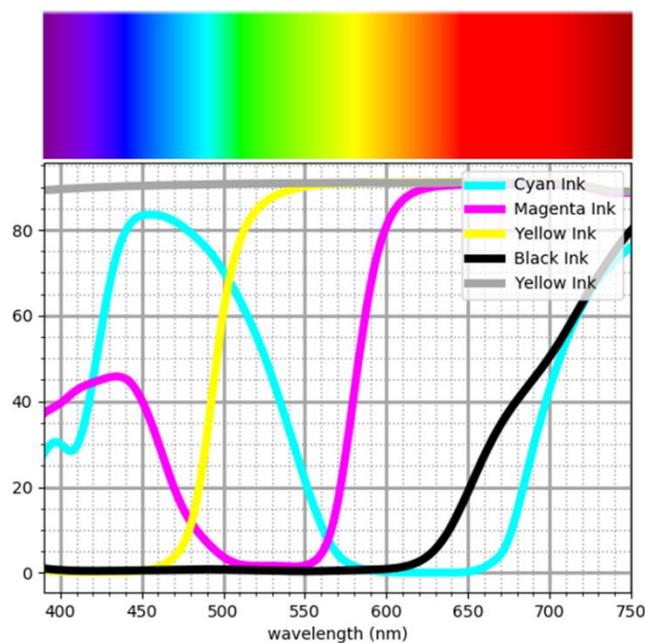
10.     'darkgreen': [0/255,176/255,80/255,1] ,
11.     'lightblue': [0/255,176/255,240/255,1] ,
12.     'blue': [0/255,112/255,96/255,1] ,
13.     'darkblue': [0/255,32/255,96/255,1] ,
14.     'purple': [112/255,48/255,160/255,1] ,
15.     'lightgrey': [167/255,167/255,167/255,1]}
16. # %% Create Vectors
17. t=np.array([0,10,15,20,22.5,30])
18. v=np.array([0,227.04,362.78,517.35,602.97,901.67])
19. # %% Plotting
20. plt.close('all')
21. plt.figure()
22. plt.plot(t,v,color=colors['blue'],linewidth=3.5,
23.         linestyle=':',marker='8',markersize=15,
24.         markeredgewidth=3,fillstyle='left',
25.         markeredgewidth=3,fillstyle='left',
26.         markeredgewidth=3,fillstyle='left',
27.         markeredgewidth=3,fillstyle='left',
28.         markeredgewidth=3,fillstyle='left',
29.         label='Rocket 1')
30. plt.grid(axis='both',which='major',
31.         color=colors['lightgrey'],
32.         linestyle='-', linewidth=2)
33. plt.minorticks_on()
34. plt.grid(axis='both',which='minor',
35.         color=colors['lightgrey'],
36.         linestyle=':', linewidth=1)
37. plt.legend(loc='upper left')
38. plt.xlabel('time (s)')
39. plt.ylabel('velocity (m/s)')
40. plt.title('Projectile of a Rocket')
41. plt.show()

```



Subtractive Addition of Light

In print media, subtractive addition of light is used. In subtractive addition room light is not generated but rather it is subtracted from room light. Room light is incident on a reflective surface such as a sheet piece of paper. This piece of paper appears to be white as all incident light is reflected. Inks are used to colour the paper, first of all there is cyan dye which has chemicals which interacts and removes light in the red while reflecting blue and green light giving the cyan appearance. This can be thought of as $\text{white} - \text{red} = \text{green} + \text{blue} = \text{cyan}$. Then there is the magenta dye which removes light in the green and passes light in the red and blue. This can be thought of as $\text{white} - \text{green} = \text{red} + \text{blue} = \text{magenta}$. Then there is the yellow dye which removes the blue light and reflects the green and red light. This can be thought of as $\text{white} - \text{blue} = \text{green} + \text{red} = \text{yellow}$. Finally as these three inks do not act as perfect “primary color removers”, there is usually a black ink which removes all the white light and the absence of light appears black. Once again by fine tuning the levels and concentration of dyes, any color can be produced although in practice is more limited as there is variation in room light and day light from place to place and time to time. Subtractive addition is the basis of the arts.



Matrices

Practical Applications of Matrices

Matrices are often taught to be quite abstract objects in high school mathematics and very few practical applications are given even though most of you interact with several matrices daily.

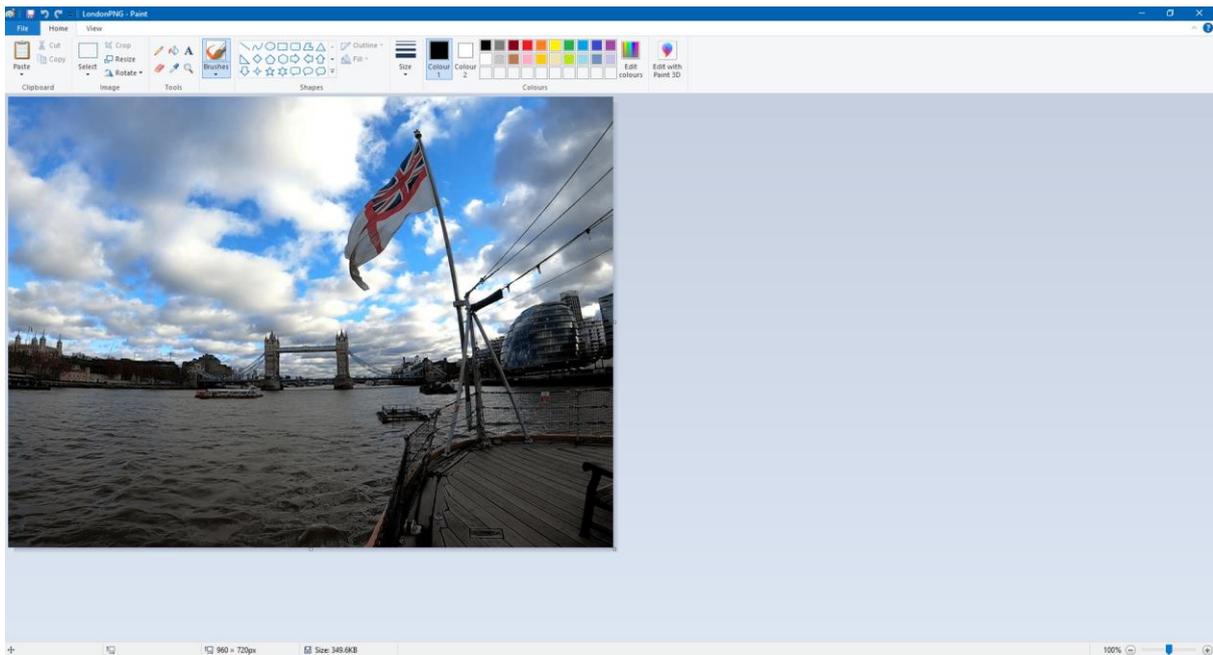
We have discussed the creation of light of any color using three LEDs already. Routinely sets of three primary colored LEDs are miniaturised into electronics to make a pixel and a series of pixels are constructed to form a 2D array or matrix. These can be considered the basis behind the screen on a laptop, phone and a camera. Here are three objects, I interact with daily and I am sure you can relate to similar objects and indeed are probably using one to read this document.



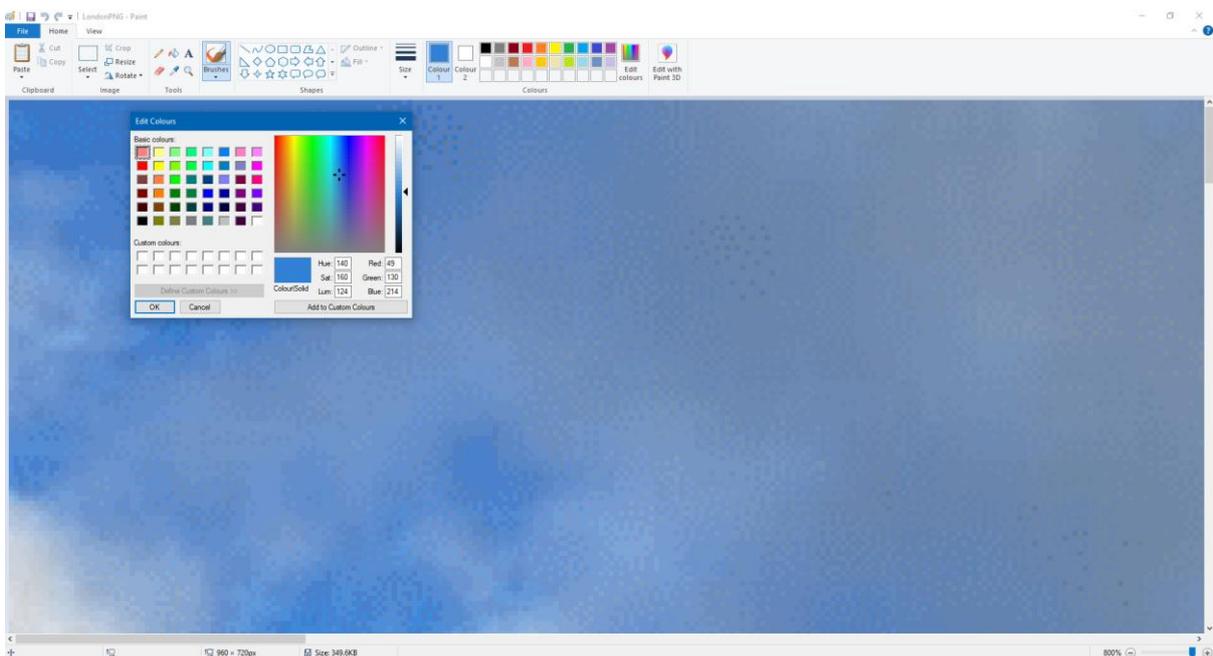
Screen resolution is usually defined in terms of pixels, in the case of the Dell XPS 13 this is 3200×1600 pixels which means 1600 rows and 3200 columns. A graphical computer program on this XPS 13 9365 for instance will send a command to the computer to process a 1800 row by 3200 column matrix of red values, a 1800 row by 3200 column matrix of green values, a 1800 row by 3200 column matrix of blue values and if it is running at 60 frames per second will do this every 0.017 s.



It can be opened in the most common photo editing software, Microsoft Paint.



If zoomed into maximum, we can see the image appears to be pixelated. We can open the color picture and select a pixel in the upper left corner. This gives the red, green and blue value for the selected pixel in the unsigned integer 8. In other words, this image is just an overlaid red, green and blue matrices of numbers.



A printer also treats a piece of paper as a matrix that it applies a series of dots to. In the past a series of printers that printed in black and white only were appropriately called dot matrix printer. This name described perfectly the operation of the printer; in that they prescribed a series of equally spaced black dots across two dimensions on a piece of white paper. Each dot of ink is applied to a pixel on the paper and can be thought of as a numerical quantity of ink ranging from 0 where no ink is applied to 255 where the pixel is completely covered by ink. Nowadays printer specifications are given as dots per inch. This HP has a resolution of 600 dots per inch by 600 dots. If we take the paper size to be A4

which is 11.69 inches by 8.27 inches we can calculate the piece of paper (assuming no margins) to be treated by the printer as 7014 rows by 4962 columns.

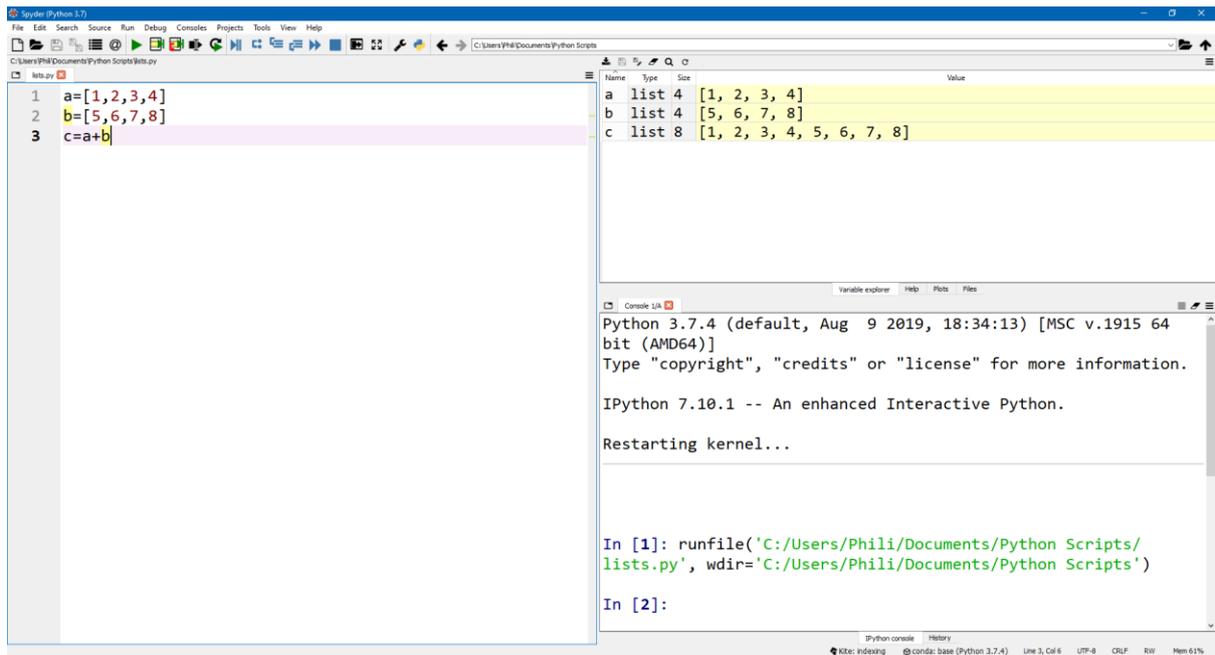


Before looking at such complicated objects it is worthwhile revising high school matrix operations and how to carry out such operations using Python.

Lists vs NumPy Arrays

So far we have looked at list creation using core Python.

1. `a=[1, 2, 3, 4]`
2. `b=[5, 6, 7, 8]`
3. `c=a+b`



Note that when we perform an operation using the `+` operator on a list that we instead concatenate the list after the `+` operator to the end of the list before the `+` operator. This is known as concatenation.

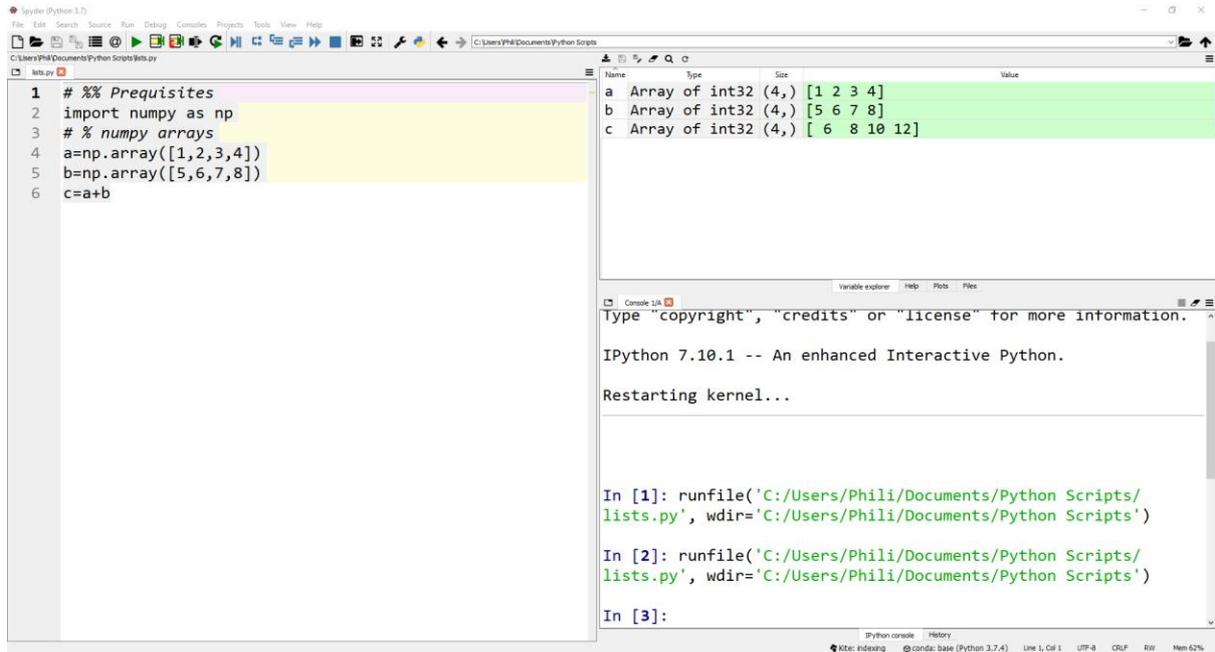
$$\begin{aligned}
 a &= [1,2,3,4] \\
 b &= [5,6,7,8] \\
 c &= a + b \\
 c &= [1,2,3,4,5,6,7,8]
 \end{aligned}$$

NumPy arrays will act slightly differently and the `+` operator will instead perform an addition.

```

1. # %% Prerequisites
2. import numpy as np
3. # % numpy arrays
4. a=np.array([1,2,3,4])
5. b=np.array([5,6,7,8])
6. c=a+b

```



Note for the addition to take place the dimensions of both lists must match as the $+$ operation occurs element by element.

$$a = [1, 2, 3, 4]$$

$$b = [5, 6, 7, 8]$$

$$c = a + b = [1, 2, 3, 4] + [5, 6, 7, 8]$$

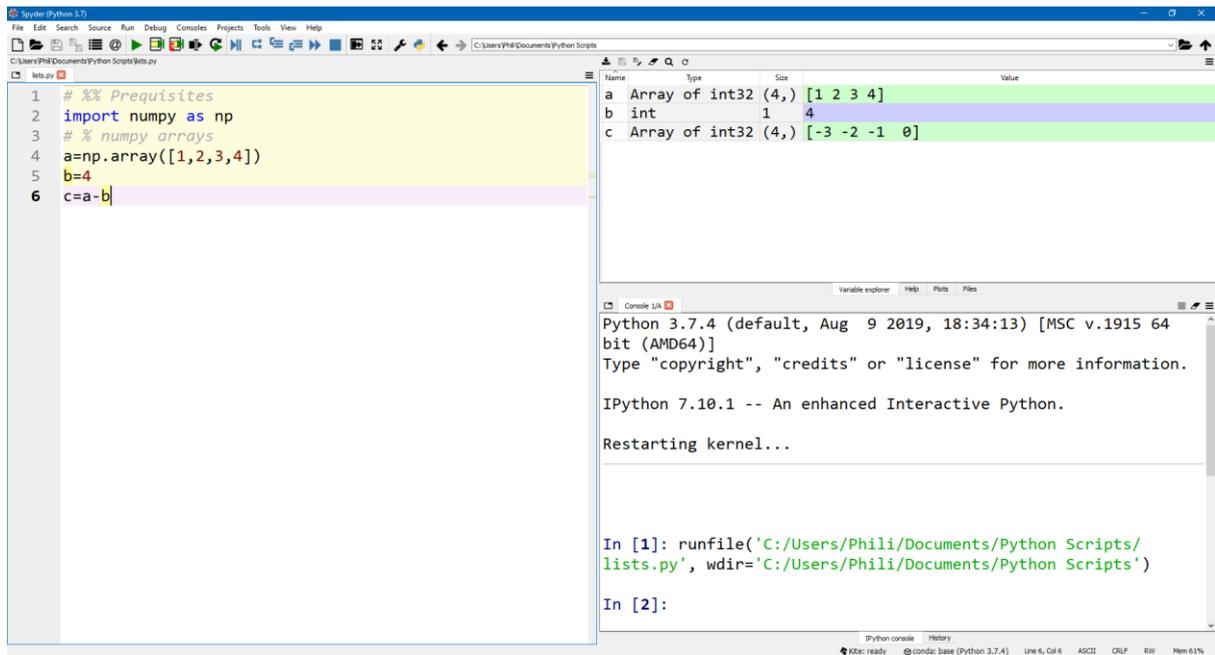
$$c = [1+5, 2+6, 3+7, 4+8] = [6, 8, 10, 12]$$

The only case when the dimensions can differ is if one of the values is a scalar as it will be explicitly expanded. For instance:

```

1. # %% Prerequisites
2. import numpy as np
3. # % numpy arrays
4. a=np.array([1, 2, 3, 4])
5. b=4
6. c=a-b

```



$$a = [1, 2, 3, 4]$$

$$b = 4$$

$$c = a + b = [1, 2, 3, 4] - 4 = [1, 2, 3, 4] - [4, 4, 4, 4]$$

$$c = [1 - 4, 2 - 4, 3 - 4, 4 - 4] = [-3, -2, -1, 0]$$

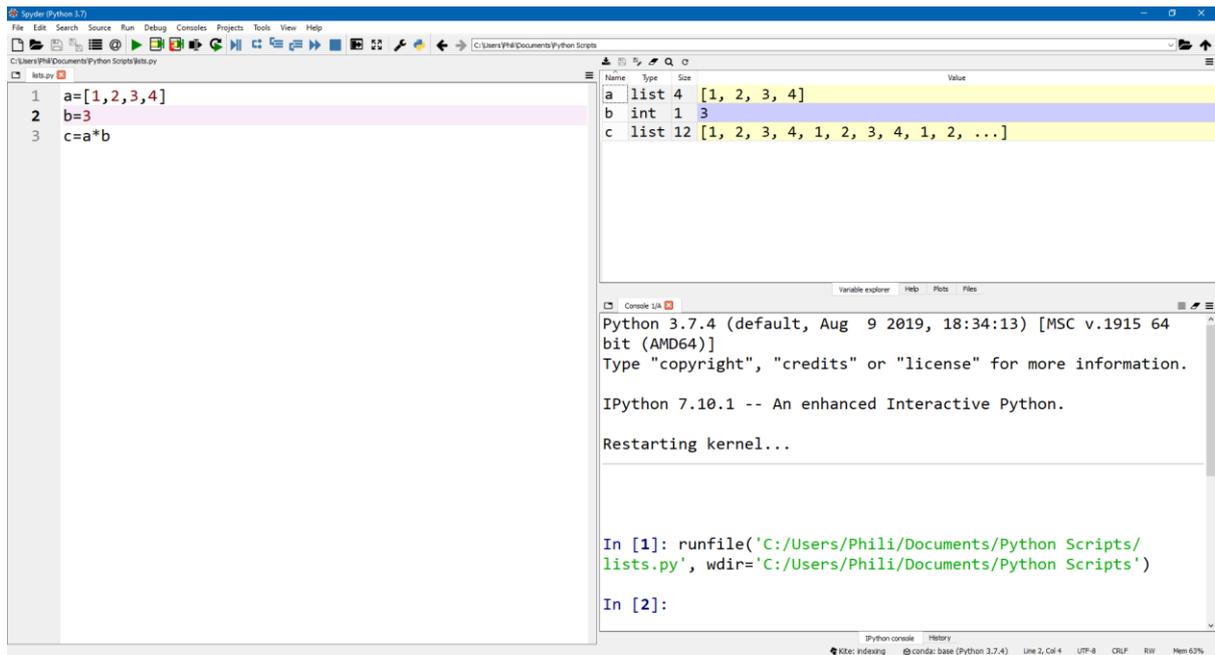
For a list the multiplication operator `*` will only work with a scalar integer and concatenate the list by multiple copies of itself.

1. `a = [1, 2, 3, 4]`
2. `b = 3`
3. `c = a * b`

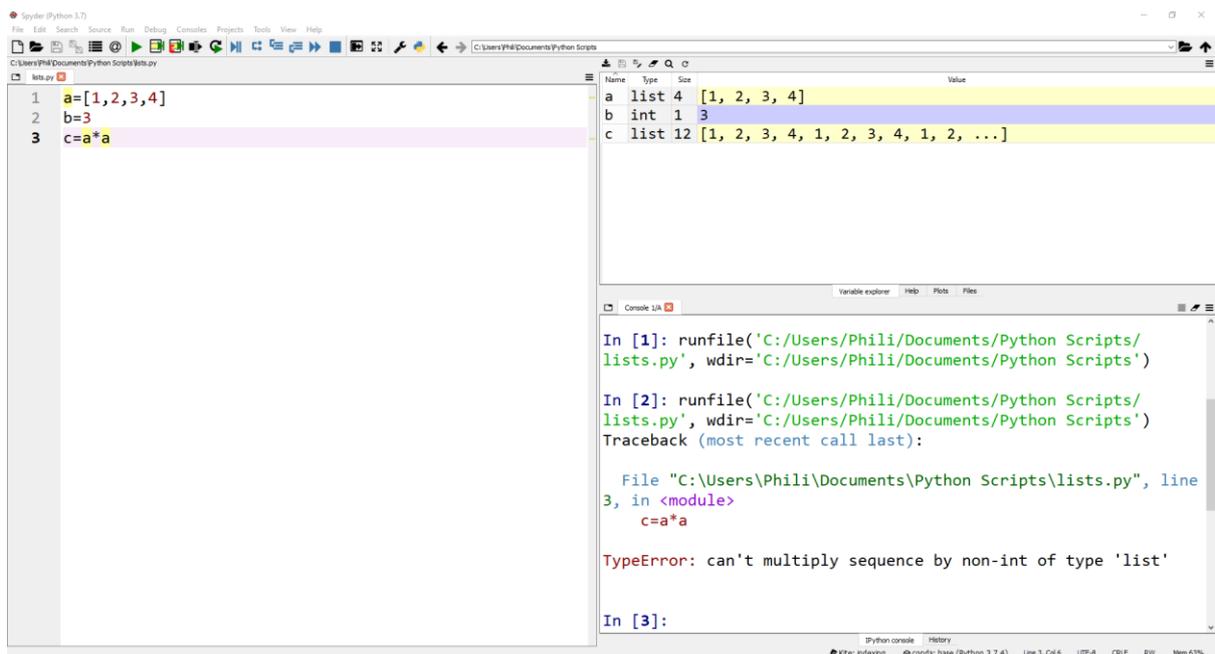
$$a = [1, 2, 3, 4]$$

$$b = 3$$

$$c = a * b = [1, 2, 3, 4] * 3 = [1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]$$



If a list is attempted to be multiplied by another list for example `a` by itself an error will instead show. `TypeError: can't multiply sequence by non-int of type 'list'`.



For numpy arrays element by element multiplication is possible using the `*` operator, element by element exponentiation is possible using the `**` operator and element by element float division is possible using the `/` operator.

```

1. # %% Prerequisites
2. import numpy as np
3. # % numpy arrays
4. a=np.array([1,2,3,4])
5. b=4
6. c=a*4

```

7. $d=a*a$
8. $e=a**2$
9. $f=e/e$

The screenshot shows a Jupyter Notebook with the following code in the cell:

```

1  %% Prequisites
2  import numpy as np
3  %% numpy arrays
4  a=np.array([1,2,3,4])
5  b=4
6  c=a*4
7  d=a*a
8  e=a**2
9  f=a/a

```

The Variable Explorer on the right shows the following variables:

Name	Type	Size	Value
a	Array of int32	(4,)	[1 2 3 4]
b	int	1	4
c	Array of int32	(4,)	[4 8 12 16]
d	Array of int32	(4,)	[1 4 9 16]
e	Array of int32	(4,)	ndarray object of numpy module
f	Array of float64	(4,)	[1. 1. 1. 1.]

The Console shows the following output:

```

Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/
lists.py', wdir='C:/Users/Phili/Documents/Python Scripts')

In [2]: e
Out[2]: array([ 1,  4,  9, 16], dtype=int32)

In [3]: d==e
Out[3]: array([ True,  True,  True,  True])

In [4]:

```

Note e for some reason does not display correctly in the variable explorer despite being the same values as d .

$$a = [1, 2, 3, 4]$$

$$b = 4$$

$$c = a * b = [1, 2, 3, 4] * 4 = [1, 2, 3, 4] * [4, 4, 4, 4]$$

$$c = [1 * 4, 2 * 4, 3 * 4, 4 * 4] = [4, 8, 12, 16]$$

$$d = a * a = [1, 2, 3, 4] * [1, 2, 3, 4]$$

$$d = [1 * 1, 2 * 2, 3 * 3, 4 * 4] = [1, 4, 9, 16]$$

$$e = a ** 2 = [1^2, 2^2, 3^2, 4^2]$$

$$e = [1, 4, 9, 16]$$

$$f = a/a = [1, 2, 3, 4]/[1, 2, 3, 4]$$

$$f = [1/1, 2/2, 3/3, 4/4] = [1, 1, 1, 1]$$

As `d` and `e` are the same object, conditional logic can be used to compare them. Note the output is again given element by element.

```
g = d == e
g = [1,4,9,16] == [1,4,9,16]
g = [1==1,4==4,9==9,16==16]
g = [True, True, True, True]
```

Creation of Matrices

Supposing we want to create a matrix:

$$a = \begin{bmatrix} 2 & 4 & 6 \\ 3 & 1 & 5 \end{bmatrix}$$

This can be thought of as a list of lists:

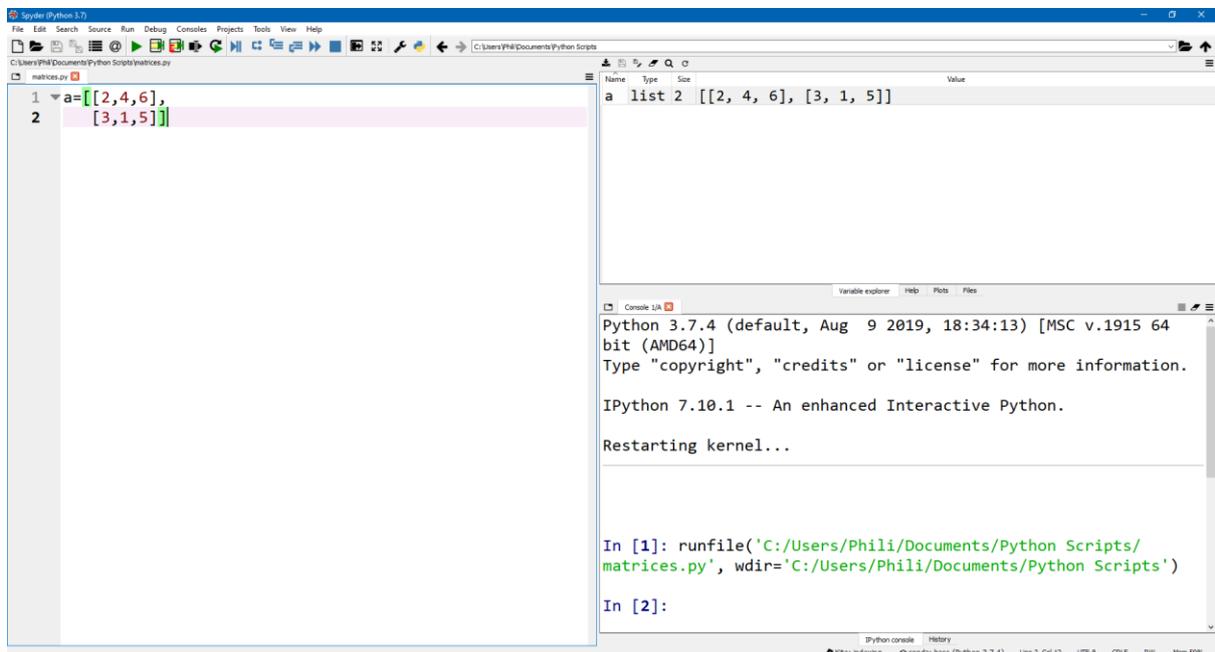
$$a = \begin{bmatrix} 2 & 4 & 6 \\ 3 & 1 & 5 \end{bmatrix}$$

In Python it will be input using:

```
1. a=[[2,4,6],[3,1,5]]
```

However this can be split over multiple lines for readability:

```
1. a=[[2,4,6],
2.    [3,1,5]]
```



Note this shows up on the variable explorer as a list of size 2 (the outside list). It can be opened up to view it in more detail.

Index	Type	Size	Value
0	list	3	[2, 4, 6]
1	list	3	[3, 1, 5]

Here we see that each list has a nested list. These can also be expanded.

Index	Type	Size	Value
0	int	1	2
1	int	1	4
2	int	1	6

We can index into an element in this nested list by first indexing into the outside list and then indexing into the nested list.

$$a = \begin{bmatrix} 2 & 4 & 6 \\ 3 & 1 & 5 \end{bmatrix}$$

The highlighted element is in the 0th row and 2nd column. This can be indexed by selecting the 0th from the outside list and then selecting the 2nd element from the nested list.

a[0][2]

```

1 a = [[2,4,6],
2       [3,1,5]]

```

Index	Type	Size	Value
0	list	3	[2, 4, 6]
1	list	3	[3, 1, 5]

Index	Type	Size	Value
0	int	1	2
1	int	1	4
2	int	1	6

```

Name Type Size
a list 2 [[2, 4, 6], [3, 1, 5]]

```

```

Type "copyright", "credits" or "license" for more information.

IPython 7.10.1 -- An enhanced Interactive Python.
Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/
matrices.py', wdir='C:/Users/Phili/Documents/Python Scripts')

In [2]: a[0][2]
Out[2]: 6

In [3]:

```

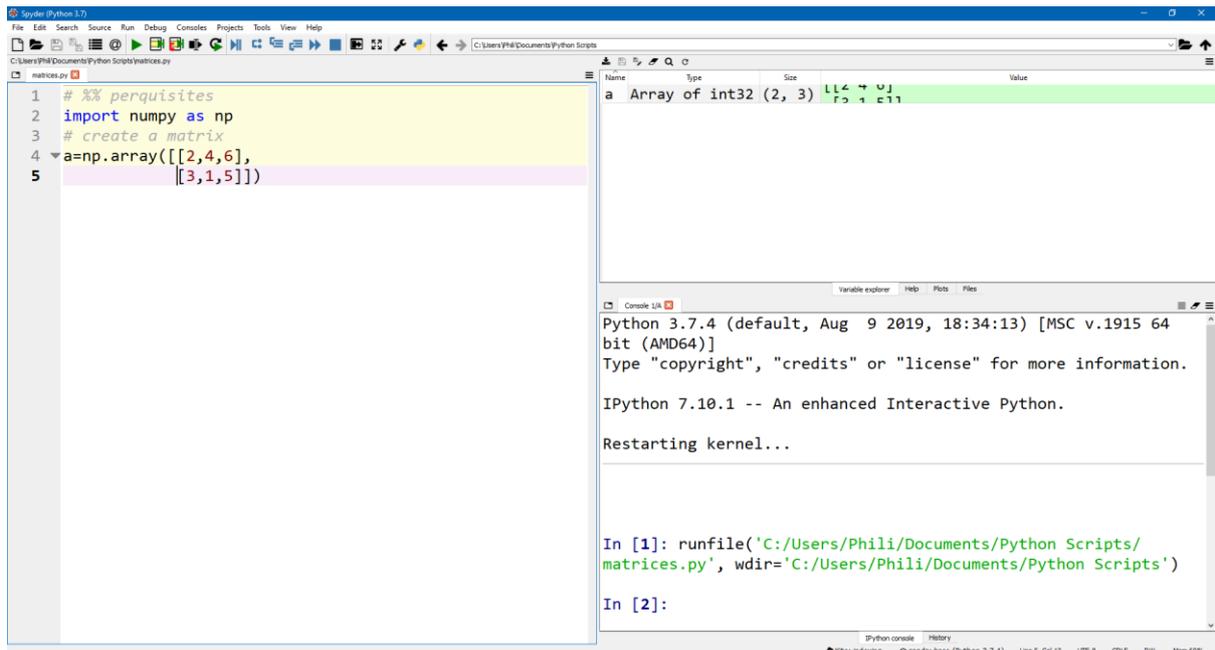
Now let's compare this with a matrix created as a numpy array.

```

1. # %% prerequisites
2. import numpy as np
3. # create a matrix
4. a=np.array([[2,4,6],
              [3,1,5]])

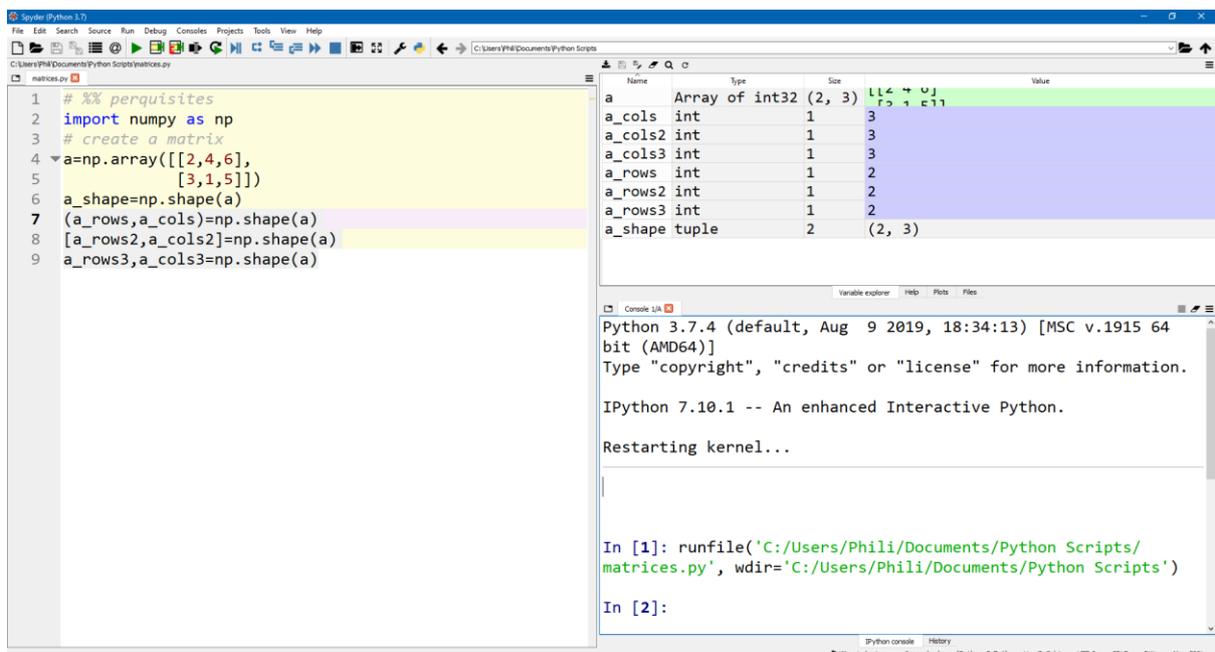
```

In this case the nested list is the input argument for the function `np.array()`.



Note the color difference on the variable explorer and the size of the numpy array is listed as a tuple `(2, 3)` opposed to a value of `2`. This tuple can be accessed using the function `np.shape()` and can explicitly be packed into rows and columns by explicitly assigning the output to a tuple or list or two individual output arguments:

1. `a_shape=np.shape(a)`
2. `(a_rows,a_cols)=np.shape(a)`
3. `[a_rows2,a_cols2]=np.shape(a)`
4. `a_rows3,a_cols3=np.shape(a)`



This can be expanded in the variable explorer.

	0	1	2
0	2	4	6
1	3	1	5

Format Resize Background color

Save and Close Close

Note all elements show together alongside their row numbers and column numbers.

We can index into an element into this by first indexing into the row number and then indexing into the column number.

$$a = \begin{bmatrix} 2 & 4 & 6 \\ 3 & 1 & 5 \end{bmatrix}$$

The highlighted element is in the 0th row and 2nd column. i.e. it is of the same form as the nested list however there is no need to use two individual square brackets.

```
a[0][2]
```

```
a[0,2]
```

The screenshot shows the Spyder Python IDE interface. On the left, a script named 'matrices.py' contains the following code:

```
1 %% perquisites
2 import numpy as np
3 # create a matrix
4 a=np.array([[2,4,6],
5             [3,1,5]])
```

On the right, the Variable explorer shows the variable 'a' as an 'Array of int32 (2, 3)' with the values $\begin{bmatrix} 2 & 4 & 6 \\ 3 & 1 & 5 \end{bmatrix}$. The console output shows the following interactions:

```
Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/
matrices.py', wdir='C:/Users/Phili/Documents/Python Scripts')

In [2]: a[0][2]
Out[2]: 6

In [3]: a[0,2]
Out[3]: 6

In [4]: |
```

It is also possible to index entire rows and entire columns using the colon. For example, the following could be x and y data for a plot:

$$a = \begin{bmatrix} 2 & 4 & 6 \\ 3 & 1 & 5 \end{bmatrix}$$

In which case all rows would be selected:

```
x=a[:,0]
y=a[:,1]
```

The screenshot shows a Python IDE window with a code editor on the left and a variable explorer on the right. The code in the editor is:

```
1 # %% prerequisites
2 import numpy as np
3 # create a matrix
4 a=np.array([[2,4,6],
5             [3,1,5]])
```

The variable explorer on the right shows the following variables:

Name	Type	Size	Value
a	Array of int32 (2, 3)	[[2 4 6] [3 1 5]]	
x	Array of int32 (2,)	[2 3]	
y	Array of int32 (2,)	[4 1]	

The console window below shows the execution of the code:

```
IPython 7.10.1 -- An enhanced Interactive Python.
Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/
matrices.py', wdir='C:/Users/Phili/Documents/Python Scripts')

In [2]: x=a[:,0]

In [3]: y=a[:,1]

In [4]: |
```

1. # %% prerequisites
2. **import** numpy **as** np
3. # create a matrix
4. a=np.array([[2,4,6],[3,1,5]])
5. col_0=a[:,0]
6. col_1=a[:,1]
7. row_0=a[0,:]
8. row_1=a[1,:]

The screenshot shows a Python IDE window with a code editor on the left and a variable explorer on the right. The code in the editor is:

```
1 # %% prerequisites
2 import numpy as np
3 # create a matrix
4 a=np.array([[2,4,6],
5             [3,1,5]])
6 col_0=a[:,0]
7 col_1=a[:,1]
8 row_0=a[0,:]
9 row_1=a[1,:]
```

The variable explorer on the right shows the following variables:

Name	Type	Size	Value
a	Array of int32 (2, 3)	[[2 4 6] [3 1 5]]	
col_0	Array of int32 (2,)	[2 3]	
col_1	Array of int32 (2,)	[4 1]	
row_0	Array of int32 (3,)	[2 4 6]	
row_1	Array of int32 (3,)	[3 1 5]	

The console window below shows the execution of the code:

```
Python 3.7.4 (default, Aug 9 2019, 18:34:13) [MSC v.1915 64
bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

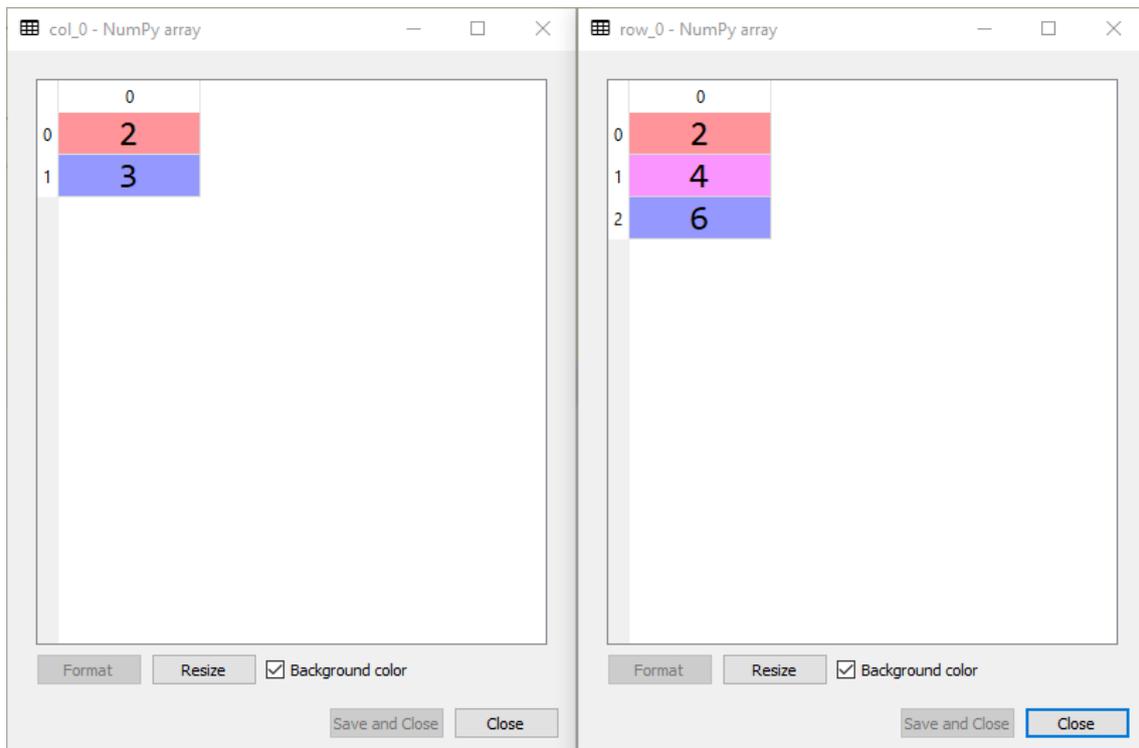
IPython 7.10.1 -- An enhanced Interactive Python.

Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/
matrices.py', wdir='C:/Users/Phili/Documents/Python Scripts')

In [2]:
```

Note the size of `col_0` and `col_1` as `(2,)` and `row_0` and `row_1` as `(3,)` which can be confusing when first encountered. Looking at the variable explorer they are all displayed as rows however when expanded they are instead displayed as columns...



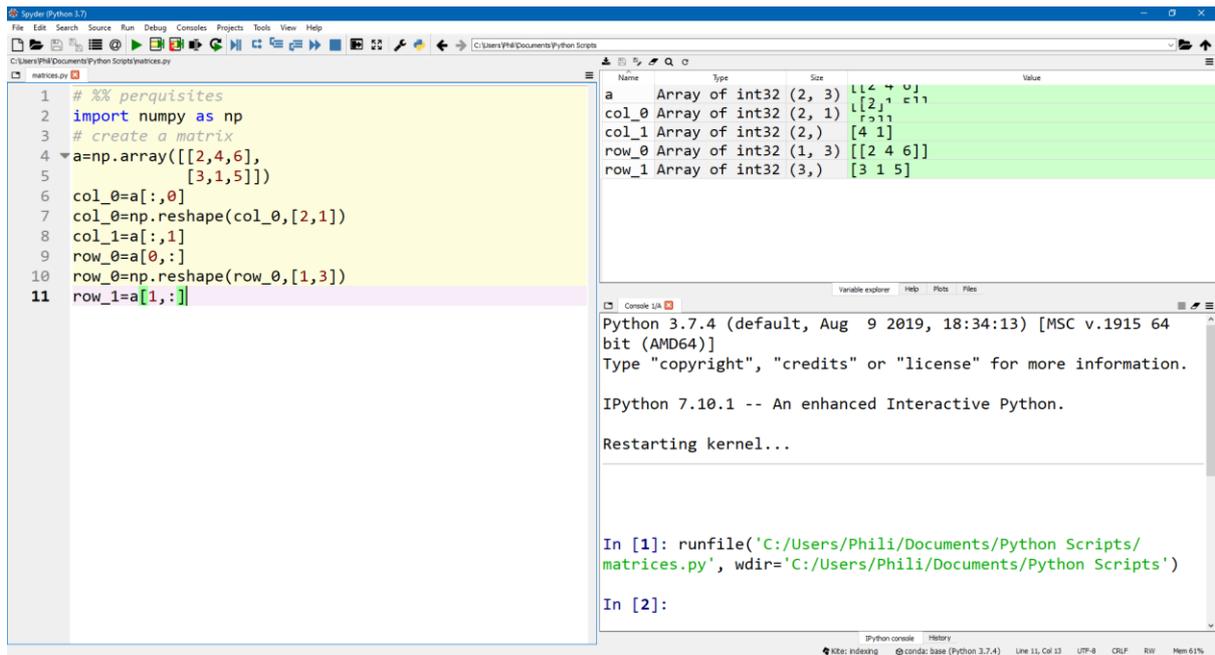
Vectors by default are taken to only have a solo dimension as the other dimension is 1 and will usually be displayed or used as most convenient. In a number of cases it is useful to explicitly express them as columns and rows respectively. This can be done using the `np.reshape()` function which has two input arguments. The input array to be reshaped and a vector specifying the new number of rows and number of columns respectively.

```
new_array=np.reshape(old_array,[n_cols,n_rows])
```

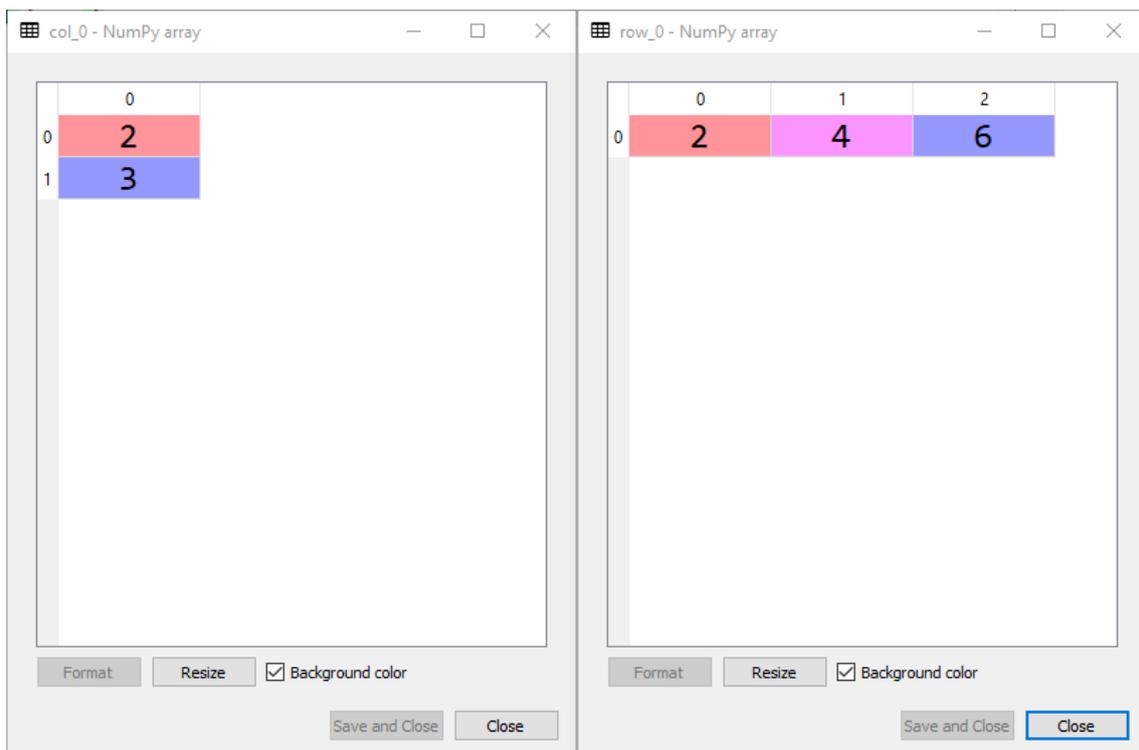
In this example we can reshape `col_0` as a column and `row_0` as a row and instead of assigning these to new variable names we can reassign them to the original variable names.

```
1. # %% prerequisites
2. import numpy as np
3. # create a matrix
4. a=np.array([[2,4,6],
5.             [3,1,5]])
6. col_0=a[:,0]
7. col_0=np.reshape(col_0,[2,1])
8. col_1=a[:,1]
9. row_0=a[0,:]
10. row_0=np.reshape(row_0,[1,3])
11. row_1=a[1,:]
```

Note now in the variable explorer the sizes of `col_0` and `row_0` are now specifically `(2,1)` and `(1,3)` respectively and they show as a column and row respectively.



This can be seen when they are expanded from the variable explorer.



Addition, subtraction, multiplication, exponentiation, division and logical comparisons can be performed on matrices as previously demonstrated with lists using the operators `+`, `-`, `*`, `**`, `/`, `<`, `>`, `<=`, `>=`, `==` and `!=`. For this to happen, both matrices must have matching shapes, there are some exceptions however. If one of these operations is used with a single value known as a scalar than scalar expansion along both axes is performed and the scalar is treated as a matrix with equal dimensions. Additionally if one of these operations is performed using a vector which has the same dimension as the matrix along one of the dimensions, vector expansion is carried out along the other dimension giving a matrix of the same dimensions. **Care should be taken when dealing with square matrices and**

vector expansion as it may expand along the other axes and you should explicitly reshape the vector as a row or column before to avoid confusion.

Let us look at some practical examples.

Element by Element Operations

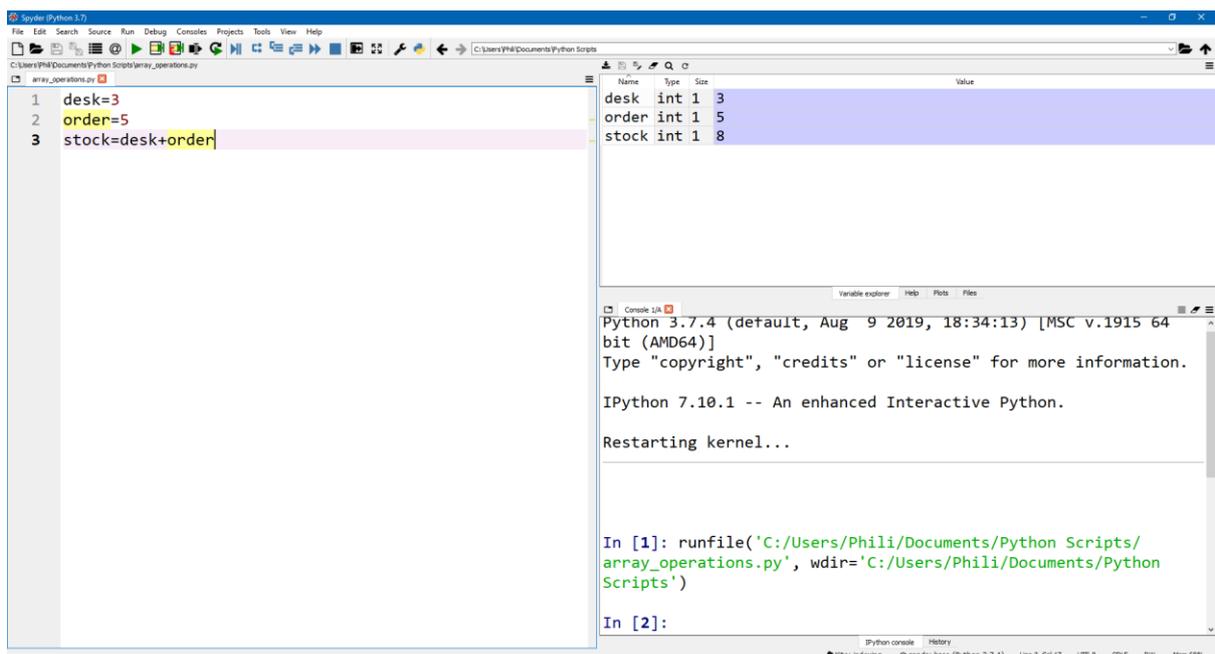


When looking at these problems for the first time it is worth equating them to physical objects so you can see what is going on. Assume you have 3 pens on your desk, and you order 3 pens online. When they arrive, this gives you:

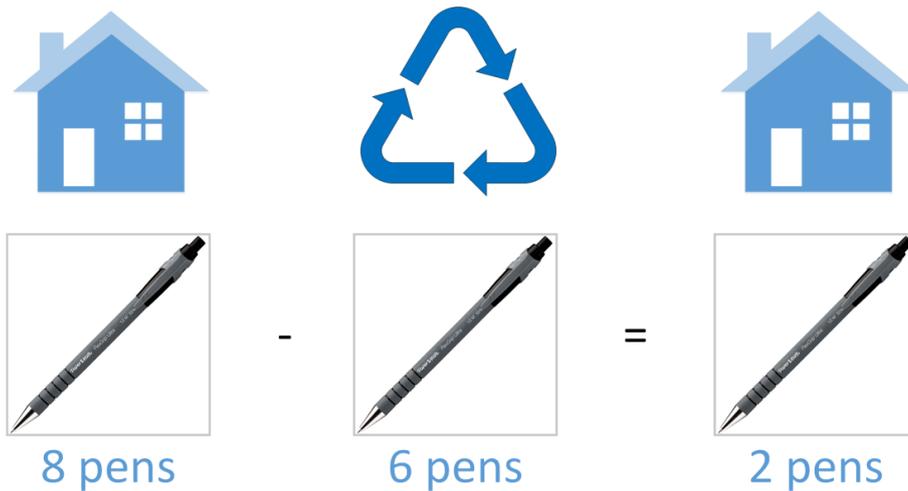
$$3 \text{ pens} + 5 \text{ pens} = 8 \text{ pens}$$

This can be coded in Python by use of variable names.

1. `desk=3`
2. `order=5`
3. `stock=desk+order`



We can get rid of the variable desk and instead assign it to stock and later update stock.



Now as pens are a consumable and eventually run out of ink. We can assume we start with 8 pens and use up 6 pens and can quickly calculate how many we have left.

$$8 \text{ pens} - 6 \text{ pens} = 2 \text{ pens}$$

1. `stock=3`
2. `order=5`
3. `stock=stock+order`
4. `recycle=6`
5. `stock=stock-recycle`

The screenshot shows the Spyder Python IDE interface. The editor on the left contains the following code:

```

1 stock=3
2 order=5
3 stock=stock+order
4 recycle=6
5 stock=stock-recycle

```

The Variable explorer on the right shows the state of the variables after execution:

Name	Type	Size	Value
order	int	1	5
recycle	int	1	6
stock	int	1	2

The console at the bottom shows the execution output:

```

Python 3.7.4 (default, Aug 9 2019, 18:34:13) [MSC v.1915 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

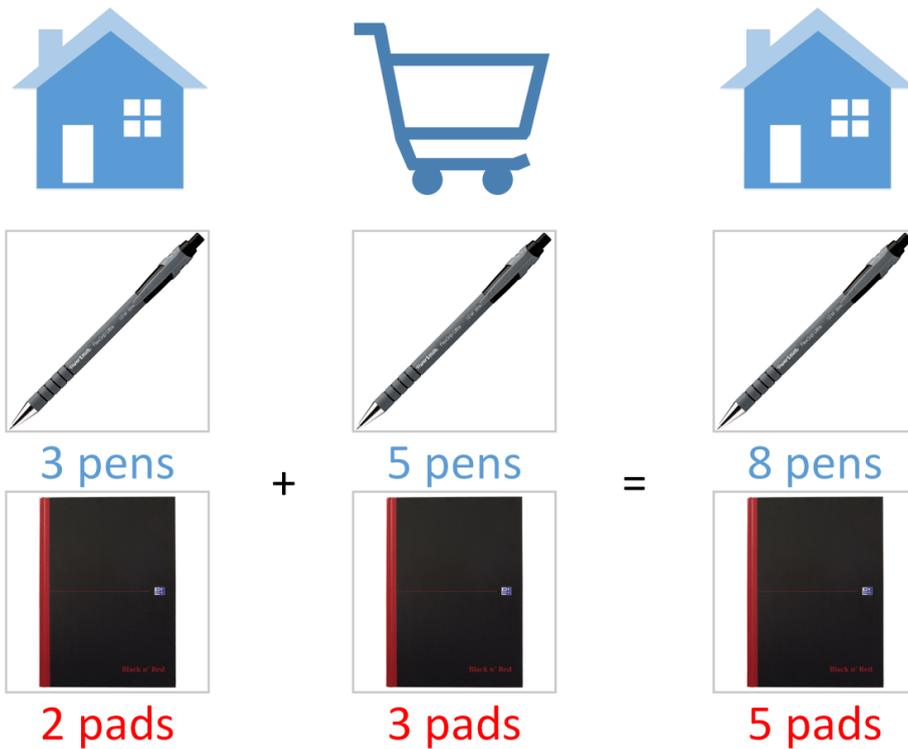
IPython 7.10.1 -- An enhanced Interactive Python.

Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/array_operations.py', wdir='C:/Users/Phili/Documents/Python Scripts')
In [2]:

```

Now let's instead look at a more complicated sample. Instead of looking at a single object type. Let's look at two pens and pads. It is impossible to convert a pen into a pad and vice versa so we mustn't do this when we input them.



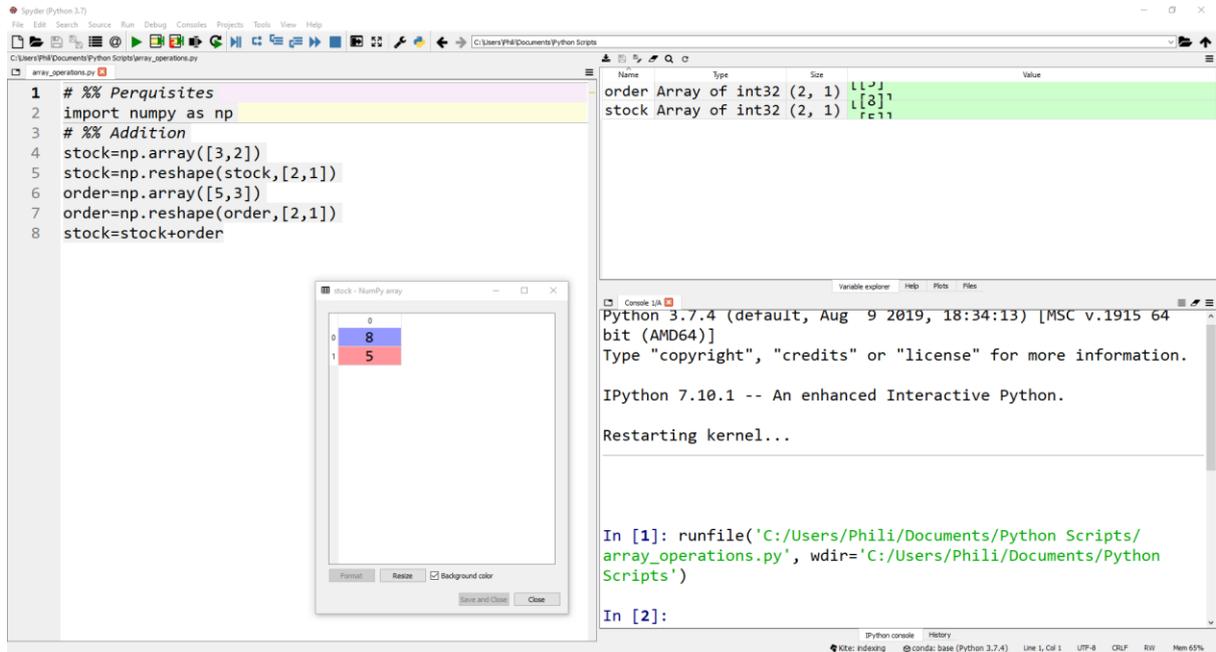
$$\begin{bmatrix} 3 \text{ pens} \\ 2 \text{ pads} \end{bmatrix} + \begin{bmatrix} 5 \text{ pens} \\ 3 \text{ pads} \end{bmatrix} = \begin{bmatrix} 8 \text{ pens} \\ 5 \text{ pads} \end{bmatrix}$$

In this case we will explicitly specify columns using the `np.reshape()` function.

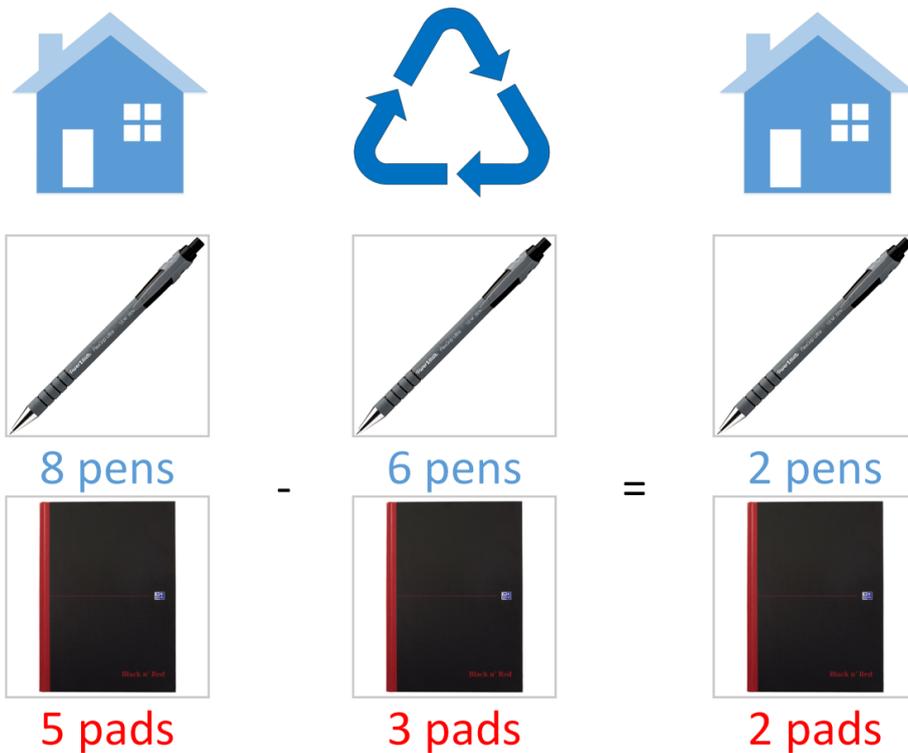
```

1. # %% Perquisites
2. import numpy as np
3. # %% Addition
4. stock=np.array([3,2])
5. stock=np.reshape(stock,[2,1])
6. order=np.array([5,3])
7. order=np.reshape(order,[2,1])
8. stock=stock+order

```



Once again both are consumables so we can recycle 6 pens and 3 pads.



$$\begin{bmatrix} 8 \text{ pens} \\ 5 \text{ pads} \end{bmatrix} + \begin{bmatrix} 6 \text{ pens} \\ 3 \text{ pads} \end{bmatrix} = \begin{bmatrix} 2 \text{ pens} \\ 2 \text{ pads} \end{bmatrix}$$

```

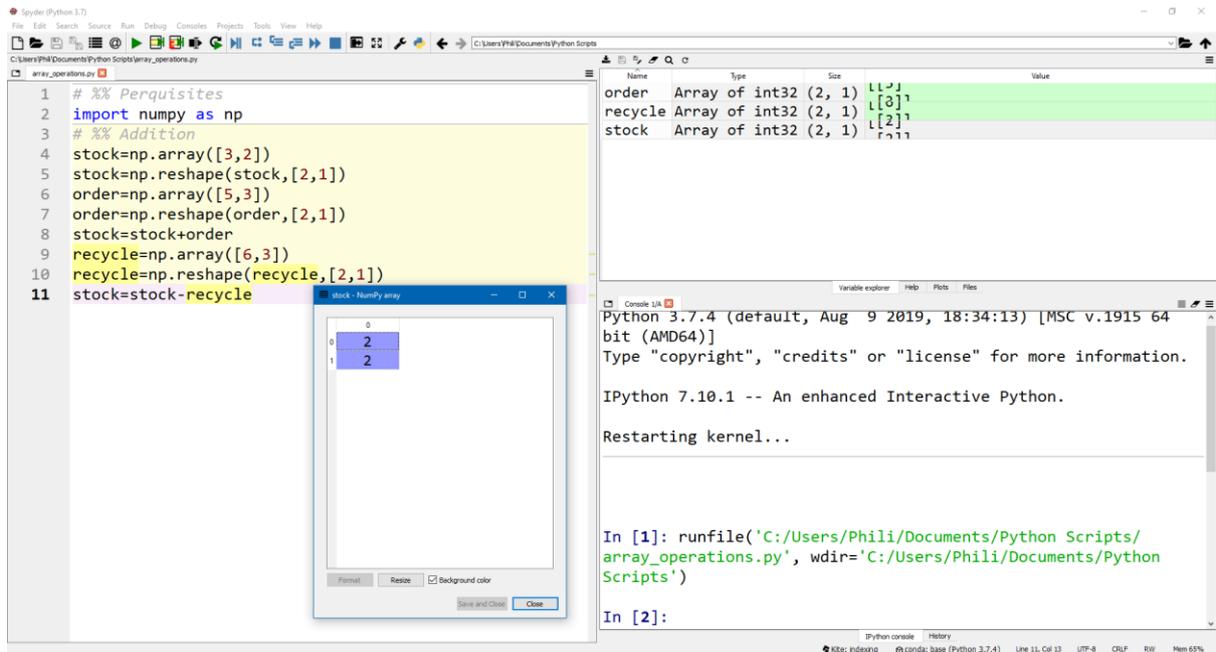
1. # %% Perquisites
2. import numpy as np
3. # %% Addition
4. stock=np.array([3,2])

```

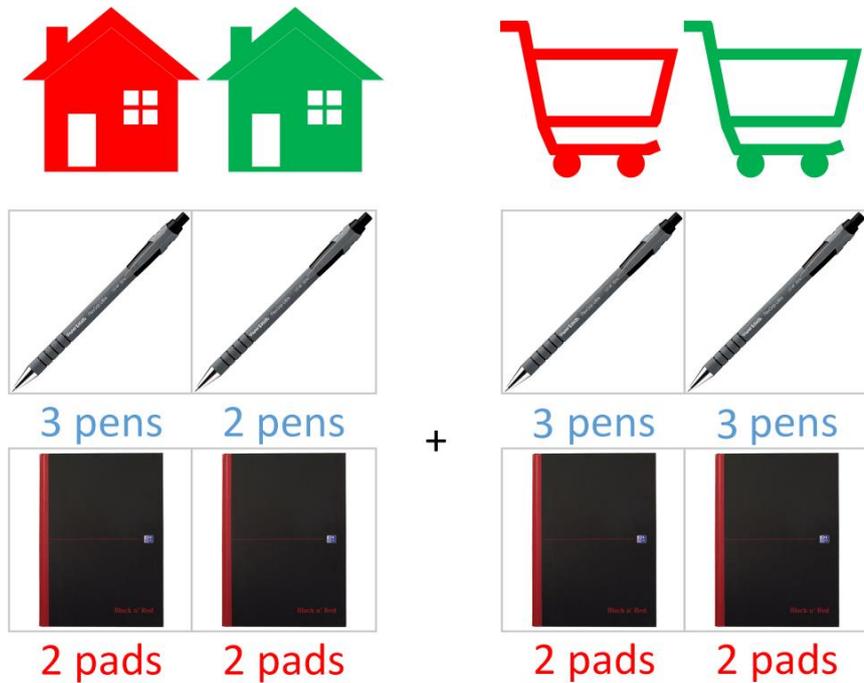
```

5. stock=np.reshape(stock,[2,1])
6. order=np.array([5,3])
7. order=np.reshape(order,[2,1])
8. stock=stock+order
9. recycle=np.array([6,3])
10.recycle=np.reshape(recycle,[2,1])
11.stock=stock-recycle

```



So far, so good, let's now look at a matrix.



In this case we have two homes which each have their own inventory and both homes have made the same blanket order.

$$\begin{bmatrix} 3 \text{ pens} & 2 \text{ pads} \\ 2 \text{ pads} & 2 \text{ pads} \end{bmatrix} + \begin{bmatrix} 3 \text{ pens} \\ 2 \text{ pads} \end{bmatrix}$$

First of all vector expansion (duplicating the column):

$$= \begin{bmatrix} 3 \text{ pens} & 2 \text{ pads} \\ 2 \text{ pads} & 2 \text{ pads} \end{bmatrix} + \begin{bmatrix} 3 \text{ pens} & 3 \text{ pens} \\ 2 \text{ pads} & 2 \text{ pads} \end{bmatrix}$$

Then addition (element by element)

$$= \begin{bmatrix} 3 \text{ pens} & 2 \text{ pads} \\ 2 \text{ pads} & 2 \text{ pads} \end{bmatrix} + \begin{bmatrix} 3 \text{ pens} & 3 \text{ pens} \\ 2 \text{ pads} & 2 \text{ pads} \end{bmatrix}$$

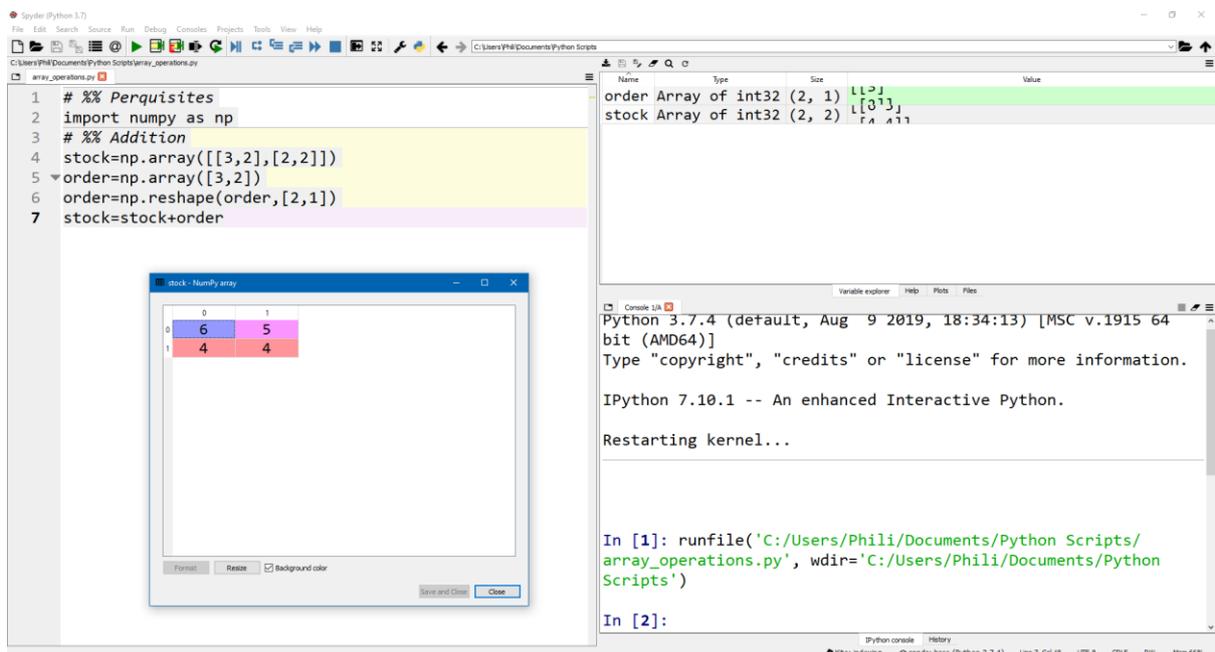
$$= \begin{bmatrix} 3 \text{ pens} + 3 \text{ pens} & 2 \text{ pads} + 3 \text{ pens} \\ 2 \text{ pads} + 2 \text{ pads} & 2 \text{ pads} + 2 \text{ pads} \end{bmatrix}$$

$$= \begin{bmatrix} 6 \text{ pens} & 5 \text{ pens} \\ 4 \text{ pads} & 4 \text{ pads} \end{bmatrix}$$

```

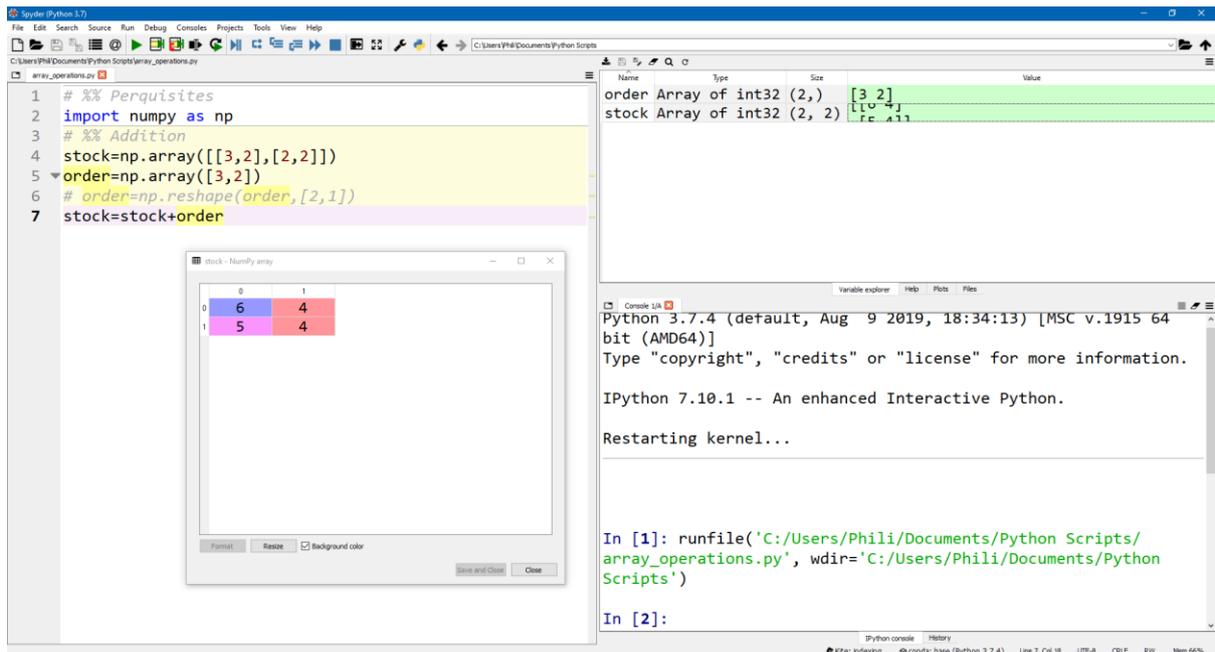
1. # %% Perquisites
2. import numpy as np
3. # %% Addition
4. stock=np.array([[3,2],[2,2]])
5. order=np.array([3,2])
6. order=np.reshape(order,[2,1])
7. stock=stock+order

```



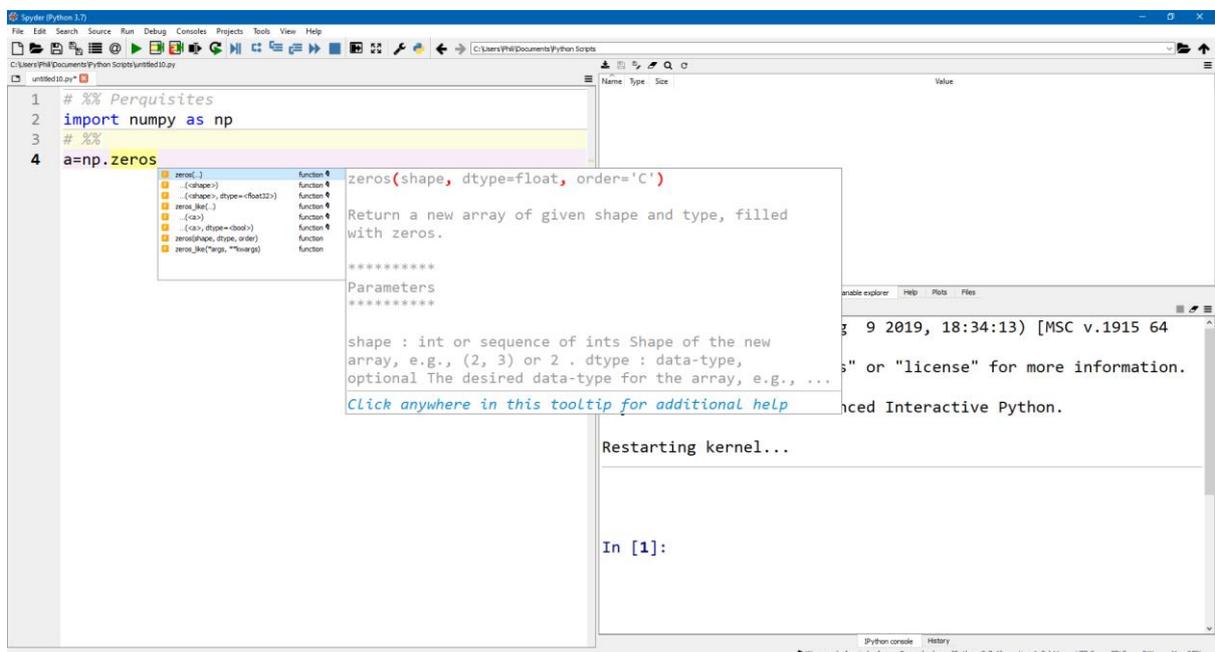
Notice what happens if line 5 is commented out, in such a scenario we get the wrong value. This is because our vector `order` was assumed by Python to be a row, but we intended it to be a column. This meant that the vector expansion was carried out using the wrong axes.

$$\begin{bmatrix} 3 & 2 \\ 2 & 2 \end{bmatrix} + [3 \quad 2]$$



Functions for Rapid Array Generation

The `numpy` library has several functions for rapid array generation. The function `np.zeros()` and `np.ones()` take in a shape as their input argument(s) and generate arrays where each element is `0` and `1` respectively. We can take advantage of scalar expansion and `np.ones()` to quickly generate an array where each element is a different constant value.



For a vector only a single scalar input argument is required however for a matrix a tuple (or list) can be input with the 0th index denoting the number of rows (axis 0) and the 1st index denoting the number of columns (axis 1).

1. # %% Perquisites
2. `import numpy as np`
3. # %% Array Generation

```

4. a=np.zeros(4)
5. b=np.zeros((2,3))
6. c=np.ones((2,3))
7. d=2*np.ones((2,3))

```

Note how the input scalar or input tuple corresponds to the size of the array shown on the variable explorer.

The screenshot shows the Spyder Python IDE with a script named 'rapid_array_generation.py'. The code in the editor is as follows:

```

1 # %% Perquisites
2 import numpy as np
3 # %% Array Generation
4 a=np.zeros(4)
5 b=np.zeros((2,3))
6 c=np.ones((2,3))
7 d=2*np.ones((2,3))

```

The Variable explorer on the right shows the following variables:

Name	Type	Size	Value
a	Array of float64	(4,)	[0. 0. 0. 0.]
b	Array of float64	(2, 3)	[[0. 0. 0.] [0. 0. 0.]]
c	Array of float64	(2, 3)	[[1. 1. 1.] [1. 1. 1.]]
d	Array of float64	(2, 3)	[[2. 2. 2.] [2. 2. 2.]]

Below the code editor, four NumPy array visualizers are shown, each displaying a 2D grid of values:

- a**: A 1x4 grid of zeros.
- b**: A 2x3 grid of zeros.
- c**: A 2x3 grid of ones.
- d**: A 2x3 grid of twos.

The console at the bottom shows the execution of the script:

```

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/rapid_array_generation.py', wdir='C:/Users/Phili/Documents/Python Scripts')
In [2]:

```

The function `np.diag()` can be used to with an input vector to generate a square matrix where the diagonal is the list and every other element is 0. Alternatively, if the input is a square matrix then it will read the diagonal. The anti-diagonal is less common but can be accessed using the function `flipplr()` or `flipud()` which flip a matrix horizontally and vertically respectively. `flip()` is used for a vector.

```

1. # %% Perquisites
2. import numpy as np
3. # %% Array Generation
4. # a=np.zeros(4)
5. # b=np.zeros((2,3))
6. # c=np.ones((2,3))
7. # d=2*np.ones((2,3))
8. e=np.diag([1,2,3,4])
9. f=np.array([[1,2,3,4],
10.            [5,6,7,8],
11.            [9,10,11,12],
12.            [13,14,15,16]])
13. g=np.fliplr(f)
14. h=np.flipud(f)
15. i=np.diag(f)
16. j=np.diag(g)

```

As we can see, `e` has a diagonal of `[1, 2, 3, 4]` with every element being assigned to `0`.

	0	1	2	3
0	1	0	0	0
1	0	2	0	0
2	0	0	3	0
3	0	0	0	4

We can compare `f` to `g` which is flipped horizontally:

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

	0	1	2	3
0	4	3	2	1
1	8	7	6	5
2	12	11	10	9
3	16	15	14	13

We can compare `f` to `h` which is flipped vertically:

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

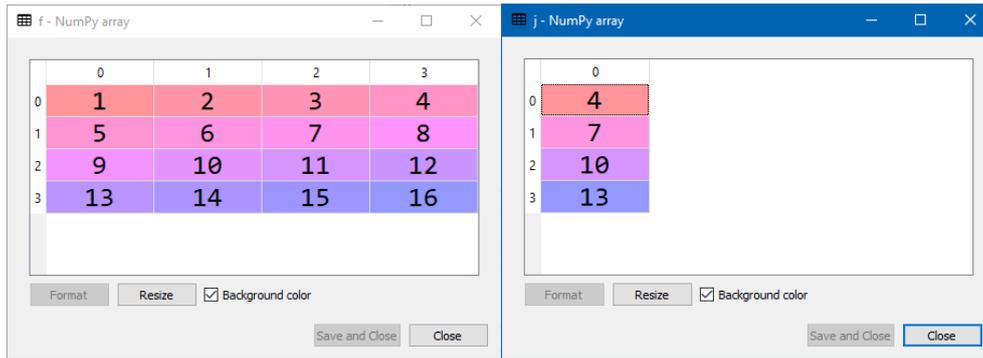
	0	1	2	3
0	13	14	15	16
1	9	10	11	12
2	5	6	7	8
3	1	2	3	4

We can see that `i` is the diagonal of `f`:

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

	0
0	1
1	6
2	11
3	16

And `j` is the antidiagonal of `f`:

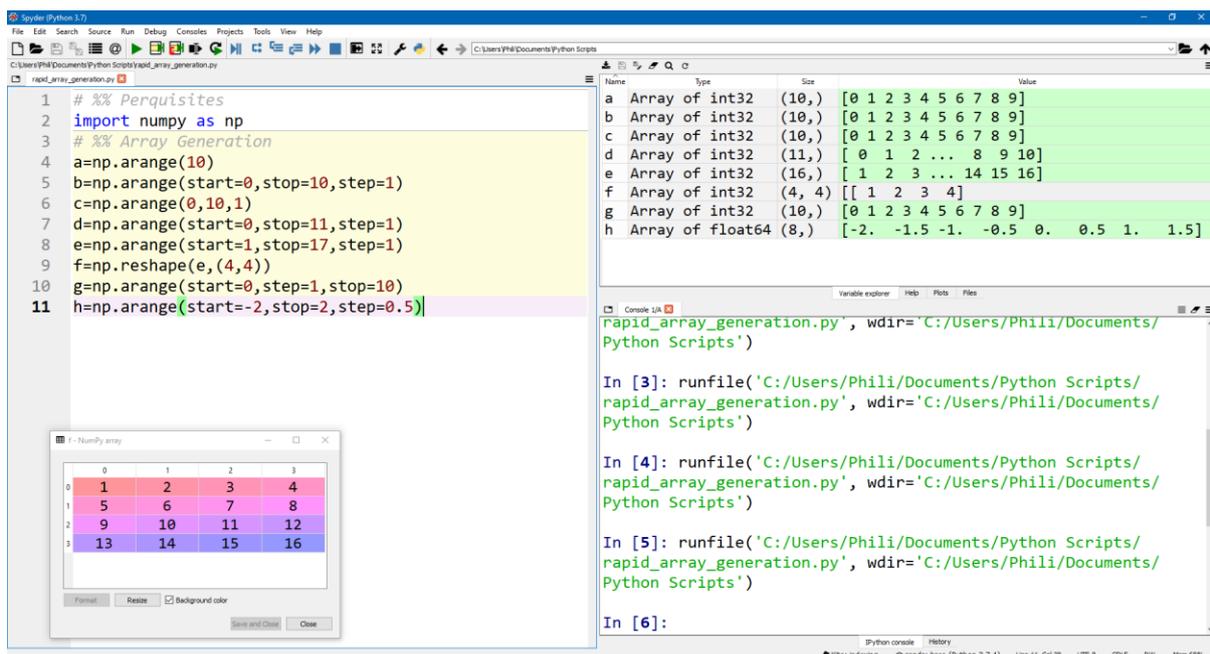


The function `np.arange(10)` can be called using a single input argument in which it will automatically set `start=0` and `step=1` or it can be called with three keyword arguments `np.arange(start=0, stop=10, step=1)`. Keyword arguments can be called in any order (line 10) however this function can also be called using just three numbers `np.arange(0, 10, 1)` (line 6) and when it is the 0th number will automatically be assumed to be the start, the 1st number will be assumed to be the stop and the 2nd number will be assumed to be the step. Take into account of zero order indexing when using this function line 4,5 and 6 will go to 10 but in steps of 1 but not reach 10 terminating at 9. In contrast line 6 will go to 11 in steps of 1 and terminate at 10.

```

1. # %% Perquisites
2. import numpy as np
3. # %% Array Generation
4. a=np.arange(10)
5. b=np.arange(start=0, stop=10, step=1)
6. c=np.arange(0, 10, 1)
7. d=np.arange(start=0, stop=11, step=1)
8. e=np.arange(start=1, stop=17, step=1)
9. f=np.reshape(e, (4, 4))
10. g=np.arange(start=0, step=1, stop=10)
11. h=np.arange(start=-2, stop=2, step=0.5)

```



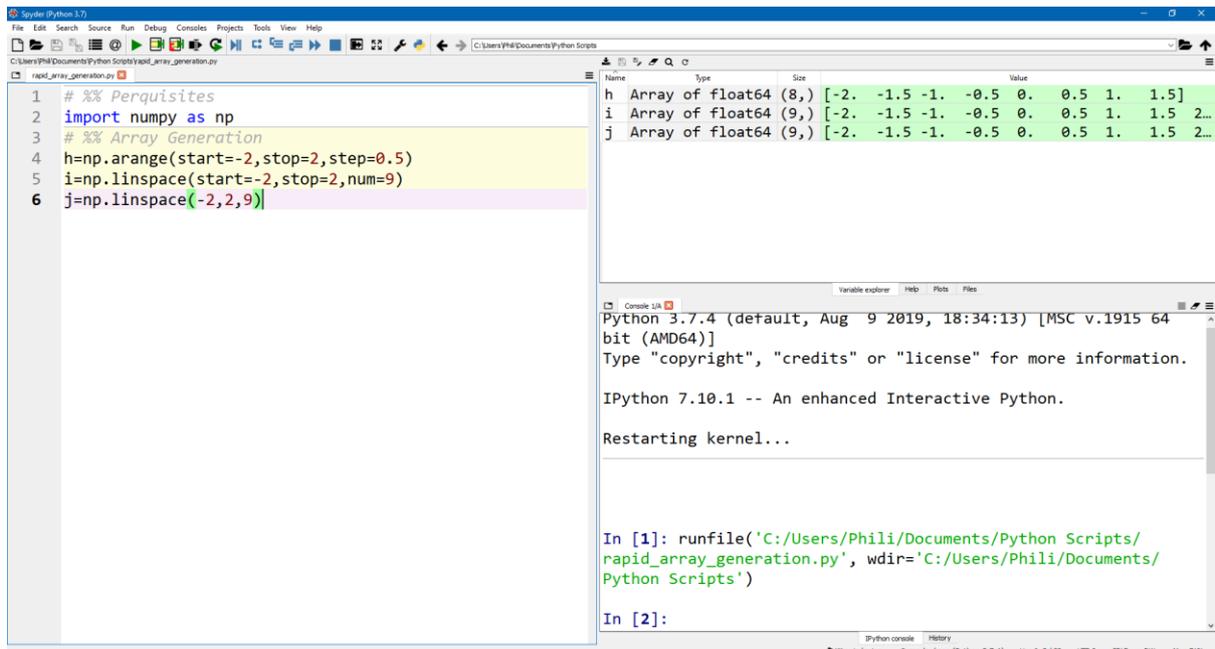
The square matrix which we created before when looking at the `np.diag()`, `np.fliplr()` and `np.flipud()` functions can be recreated using `np.arange()` (line 8) and `np.reshape()` (line 9). Note in line 9 we need to explicitly start at 1 opposed to the default of 0 and stop at 17 as a consequence of zero-order indexing.

In line 11 we tried to create `h` with equally spaced values of 0.5 between -2 and 2. In such a case we should use the function `np.linspace()` instead of `np.arange()`. These functions are similar and the first two keyword input arguments are the same `start` and `stop` however the third keyword input argument is `num` and `np.linspace()` does not take into account zero order indexing going creating values which start and end at the values given by the keyword input arguments `start` and `stop`.

```

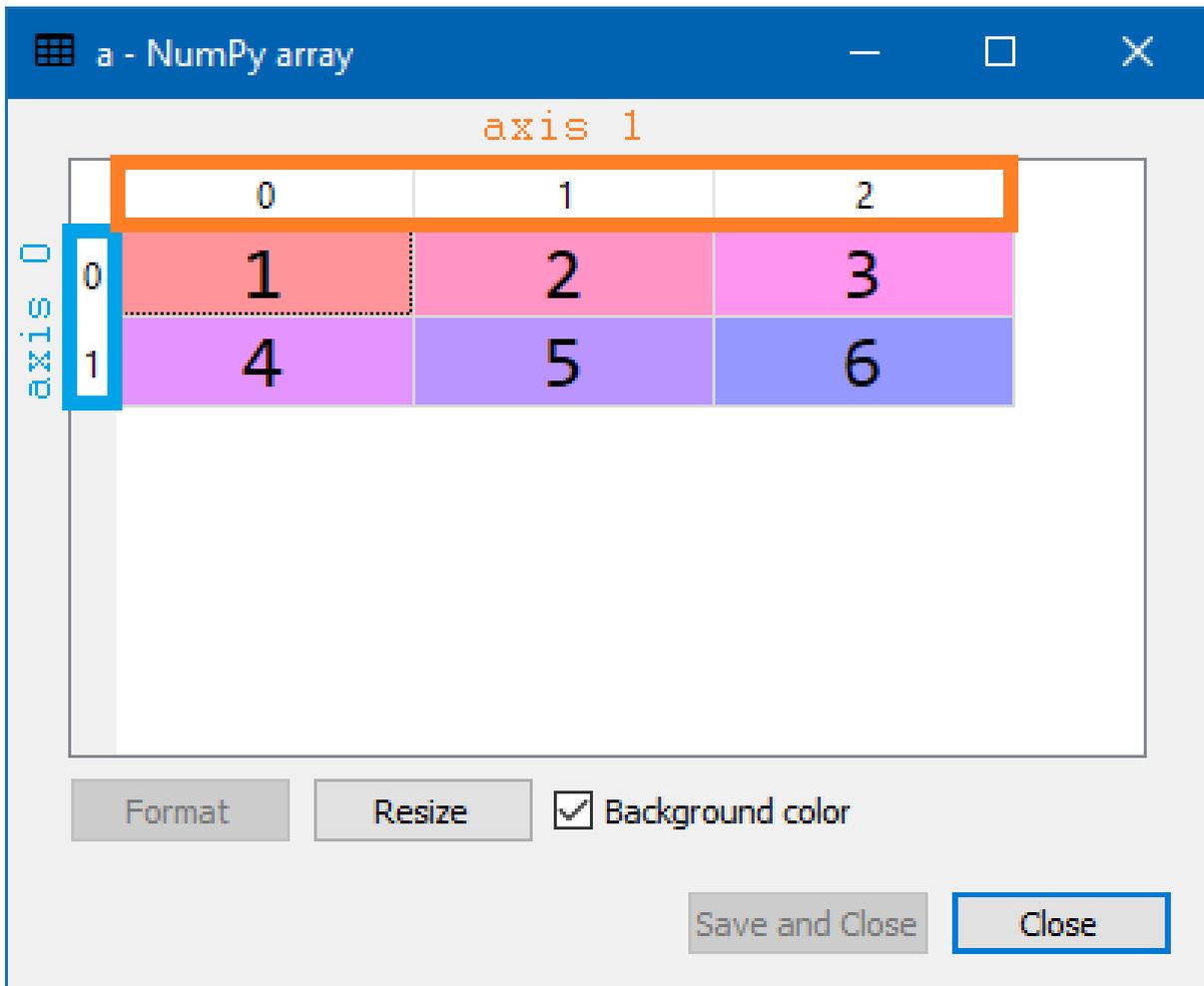
1. # %% Perquisites
2. import numpy as np
3. # %% Array Generation
4. h=np.arange(start=-2,stop=2,step=0.5)
5. i=np.linspace(start=-2,stop=2,num=9)
6. j=np.linspace(-2,2,9)

```



Concatenation of NumPy Arrays

We seen earlier that the operator `+` concatenated lists but performed addition in numpy arrays. The function `np.concatenate([array1,array2],axis=0)` can be used to concatenate two numpy arrays together. The first input argument is a list of the numpy arrays to be concatenated and an additional keyword input argument can be used to specify the axis to concatenate with. Recall that when we index, `axis=0` denotes the rows and `axis=1` denotes the columns.



If we concatenate along `axis=0`, then we act along columns (meaning we have the requirement that the rows must have the same dimension). This means we add the first matrix being concatenated below the zeroth matrix. For example:

$$\begin{aligned}
 a &= \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \\
 b &= \begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} \\
 ab_concat_axis0 &= \begin{bmatrix} a \\ b \end{bmatrix} \\
 ab_concat_axis0 &= \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}
 \end{aligned}$$

Alternatively, if we concatenate along `axis=1`, then we act along rows (meaning we have the requirement that the columns must have the same dimension). This means we add the first matrix being concatenated right of the zeroth matrix. For example:

$$\begin{aligned}
 ab_concat_axis1 &= [a \quad b] \\
 ab_concat_axis1 &= \begin{bmatrix} 1 & 2 & 3 & 7 & 8 & 9 \\ 4 & 5 & 6 & 10 & 11 & 12 \end{bmatrix}
 \end{aligned}$$

We can use the function `np.arange()` to generate two 2 by 3 arrays (line 4-7) and then concatenate these (line 8-10).

```
1. # %% Perquisites
2. import numpy as np
3. # %% Array Generation
4. a=np.arange(start=1,stop=7,step=1)
5. a=np.reshape(a,(2,3))
6. b=np.arange(start=7,stop=13,step=1)
7. b=np.reshape(b,(2,3))
8. c=np.concatenate([a,b],axis=0)
9. d=np.concatenate([a,b],axis=1)
10. e=np.concatenate([a,b])
```

The screenshot shows the Spyder Python IDE interface. The left pane contains the Python code from the previous block. The right pane is split into two sections: a variable explorer and a console.

Variable Explorer:

Name	Type	Size	Value
a	Array of int32 (2, 3)		$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$
b	Array of int32 (2, 3)		$\begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$
c	Array of int32 (4, 3)		$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$
d	Array of int32 (2, 6)		$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \end{bmatrix}$
e	Array of int32 (4, 3)		$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$

Console:

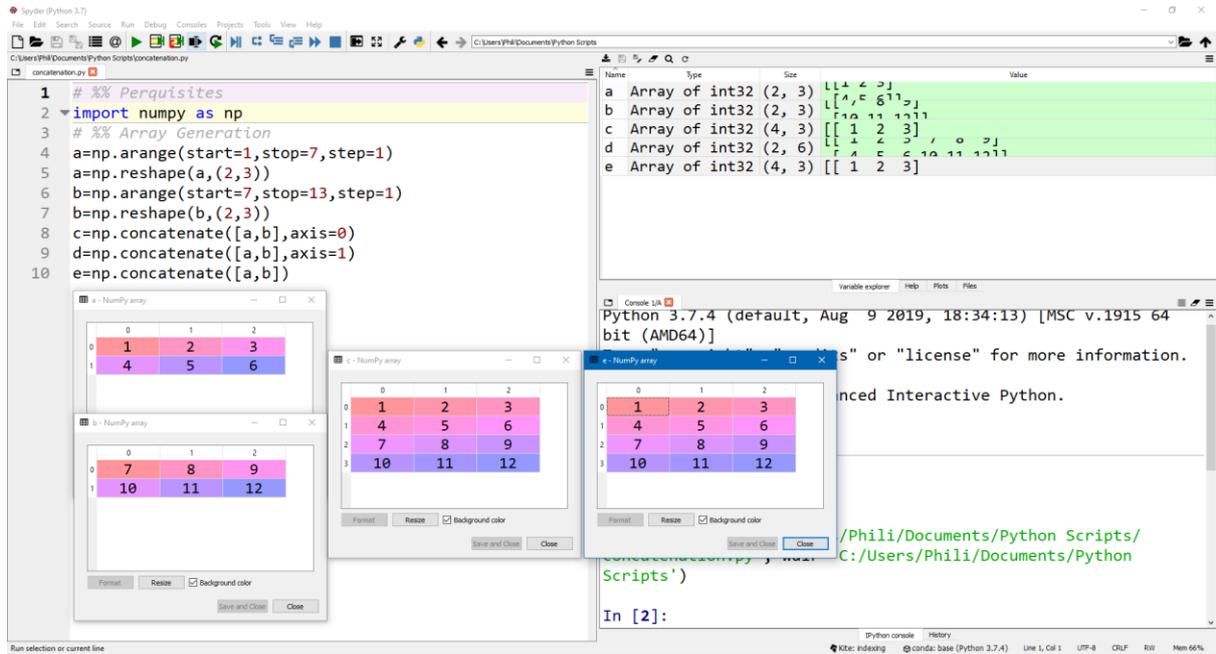
```
Python 3.7.4 (default, Aug 9 2019, 18:34:13) [MSC v.1915 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.10.1 -- An enhanced Interactive Python.

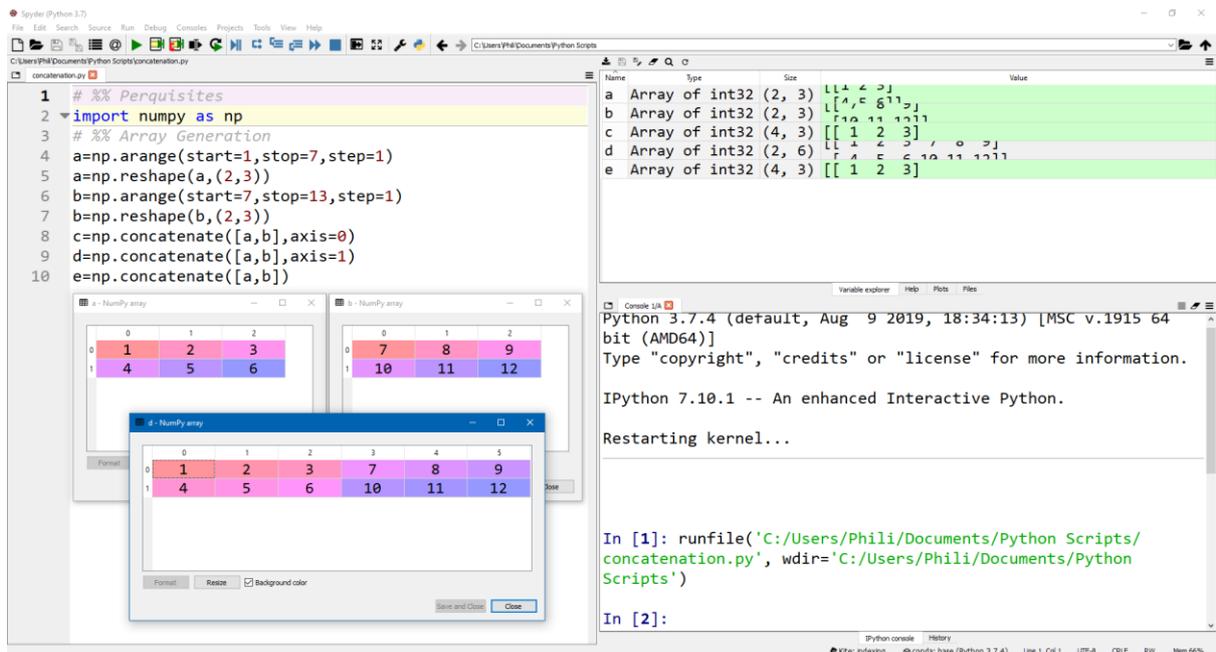
Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/rapid_array_generation.py', wdir='C:/Users/Phili/Documents/Python Scripts')
In [2]:
```

In this case `c` and `e` are the same as expected as the default keyword value for `axis=0` meaning that by default we act along columns.



When the `axis=1` is instead selected we get `d`.



Before proceeding it is recommended to spend some time practicing indexing, slicing and concatenating matrices.

Create the following matrix `m` using `np.arange()` and `np.reshape()`.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

Index into `m` and save the following slices using the variable names `yellow`, `red`, `magenta`, `green` and `cyan`.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

For the slices that are vectors, use the `np.reshape()` function to explicitly assign them to row vectors or column vectors.

Use concatenation to create the matrix `n`:

$$\begin{bmatrix} 2 & 3 & 4 & 1 \\ 12 & 10 & 11 & 5 \\ 16 & 14 & 15 & 9 \\ 6 & 7 & 8 & 13 \end{bmatrix}$$

For additional practice start with `n` and work your way back to `m`.

```
1. # Perquisites
2. import numpy as np
3. m=np.reshape(np.arange(start=1,stop=17,step=1),(4,4))
4. # Selections
5. yellow=m[:,0]
6. yellow=np.reshape(yellow,(4,1))
7. red=m[0,1:]
8. red=np.reshape(red,(1,3))
9. magenta=m[1,1:]
10. magenta=np.reshape(magenta,(1,3))
11. green=m[2:,1:3]
12. cyan=m[2:,3]
13. cyan=np.reshape(cyan,(2,1))
14. # Concatenation
15. cyangreen=np.concatenate([cyan,green],axis=1)
16. redcyangreenmagenta=np.concatenate(
17.     [red,cyangreen,magenta],axis=0)
18. n=np.concatenate([redcyangreenmagenta,yellow],axis=1)
```

The screenshot shows the Spyder Python IDE with a script named 'concatenation.py'. The script performs various NumPy operations including reshaping and concatenating arrays. The variable explorer on the right shows the state of the variables: cyan, cyangreen, green, m, magenta, n, red, redcyangreenmagenta, and yellow. Below the explorer, the console shows the execution of the script, including the command 'runfile' used to execute the script.

```

1 # Perquisites
2 import numpy as np
3 m=np.reshape(np.arange(start=1,stop=17,step=1),(4,4))
4 # Selections
5 yellow=m[:,0]
6 yellow=np.reshape(yellow,(4,1))
7 red=m[0,1:]
8 red=np.reshape(red,(1,3))
9 magenta=m[1,1:]
10 magenta=np.reshape(magenta,(1,3))
11 green=m[2:,3]
12 cyan=m[2:,3]
13 cyan=np.reshape(cyan,(2,1))
14 # Concatenation
15 cyangreen=np.concatenate([cyan,green],axis=1)
16 redcyangreenmagenta=np.concatenate(
17     [red,cyangreen,magenta],axis=0)
18 n=np.concatenate([redcyangreenmagenta,yellow],axis=1)

```

Name	Type	Size	Value
cyan	Array of int32	(2, 1)	[[4]]
cyangreen	Array of int32	(2, 3)	[[12 10 11]]
green	Array of int32	(2, 2)	[[16 11]]
m	Array of int32	(4, 4)	[[1 2 3 4]]
magenta	Array of int32	(1, 3)	[[6 7 8]]
n	Array of int32	(4, 4)	[[2 3 4 1]]
red	Array of int32	(1, 3)	[[2 3 4]]
redcyangreenmagenta	Array of int32	(4, 3)	[[2 3 4]]
yellow	Array of int32	(4, 1)	[[1]]

Sorting Data in Matrices

There are several numpy functions which work on vectors and matrices but differ in the way they act depending on the input argument(s).

1. # %% Prequisites
2. `import numpy as np`
3. # Vectors
4. `v=np.array([4,6,5])`
5. `v_sorted=np.sort(v)`
6. `v_arg_sorted=np.argsort(v)`
7. `v_sorted_reverse=np.flip(v_sorted)`

The screenshot shows the Spyder Python IDE with a script named 'sorting.py'. The script demonstrates sorting a vector and its arguments. The variable explorer on the right shows the state of the variables: v, v_arg_sorted, v_sorted, and v_sorted_reverse. Below the explorer, the console shows the execution of the script, including the command 'runfile' used to execute the script.

```

1 # %% Prequisites
2 import numpy as np
3 # Vectors
4 v=np.array([4,6,5])
5 v_sorted=np.sort(v)
6 v_arg_sorted=np.argsort(v)
7 v_sorted_reverse=np.flip(v_sorted)
8

```

Name	Type	Size	Value
v	Array of int32	(3,)	[4 6 5]
v_arg_sorted	Array of int64	(3,)	[0 2 1]
v_sorted	Array of int32	(3,)	[4 5 6]
v_sorted_reverse	Array of int32	(3,)	[6 5 4]

Here we can see:

$$v = [4 \ 6 \ 5]$$

Gets sorted to:

$$v_sorted = [4 \ 5 \ 6]$$

And if we flip this, we get the reverse order:

$$v_sorted_reverse = [6 \ 5 \ 4]$$

And we can see that argsort looks at v , looks for the lowest value:

$$v = \begin{bmatrix} 4 & 6 & 5 \\ 0 & 1 & 2 \end{bmatrix}$$

Computes its index in this case $v[0]$

$$v_arg_sorted = [0 \ \dots \ \dots]$$

Then looks for the next lowest value:

$$v = \begin{bmatrix} 4 & 6 & 5 \\ 0 & 1 & 2 \end{bmatrix}$$

Computes its index in this case $v[2]$

$$v_arg_sorted = [0 \ 2 \ \dots]$$

Then looks for the next lowest value:

$$v = \begin{bmatrix} 4 & 6 & 5 \\ 0 & 1 & 2 \end{bmatrix}$$

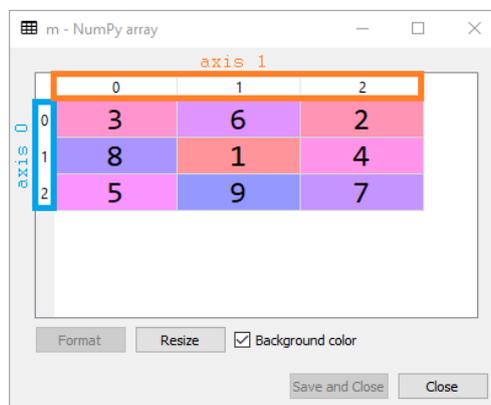
Computes its index in this case $v[1]$

$$v_arg_sorted = [0 \ 2 \ 1]$$

We can now look at using this function with a matrix.

$$m = \begin{bmatrix} 3 & 6 & 2 \\ 8 & 1 & 4 \\ 5 & 9 & 7 \end{bmatrix}$$

In the case of a matrix we have multiple axes which we can sort by.



We can sort by $axis=0$ which sorts along columns:

$$m_{\text{axis0}} = \begin{bmatrix} 3 & 6 & 2 \\ 8 & 1 & 4 \\ 5 & 9 & 7 \end{bmatrix}$$

$$m_{\text{sorted_axis0}} = \begin{bmatrix} m[0,0] & m[0,1] & m[0,2] \\ \hat{3} & \hat{1} & \hat{2} \\ m[2,0] & m[1,1] & m[1,2] \\ \hat{5} & \hat{6} & \hat{4} \\ m[1,0] & m[2,1] & m[2,2] \\ \hat{8} & \hat{9} & \hat{7} \end{bmatrix}$$

$$m_{\text{argsorted_axis0}} = \begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 1 \\ 1 & 2 & 2 \end{bmatrix}$$

or `axis=1` which sorts along rows:

$$m_{\text{axis1}} = \begin{bmatrix} 3 & 6 & 2 \\ 8 & 1 & 4 \\ 5 & 9 & 7 \end{bmatrix}$$

$$m_{\text{sorted_axis1}} = \begin{bmatrix} m[0,2] & m[0,0] & m[0,1] \\ \hat{2} & \hat{3} & \hat{6} \\ m[1,1] & m[1,2] & m[1,0] \\ \hat{1} & \hat{4} & \hat{8} \\ m[2,0] & m[2,2] & m[2,1] \\ \hat{5} & \hat{7} & \hat{9} \end{bmatrix}$$

$$m_{\text{argsorted_axis1}} = \begin{bmatrix} 2 & 0 & 1 \\ 1 & 2 & 0 \\ 0 & 2 & 1 \end{bmatrix}$$

In this case, `axis` is a keyword argument which has a default value of `axis=0` as it is more common to group data along columns than rows.

```

1. # %% Perquisites
2. import numpy as np
3. # Matrices
4. m=np.array([[3,6,2],[8,1,4],[5,9,7]])
5. m_sorted_axis0=np.sort(m,axis=0)
6. m_argsorted_axis0=np.argsort(m,axis=0)
7. indices_axis0=np.array([[0,1,2],[0,1,2],[0,1,2]])
8. m_sorted_axis0_2=m[m_argsorted_axis0,indices_axis0]

```

```

1 # %% Perquisites
2 import numpy as np
3 # Matrices
4 m=np.array([[3,6,2],[8,1,4],[5,9,7]])
5 m_sorted_axis0=np.sort(m,axis=0)
6 m_argsorted_axis0=np.argsort(m,axis=0)
7 indices_axis0=np.array([[0,1,2],[0,1,2],[0,1,2]])
8 m_sorted_axis0_2=m[m_argsorted_axis0,indices_axis0]
9

```

Name	Type	Size	Value
indices_axis0	Array of int32	(3, 3)	[[0 1 2]
m	Array of int32	(3, 3)	[[3 6 2]
m_argsorted_axis0	Array of int64	(3, 3)	[[0 1 0]
m_sorted_axis0	Array of int32	(3, 3)	[[3 1 2]
m_sorted_axis0_2	Array of int32	(3, 3)	[[3 1 2]

```

1. # %% Perquisites
2. import numpy as np
3. # Matrices
4. m=np.array([[3,6,2],[8,1,4],[5,9,7]])
5. m_sorted_axis1=np.sort(m,axis=1)
6. m_argsorted_axis1=np.argsort(m,axis=1)
7. indices_axis1=np.array([[0,0,0],[1,1,1],[2,2,2]])
8. m_sorted_axis1_2=m[indices_axis1,m_argsorted_axis1]

```

```

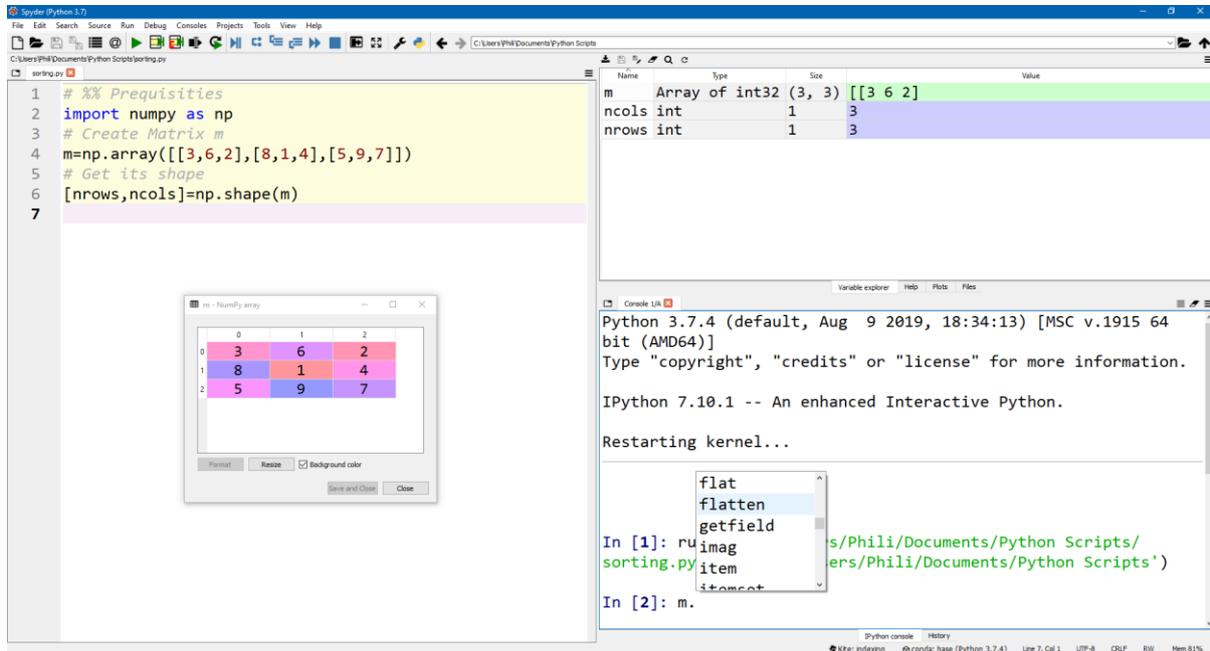
1 # %% Perquisites
2 import numpy as np
3 # Matrices
4 m=np.array([[3,6,2],[8,1,4],[5,9,7]])
5 m_sorted_axis1=np.sort(m,axis=1)
6 m_argsorted_axis1=np.argsort(m,axis=1)
7 indices_axis1=np.array([[0,0,0],[1,1,1],[2,2,2]])
8 m_sorted_axis1_2=m[indices_axis1,m_argsorted_axis1]
9

```

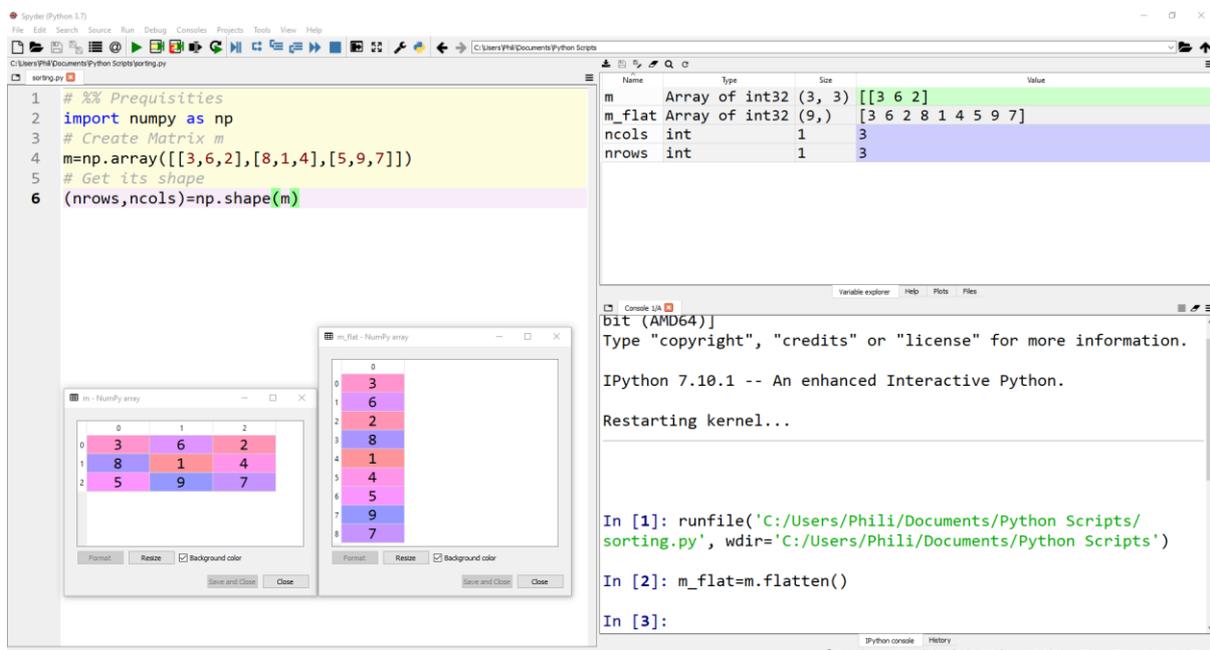
Name	Type	Size	Value
indices_axis1	Array of int32	(3, 3)	[[0 0 0]
m	Array of int32	(3, 3)	[[3 6 2]
m_argsorted_axis1	Array of int64	(3, 3)	[[2 0 1]
m_sorted_axis1	Array of int32	(3, 3)	[[2 3 6]
m_sorted_axis1_2	Array of int32	(3, 3)	[[2 3 6]

It is less common the sort the entire matrix by order so there is no option to select `axis='all'`. To do this we need to calculate the shape of the matrix, flatten it to a vector which we can sort, then we can reshape this back to the original shape.

Let's create the matrix `m` and then compute the number of rows and columns by assigning a tuple to `np.shape(m)`. Previously when we looked at lists we could access a number of list functions (or methods) by typing in the list name followed by a `.` and then a `[]`. This can also be done with numpy arrays. In this case we are interested in using the numpy array method `flatten()`.

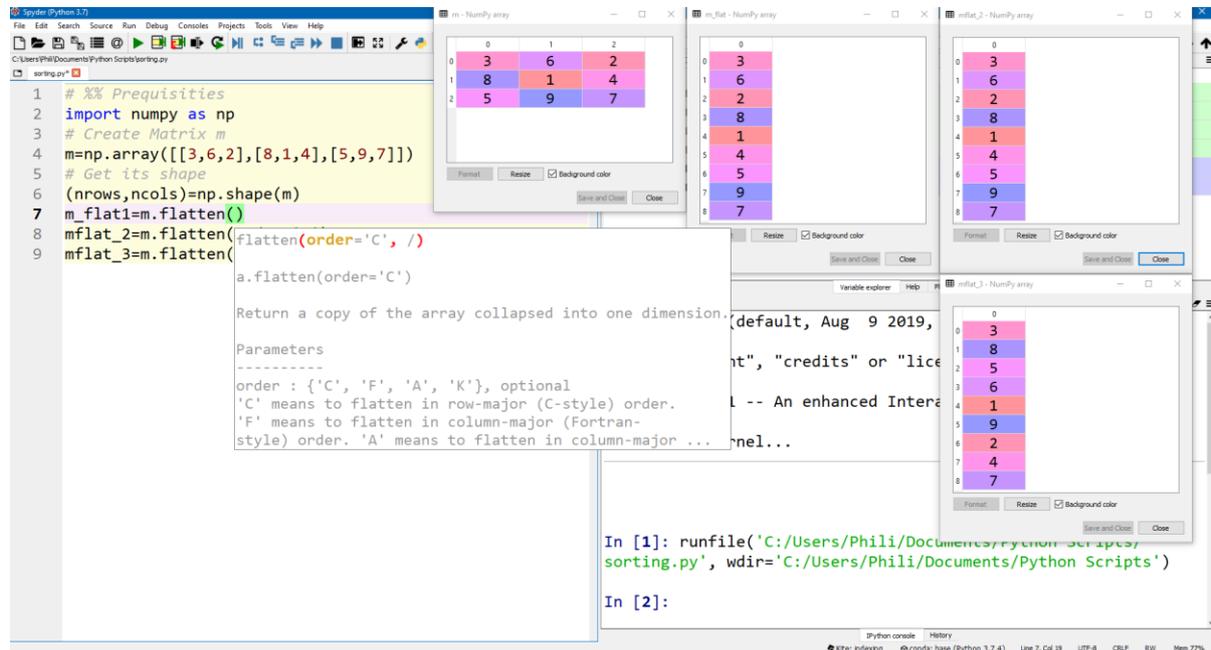


This array gets flattened by creating a vector of the 0th row and to this concatenating the vector of the 1st row and 2nd row and so on. Because it is a vector, it shows up on in the variable explorer as a row and shows up as a column when the variable is opened in its own window.



This function has no positional input arguments as it is called as a numpy array function (method) however it can have keyword input argument to change the way the array gets flattened. Unfortunately it does not use `axis=0` and `axis=1` like many other numpy functions uses. Instead it uses `order='C'` by default which does not denote columns, it instead denotes the programming

language C which flattens by rows. This can be changed to `order='F'` which flattens by columns. Recall that more details about the input arguments can be found by typing the function without any input arguments.



In our case it doesn't matter what one we use as we are going to sort the flattened vector. We can use the `sort` function to do this that we used on the vector `v` or we can call the method `sort()` directly from the numpy array.

Although they look similar if we compare the function (1) `n=np.sort(m,axis=0)` versus the method (2) `n=m.sort(axis=0)` we see a few differences. The first difference is if we look at the value returned, we get the sorted list when we use (1) but we get `NoneType` when we use (2). This is because (2) mutates the original list (in the screenshot below you can see that `m_flat` has been altered) and does not return any output value. Recall earlier on when we made a custom function and in the custom function where we did not define an output that when we attempted to call our custom function and assign it to an output variable, the value of this was `NoneType` well in this case the same thing has happened using the method `sort()` called from the matrix (numpy array) `m`.

The screenshot shows the Spyder Python IDE interface. On the left, a script named 'script.py' contains the following code:

```

1  %% Prequisites
2  import numpy as np
3  # Create Matrix m
4  m=np.array([[3,6,2],[8,1,4],[5,9,7]])
5  # Get its shape
6  (nrows,ncols)=np.shape(m)

```

On the right, the Variable explorer shows the state of the variables:

Name	Type	Size	Value
m	Array of int32 (3, 3)		[[3 6 2]
m_flat	Array of int32 (9,)		[1 2 3 4 5 6 7 8 9]
m_flat_sort	NoneType	1	NoneType object
ncols	int	1	3
nrows	int	1	3

Below the Variable explorer, the IPython console shows the execution history:

```

Out[2]: array([[3, 6, 2],
              [8, 1, 4],
              [5, 9, 7]])

In [3]: m_flat=m.flatten()

In [4]: m_flat
Out[4]: array([3, 6, 2, 8, 1, 4, 5, 9, 7])

In [5]: m_flat_sort=m_flat.sort()

In [6]: m_flat_sort

In [7]: m_flat
Out[7]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])

```

Although both of these are numpy array methods `n=m.flatten()` assigns the flattened matrix to a new output variable, while `n=m.sort()` mutates `m` and does not assign anything to an output variable. This difference in behaviour can be very confusing to a beginner and more details will be given later when we look at defining a custom class and methods.

We can put this together to get:

```

1.  %% Prequisites
2.  import numpy as np
3.  # Create Matrix m
4.  m=np.array([[3,6,2],[8,1,4],[5,9,7]])
5.  # Get its shape
6.  (nrows,ncols)=np.shape(m)
7.  # Flatten the Matrix (output variable)
8.  m_flat=m.flatten()
9.  # Sort the Flattened Vector (acts on m directly)
10. m_flat.sort()
11. # Reshape Flattened and Sorted Vector
12. n=np.reshape(m_flat, (nrows,ncols))

```

The screenshot shows the Spyder Python IDE interface. The main editor contains the following Python code:

```

1  # %% Prerequisites
2  import numpy as np
3  # Create Matrix m
4  m=np.array([[3,6,2],[8,1,4],[5,9,7]])
5  # Get its shape
6  (nrows,ncols)=np.shape(m)
7  # Flatten the Matrix (output variable)
8  m_flat=m.flatten()
9  # Sort the Flattened Vector (acts on m directly)
10 m_flat.sort()
11 # Reshape Flattened and Sorted Vector
12 n=np.reshape(m_flat,(nrows,ncols))

```

The Variable Explorer on the right shows the following variables:

Name	Type	Size	Value
m	Array of int32	(3, 3)	[[3 6 2]
m_flat	Array of int32	(9,)	[1 2 3 4 5 6 7 8 9]
n	Array of int32	(3, 3)	[[1 2 3]
ncols	int	1	3
nrows	int	1	3

Below the code editor, two small windows display the matrix 'm' and the flattened matrix 'm_flat' as 2D arrays. The matrix 'm' is:

0	3	6	2
1	8	1	4
2	5	9	7

The flattened matrix 'm_flat' is:

0	1	2	3
1	4	5	6
2	7	8	9

The console output shows the IPython environment and the execution of the script:

```

Python 3.7.4 (default, Aug 9 2019, 18:34:13) [MSC v.1915 64
bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.10.1 -- An enhanced Interactive Python.

Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/
sorting.py', wdir='C:/Users/Phili/Documents/Python Scripts')

In [2]:

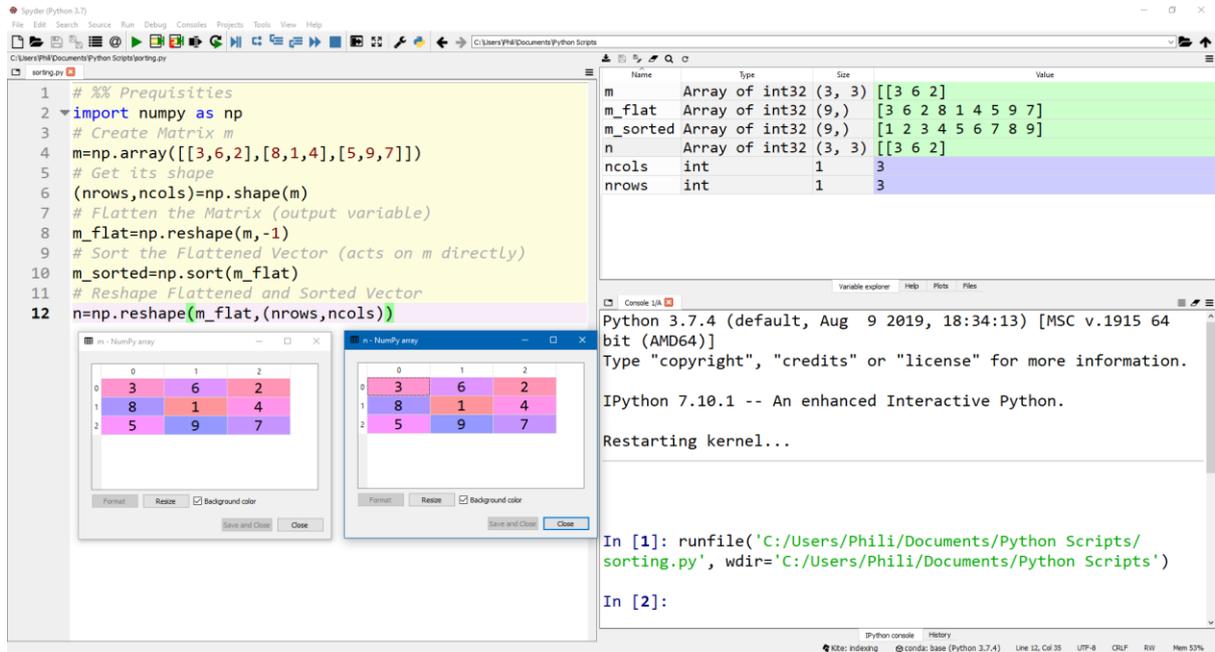
```

To flatten the matrix to a list we can also use the reshape function but instead of specifying the number of rows and then number of columns we can use `-1` to specify a vector for example `m_flat=np.reshape(m,-1)` and then we can sort using `m_sorted=np.sort(m_flat)`.

```

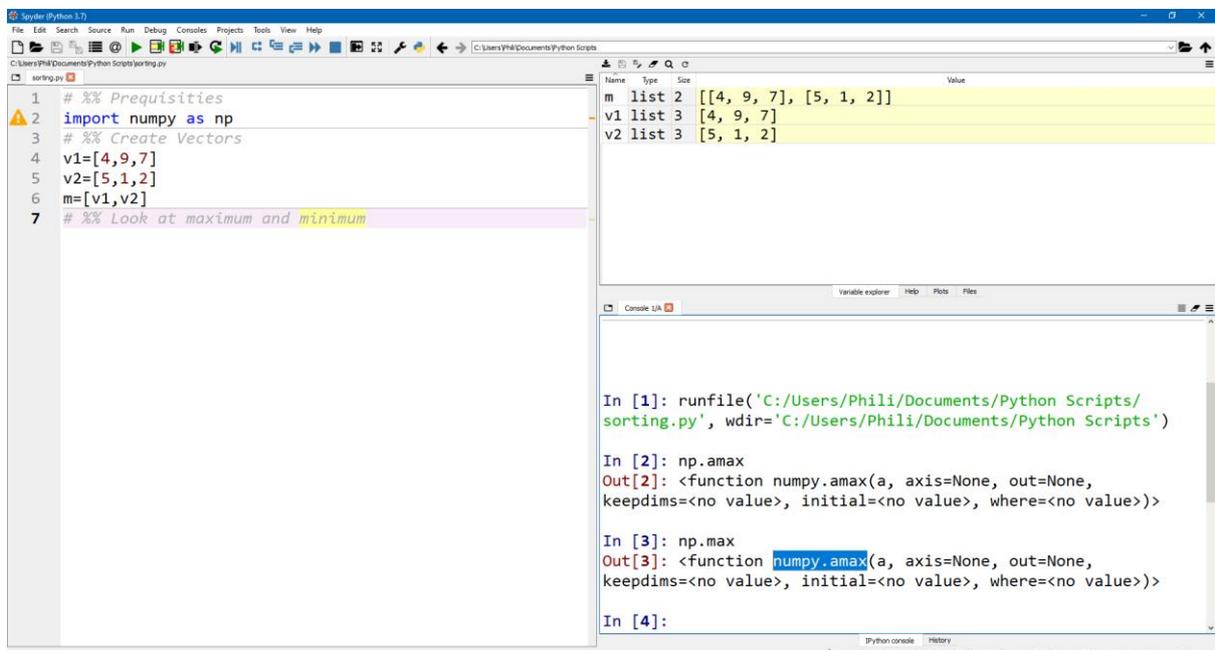
1. # %% Prerequisites
2. import numpy as np
3. # Create Matrix m
4. m=np.array([[3,6,2],[8,1,4],[5,9,7]])
5. # Get its shape
6. (nrows,ncols)=np.shape(m)
7. # Flatten the Matrix (output variable)
8. m_flat=np.reshape(m,-1)
9. # Sort the Flattened Vector (acts on m directly)
10. m_sorted=np.sort(m_flat)
11. # Reshape Flattened and Sorted Vector
12. n=np.reshape(m_flat,(nrows,ncols))

```



Associated with sorting is finding the minimum and maximum values in an array.

We can also look at the minimum and maximum value using the functions `np.amin()` and `np.amax()` respectively. These functions have the alias `np.min()` and `np.max()` respectively. We can verify this by calling the functions without following by parenthesis which will give details about the function. Note when we call `np.max` we get `<function numpy.amax(...)>` showing that it is an alias. We can also use `np.argmin()` and `np.argmax()` to find the index of the minimum and maximum argument respectively.



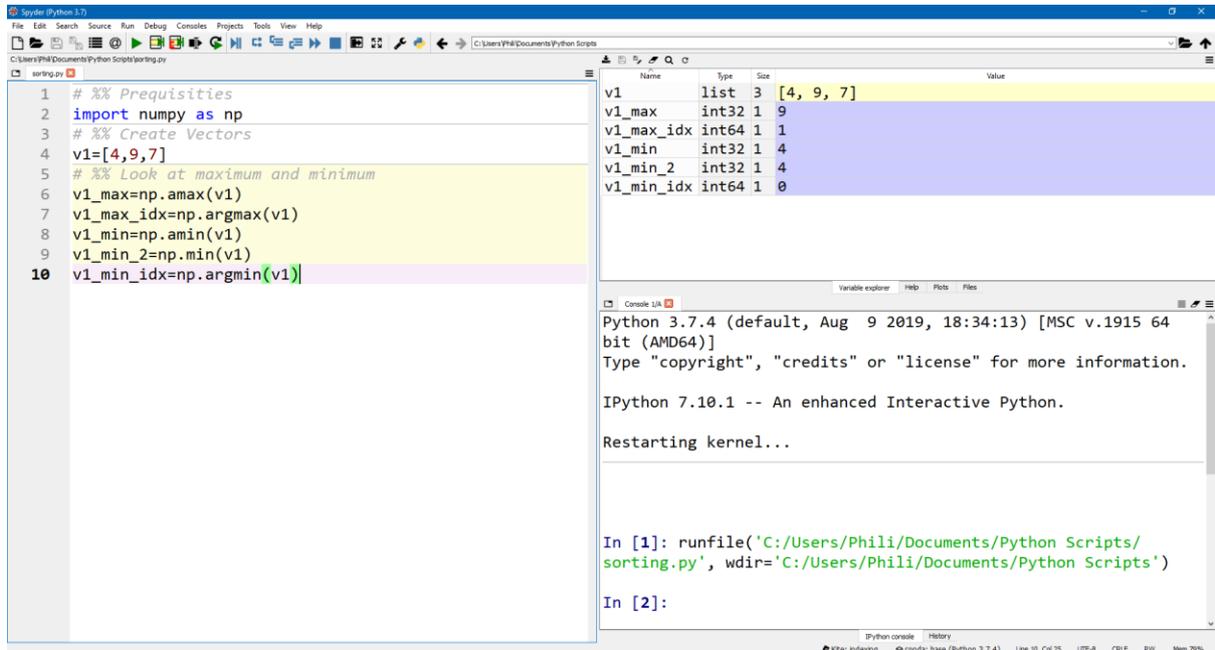
Let's create a simple vector and have a look at the minimum and maximum values using:

1. # %% Prerequisites
2. import numpy as np
3. # %% Create Vectors

```

4. v1=[4,9,7]
5. # %% Look at maximum and minimum
6. v1_max=np.amax(v1)
7. v1_max_idx=np.argmax(v1)
8. v1_min=np.amin(v1)
9. v1_min_2=np.min(v1)
10. v1_min_idx=np.argmin(v1)

```



$v1 = [4 \quad 9 \quad 7]$

$v1_max = 9$

$v1 = \begin{bmatrix} 4 & 9 & 7 \\ 0 & 1 & 2 \end{bmatrix}$

$v1_max_idx = 1$

$v1 = [4 \quad 9 \quad 7]$

$v1_min = 4$

$v1 = \begin{bmatrix} 4 & 9 & 7 \\ 0 & 1 & 2 \end{bmatrix}$

$v1_min_idx = 0$

When looking at a matrix we must once again select the axis to use. If we set `axis=0` we will once again keep columns together and then find the maximum value of each one.

$m1 = \begin{bmatrix} 4 & 9 & 7 \\ 5 & 1 & 2 \end{bmatrix}$

$m1 = \begin{bmatrix} 0 \\ 4 & 9 & 7 \\ 5 & 1 & 2 \\ 1 \end{bmatrix}$

$m1_max_axis0 = [5 \quad \dots \quad \dots]$

`m1_argmax_axis0 = [1]`

$$m1 = \begin{bmatrix} 4 & 9 & 7 \\ 5 & 1 & 2 \end{bmatrix}$$

$$m1 = \begin{bmatrix} 4 & \overset{0}{9} & 7 \\ 5 & \underset{1}{1} & 2 \end{bmatrix}$$

`m1_max_axis0 = [5 9 ...]`

`m1_argmax_axis0 = [1 0 ...]`

$$m1 = \begin{bmatrix} 4 & 9 & 7 \\ 5 & 1 & 2 \end{bmatrix}$$

$$m1 = \begin{bmatrix} 4 & 9 & \overset{0}{7} \\ 5 & 1 & \underset{1}{2} \end{bmatrix}$$

`m1_max_axis0 = [5 9 7]`

`m1_argmax_axis0 = [1 0 0]`

In other words `np.amax()` and `np.argmin()` when `axis=0`, it will act on each column of `m1` individually to compute a row of maximum values. We can type this into Python using the following. Note the output vector will be converted explicitly to a row using `np.reshape()`.

```
1. # %% Prerequisites
2. import numpy as np
3. # %% Create Vectors and Matrices
4. v1=np.array([4,9,7])
5. v2=np.array([5,1,2])
6. m1=np.array([v1,v2])
7. # %% Look at maximum and minimum
8. m1_max_axis0=np.amax(m1,axis=0)
9. m1_argmax_axis0=np.argmax(m1,axis=0)
10. (nrows,)=np.shape(m1_max_axis0)
11. m1_max_axis0=np.reshape(m1_max_axis0,(1,nrows))
12. m1_argmax_axis0=np.reshape(m1_argmax_axis0,(1,nrows))
```

```

1  # %% Prerequisites
2  import numpy as np
3  # %% Create Vectors and Matrices
4  v1=np.array([4,9,7])
5  v2=np.array([5,1,2])
6  m1=np.array([v1,v2])
7  # %% Look at maximum and minimum
8  m1_max_axis0=np.amax(m1,axis=0)
9  m1_argmax_axis0=np.argmax(m1,axis=0)
10 (nrows,)=np.shape(m1_max_axis0)
11 m1_max_axis0=np.reshape(m1_max_axis0,(1,nrows))
12 m1_argmax_axis0=np.reshape(m1_argmax_axis0,(1,nrows))

```

Name	Type	Size	Value
m1	Array of int32	(2, 3)	$\begin{bmatrix} 4 & 9 & 7 \\ 5 & 1 & 2 \end{bmatrix}$
m1_argmax_axis0	Array of int64	(1, 3)	$\begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$
m1_max_axis0	Array of int32	(1, 3)	$\begin{bmatrix} 5 & 9 & 7 \end{bmatrix}$
nrows	int	1	3
v1	Array of int32	(3,)	$[4 \ 9 \ 7]$
v2	Array of int32	(3,)	$[5 \ 1 \ 2]$

```

Python 3.7.4 (default, Aug 9 2019, 18:34:13) [MSC v.1915 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.10.1 -- An enhanced Interactive Python.

Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/
sorting.py', wdir='C:/Users/Phili/Documents/Python Scripts')

In [2]:

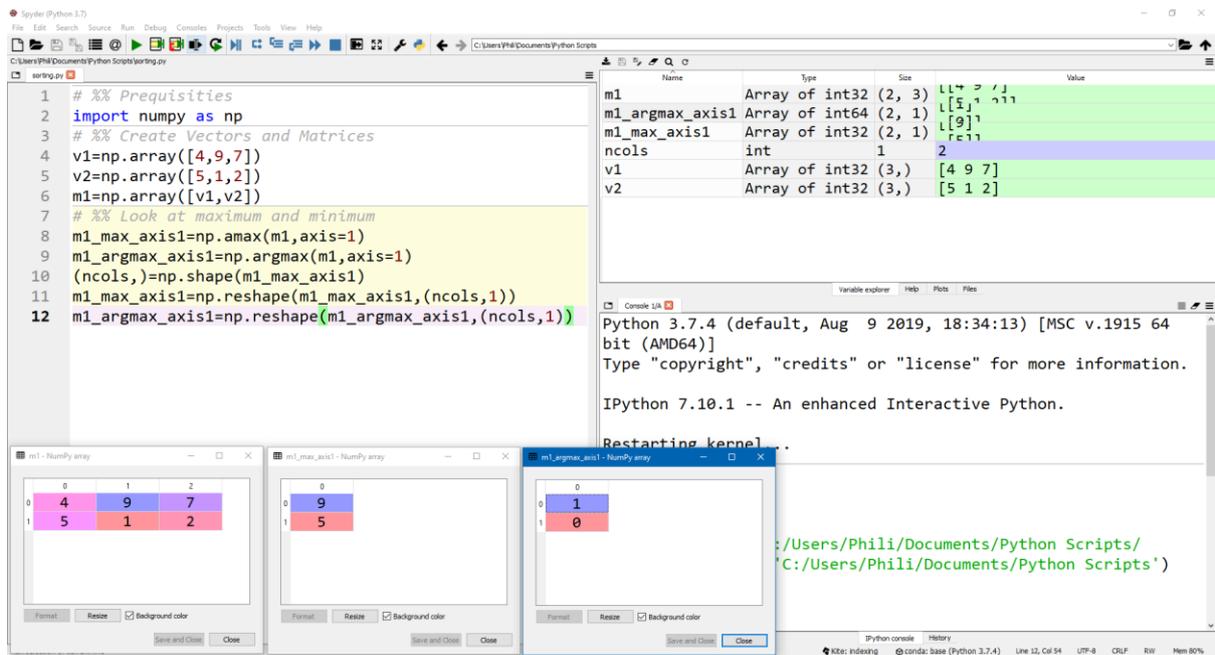
```

Note although we have looked for the maximum value along `axis=0` for vector `m1`, that `m1` is made using `v1` and `v2` and we have also found the maximum value between `v1` and `v2` for each index. This can be repeated for `axis=1` which will instead keep the rows together, find the maximum value of each row and return the maximum of each row as a column:

```

1.  # %% Prerequisites
2.  import numpy as np
3.  # %% Create Vectors and Matrices
4.  v1=np.array([4,9,7])
5.  v2=np.array([5,1,2])
6.  m1=np.array([v1,v2])
7.  # %% Look at maximum and minimum
8.  m1_max_axis1=np.amax(m1,axis=1)
9.  m1_argmax_axis1=np.argmax(m1,axis=1)
10. (ncols,)=np.shape(m1_max_axis1)
11. m1_max_axis1=np.reshape(m1_max_axis1,(ncols,1))
12. m1_argmax_axis1=np.reshape(m1_argmax_axis1,(ncols,1))

```



To find the absolute minimum or maximum it is possible to first act upon `axis=0` and then subsequently act upon the list generated.

```

1. # %% Prerequisites
2. import numpy as np
3. # %% Create Vectors and Matrices
4. v1=np.array([4,9,7])
5. v2=np.array([5,1,2])
6. m1=np.array([v1,v2])
7. # %% Look at maximum and minimum
8. m1_max_axis0=np.amax(m1,axis=0)
9. m1_argmax_axis0=np.argmax(m1,axis=0)
10. m1_max=np.amax(m1_max_axis0)
11. m1_argmaxrow=np.argmax(m1_argmax_axis0)
12. m1_argmaxcol=m1_argmax_axis0[m1_argmaxrow]
13. max_index=(m1_argmaxrow,m1_argmaxcol)
14. print(m1[max_index])
15. print(max_index)

```

```

1 # %% Prerequisites
2 import numpy as np
3 # %% Create Vectors and Matrices
4 v1=np.array([4,9,7])
5 v2=np.array([5,1,2])
6 m1=np.array([v1,v2])
7 # %% Look at maximum and minimum
8 m1_max_axis0=np.amax(m1,axis=0)
9 m1_argmax_axis0=np.argmax(m1,axis=0)
10 m1_max=np.amax(m1_max_axis0)
11 m1_argmaxrow=np.argmax(m1_argmax_axis0)
12 m1_argmaxcol=m1_argmax_axis0[m1_argmaxrow]
13 max_index=(m1_argmaxrow,m1_argmaxcol)
14 print(m1[max_index])
15 print(max_index)

```

Name	Type	Size	Value
m1	Array of int32 (2, 3)		$\begin{bmatrix} 4 & 9 & 7 \\ 5 & 1 & 2 \end{bmatrix}$
m1_argmax_axis0	Array of int64 (3,)		$[1 \ 0 \ 0]$
m1_argmaxcol	int64	1	1
m1_argmaxrow	int64	1	0
m1_max	int32	1	9
m1_max_axis0	Array of int32 (3,)		$[5 \ 9 \ 7]$
max_index	tuple	2	(0, 1)
v1	Array of int32 (3,)		$[4 \ 9 \ 7]$
v2	Array of int32 (3,)		$[5 \ 1 \ 2]$

Console output:
Type "copyright", "credits" or "license" for more information.
IPython 7.10.1 -- An enhanced Interactive Python.
Restarting kernel...
In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/sorting.py', wdir='C:/Users/Phili/Documents/Python Scripts')
9
(0, 1)
In [2]:

The functions `np.maximum(m1,m2)` and `np.minimum(m1,m2)` can be used to directly find the maximum and minimum values for each matching element in `m1` and `m2`. In this case if we look at the maximum value of

$$m1 = \begin{bmatrix} 1 & 8 \\ 6 & 3 \end{bmatrix}, m2 = \begin{bmatrix} 5 & 2 \\ 7 & 4 \end{bmatrix}, m_max = \begin{bmatrix} 5 & 8 \\ 7 & 4 \end{bmatrix}$$

1. # %% Prerequisites
2. `import numpy as np`
3. # %% Create Vectors and Matrices
4. `m1=np.array([[1,8],[6,3]])`
5. `m2=np.array([[5,2],[7,4]])`
6. # %% Look at Maximum
7. `m_max=np.maximum(m1,m2)`

```

1 # %% Prerequisites
2 import numpy as np
3 # %% Create Vectors and Matrices
4 m1=np.array([[1,8],[6,3]])
5 m2=np.array([[5,2],[7,4]])
6 # %% Look at Maximum
7 m_max=np.maximum(m1,m2)

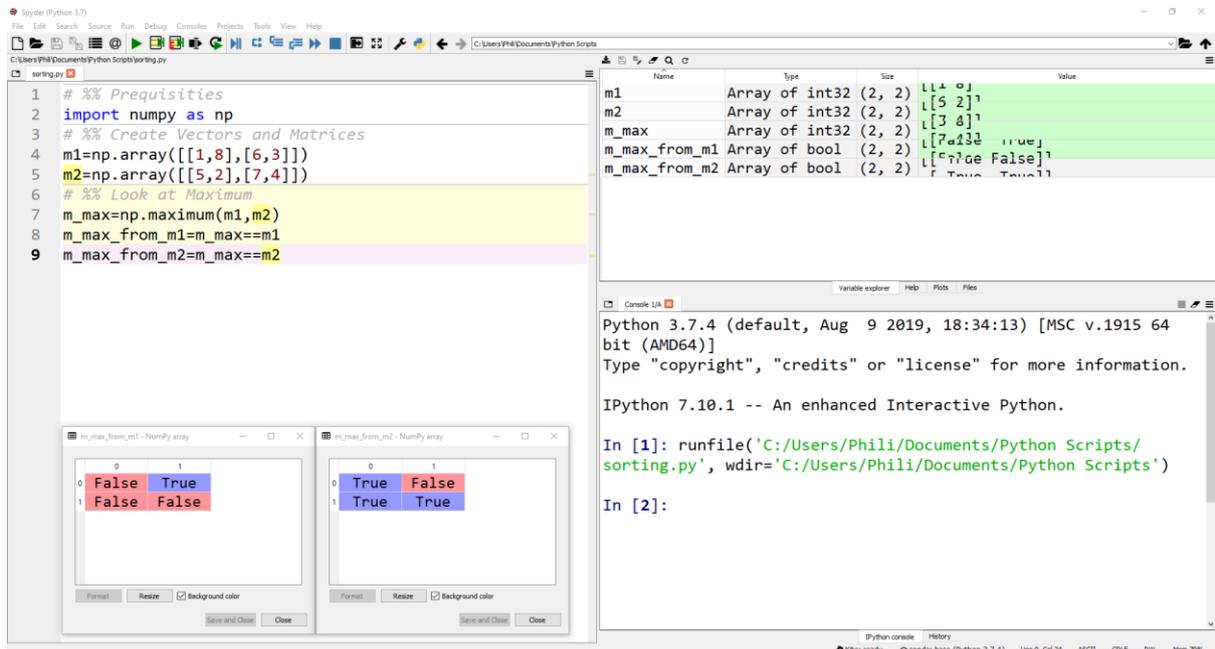
```

Name	Type	Size	Value
m1	Array of int32 (2, 2)		$\begin{bmatrix} 1 & 8 \\ 6 & 3 \end{bmatrix}$
m2	Array of int32 (2, 2)		$\begin{bmatrix} 5 & 2 \\ 7 & 4 \end{bmatrix}$
m_max	Array of int32 (2, 2)		$\begin{bmatrix} 5 & 8 \\ 7 & 4 \end{bmatrix}$

Console output:
Python 3.7.4 (default, Aug 9 2019, 18:34:13) [MSC v.1915 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.
IPython 7.10.1 -- An enhanced Interactive Python.
Restarting kernel...
In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/sorting.py', wdir='C:/Users/Phili/Documents/Python Scripts')
In [2]:

We can also use conditional logic to determine whether we got the element from `m1` or from `m2`.

8. `m_max_from_m1=m_max==m1`
9. `m_max_from_m2=m_max==m2`



Statistical Functions

So far, we have seen sorting and finding the minimum and maximum value in a vector and matrix where the operation on the matrix used the additional keyword input argument `axis=0` which kept columns together and acted upon them. This could be changed to `axis=1` which instead kept rows together and acted upon them. There are several additional statistical functions which follow a similar convention. `np.sum()` will find the sum of all values in a vector or along an axis in a matrix.

$$v1 = [4 \quad 9 \quad 7]$$

$$\text{sum}_v1 = 4 + 9 + 7 = 20$$

$$m1 = \begin{bmatrix} 4 & 9 & 7 \\ 5 & 1 & 3 \end{bmatrix}$$

$$\text{sum}_m1_{\text{axis}0} = [4+5 \quad 9+1 \quad 7+3]$$

$$\text{sum}_m1_{\text{axis}0} = [9 \quad 10 \quad 10]$$

$$m1 = \begin{bmatrix} 4 & 9 & 7 \\ 5 & 1 & 3 \end{bmatrix}$$

$$\text{sum}_m1_{\text{axis}1} = [4+9+7 \quad 5+1+3]$$

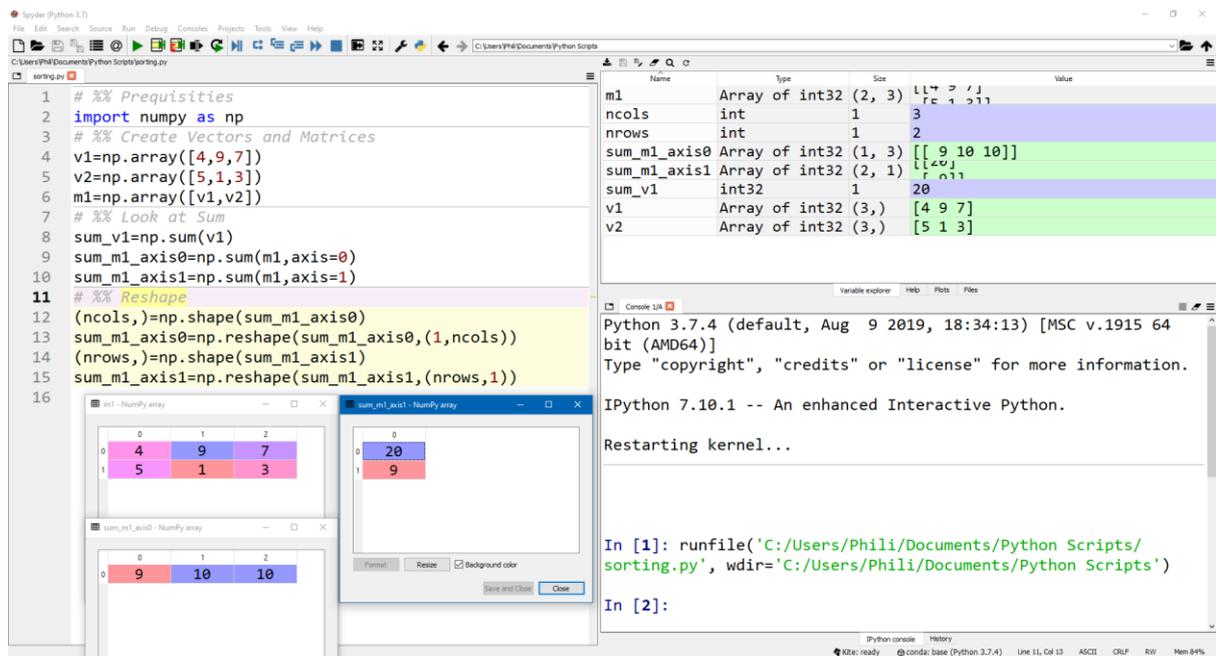
$$\text{sum}_m1_{\text{axis}1} = [20 \quad 9]$$

1. `# %% Prerequisites`
2. `import numpy as np`
3. `# %% Create Vectors and Matrices`
4. `v1=np.array([4,9,7])`
5. `v2=np.array([5,1,3])`

```

6. m1=np.array([v1,v2])
7. # %% Look at Sum
8. sum_v1=np.sum(v1)
9. sum_m1_axis0=np.sum(m1,axis=0)
10. sum_m1_axis1=np.sum(m1,axis=1)
11. # %% Reshape
12. (ncols,)=np.shape(sum_m1_axis0)
13. sum_m1_axis0=np.reshape(sum_m1_axis0,(1,ncols))
14. (nrows,)=np.shape(sum_m1_axis1)
15. sum_m1_axis1=np.reshape(sum_m1_axis1,(nrows,1))

```



We also have the function `np.mean()` which computes the sum divided by the number of values used to compute the sum.

$$v1 = [4 \quad 9 \quad 7]$$

$$\text{sum_v1} = 4 + 9 + 7 = 20$$

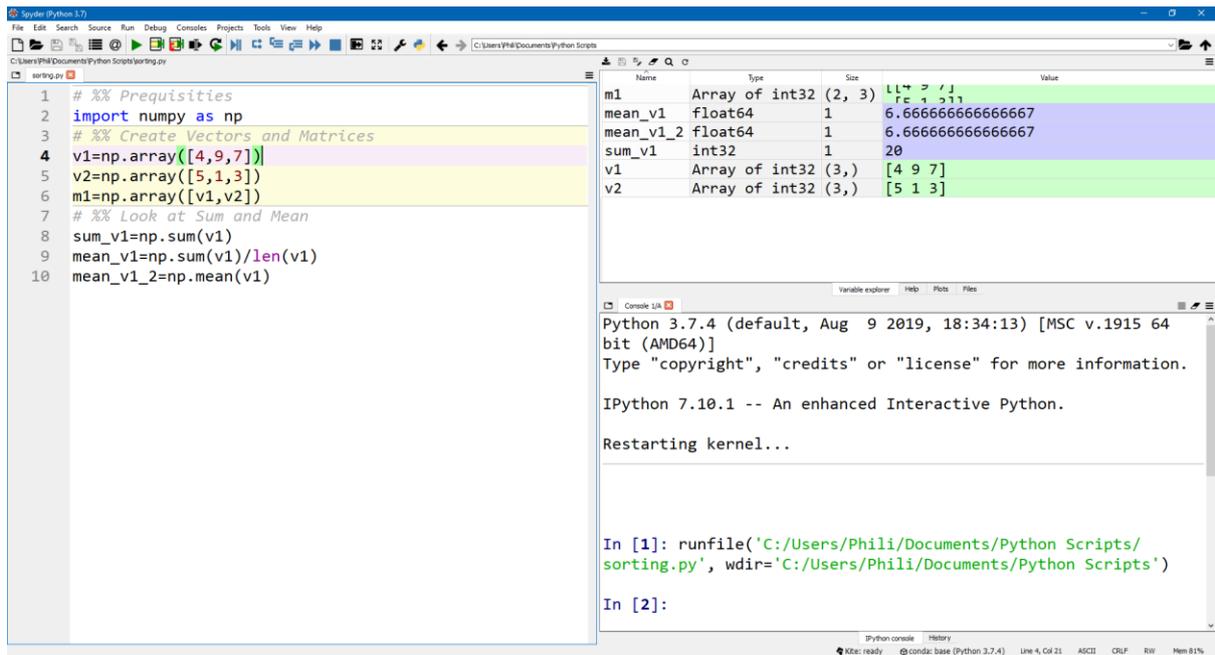
$$\text{len_v1} = 3$$

$$\text{mean_v1} = \frac{20}{3} = 6.6667$$

```

1. # %% Prerequisites
2. import numpy as np
3. # %% Create Vectors and Matrices
4. v1=np.array([4,9,7])
5. v2=np.array([5,1,3])
6. m1=np.array([v1,v2])
7. # %% Look at Sum and Mean
8. sum_v1=np.sum(v1)
9. mean_v1=np.sum(v1)/len(v1)
10. mean_v1_2=np.mean(v1)

```



$$m_1 = \begin{bmatrix} 4 & 9 & 7 \\ 5 & 1 & 3 \end{bmatrix}$$

$$(nrows, ncols) = (2, 3)$$

$$\text{sum_m1_axis0} = [4+5 \quad 9+1 \quad 7+3] = [9 \quad 10 \quad 10]$$

$$\text{mean_sum1_axis0} = \text{sum_m1_axis0}/nrows = \begin{bmatrix} 9 & 10 & 10 \\ 2 & 2 & 2 \end{bmatrix} = [4.5 \quad 5.0 \quad 5.0]$$

$$m_1 = \begin{bmatrix} 4 & 9 & 7 \\ 5 & 1 & 3 \end{bmatrix}$$

$$(nrows, ncols) = (2, 3)$$

$$\text{sum_m1_axis1} = \begin{bmatrix} 4+9+7 \\ 5+1+3 \end{bmatrix} = \begin{bmatrix} 20 \\ 9 \end{bmatrix}$$

$$\text{mean_sum1_axis1} = \text{sum_m1_axis1}/ncols = \begin{bmatrix} 20 \\ 3 \\ 9 \\ 3 \end{bmatrix} = \begin{bmatrix} 6.666666666666667 \\ 3.0 \end{bmatrix}$$

```

1. # %% Prerequisites
2. import numpy as np
3. # %% Create Vectors and Matrices
4. v1=np.array([4,9,7])
5. v2=np.array([5,1,3])
6. m1=np.array([v1,v2])
7. # %% Look at Sum and Mean
8. (nrows,ncols)=np.shape(m1)
9. sum_m1_axis0=np.sum(m1,axis=0)
10. sum_m1_axis0=np.reshape(sum_m1_axis0,(1,ncols))
11. mean_m1_axis0=sum_m1_axis0/nrows
12. mean_m1_axis0_2=np.mean(m1,axis=0)
13. mean_m1_axis0_2=np.reshape(mean_m1_axis0_2,(1,ncols))

```

```

1 # %% Prerequisites
2 import numpy as np
3 # %% Create Vectors and Matrices
4 v1=np.array([4,9,7])
5 v2=np.array([5,1,3])
6 m1=np.array([v1,v2])
7 # %% Look at Sum and Mean
8 (nrows,ncols)=np.shape(m1)
9 sum_m1_axis0=np.sum(m1,axis=0)
10 sum_m1_axis0=np.reshape(sum_m1_axis0,(1,ncols))
11 mean_m1_axis0=sum_m1_axis0/nrows
12 mean_m1_axis0_2=np.mean(m1,axis=0)
13 mean_m1_axis0_2=np.reshape(mean_m1_axis0_2,(1,ncols))

```

Name	Type	Size	Value
m1	Array of int32	(2, 3)	$\begin{bmatrix} 4 & 9 & 7 \\ 5 & 1 & 3 \end{bmatrix}$
mean_m1_axis0	Array of float64	(1, 3)	$\begin{bmatrix} 4.5 & 5. & 5. \end{bmatrix}$
mean_m1_axis0_2	Array of float64	(1, 3)	$\begin{bmatrix} 4.5 & 5. & 5. \end{bmatrix}$
ncols	int	1	3
nrows	int	1	2
sum_m1_axis0	Array of int32	(1, 3)	$\begin{bmatrix} 9 & 10 & 10 \end{bmatrix}$
v1	Array of int32	(3,)	$\begin{bmatrix} 4 & 9 & 7 \end{bmatrix}$
v2	Array of int32	(3,)	$\begin{bmatrix} 5 & 1 & 3 \end{bmatrix}$

1. # %% Prerequisites
2. `import numpy as np`
3. # %% Create Vectors and Matrices
4. `v1=np.array([4,9,7])`
5. `v2=np.array([5,1,3])`
6. `m1=np.array([v1,v2])`
7. # %% Look at Sum and Mean
8. `(nrows,ncols)=np.shape(m1)`
9. `sum_m1_axis1=np.sum(m1,axis=1)`
10. `sum_m1_axis1=np.reshape(sum_m1_axis1,(nrows,1))`
11. `mean_m1_axis1=sum_m1_axis1/ncols`
12. `mean_m1_axis1_2=np.mean(m1,axis=1)`
13. `mean_m1_axis1_2=np.reshape(mean_m1_axis1_2,(nrows,1))`

```

1 # %% Prerequisites
2 import numpy as np
3 # %% Create Vectors and Matrices
4 v1=np.array([4,9,7])
5 v2=np.array([5,1,3])
6 m1=np.array([v1,v2])
7 # %% Look at Sum and Mean
8 (nrows,ncols)=np.shape(m1)
9 sum_m1_axis1=np.sum(m1,axis=1)
10 sum_m1_axis1=np.reshape(sum_m1_axis1,(nrows,1))
11 mean_m1_axis1=sum_m1_axis1/ncols
12 mean_m1_axis1_2=np.mean(m1,axis=1)
13 mean_m1_axis1_2=np.reshape(mean_m1_axis1_2,(nrows,1))

```

Name	Type	Size	Value
m1	Array of int32	(2, 3)	$\begin{bmatrix} 4 & 9 & 7 \\ 5 & 1 & 3 \end{bmatrix}$
mean_m1_axis1	Array of float64	(2, 1)	$\begin{bmatrix} 6.66666667 \\ 3. \end{bmatrix}$
mean_m1_axis1_2	Array of float64	(2, 1)	$\begin{bmatrix} 6.66666667 \\ 3. \end{bmatrix}$
ncols	int	1	3
nrows	int	1	2
sum_m1_axis1	Array of int32	(2, 1)	$\begin{bmatrix} 14 \\ 8 \end{bmatrix}$
v1	Array of int32	(3,)	$\begin{bmatrix} 4 & 9 & 7 \end{bmatrix}$
v2	Array of int32	(3,)	$\begin{bmatrix} 5 & 1 & 3 \end{bmatrix}$

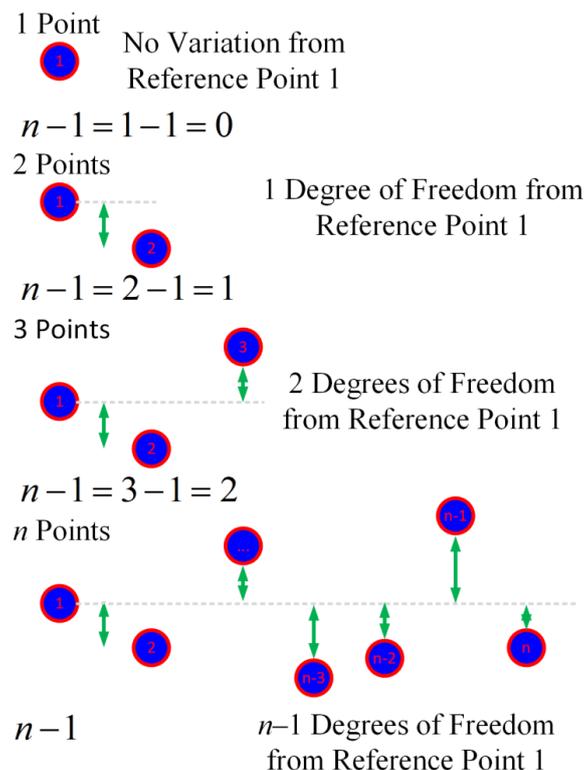
In addition to the mean μ_x

$$\mu_x = \frac{\sum_{i=0}^n x}{n}$$

A metric which expresses how much members of a group differ from the mean value is useful. Because by definition the sum of the difference of the mean and each value will equal 0 we instead look at the difference of each value to the mean and square it.

$$var = \frac{\sum_{i=0}^n (x - \mu_x)^2}{n}$$

Here the denominator n is expressed using 0 degrees of freedom however for a dataset spanning across a single dimension such as the measurement of a length it is more typical to use 1 degree of freedom. This is because what we typically measure and analyse is the mean of a sample and not the true mean of the population being studied (which would typically require measurement of the sample infinite times). In other words, if I took three measurements 1,2 and x and told you that I obtained a mean of 2 you would be able to determine the last number x would be 2 so the sample set and the sample mean provides $n - 1$ pieces of information opposed to n pieces of information because the definition of the sample mean contains shares the same information as the last value. Practically we can see how this works if we take the simplest of cases where we have only a single sample and let's use the denominator with no degrees of freedom, the numerator in the equation above would be 0 as the mean of a single sample is the mean and the denominator would be 1. If we use the variance to estimate the error, we would have an error of 0 however in such a scenario we only measured a value once and don't know if our measurement was just a one-off by chance in a distribution or whether we measured a static value correctly or made some other experimental error. Conversely if we use one degree of freedom the denominator would therefore be 0 and dividing by 0 means the error is infinite or undefined.



Schematically $n - 1$ can be depicted below where each data point is a bead on a string. If we take a single point as a reference point (in this case the 0th point) and compare every other point to it. We see we will get $n - 1$ green lines to compare it to. Now if we take 1 degree of freedom and compare the variance to the mean, we see that the units of the mean μ_x correspond to the unit of the object x being measured however the variance corresponds to squared units.

$$\mu_x = \frac{\sum_{i=0}^n x}{n}$$

$$var = \frac{\sum_{i=0}^n (x - \mu_x)^2}{n - ddof}$$

It is therefore more common to measure the square root of this value to keep dimensionality and this is the definition of the standard deviation:

$$std = \sqrt{\frac{\sum_{i=0}^n (x - \mu_x)^2}{n - ddof}} = \sqrt{var}$$

The smaller the standard deviation, the smaller the spread within the measurements.

$$v1 = [4 \quad 9 \quad 7]$$

From earlier:

$$\text{len_v1} = 3$$

$$\text{mean_v1} = \frac{20}{3} = 6.6667$$

$$\text{var_v1} = \frac{(4 - \text{mean_v1})^2 + (9 - \text{mean_v1})^2 + (7 - \text{mean_v1})^2}{\text{len_v1} - 1}$$

$$\text{var_v1} = \frac{(4 - 6.6667)^2 + (9 - 6.6667)^2 + (7 - 6.6667)^2}{3 - 1}$$

$$\text{var_v1} = \frac{7.1113 + 5.444 + 0.1111}{2} = \frac{12.6667}{2} = 6.3333$$

$$\text{std_v1} = \sqrt{\text{var_v1}}$$

$$\text{std_v1} = \sqrt{6.3333} = 2.5166$$

In Python, this can be directly calculated using:

```

1. # %% Prerequisites
2. import numpy as np
3. # %% Create Vectors and Matrices
4. v1=np.array([4,9,7])
5. v2=np.array([5,1,3])
6. m1=np.array([v1,v2])
7. # %% Look at Var and Std
8. var_v1=np.var(v1,ddof=1)
9. std_m1=np.std(v1,ddof=1)

```

Which gives the same value as expected:

```

1  # %% Prerequisites
2  import numpy as np
3  # %% Create Vectors and Matrices
4  v1=np.array([4,9,7])
5  v2=np.array([5,1,3])
6  m1=np.array([v1,v2])
7  # %% Look at Var and Std
8  var_v1=np.var(v1,ddof=1)
9  std_m1=np.std(v1,ddof=1)

```

Name	Type	Size	Value
m1	Array of int32	(2, 3)	$\begin{bmatrix} 4 & 9 & 7 \\ 5 & 1 & 3 \end{bmatrix}$
std_m1	float64	1	2.516611478423583
v1	Array of int32	(3,)	[4 9 7]
v2	Array of int32	(3,)	[5 1 3]
var_v1	float64	1	6.333333333333333

```

Python 3.7.4 (default, Aug 9 2019, 18:34:13) [MSC v.1915 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.10.1 -- An enhanced Interactive Python.

Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/statistics.py', wdir='C:/Users/Phili/Documents/Python Scripts')
In [2]:

```

Note the keyword argument `ddof` is assigned to `1`. If these functions are called without assigning `ddof` then its default value of `0` is taken.

```

1  # %% Prerequisites
2  import numpy as np
3  # %% Create Vectors and Matrices
4  v1=np.array([4,9,7])
5  v2=np.array([5,1,3])
6  m1=np.array([v1,v2])
7  # %% Look at Var and Std
8  var_v1=np.var(v1)
9  std_m1=np.std(v1)

```

Name	Type	Size	Value
m1	Array of int32	(2, 3)	$\begin{bmatrix} 4 & 9 & 7 \\ 5 & 1 & 3 \end{bmatrix}$
std_m1	float64	1	2.0548046676563256
v1	Array of int32	(3,)	[4 9 7]
v2	Array of int32	(3,)	[5 1 3]
var_v1	float64	1	4.222222222222222

```

Python 3.7.4 (default, Aug 9 2019, 18:34:13) [MSC v.1915 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.10.1 -- An enhanced Interactive Python.

Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/statistics.py', wdir='C:/Users/Phili/Documents/Python Scripts')
In [2]:

```

When used on a matrix once again the keyword input argument `axis` can be used which has a default value of `0`. Once again when `axis=0` we keep the columns together and act upon them and when `axis=1` we keep the rows together and act upon them.

```

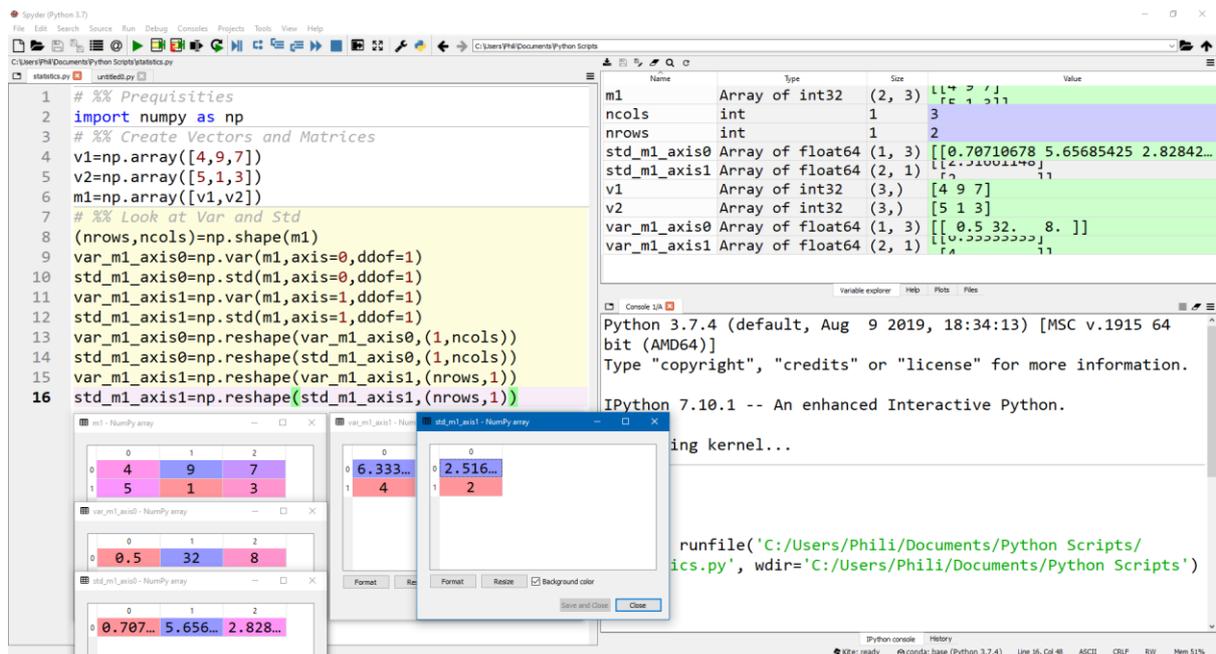
1. # %% Prerequisites
2. import numpy as np
3. # %% Create Vectors and Matrices
4. v1=np.array([4,9,7])
5. v2=np.array([5,1,3])
6. m1=np.array([v1,v2])

```

```

7. # %% Look at Var and Std
8. (nrows,ncols)=np.shape(m1)
9. var_m1_axis0=np.var(m1,axis=0,ddof=1)
10. std_m1_axis0=np.std(m1,axis=0,ddof=1)
11. var_m1_axis1=np.var(m1,axis=1,ddof=1)
12. std_m1_axis1=np.std(m1,axis=1,ddof=1)
13. var_m1_axis0=np.reshape(var_m1_axis0,(1,ncols))
14. std_m1_axis0=np.reshape(std_m1_axis0,(1,ncols))
15. var_m1_axis1=np.reshape(var_m1_axis1,(nrows,1))
16. std_m1_axis1=np.reshape(std_m1_axis1,(nrows,1))

```

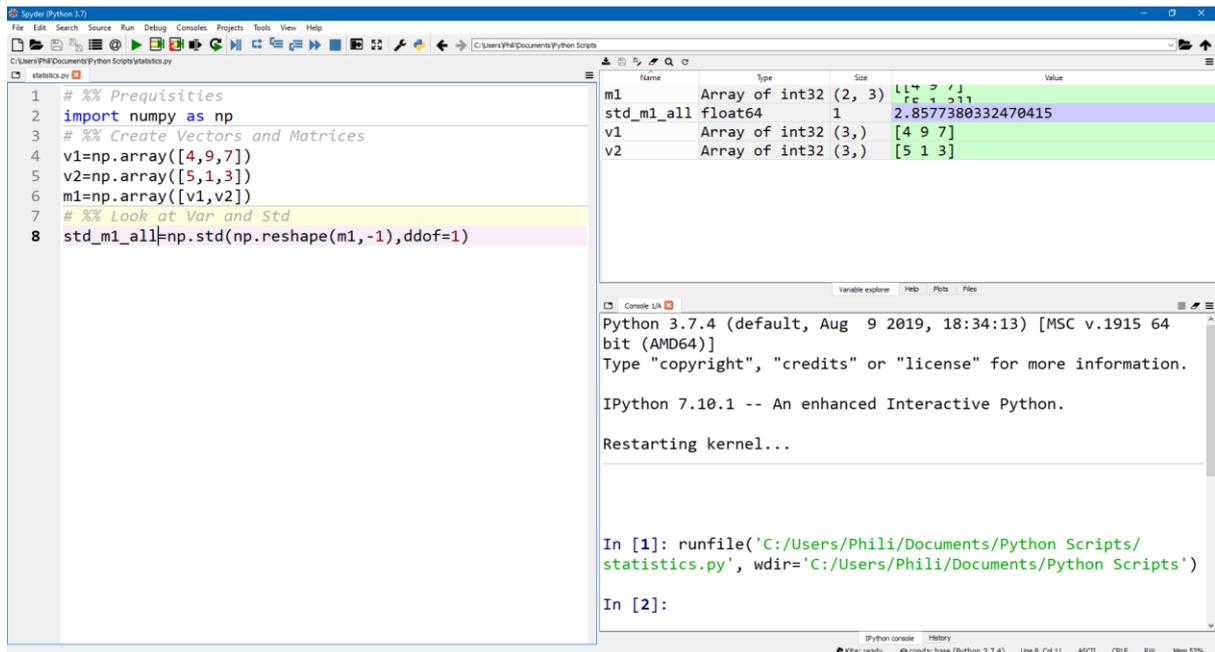


To find the standard deviation of the entire matrix we can reshape the matrix into a list and then find the standard deviation of the list.

```

1. # %% Prerequisites
2. import numpy as np
3. # %% Create Vectors and Matrices
4. v1=np.array([4,9,7])
5. v2=np.array([5,1,3])
6. m1=np.array([v1,v2])
7. # %% Look at Var and Std
8. std_m1_all=np.std(np.reshape(m1,-1),ddof=1)

```



So far, we have calculated both the mean and the standard deviation as it is very common to use both to describe a dataset. Moreover, as both have the same dimensionality of the data values being measured we typically use `mean±std` to describe a dataset. For a small dataset, the mean is very susceptible to being skewed by outliers and in such cases, it may be more appropriate to use the median.

$$m = \begin{bmatrix} 1 & 3 & 4 & 7 \\ 2 & 8 & 9 & 100 \\ 5 & 6 & 10 & 11 \end{bmatrix}$$

If we look at the mean acting along the columns (`axis=0`).

$$m = \begin{bmatrix} 1 & 3 & 4 & 7 \\ 2 & 8 & 9 & 100 \\ 5 & 6 & 10 & 11 \end{bmatrix}$$

$$m_mean_axis0 = [2.6667 \quad 5.6667 \quad 7.6667 \quad 39.3333]$$

We see that the outlier in this case `100` has severely influenced the mean:

$$m_mean_axis0 = [2.6667 \quad 5.6667 \quad 7.6667 \quad 39.3333]$$

If on the other hand, we sort along the columns (`axis=0`).

$$m = \begin{bmatrix} 1 & 3 & 4 & 7 \\ 2 & 8 & 9 & 100 \\ 5 & 6 & 10 & 11 \end{bmatrix}$$

$$m_sorted_axis0 = \begin{bmatrix} 1 & 3 & 4 & 7 \\ 2 & 6 & 9 & 11 \\ 5 & 8 & 10 & 100 \end{bmatrix}$$

We can then take the median or middle value along the sorted columns (`axis=0`).

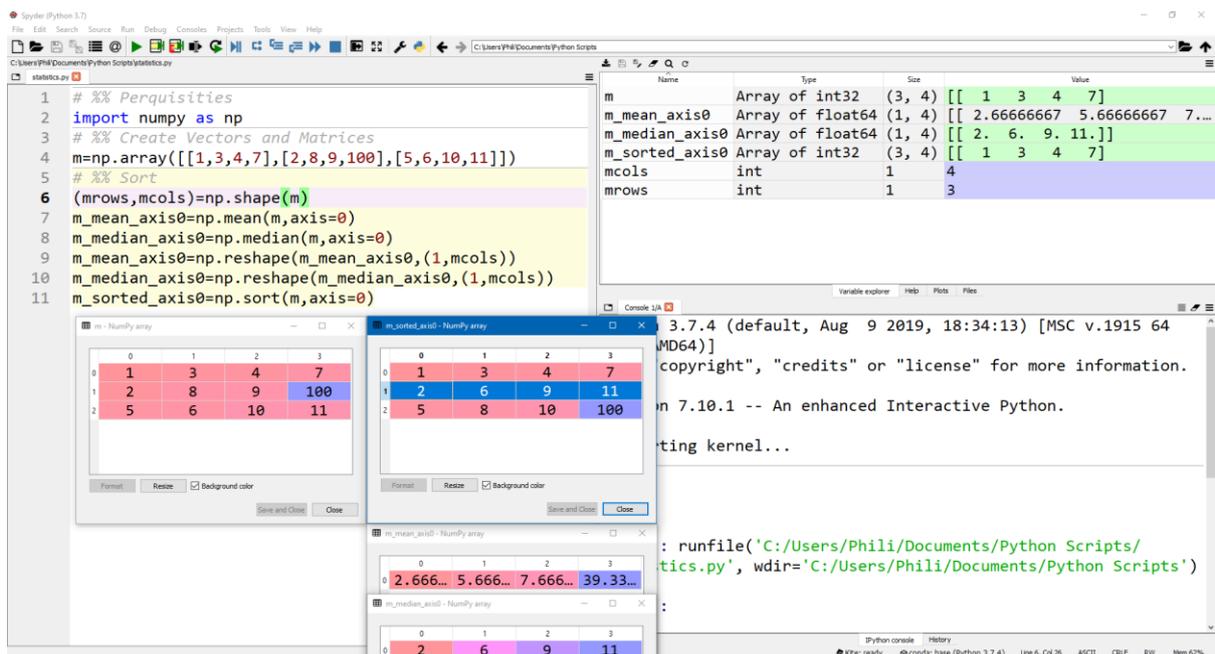
$$m_median_axis0 = [2 \quad 6 \quad 9 \quad 11] = [2 \quad 6 \quad 9 \quad 11]$$

Which is less influenced by the outlier.

```

1. # %% Perquisites
2. import numpy as np
3. # %% Create Vectors and Matrices
4. m=np.array([[1,3,4,7],[2,8,9,100],[5,6,10,11]])
5. # %% Sort
6. (mrows,mcols)=np.shape(m)
7. m_mean_axis0=np.mean(m,axis=0)
8. m_median_axis0=np.median(m,axis=0)
9. m_mean_axis0=np.reshape(m_mean_axis0,(1,mcols))
10. m_median_axis0=np.reshape(m_median_axis0,(1,mcols))
11. m_sorted_axis0=np.sort(m,axis=0)

```



$$m = \begin{bmatrix} 1 & 3 & 4 & 7 \\ 2 & 8 & 9 & 100 \\ 5 & 6 & 10 & 11 \end{bmatrix}$$

If we look at the mean acting along the columns ($axis=1$).

$$m = \begin{bmatrix} 1 & 3 & 4 & 7 \\ 2 & 8 & 9 & 100 \\ 5 & 6 & 10 & 11 \end{bmatrix}$$

$$m_mean_axis1 = \begin{bmatrix} 3.75 \\ 29.75 \\ 8 \end{bmatrix} = \begin{bmatrix} 3.75 \\ 29.75 \\ 8 \end{bmatrix}$$

We see that the outlier in this case 100 has severely influenced the mean:

If on the other hand, we sort along the columns ($axis=1$).

$$m = \begin{bmatrix} 1 & 3 & 4 & 7 \\ 2 & 8 & 9 & 100 \\ 5 & 6 & 10 & 11 \end{bmatrix}$$

$$m_sorted_axis1 = \begin{bmatrix} 1 & 3 & 4 & 7 \\ 2 & 8 & 9 & 100 \\ 5 & 6 & 10 & 11 \end{bmatrix}$$

We can then take the median or middle value along the sorted columns (`axis=1`).

$$m_sorted_axis1 = \begin{bmatrix} 1 & 3 & 4 & 7 \\ 2 & 8 & 9 & 100 \\ 5 & 6 & 10 & 11 \end{bmatrix}$$

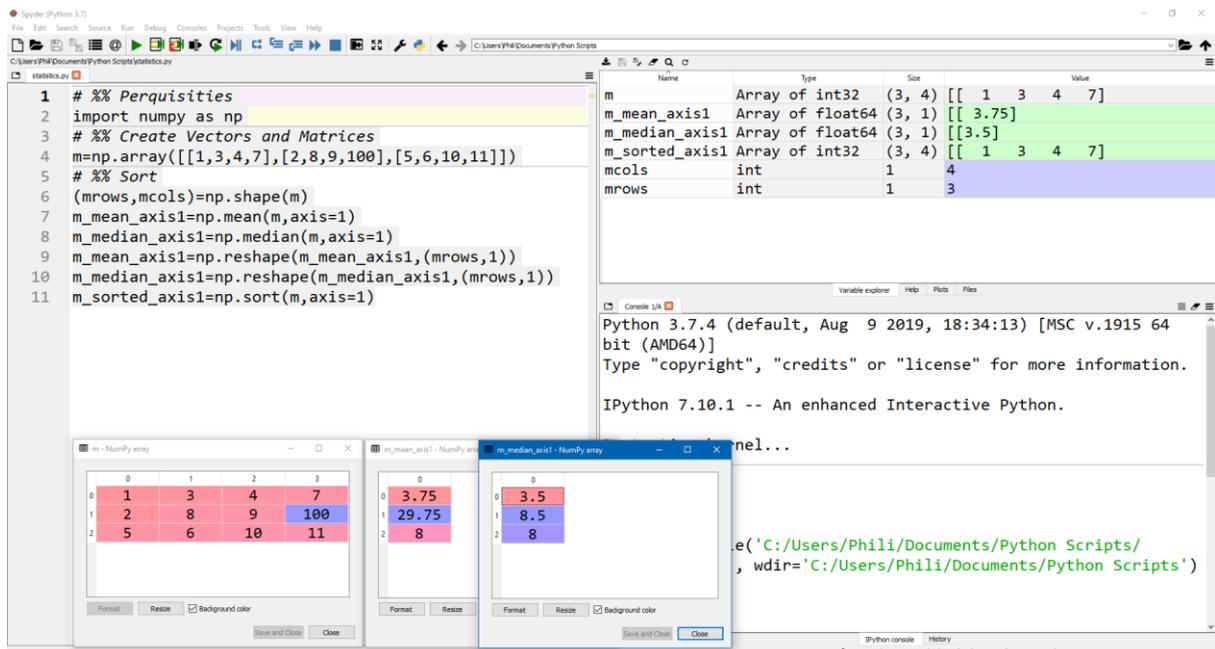
$$m_median_axis1 = \begin{bmatrix} \frac{3+4}{2} \\ \frac{8+9}{2} \\ \frac{6+10}{2} \end{bmatrix} = \begin{bmatrix} 3.5 \\ 8.5 \\ 8.0 \end{bmatrix} = \begin{bmatrix} 3.5 \\ 8.0 \end{bmatrix}$$

Which is less influenced by the outlier:

```

1. # %% Perquisites
2. import numpy as np
3. # %% Create Vectors and Matrices
4. m=np.array([[1,3,4,7],[2,8,9,100],[5,6,10,11]])
5. # %% Sort
6. (mrows,mcols)=np.shape(m)
7. m_mean_axis1=np.mean(m,axis=1)
8. m_median_axis1=np.median(m,axis=1)
9. m_mean_axis1=np.reshape(m_mean_axis1,(mrows,1))
10. m_median_axis1=np.reshape(m_median_axis1,(mrows,1))
11. m_sorted_axis1=np.sort(m,axis=1)

```



`np.prod()` will find the product of all values in a vector or along an axis in a matrix and has a similar form to `np.sum()`.

$$v1 = [4 \quad 9 \quad 7]$$

$$\text{prod_v1} = 4 * 9 * 7 = 252$$

$$m1 = \begin{bmatrix} 4 & 9 & 7 \\ 5 & 1 & 3 \end{bmatrix}$$

$$\text{prod_m1_axis0} = [4*5 \quad 9*1 \quad 7*3]$$

$$\text{prod_m1_axis0} = [20 \quad 9 \quad 21]$$

$$m1 = \begin{bmatrix} 4 & 9 & 7 \\ 5 & 1 & 3 \end{bmatrix}$$

$$\text{prod_m1_axis1} = \begin{bmatrix} 4*9*7 \\ 5*1*3 \end{bmatrix}$$

$$\text{prod_m1_axis1} = \begin{bmatrix} 252 \\ 15 \end{bmatrix}$$

```

1. # %% Perquisites
2. import numpy as np
3. # %% Create Vectors and Matrices
4. v1=np.array([4,9,7])
5. v2=np.array([5,1,3])
6. m1=np.array([v1,v2])
7. # %% Look at Prod
8. prod_v1=np.prod(v1)
9. prod_m1_axis0=np.prod(m1,axis=0)
10. prod_m1_axis1=np.prod(m1,axis=1)
11. # %% Reshape
12. (ncols,)=np.shape(prod_m1_axis0)
13. prod_m1_axis0=np.reshape(prod_m1_axis0,(1,ncols))
14. (nrows,)=np.shape(prod_m1_axis1)
15. prod_m1_axis1=np.reshape(prod_m1_axis1,(nrows,1))

```

The screenshot displays a Jupyter Notebook environment with the following components:

- Code Editor:** Contains the Python code from the previous block, with line numbers 1 through 15. The code defines vectors, a matrix, and performs various operations including product calculation and reshaping.
- Variable Explorer:** A table showing the state of variables in memory:

Name	Type	Size	Value
m1	Array of int32	(2, 3)	$\begin{bmatrix} 4 & 9 & 7 \\ 5 & 1 & 3 \end{bmatrix}$
ncols	int	1	3
nrows	int	1	2
prod_m1_axis0	Array of int32	(1, 3)	$[20 \quad 9 \quad 21]$
prod_m1_axis1	Array of int32	(2, 1)	$\begin{bmatrix} 252 \\ 15 \end{bmatrix}$
prod_v1	int32	1	252
v1	Array of int32	(3,)	$[4 \quad 9 \quad 7]$
v2	Array of int32	(3,)	$[5 \quad 1 \quad 3]$
- Console:** Shows the execution of the code, including the IPython version (7.10.1) and the command used to run the notebook: `runfile('C:/Users/Phili/Documents/Python Scripts/statistics.py', wdir='C:/Users/Phili/Documents/Python Scripts')`.
- Visualizations:** Several small windows show the NumPy arrays as 2D grids. For example, the 'prod_m1_axis0' window shows a single row with values 20, 9, and 21. The 'prod_m1_axis1' window shows a single column with values 252 and 15.

The functions `np.sum()` and `np.prod()` only calculate one value for each column or row depending if `axis=0` or `axis=1` is selected. We can also calculate the cumulative sum or cumulative product as we go along an axis using `cumsum()` and `cumprod()` respectively.

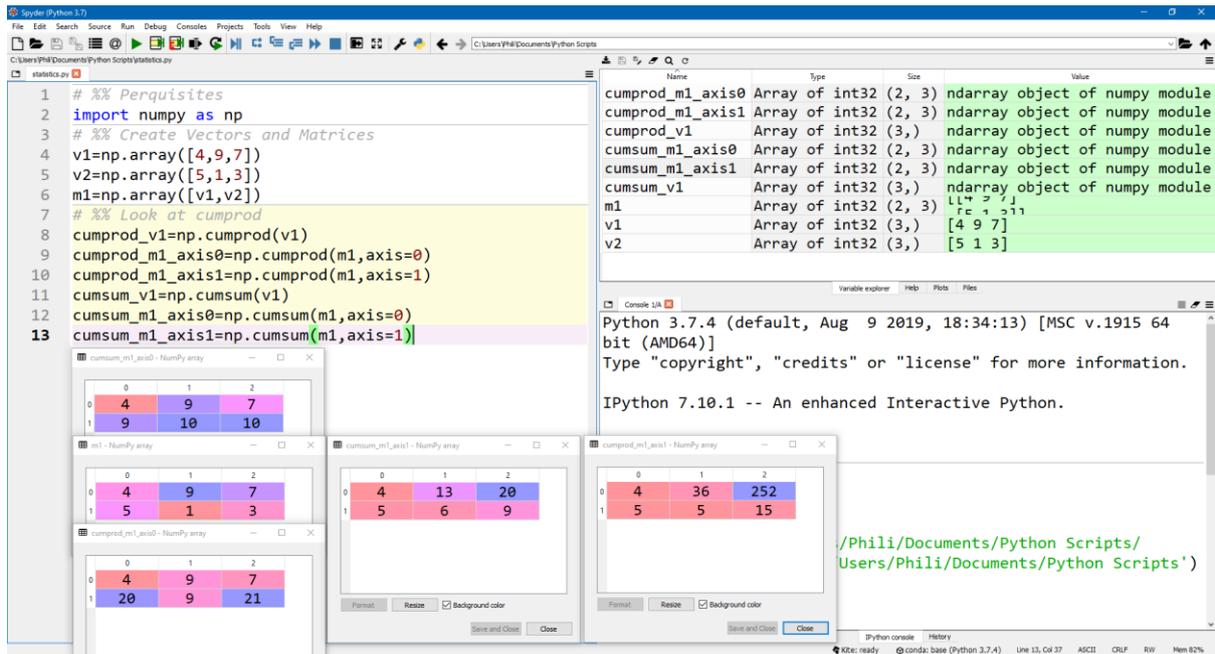
$$\begin{aligned}
 v1 &= [4 \quad 9 \quad 7] \\
 \text{cumprod_v1} &= [4 \quad 4 * 9 \quad 4 * 9 * 7] = [4 \quad 36 \quad 252] \\
 m1 &= \begin{bmatrix} 4 & 9 & 7 \\ 5 & 1 & 3 \end{bmatrix} \\
 \text{cumprod_m1_axis0} &= \begin{bmatrix} 4 & 9 & 7 \\ 4 * 5 & 9 * 1 & 7 * 3 \end{bmatrix} \\
 \text{cumprod_m1_axis0} &= \begin{bmatrix} 4 & 9 & 7 \\ 20 & 9 & 21 \end{bmatrix} \\
 m1 &= \begin{bmatrix} 4 & 9 & 7 \\ 5 & 1 & 3 \end{bmatrix} \\
 \text{cumprod_m1_axis1} &= \begin{bmatrix} 4 & 4 * 9 & 4 * 9 * 7 \\ 5 & 5 * 1 & 5 * 1 * 3 \end{bmatrix} \\
 \text{cumprod_m1_axis1} &= \begin{bmatrix} 4 & 36 & 252 \\ 5 & 5 & 15 \end{bmatrix} \\
 v1 &= [4 \quad 9 \quad 7] \\
 \text{cumsum_v1} &= [4 \quad 4 + 9 \quad 4 + 9 + 7] = [4 \quad 13 \quad 20] \\
 m1 &= \begin{bmatrix} 4 & 9 & 7 \\ 5 & 1 & 3 \end{bmatrix} \\
 \text{cumsum_m1_axis0} &= \begin{bmatrix} 4 & 9 & 7 \\ 4 + 5 & 9 + 1 & 7 + 3 \end{bmatrix} \\
 \text{cumsum_m1_axis0} &= \begin{bmatrix} 4 & 9 & 7 \\ 9 & 10 & 10 \end{bmatrix} \\
 m1 &= \begin{bmatrix} 4 & 9 & 7 \\ 5 & 1 & 3 \end{bmatrix} \\
 \text{cumsum_m1_axis1} &= \begin{bmatrix} 4 & 4 + 9 & 4 + 9 + 7 \\ 5 & 5 + 1 & 5 + 1 + 3 \end{bmatrix} \\
 \text{cumprod_m1_axis1} &= \begin{bmatrix} 4 & 13 & 20 \\ 5 & 6 & 9 \end{bmatrix}
 \end{aligned}$$

```

1. # %% Perquisites
2. import numpy as np
3. # %% Create Vectors and Matrices
4. v1=np.array([4,9,7])
5. v2=np.array([5,1,3])
6. m1=np.array([v1,v2])
7. # %% Look at cumprod
8. cumprod_v1=np.cumprod(v1)
9. cumprod_m1_axis0=np.cumprod(m1,axis=0)
10. cumprod_m1_axis1=np.cumprod(m1,axis=1)
11. cumsum_v1=np.cumsum(v1)
12. cumsum_m1_axis0=np.cumsum(m1,axis=0)

```

13. `cumsum_m1_axis1=np.cumsum(m1,axis=1)`



The function `np.diff()` calculates the difference as you go along columns (`axis=0`) or rows (`axis=1`). It will be smaller than the original by 1 row or 1 column when using (`axis=0`) or (`axis=1`) respectively.

$$v1 = [4 \ 9 \ 7]$$

$$\text{diff}_v1 = [9 - 4 \ 7 - 9] = [5 \ -2]$$

$$m1 = \begin{bmatrix} 4 & 9 & 7 \\ 5 & 1 & 3 \end{bmatrix}$$

$$\text{diff}_m1_axis0 = [5 - 4 \ 1 - 9 \ 3 - 7] = [-1 \ -8 \ -4]$$

$$m1 = \begin{bmatrix} 4 & 9 & 7 \\ 5 & 1 & 3 \end{bmatrix}$$

$$\text{diff}_m1_axis1 = \begin{bmatrix} 9 - 4 & 7 - 9 \\ 1 - 5 & 3 - 1 \end{bmatrix} = \begin{bmatrix} -5 & -2 \\ -4 & 2 \end{bmatrix}$$

```

1. # %% Perquisites
2. import numpy as np
3. # %% Create Vectors and Matrices
4. v1=np.array([4,9,7])
5. v2=np.array([5,1,3])
6. m1=np.array([v1,v2])
7. # %% Look at diff
8. diff_v1=np.diff(v1)
9. diff_m1_axis0=np.diff(m1,axis=0)
10. diff_m1_axis1=np.diff(m1,axis=1)

```

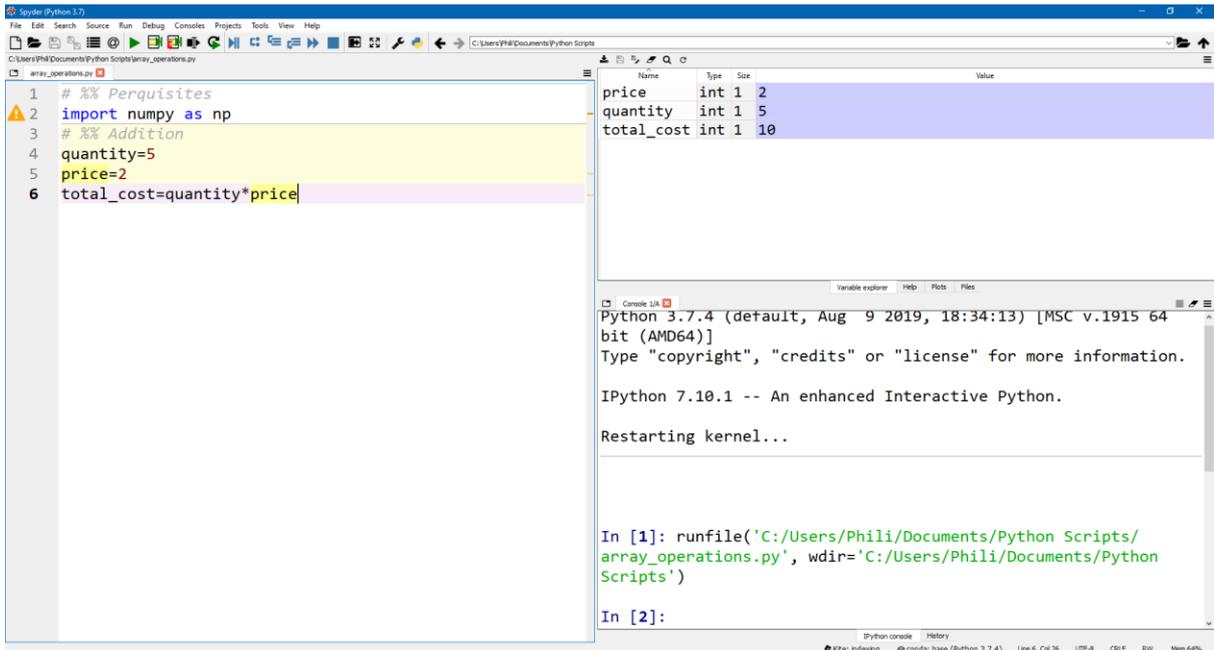

$$5 \text{ pen} * 2 \text{ £/pen} = 10 \text{ £} \left(\frac{\text{pen}}{\text{pen}} \right) = 10 \text{ £}$$

Units are added at the end and also multiplied for a dimensionality check.

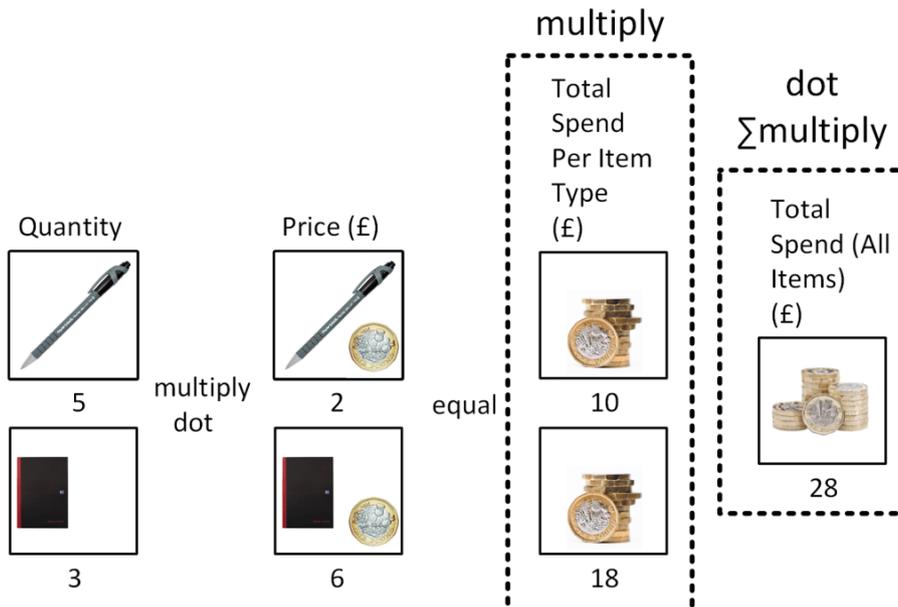
```

1. # %% Perquisites
2. import numpy as np
3. # %% Addition
4. quantity=5
5. price=2
6. total_cost=quantity*price

```



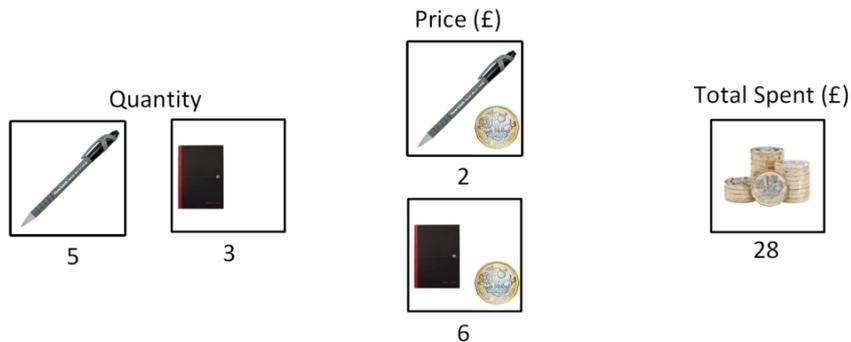
Now if instead we look at a vector, we see there are two ways that we can multiply. We can multiply element by element to get the total cost of each item type or we can calculate the total cost that we spend (which is known as array multiplication).



In element by element multiplication we must have matching dimensions:

$$\begin{bmatrix} 5 \text{ pen} \\ 3 \text{ pad} \end{bmatrix} * \begin{bmatrix} 2 \frac{\text{£}}{\text{pen}} \\ 6 \frac{\text{£}}{\text{pad}} \end{bmatrix} = \begin{bmatrix} 10 \text{ £} \left(\frac{\text{pen}}{\text{pen}} \right) \\ 18 \text{ £} \left(\frac{\text{pad}}{\text{pad}} \right) \end{bmatrix} = \begin{bmatrix} 10 \text{ £} \\ 18 \text{ £} \end{bmatrix}$$

For array multiplication the number of columns of the array on the left-hand side must match the number of rows of the array on the right-hand side. In other words, the inner dimensions of the array must match. The problem is thus laid out this way:



$$\begin{bmatrix} 5 \text{ pen} & 3 \text{ pad} \end{bmatrix} @ \begin{bmatrix} 2 \frac{\text{£}}{\text{pen}} \\ 6 \frac{\text{£}}{\text{pad}} \end{bmatrix} = \begin{bmatrix} 5 \text{ pen} * 2 \frac{\text{£}}{\text{pen}} \\ 3 \text{ pad} * 6 \frac{\text{£}}{\text{pad}} \end{bmatrix} = [10 \text{ £} + 18 \text{ £}] = [28]$$

The screenshot shows the Spyder Python IDE. The code editor contains the following code:

```

1. # %% Perquisites
2. import numpy as np
3. # %% Addition
4. quantity=np.array([5,3])
5. price=np.array([2,6])
6. # %% Element by Element
7. quantity=np.reshape(quantity,[2,1])
8. price=np.reshape(price,[2,1])
9. total_cost_per_item=quantity*price
10. # %% Array
11. quantity=np.reshape(quantity,[1,2])
12. price=np.reshape(price,[2,1])
13. total_cost=quantity@price
  
```

The Variable explorer shows the following variables:

Name	Type	Size	Value
price	Array of int32 (2, 1)	[2, 1]	[[2 6]]
quantity	Array of int32 (1, 2)	[1, 2]	[[5 3]]
total_cost	Array of int32 (1, 1)	[1, 1]	[[28]]
total_cost_per_item	Array of int32 (2, 1)	[2, 1]	[[10 18]]

The console output shows the following:

```

Python 3.7.4 (default, Aug 9 2019, 18:34:13) [MSC v.1915 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.10.1 -- An enhanced Interactive Python.
Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/array_operations.py', wdir='C:/Users/Phili/Documents/Python Scripts')

In [2]:
  
```

```

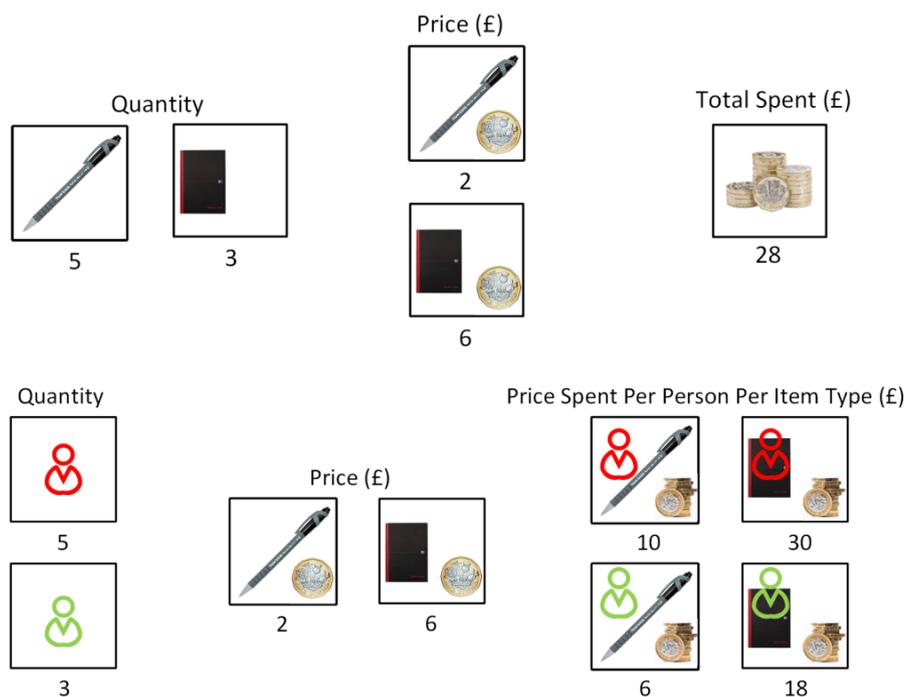
1. # %% Perquisites
2. import numpy as np
3. # %% Addition
4. quantity=np.array([5,3])
5. price=np.array([2,6])
6. # %% Element by Element
7. quantity=np.reshape(quantity,[2,1])
  
```

```

8. price=np.reshape(price,[2,1])
9. total_cost_per_item=quantity*price
10. # %% Array
11. quantity=np.reshape(quantity,[1,2])
12. price=np.reshape(price,[2,1])
13. total_cost=quantity@price

```

Because vectors are used, we must be careful with array multiplication and ensure that we reshape them so that they have the correct dimensionality. In the problem above we performed inner multiplication where the vector dimensions were $[1, n]$ and $[n, 1]$ respectively leading to an output array that was $[1, 1]$.



It is possible in another problem to perform multiplication where we use outer multiplication instead. In this case we would have:

$$\begin{bmatrix} 5 \text{ person (office 1)} \\ 3 \text{ person (office 2)} \end{bmatrix} @ \begin{bmatrix} 2 \frac{\text{£}}{\text{person}} \text{ (pens)} & 6 \frac{\text{£}}{\text{person}} \text{ (pads)} \end{bmatrix}$$

Let's look at the 0th row (left hand side matrix) and 0th column (right hand side matrix) to give the 0th row and 0th column of the output matrix:

$$\begin{bmatrix} 5 \text{ person (office 1)} \\ 3 \text{ person (office 2)} \end{bmatrix} @ \begin{bmatrix} 2 \frac{\text{£}}{\text{person}} \text{ (pens)} & 6 \frac{\text{£}}{\text{person}} \text{ (pads)} \end{bmatrix} = \begin{bmatrix} 5 \text{ person (office 1)} * 2 \frac{\text{£}}{\text{person}} \text{ (pens)} & \dots \\ \vdots & \ddots \end{bmatrix} = \begin{bmatrix} 10 \text{ £ spent on pens in office 1} & \dots \\ \vdots & \ddots \end{bmatrix}$$

Now let's look at the 0th row (left hand side matrix) and 1st column (right hand side matrix) of the output matrix:

$$\begin{bmatrix} 5 \text{ person (office 1)} \\ 3 \text{ person (office 2)} \end{bmatrix} @ \begin{bmatrix} 2 \frac{\text{£}}{\text{person}} \text{ (pens)} & 6 \frac{\text{£}}{\text{person}} \text{ (pads)} \end{bmatrix}$$

$$= \begin{bmatrix} 10 \text{ £ spent on pens in office 1} & 30 \text{ £ spent on pads in office 1} \\ \vdots & \ddots \end{bmatrix}$$

Now let's look at the 1st row (left hand side matrix) and 0th column (right hand side matrix) to give to give the 1st row and 0th column of the output matrix:

$$\begin{bmatrix} 5 \text{ person (office 1)} \\ 3 \text{ person (office 2)} \end{bmatrix} @ \begin{bmatrix} 2 \frac{\text{£}}{\text{person}} \text{ (pens)} & 6 \frac{\text{£}}{\text{person}} \text{ (pads)} \end{bmatrix}$$

$$= \begin{bmatrix} 10 \text{ £ spent on pens in office 1} & 30 \text{ £ spent on pads in office 1} \\ 6 \text{ £ spent on pens in office 2} & \ddots \end{bmatrix}$$

Finally let's look at the 1st row (left hand side matrix) and 1st column (right hand side matrix) to give to give the 1st row and 1st column of the output matrix:

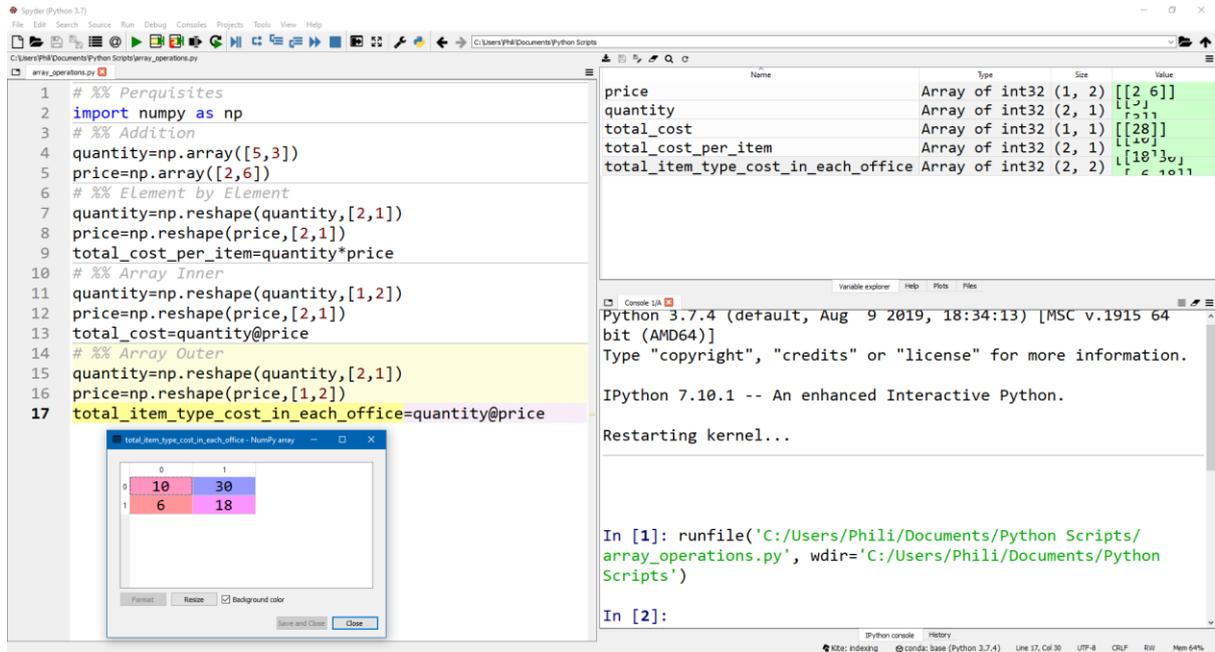
$$\begin{bmatrix} 5 \text{ person (office 1)} \\ 3 \text{ person (office 2)} \end{bmatrix} @ \begin{bmatrix} 2 \frac{\text{£}}{\text{person}} \text{ (pens)} & 6 \frac{\text{£}}{\text{person}} \text{ (pads)} \end{bmatrix}$$

$$= \begin{bmatrix} 10 \text{ £ spent on pens in office 1} & 30 \text{ £ spent on pads in office 1} \\ 6 \text{ £ spent on pens in office 2} & 18 \text{ £ spent on pads in office 2} \end{bmatrix}$$

```

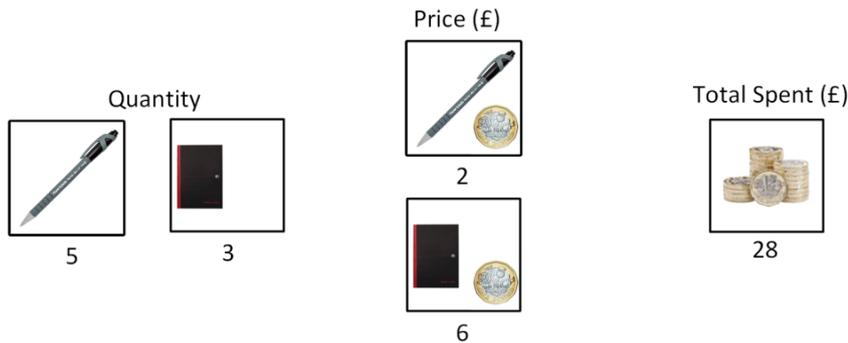
1. # %% Perquisites
2. import numpy as np
3. # %% Addition
4. quantity=np.array([5,3])
5. price=np.array([2,6])
6. # %% Element by Element
7. quantity=np.reshape(quantity,[2,1])
8. price=np.reshape(price,[2,1])
9. total_cost_per_item=quantity*price
10. # %% Array Inner
11. quantity=np.reshape(quantity,[1,2])
12. price=np.reshape(price,[2,1])
13. total_cost=quantity@price
14. # %% Array Outer
15. quantity=np.reshape(quantity,[2,1])
16. price=np.reshape(price,[1,2])
17. total_item_type_cost_in_each_office=quantity@price

```



Element by Element Division/Interpolation

Array division is the opposite process of multiplication, so let's have a look at the two arrays we have just created. Now let us assume we know the value on the left-hand side and the value on the right hand side and are trying to instead calculate the value in the middle:



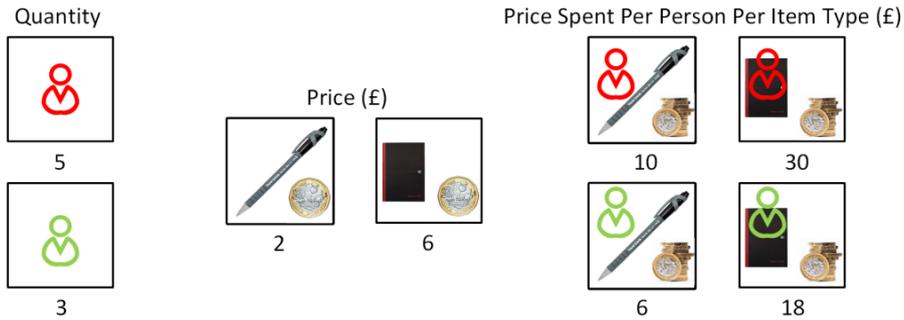
What we have is:

$$[5 \quad 3] @ \begin{bmatrix} x \\ y \end{bmatrix} = [28]$$

Or:

$$5x + 3y = 28$$

Here we run into a problem, we have two unknowns and only a single equation. Now our solution of $x = 2$ and $y = 6$ obviously work as we just back tracked from it however we never specified that we had to have a solution which yielded full values of £s if pennies are included then the solution $x = 3.8$ and $y = 9$ could also work. In other words, from the limited data we have we have come up with multiple solutions for the problem.



If on the other hand, we have the following:

$$\begin{bmatrix} 5 \\ 2 \end{bmatrix} @ [x \quad y] = \begin{bmatrix} 10 & 30 \\ 6 & 18 \end{bmatrix}$$

Then we have four equations for only two unknowns.

$$5x = 10 \therefore x = \frac{10}{5} = 2$$

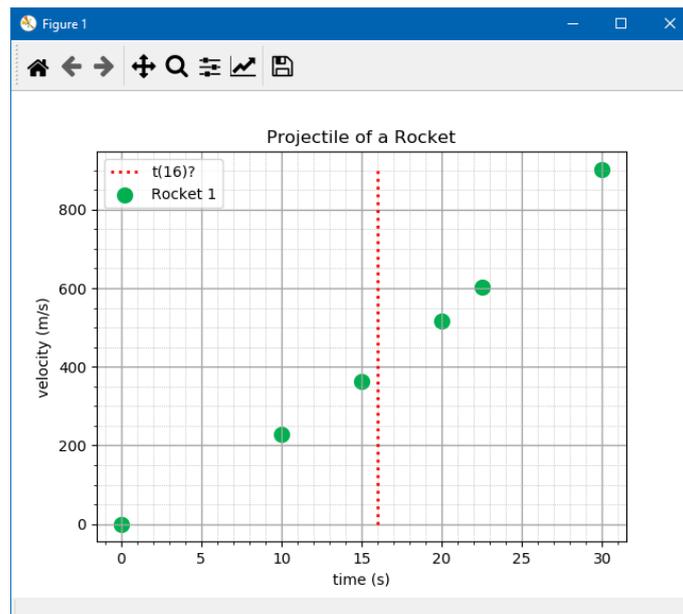
$$5y = 30 \therefore y = \frac{30}{5} = 6$$

$$3x = 6 \therefore x = \frac{6}{3} = 2$$

$$3y = 18 \therefore y = \frac{18}{3} = 6$$

Note that we can use two of these to confirm the value of x and y and the other equations just confirm the results. To solve a linear system of equations with accuracy we require the same number of equations as unknowns.

If we return to the data which we used as an example when we looked at plotting and ask the question, what is the velocity at a $t=16$ s?



time (s)	velocity (m/s)
0	0
10	227.04
15	362.78
20	517.35
22.5	602.97
30	901.67

We can select a single point to perform the nearest point interpolation.

time (s)	Δt (s)	velocity (m/s)
0	0-16=-16	0
10	10-16=-6	227.04
15	15-16=-1	362.78
20	20-16=+4	517.35
22.5	22.5-16=+6.5	602.97
30	30-16=+14	901.67

Let's look at how to calculate this manually in Python using the existing data. For this we can calculate dt by subtracting a scalar 16 from v . To find a minimum we can use the `np.min()` function however because we have both negative and positive values we should instead find the absolute minimum which can be done using the function `abs` alongside `np.min()`. Note however that this results in a loss of the sign. If we want to retain this sign what we can instead do is find the index of the absolute minimum using `np.argmin()` and use this to index into dt and v to find the absolute minimum value of dt and the nearest v to our unknown data point at v_{16} .

```

1. # %% Perquisites
2. import numpy as np
3. import matplotlib.pyplot as plt
4. import scipy as sp
5. # %% Create Vectors
6. t=np.array([0,10,15,20,22.5,30])
7. v=np.array([0,227.04,362.78,517.35,602.97,901.67])
8. dt=t-16
9. # %% Calculate minimum dt
10. dt_min=np.min(dt)
11. print(f'dt_min={dt_min}')
12. print('we need the absolute minimum')
13. dt_min=np.min(abs(dt))
14. print(f'dt_min={dt_min}')
15. idx_min=np.argmin(abs(dt))
16. print('when we use this we lose the sign')
17. print('finding the absolute minimum index')
18. print(f'idx_min={idx_min}')
19. print('allows us to index into dt')
20. print('to find the absolute minimum value with sign')
21. dt_min=dt[idx_min]
22. print(f'dt_min={dt_min}')
23. print('we can also use this to index into v')
24. v16_nearest=v[idx_min]
25. print(f'v16_nearest={v16_nearest}')

```

```

1  # %% Perquisites
2  import numpy as np
3  import matplotlib.pyplot as plt
4  import scipy as sp
5  # %% Create Vectors
6  t=np.array([0,10,15,20,22.5,30])
7  v=np.array([0,227.04,362.78,517.35,602.97,901.67])
8  dt=t-16
9  # %% Calculate minimum dt
10 dt_min=np.min(dt)
11 print(f'dt_min={dt_min}')
12 print('we need the absolute minimum')
13 dt_min=np.min(abs(dt))
14 print(f'dt_min={dt_min}')
15 idx_min=np.argmin(abs(dt))
16 print('when we use this we lose the sign')
17 print('finding the absolute minimum index')
18 print(f'idx_min={idx_min}')
19 print('allows us to index into dt')
20 print('to find the absolute minimum value with sign')
21 dt_min=dt[idx_min]
22 print(f'dt_min={dt_min}')
23 print('we can also use this to index into v')
24 v16_nearest=v[idx_min]
25 print(f'v16_nearest={v16_nearest}')]

```

Name	Type	Size	Value
dt	Array of float64 (6,)	[-16. -6. -1. 4. 6.5 14...	
dt_min	float64	1	-1.0
idx_min	int64	1	2
t	Array of float64 (6,)	[0. 10. 15. 20. 22.5 30.]	
v	Array of float64 (6,)	[0. 227.04 362.78 517.35 602.9...	
v16_nearest	float64	1	362.78

```

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/
nearest_interpolation.py', wdir='C:/Users/Phili/Documents/
Python Scripts')
dt_min=-16.0
we need the absolute minimum
dt_min=1.0
when we use this we lose the sign
finding the absolute minimum index
idx_min=2
allows us to index into dt
to find the absolute minimum value with sign
dt_min=-1.0
we can also use this to index into v
v16_nearest=362.78

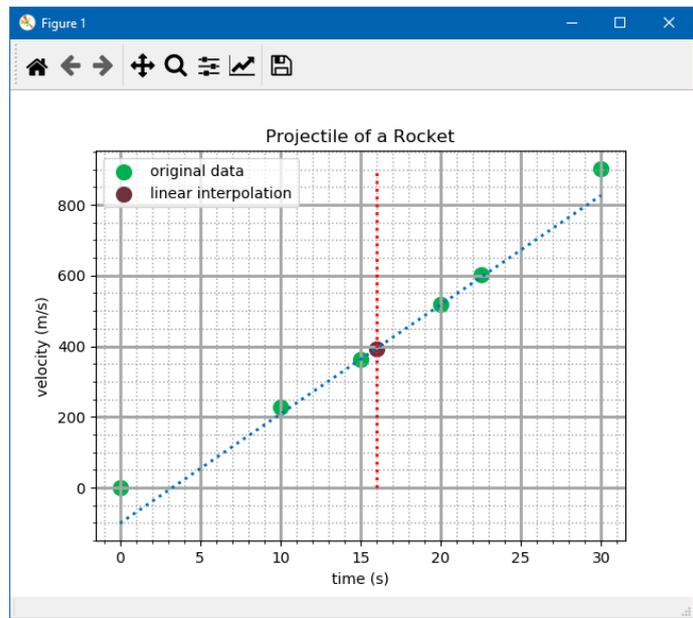
In [2]:

```

To get a more accurate estimation, we can use the two nearest points.

time (s)	Δt (s)	velocity (m/s)
0	0-16=-16	0
10	10-16=-6	227.04
15	15-16=-1	362.78
20	20-16=+4	517.35
22.5	22.5-16=+6.5	602.97
30	30-16=+14	901.67

This allows us to plot a straight line between the nearest two points and the value where this straight line intersects t=16 will yield a more accurate estimate.



The equation for a straight line is:

$$v[t] = a_0 + a_1 * t$$

To solve for the two unknowns:

$$a_0, a_1$$

We require 2 linear equations:

$$v_0 = a_0 + a_1 t_0$$

$$v_1 = a_0 + a_1 t_1$$

Which in matrix form is:

$$\begin{bmatrix} v_0 \\ v_1 \end{bmatrix} = \begin{bmatrix} a_0 + a_1 t_0 \\ a_0 + a_1 t_1 \end{bmatrix} = \begin{bmatrix} 1 & t_0 \\ 1 & t_1 \end{bmatrix} @ \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}$$

Now we can input our values from the table:

$$\begin{bmatrix} 362.78 \\ 517.35 \end{bmatrix} = \begin{bmatrix} 15^0 & 15^1 \\ 20^0 & 20^1 \end{bmatrix} @ \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}$$

$$\begin{bmatrix} 362.78 \\ 517.35 \end{bmatrix} = \begin{bmatrix} 1 & 15 \\ 1 & 20 \end{bmatrix} @ \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}$$

For convenience we can assign these arrays as:

$$v_vec = \begin{bmatrix} 362.78 \\ 517.35 \end{bmatrix}$$

$$t_mat = \begin{bmatrix} 1 & 15 \\ 1 & 20 \end{bmatrix}$$

$$a_vec = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}$$

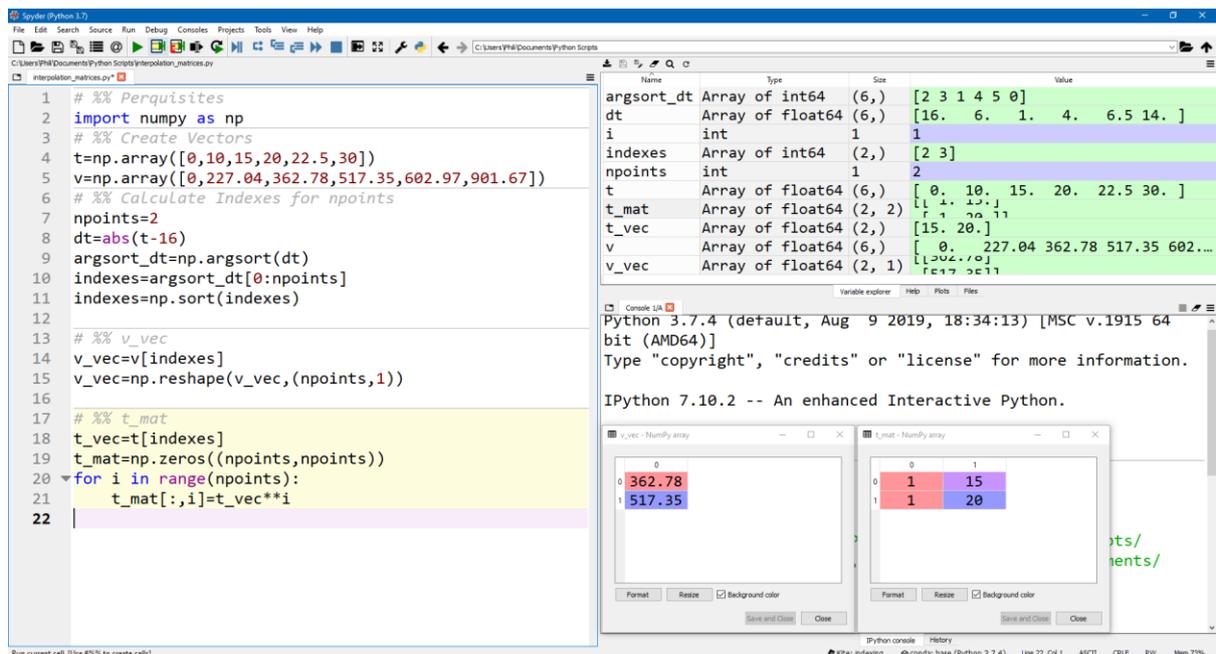
We can look at our original data and use `np.abs()`, `np.argsort()`, `np.sort()`, with a designated number `npoints` (in this case `npoints=2`) to obtain indexes for the values that have the absolute minimum values from our time point in this case 16. We can then use these indexes to extract `v_vec` and `t_vec`. We can then create `t_mat` by first initialising it to zeros using `np.zeros()` and then using a `for` loop index each column and update it to `t_vec**i`. We can set `i` to be the `range(npoints)` which in this case also corresponds to the length of the matrix and uses zero-order indexing.

```
1. # %% Perquisites
2. import numpy as np
3. # %% Create Vectors
4. t=np.array([0,10,15,20,22.5,30])
5. v=np.array([0,227.04,362.78,517.35,602.97,901.67])
6. # %% Calculate Indexes for npoints
7. npoints=2
8. dt=abs(t-16)
9. argsort_dt=np.argsort(dt)
10. indexes=argsort_dt[0:npoints]
11. indexes=np.sort(indexes)
12.
13. # %% v_vec
14. v_vec=v[indexes]
```

```

15. v_vec=np.reshape(v_vec, (npoints,1))
16.
17. # %% t_mat
18. t_vec=t[indexes]
19. t_mat=np.zeros((npoints,npoints))
20. for i in range(npoints):
21.     t_mat[:,i]=t_vec**i
22.

```



The matrix is square and to solve this we can use properties of square matrices and the identity matrix.

$$t_mat = \begin{bmatrix} 1 & 15 \\ 1 & 20 \end{bmatrix}$$

For a square matrix that is invertible:

$$m@m^{-1} = m^{-1}@m = I$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} @ \begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} @ \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The values of the inverse matrix can be expressed in terms of the original matrix using:

$$\frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

A square matrix is invertible when the determinate $ad - bc \neq 0$.

$$t_mat = \begin{bmatrix} 1 & 15 \\ 1 & 20 \end{bmatrix}$$

Therefore:

$$t_inv = t_mat^{-1} = \frac{1}{1 * 20 - 15 * 1} \begin{bmatrix} 20 & -15 \\ -1 & 1 \end{bmatrix} = \frac{1}{5} \begin{bmatrix} 20 & -15 \\ -1 & 1 \end{bmatrix} = \begin{bmatrix} 4 & -3 \\ -0.2 & 0.2 \end{bmatrix}$$

We can now check this is the inverse:

$$t_mat@t_inv = \begin{bmatrix} 1 & 15 \\ 1 & 20 \end{bmatrix} @ \begin{bmatrix} 4 & -3 \\ -0.2 & 0.2 \end{bmatrix}$$

Looking at the 0th row (left hand side matrix) and 0th column (right hand side matrix) of the output matrix:

$$t_mat@t_inv = \begin{bmatrix} 1 & 15 \\ 1 & 20 \end{bmatrix} @ \begin{bmatrix} 4 & -3 \\ -0.2 & 0.2 \end{bmatrix} = \begin{bmatrix} 1*4 + 15*-0.2 & \dots \\ \vdots & \ddots \end{bmatrix} = \begin{bmatrix} 1 & \dots \\ \vdots & \ddots \end{bmatrix}$$

Looking at the 0th row (left hand side matrix) and 1st column (right hand side matrix) of the output matrix:

$$t_mat@t_inv = \begin{bmatrix} 1 & 15 \\ 1 & 20 \end{bmatrix} @ \begin{bmatrix} 4 & -3 \\ -0.2 & 0.2 \end{bmatrix} = \begin{bmatrix} 1 & 1*-3 + 15*0.2 \\ \vdots & \ddots \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \vdots & \ddots \end{bmatrix}$$

Looking at the 1st row (left hand side matrix) and 0th column (right hand side matrix) of the output matrix:

$$t_mat@t_inv = \begin{bmatrix} 1 & 15 \\ 1 & 20 \end{bmatrix} @ \begin{bmatrix} 4 & -3 \\ -0.2 & 0.2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1*4 + 20*-0.2 & \vdots \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & \vdots \end{bmatrix}$$

Looking at the 1st row (left hand side matrix) and 1st column (right hand side matrix) of the output matrix:

$$t_mat@t_inv = \begin{bmatrix} 1 & 15 \\ 1 & 20 \end{bmatrix} @ \begin{bmatrix} 4 & -3 \\ -0.2 & 0.2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1*-3 + 20*0.2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Which is of course the identity matrix \mathbf{i} .

Likewise we can perform the check:

$$t_mat@t_inv = \begin{bmatrix} 4 & -3 \\ -0.2 & 0.2 \end{bmatrix} @ \begin{bmatrix} 1 & 15 \\ 1 & 20 \end{bmatrix}$$

Looking at the 0th row (left hand side matrix) and 0th column (right hand side matrix) of the output matrix:

$$t_mat@t_inv = \begin{bmatrix} 4 & -3 \\ -0.2 & 0.2 \end{bmatrix} @ \begin{bmatrix} 1 & 15 \\ 1 & 20 \end{bmatrix} = \begin{bmatrix} 4*1 + (-3)*1 & \dots \\ \vdots & \ddots \end{bmatrix} = \begin{bmatrix} 1 & \dots \\ \vdots & \ddots \end{bmatrix}$$

Looking at the 0th row (left hand side matrix) and 1st column (right hand side matrix) of the output matrix:

$$t_mat@t_inv = \begin{bmatrix} 4 & -3 \\ -0.2 & 0.2 \end{bmatrix} @ \begin{bmatrix} 1 & 15 \\ 1 & 20 \end{bmatrix} = \begin{bmatrix} 1 & 4*15 + (-3)*20 \\ \vdots & \ddots \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \vdots & \ddots \end{bmatrix}$$

Looking at the 1st row (left hand side matrix) and 0th column (right hand side matrix) of the output matrix:

$$t_mat@t_inv = \begin{bmatrix} 4 & -3 \\ -0.2 & 0.2 \end{bmatrix} @ \begin{bmatrix} 1 & 15 \\ 1 & 20 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ (-0.2)*1 + 0.2*1 & \vdots \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & \vdots \end{bmatrix}$$

Looking at the 1st row (left hand side matrix) and 1st column (right hand side matrix) of the output matrix:

$$t_mat@t_inv = \begin{bmatrix} 4 & -3 \\ -0.2 & 0.2 \end{bmatrix} @ \begin{bmatrix} 1 & 15 \\ 1 & 20 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & (-0.2)*15 + 0.2*20 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

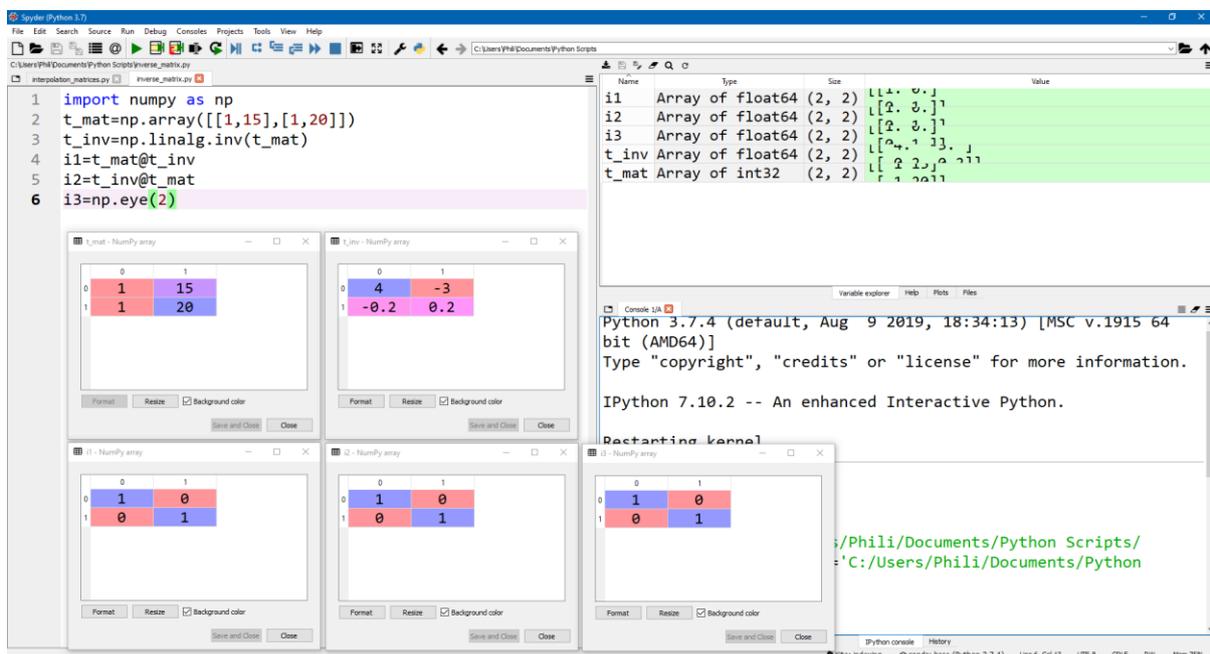
This also returns the identity matrix \mathbf{i} .

To calculate the inverse we need to use the linear algebra functions of the numpy library `np.linalg.inv()` and we can see that when we multiply the matrix by its inverse or the inverse matrix by the original matrix that we get the identity matrix which can also be created directly using the function `np.eye()` which takes only a scalar input argument because it is a square matrix.

```

1. # %% Perquisites
2. import numpy as np
3. # %% Matrix and Matrix Inverse
4. t_mat=np.array([[1,15],[1,20]])
5. t_inv=np.linalg.inv(t_mat)
6. i1=t_mat@t_inv
7. i2=t_inv@t_mat
8. i3=np.eye(2)

```



Returning to the problem at hand:

$$\begin{bmatrix} 362.78 \\ 517.35 \end{bmatrix} = \begin{bmatrix} 1 & 15 \\ 1 & 20 \end{bmatrix} @ \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}$$

$$v_vec = t_mat@a_vec$$

We can multiply through by the `t_inv` Let's look at the right-hand side:

$$t_inv@v_vec = t_inv@t_mat@a_vec$$

$$t_inv@v_vec = i@a_vec$$

Looking at the right-hand side:

$$i@a_vec = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} @ \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}$$

Looking at the 0th row (left hand side matrix) and 0th column (right hand side matrix) of the output matrix:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} @ \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} 1 * a_0 + 0 * a_1 \\ \vdots \end{bmatrix} = \begin{bmatrix} a_0 \\ \vdots \end{bmatrix}$$

Looking at the 1st row (left hand side matrix) and 0th column (right hand side matrix) of the output matrix:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} @ \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} a_0 \\ 0 * a_0 + 1 * a_1 \end{bmatrix} = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}$$

We can see that when an array is multiplied by the identity matrix, it remains unchanged. Therefore:

$$t_inv@v_vec = a_vec$$

Rearranging:

$$a_vec = t_inv@v_vec$$

Calculating the right hand side:

$$t_inv@v_vec = \begin{bmatrix} 4 & -3 \\ -0.2 & 0.2 \end{bmatrix} @ \begin{bmatrix} 362.78 \\ 517.35 \end{bmatrix}$$

Looking at the 0th row (left hand side matrix) and 0th column (right hand side matrix) of the output matrix:

$$\begin{bmatrix} 4 & -3 \\ -0.2 & 0.2 \end{bmatrix} @ \begin{bmatrix} 362.78 \\ 517.35 \end{bmatrix} = \begin{bmatrix} 4 * 362.78 + (-3) * 517.35 \\ \vdots \end{bmatrix} = \begin{bmatrix} -100.93 \\ \vdots \end{bmatrix}$$

Looking at the 1st row (left hand side matrix) and 0th column (right hand side matrix) of the output matrix:

$$\begin{bmatrix} 4 & -3 \\ -0.2 & 0.2 \end{bmatrix} @ \begin{bmatrix} 362.78 \\ 517.35 \end{bmatrix} = \begin{bmatrix} 4 * 362.78 + (-3) * 517.35 \\ (-0.2) * 362.78 + 0.2 * 517.35 \end{bmatrix} = \begin{bmatrix} -100.93 \\ 30.914 \end{bmatrix}$$

Therefore:

$$\begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} -100.93 \\ 30.914 \end{bmatrix}$$

These can be used to estimate the value of v at t=16.

$$v_16 = t_16@a_vec$$

$$v_16 = \begin{bmatrix} 1 & 16 \end{bmatrix} @ \begin{bmatrix} -100.93 \\ 30.914 \end{bmatrix} = \begin{bmatrix} 1 * -100.93 + 16 * 30.914 \end{bmatrix} = \begin{bmatrix} 363.694 \end{bmatrix}$$

```

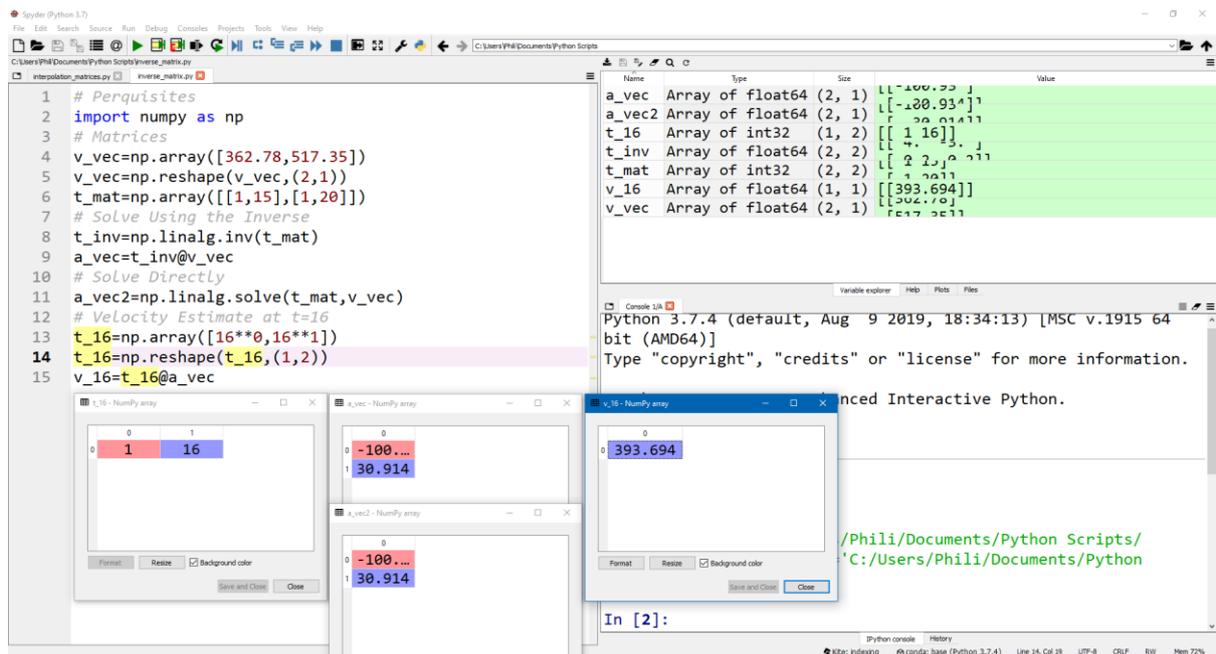
1. # Perquisites
2. import numpy as np
3. # Matrices
4. v_vec=np.array([362.78, 517.35])
5. v_vec=np.reshape(v_vec, (2,1))
6. t_mat=np.array([[1,15],[1,20]])
7. # Solve Using the Inverse
8. t_inv=np.linalg.inv(t_mat)
9. a_vec=t_inv@v_vec
10. # Solve Directly
11. a_vec2=np.linalg.solve(t_mat,v_vec)
12. # Velocity Estimate at t=16
13. t_16=np.array([16**0, 16**1])

```

```

14. t_16=np.reshape(t_16,(1,2))
15. v_16=t_16@a_vec

```



We can combine this together with our earlier script which we used to obtain `v_vec` and `t_mat` from the original data and make a function with a four input argument, the original `t` and `v` data, `npoints`, the number of points to interpolate and the timepoint we wish to perform the interpolation at. This function will return a single output `v_point` which is the interpolated value.

```

1. # %% Perquisites
2. import numpy as np
3. # %% Data
4. t=np.array([0,10,15,20,22.5,30])
5. v=np.array([0,227.04,362.78,517.35,602.97,901.67])
6. def interpolate(npoints,timepoint):
7.     """
8.     t,v are the time and velocity vectors
9.     npoints is the number of points to interpolate with
10.    timepoint is the unknown timepoint which we wish
11.    to find the return v_point
12.    """
13.    # %% Perquisites
14.    import numpy as np
15.    # %% Create Vectors
16.    # %% Calculate Indexes for npoints
17.    dt=abs(t-timepoint)
18.    argsort_dt=np.argsort(dt)
19.    indexes=argsort_dt[0:npoints]
20.    indexes=np.sort(indexes)
21.
22.    # %% v_vec
23.    v_vec=v[indexes]

```

```

24.     v_vec=np.reshape(v_vec,(npoints,1))
25.
26.     # %% t_mat
27.     t_vec=t[indexes]
28.     t_mat=np.zeros((npoints,npoints))
29.     for i in range(npoints):
30.         t_mat[:,i]=t_vec**i
31.
32.     # Calculate a_vec
33.     a_vec=np.linalg.solve(t_mat,v_vec)
34.     # Velocity Estimate at t=16
35.     t_point=np.zeros(npoints)
36.     for i in range(npoints):
37.         t_point[i]=timepoint**i
38.
39.     t_point=np.reshape(t_point,(1,npoints))
40.     v_point=t_point@a_vec
41.     return v_point
42.
43. v16_1nearest=interpolate(t,v,1,16)
44. v16_2lin=interpolate(t,v,2,16)
45. v16_3quad=interpolate(t,v,3,16)
46. v16_4cubic=interpolate(t,v,4,16)
47. v17_4cubic=interpolate(t,v,4,16)
48. v19_4cubic=interpolate(t,v,4,16)
49. v19_4cubic2=interpolate(t,2*v,4,16)

```

Where you see the versatility of making this a custom function.

The screenshot shows a Python IDE with a script editor on the left and a console/variable explorer on the right. The script defines an `interpolate` function and uses it to calculate various interpolated values. The console shows the execution of the script, including the output of the `runfile` command.

Name	Type	Size	Value
t	Array of float64 (6,)		[0. 10. 15. 20. 22.5 30.]
v	Array of float64 (6,)		[0. 227.04 362.78 517.35 602.97 901.67]
v16_1nearest	Array of float64 (1, 1)		[[362.78]]
v16_2lin	Array of float64 (1, 1)		[[393.694]]
v16_3quad	Array of float64 (1, 1)		[[392.1876]]
v16_4cubic	Array of float64 (1, 1)		[[392.057168]]
v17_4cubic	Array of float64 (1, 1)		[[422.120144]]
v19_4cubic	Array of float64 (1, 1)		[[484.733952]]
v19_4cubic2	Array of float64 (1, 1)		[[969.467904]]

Console output:

```

Python 3.7.4 (default, Aug 9 2019, 18:34:13) [MSC v.1915 64-bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.10.2 -- An enhanced Interactive Python.

Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/interpolation_matrices.py', wdir='C:/Users/Phili/Documents/Python Scripts')

In [2]:

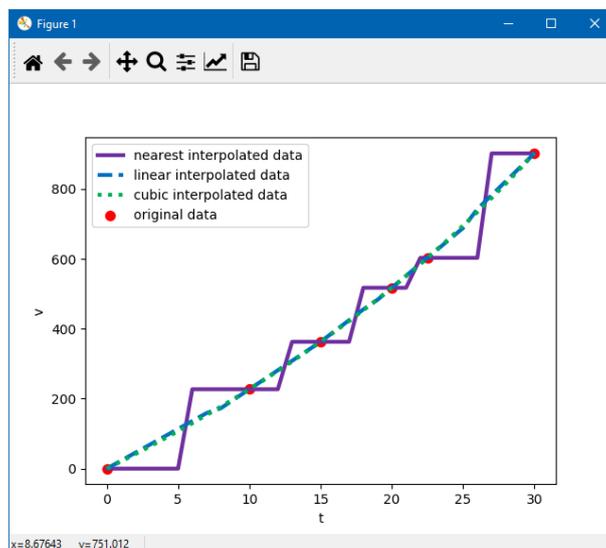
```

We can also use the function within a loop to obtain a series of interpolated data and then plot this interpolated data with respect to the original data using matplotlib.

```

43. import matplotlib.pyplot as plt
44. newt=np.arange(start=0,stop=31,step=1)
45. newv1=np.zeros(len(newt))
46. newv2=np.zeros(len(newt))
47. newv4=np.zeros(len(newt))
48. for i in newt:
49.     newv1[i]=interpolate(t,v,1,newt[i])
50.     newv2[i]=interpolate(t,v,2,newt[i])
51.     newv4[i]=interpolate(t,v,4,newt[i])
52.
53. plt.close('all')
54. plt.figure(1)
55. plt.scatter(t,v,s=50,
56.             color=[255/255,0/255,0/255,1],
57.             label='original data')
58. plt.plot(newt,newv1,
59.          color=[112/255,48/255,160/255,1],
60.          linewidth=3,linestyle='-',
61.          label='nearest interpolated data')
62. plt.plot(newt,newv2,
63.          color=[0/255,112/255,192/255,1],
64.          linewidth=3,linestyle='--',
65.          label='linear interpolated data')
66. plt.plot(newt,newv4,
67.          color=[0/255,176/255,80/255,1],
68.          linewidth=3,linestyle=':',
69.          label='cubic interpolated data')
70. plt.xlabel('t')
71. plt.ylabel('v')
72. plt.legend()
73. plt.show()

```



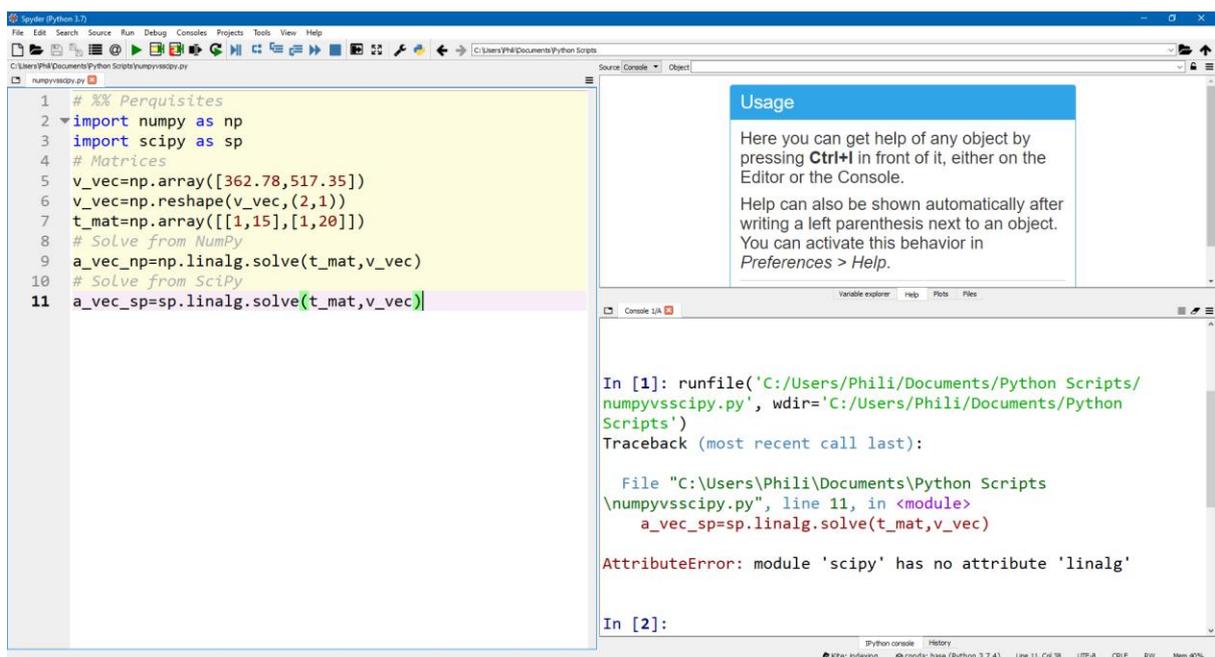
The SCientific PYthon Library (SCIPY)

We have so far used the Numerical Python Library `numpy` to perform basic element by element and array manipulation of matrices. While learning the fundamentals of these operations we created our own interpolation function above. The Interpolation function also exist in the Scientific Python library `SciPy`. SciPy includes submodules for clustering algorithms, physical and mathematical constants, fast Fourier transform routines, integration and ordinary differential equation solvers, interpolation and smoothing splines, input and output, linear algebra, N-dimensional image processing, orthogonal distance regression, optimization and root-finding routines, signal processing, sparse matrices and associated routines, spatial data structures and algorithms, special functions, statistical distributions and functions. Some of the functions were originally created as part of `numpy` before `scipy` was introduced as a separate library and are left in the `numpy` library to prevent older code from breaking. In this section we will look at only a handful of basic functions just to demonstrate the general principle behind using `scipy` subpackages.

`scipy.linalg`

The function `numpy.linalg.solve` for example that we used to perform array division is identical to `scipy.linalg.solve`. Note however if we attempt to do the following:

```
1. # %% Perquisites
2. import numpy as np
3. import scipy as sp
4. # Matrices
5. v_vec=np.array([362.78,517.35])
6. v_vec=np.reshape(v_vec,(2,1))
7. t_mat=np.array([[1,15],[1,20]])
8. # Solve from NumPy
9. a_vec_np=np.linalg.solve(t_mat,v_vec)
10. # Solve from SciPy
11. a_vec_sp=sp.linalg.solve(t_mat,v_vec)
```



That we get `AttributeError: module 'scipy' has no attribute 'linalg'`.

This error displays because the subpackages of the SciPy library need to be imported individually. The reason behind this is for efficiency. In the vast majority of cases, only one or a couple of the subpackages will be used during a single session and importing a large number of unused functions will result in the program running much slower. In the case above to use `scipy.linalg` we would have to type in:

```
import scipy.linalg
```

Or

```
import scipy.linalg as linalg
```

Or the most commonly used form:

```
from scipy import linalg
```

If we compare this to:

```
import numpy as np
```

```
1. import matplotlib.pyplot as plt
2. import scipy as sp
```

We can see that in the case of the commonly used libraries we generally use a 2-3 letter abbreviation. This is because these libraries are commonly referenced and having a 2-3 word abbreviation can save a lot of time. A single letter abbreviation is usually avoided to prevent confusion with assignment of a variable. For instance:

```
1. import numpy as n
2. import matplotlib.pyplot as p
   :
500. n=1
501. p=2
502. n.pi
```

For the specialised subpackages of the scipy library we generally use the form:

```
from scipy import linalg
```

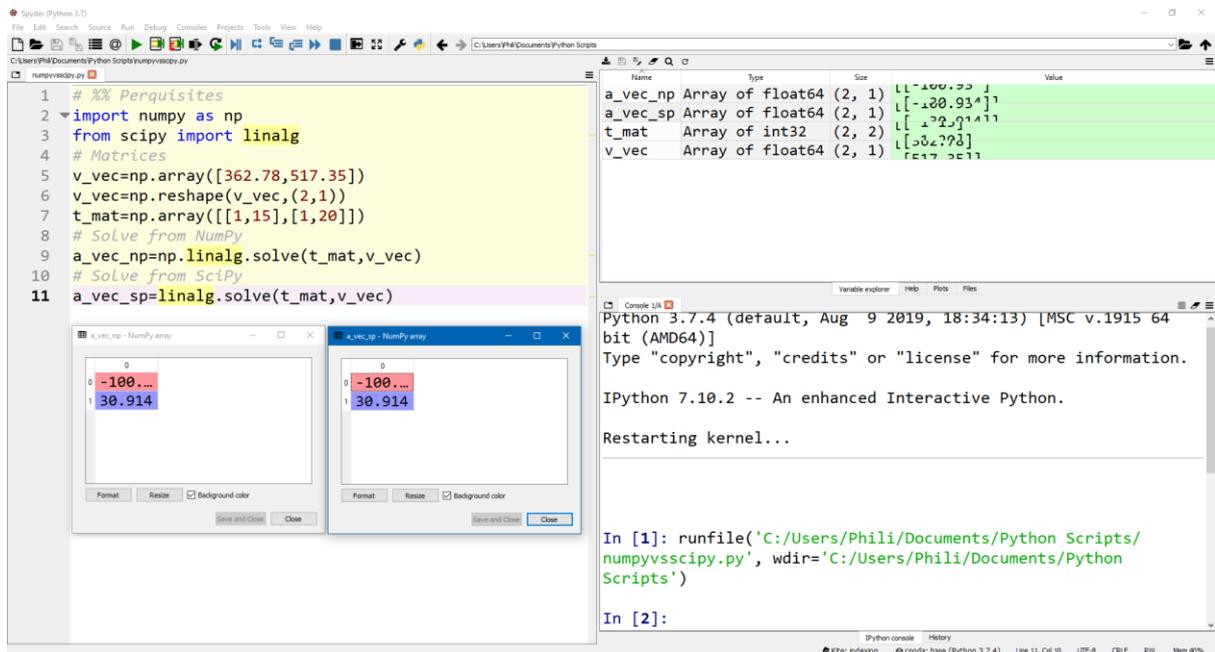
In this case we avoid abbreviating the subpackage, firstly because we are likely to only reference a handful of functions contained within the subpackage, within our code and secondly because leaving the full name makes it easier for others to follow our code than it would be had an arbitrary abbreviation been introduced. The subpackage can be imported before it is used however it is usually more common to import it at the top of the file, so others coming across your code know what modules they need installed to run your code.

```
1. # Perquisites
2. import numpy as np
3. from scipy import linalg
4. # Matrices
5. v_vec=np.array([362.78,517.35])
6. v_vec=np.reshape(v_vec,(2,1))
7. t_mat=np.array([[1,15],[1,20]])
8. # Solve from NumPy
```

```

9. a_vec_np=np.linalg.solve(t_mat,v_vec)
10. # Solve from SciPy
11. a_vec_sp=linalg.solve(t_mat,v_vec)

```



Note in this case both functions yield an identical result. In some cases however the function in SciPy is updated and may include additional information.

scipy.stats

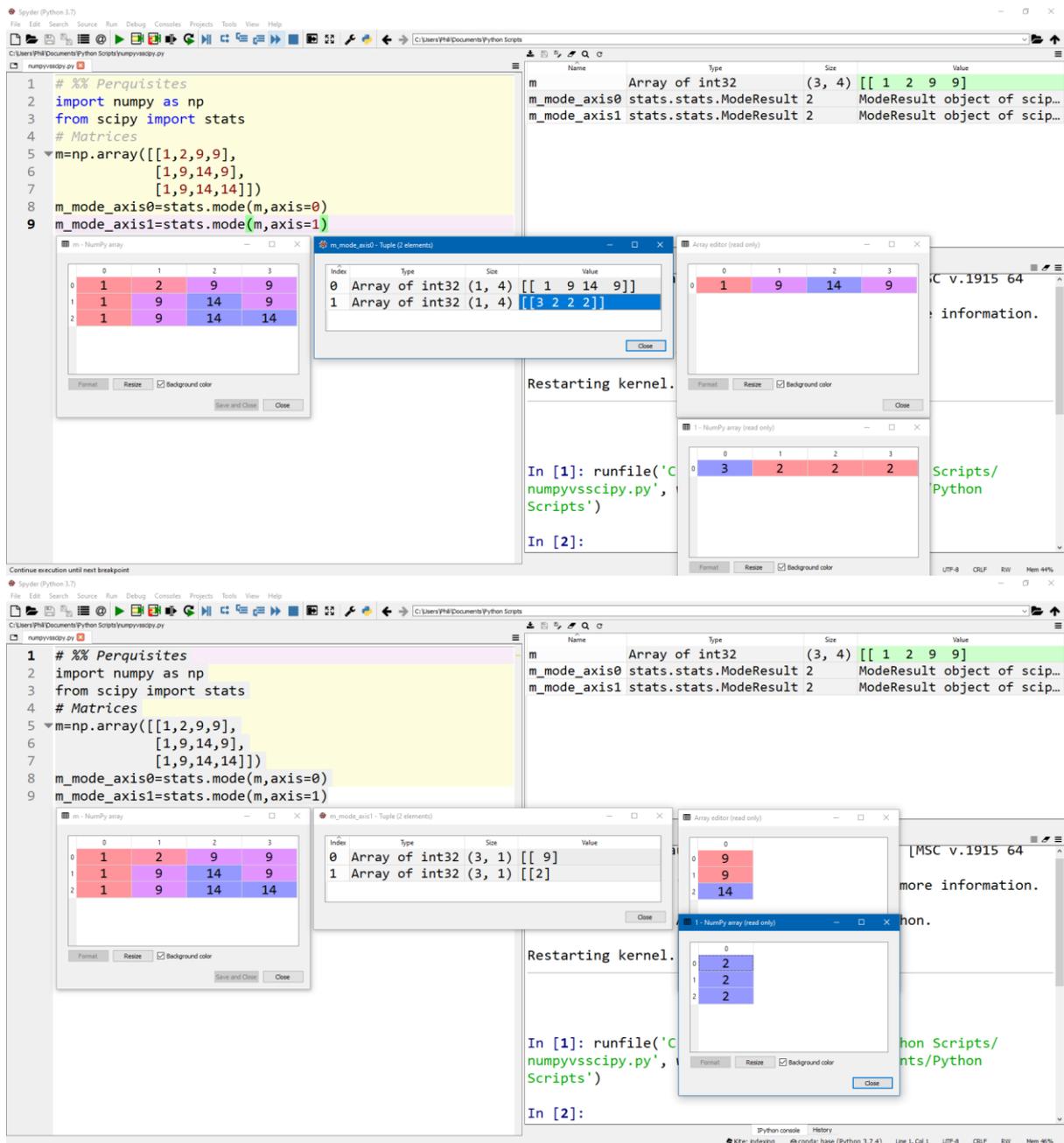
Another subpackage is the statistics subpackage called stats. We can import it and in this case just use one of the most basic statistical functions not present in numpy `stats.mode` which calculates the mode giving the number in each column (if `axis=0`) that appears the most alongside the number of times that number appears.

```

1. # %% Perquisites
2. import numpy as np
3. from scipy import stats
4. # Matrices
5. m=np.array([[1,2,9,9],
6.             [1,9,14,9],
7.             [1,9,14,14]])
8. m_mode_axis0=stats.mode(m,axis=0)
9. m_mode_axis1=stats.mode(m,axis=1)

```

The result generated is a nested tuple with the 0th element being the mode of each column (`axis=0`) and the 1st element being the occurrence of each number in each column (`axis=0`).

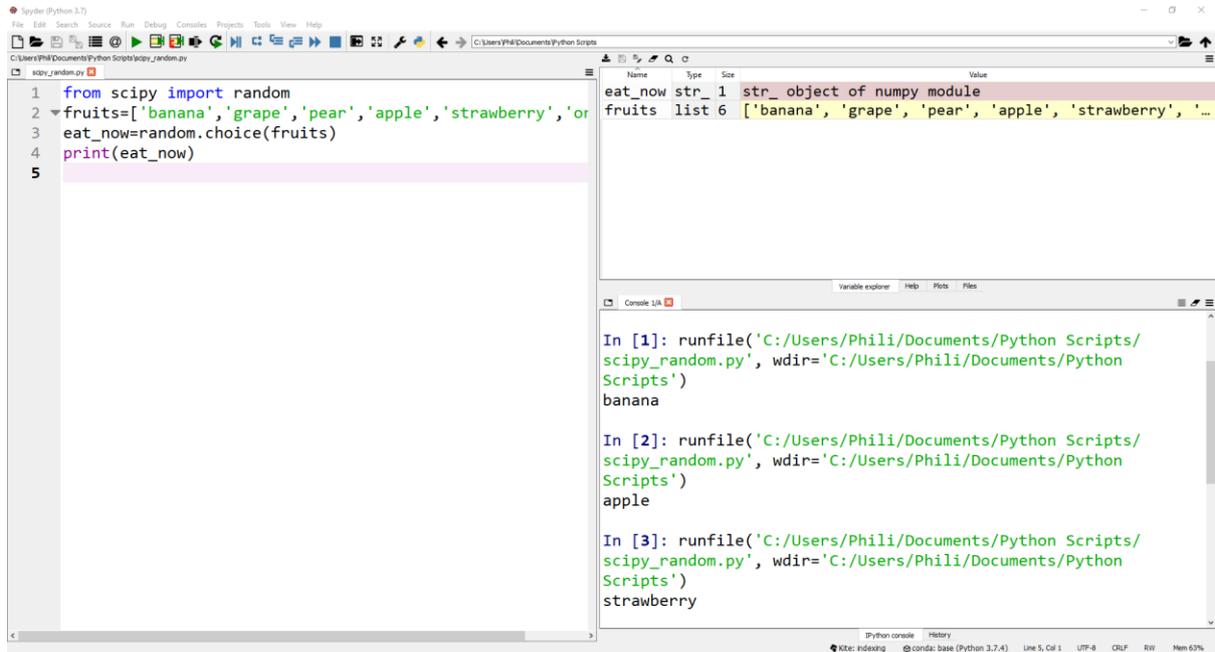


scipy.random

The `scipy` library also contains a `random` submodule. We can use the function `random.choice()` to randomly select a value from a list:

1. `from scipy import random`
2. `fruits=['banana', 'grape', 'pear', 'apple', 'strawberry', 'orange']`
3. `eat_now=random.choice(fruits)`
4. `print(eat_now)`

When ran three times we get three random values printed, `'banana'`, `'apple'` and `'strawberry'` respectively.



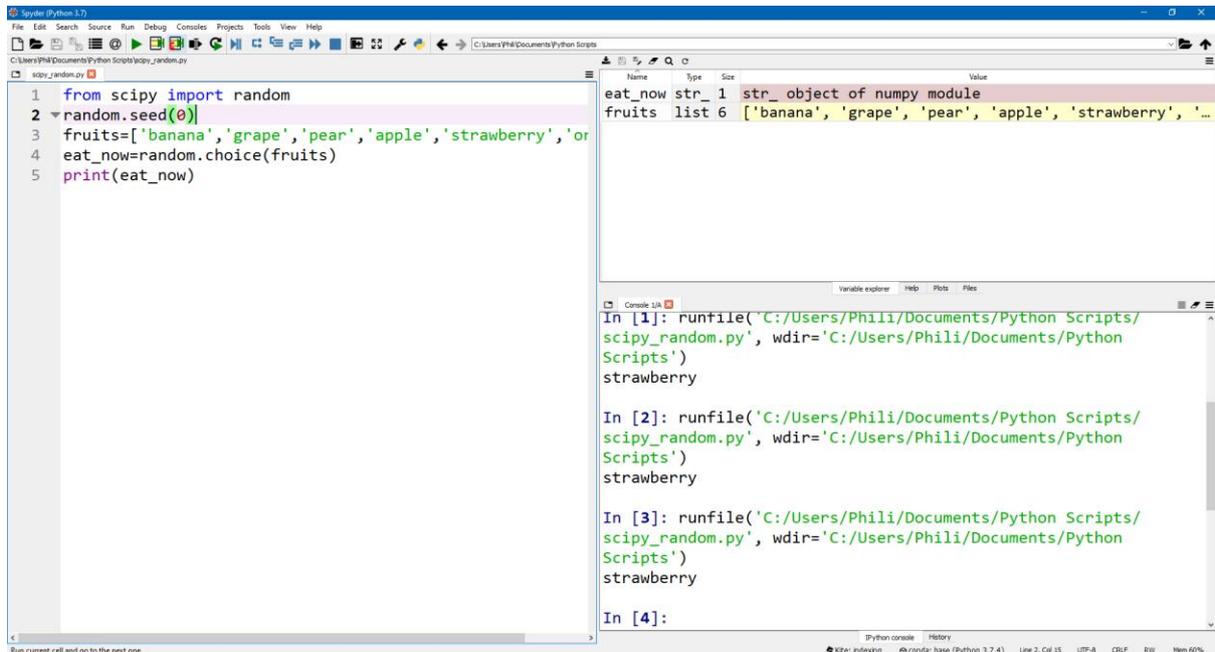
Although the function choice seems to select a completely random choice, it has a sequence of its own and this can be reset by using `random.seed(0)`.

```

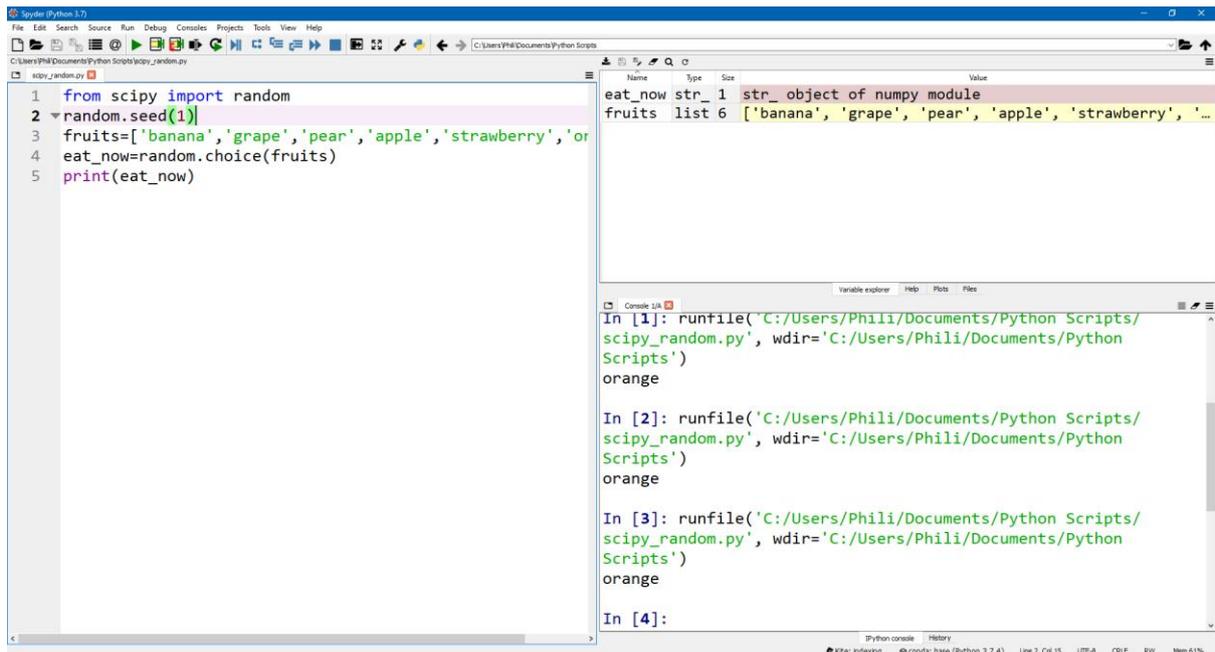
1. from scipy import random
2. random.seed(0)
3. fruits=['banana', 'grape', 'pear', 'apple', 'strawberry', 'orange']
4. eat_now=random.choice(fruits)
5. print(eat_now)

```

This will always yield the value at the start of the sequence which corresponds to `'strawberry'`.

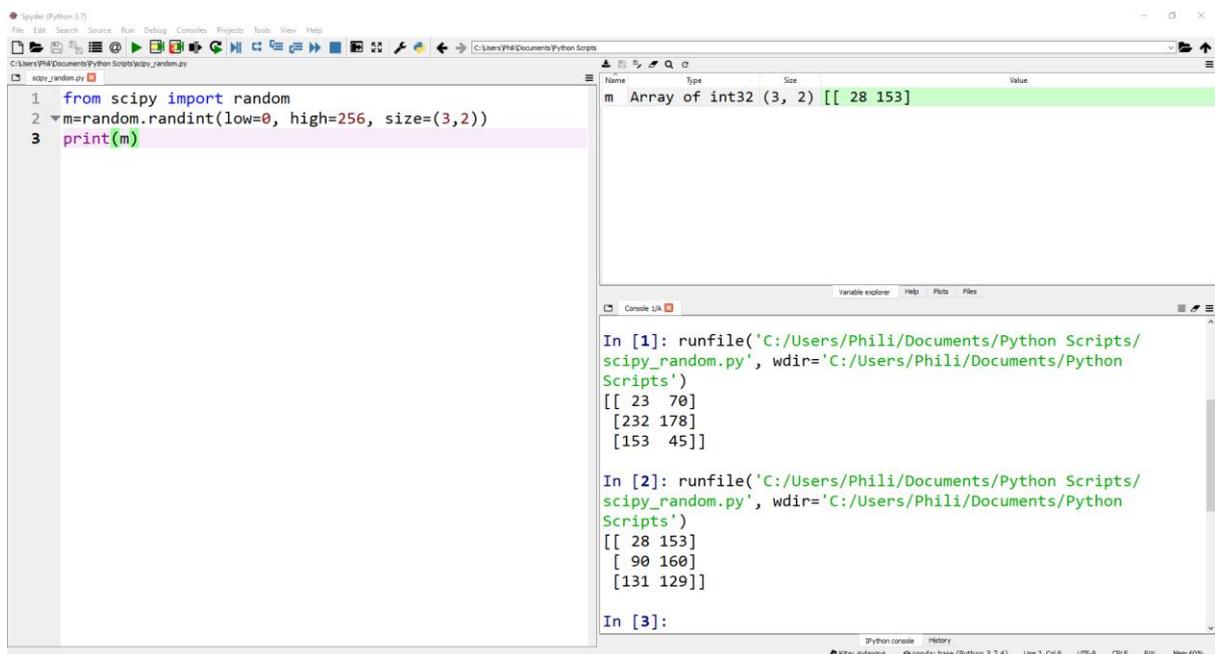


Changing the input value of `random.seed()` to another value for example `1` will change it to the set random value in the sequence and once again this value will display continuously.



We can also create an array of random integers by using the function `random.randint()` which has three keyword input arguments; `low`, `high` and `size`.

1. `from scipy import random`
2. `random.randint(low=0, high=256, size=(3,2))`
3. `print(m)`



As this function is commonly used these keyword input arguments can be dropped and taken as positional arguments, similar to the functions `np.arange()` and `np.linspace()`. Although we can rearrange the keyword input arguments it is advised to use them in the order above. As this function is commonly used, these three values can instead be directly input as positional input arguments which have to be in the correct order.

1. `from scipy import random`
2. `random.randint(0, 256, (3,2))`
3. `print(m)`

```

1 from scipy import random
2 m=random.randint(0,256,(3,2))
3 print(m)

```

Name	Type	Size	Value
m	Array of int32	(3, 2)	[[219 215]]

```

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/scipy_random.py', wdir='C:/Users/Phili/Documents/Python Scripts')
[[157 112]
 [ 72 153]
 [141 124]]

In [2]: runfile('C:/Users/Phili/Documents/Python Scripts/scipy_random.py', wdir='C:/Users/Phili/Documents/Python Scripts')
[[219 215]
 [151 52]
 [110 128]]

In [3]:

```

For the size keyword argument, if only a scalar is input opposed to a tuple, a random vector will be generated opposed to a matrix.

1. `from scipy import random`
2. `v=random.randint(0, 256, 3)`
3. `print(v)`

```

1 from scipy import random
2 v=random.randint(0, 256, 3)
3 print(v)

```

Name	Type	Size	Value
v	Array of int32	(3,)	[18 236 136]

```

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/scipy_random.py', wdir='C:/Users/Phili/Documents/Python Scripts')
[ 62 40 156]

In [2]: runfile('C:/Users/Phili/Documents/Python Scripts/scipy_random.py', wdir='C:/Users/Phili/Documents/Python Scripts')
[196 184 165]

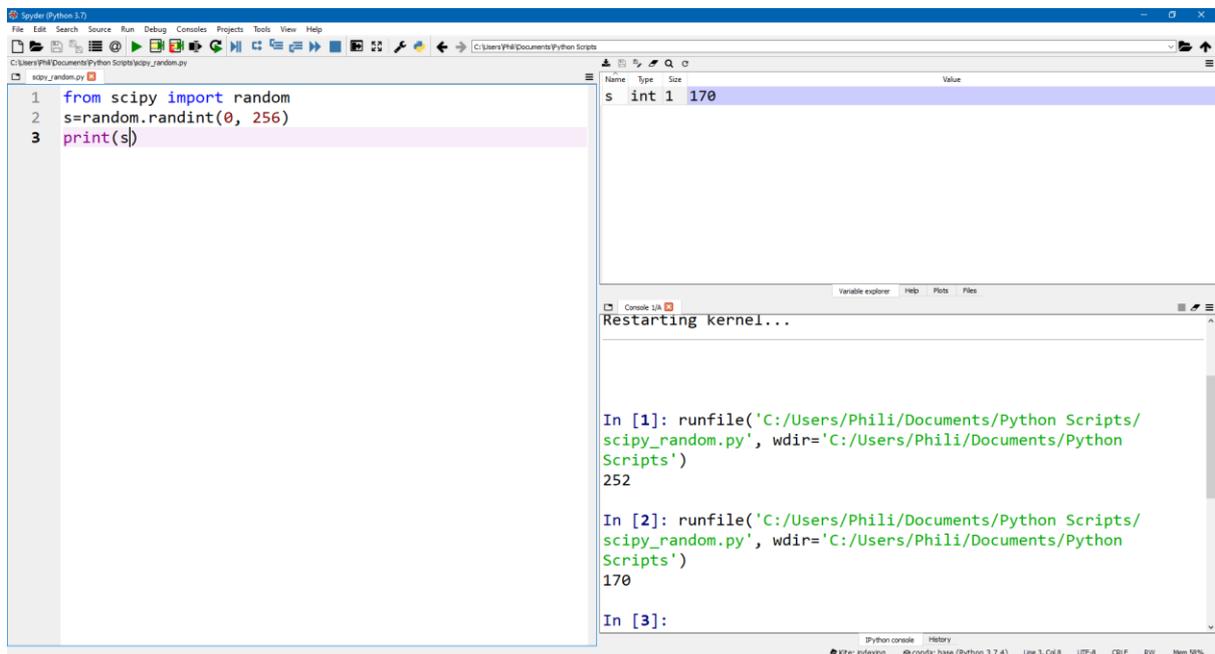
In [3]: runfile('C:/Users/Phili/Documents/Python Scripts/scipy_random.py', wdir='C:/Users/Phili/Documents/Python Scripts')
[ 18 236 136]

In [4]:

```

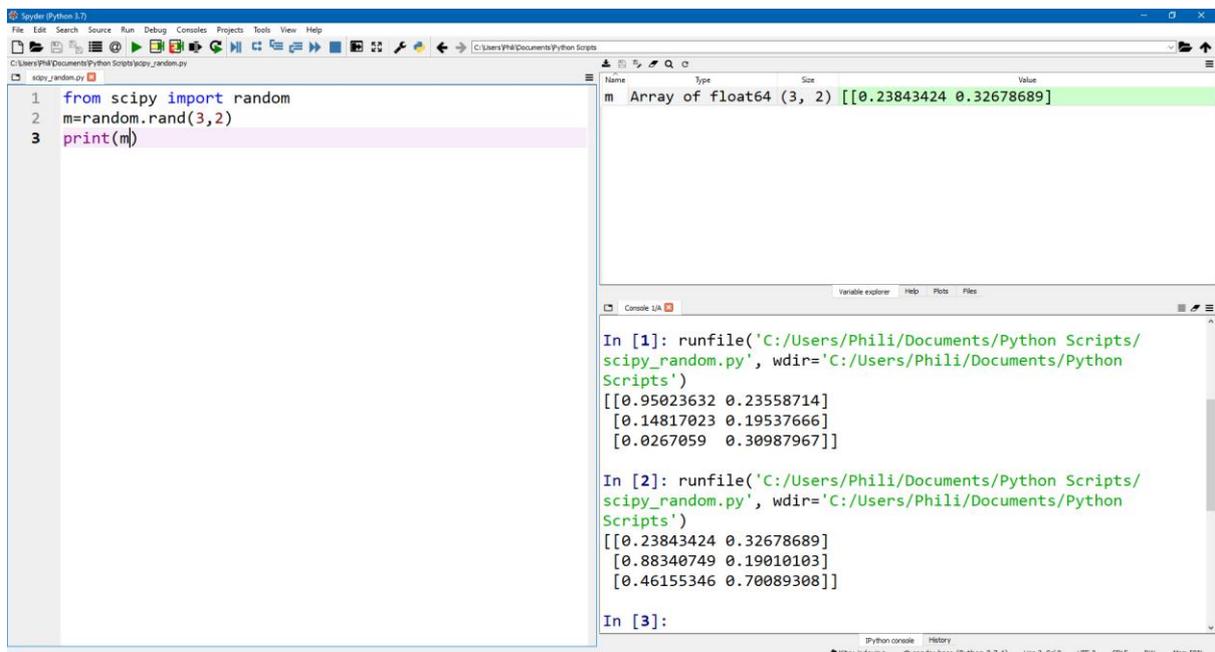
And if no number is input for this positional input argument, it will take the default value of `1` and generate a scalar.

1. `from scipy import random`
2. `s=random.randint(0, 256)`
3. `print(s)`



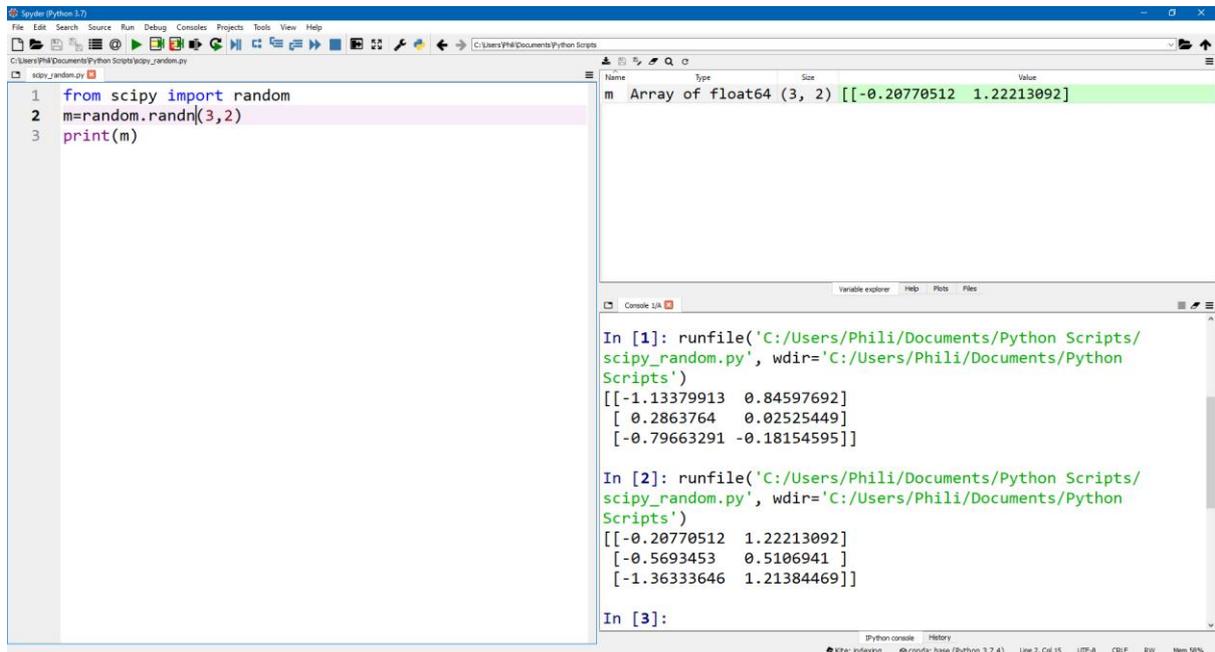
In addition to the `random.randint()` function above, there is the function `random.rand()` which generates a random number between `0` and `1`.

1. `from scipy import random`
2. `m=random.rand(3, 2)`
3. `print(m)`



There is also the function `random.randn()` which is randomly distributed about the origin.

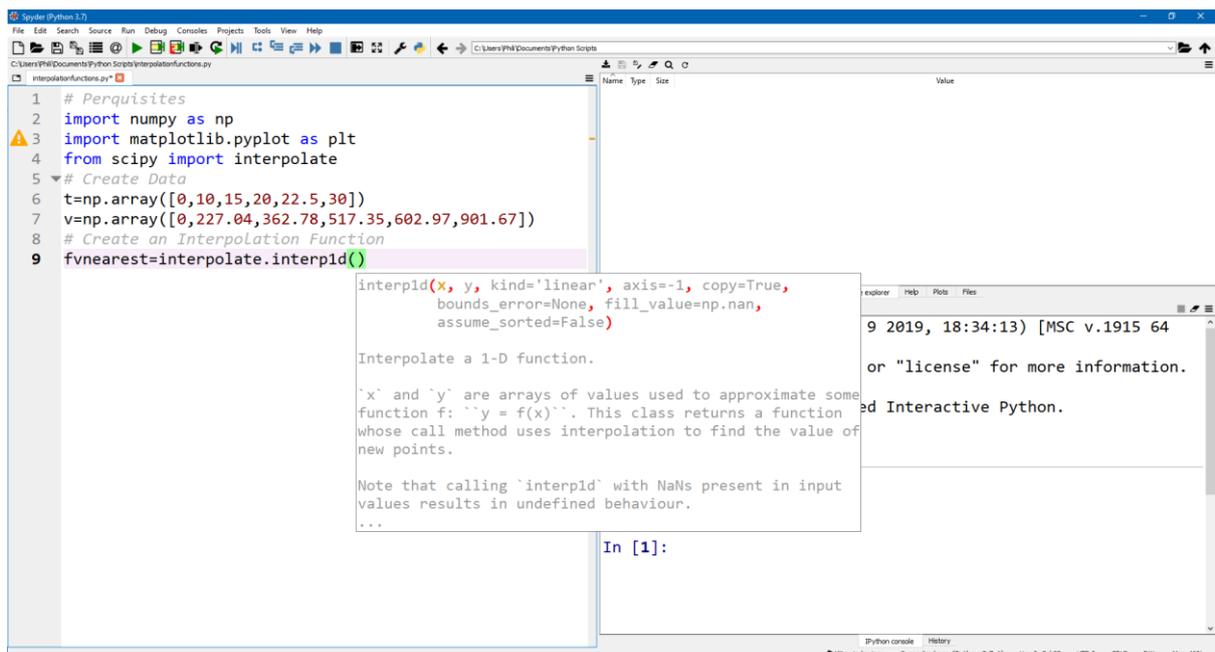
1. `from scipy import random`
2. `m=random.randn(3, 2)`
3. `print(m)`



These three functions can be used to build up additional random functions by addition and multiplication of scalars if required.

scipy.interpolate

To interpolate data we can use `scipy.interpolate` and from the `scipy.interpolate` subpackage we can use the `interp1d` function which has two input arguments, the original `x` and `y` data and a keyword input argument `kind` which has a default value `'linear'`. We can then assign this interpolation function to a name.

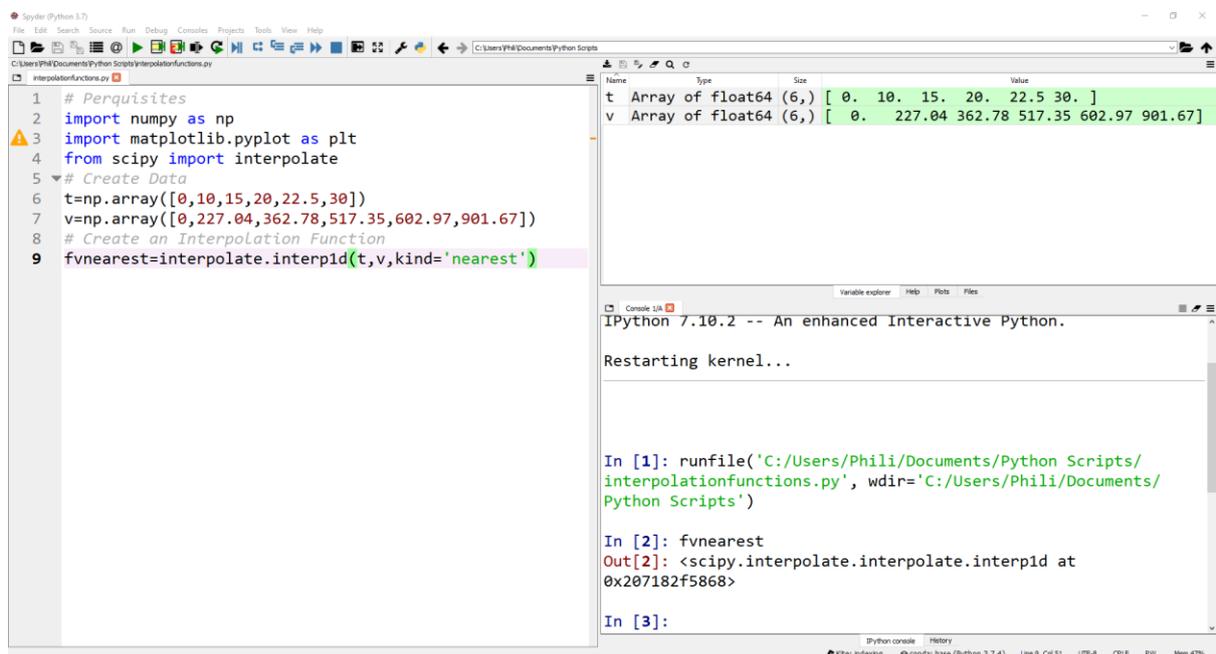


```

1. # Perquisites
2. import numpy as np
3. import matplotlib.pyplot as plt
4. from scipy import interpolate
5. # Create Data
6. t=np.array([0,10,15,20,22.5,30])
7. v=np.array([0,227.04,362.78,517.35,602.97,901.67])
8. # Create an Interpolation Function
9. fvnearest=interpolate.interp1d(t,v,kind='nearest')

```

When this code is executed however the name `fvnearest` does not show up on the variable explorer. This is because the output of `interpolate.interp1d(t,v,kind='nearest')` is itself a function and function names do not show up in the variable explorer. We can see it exists

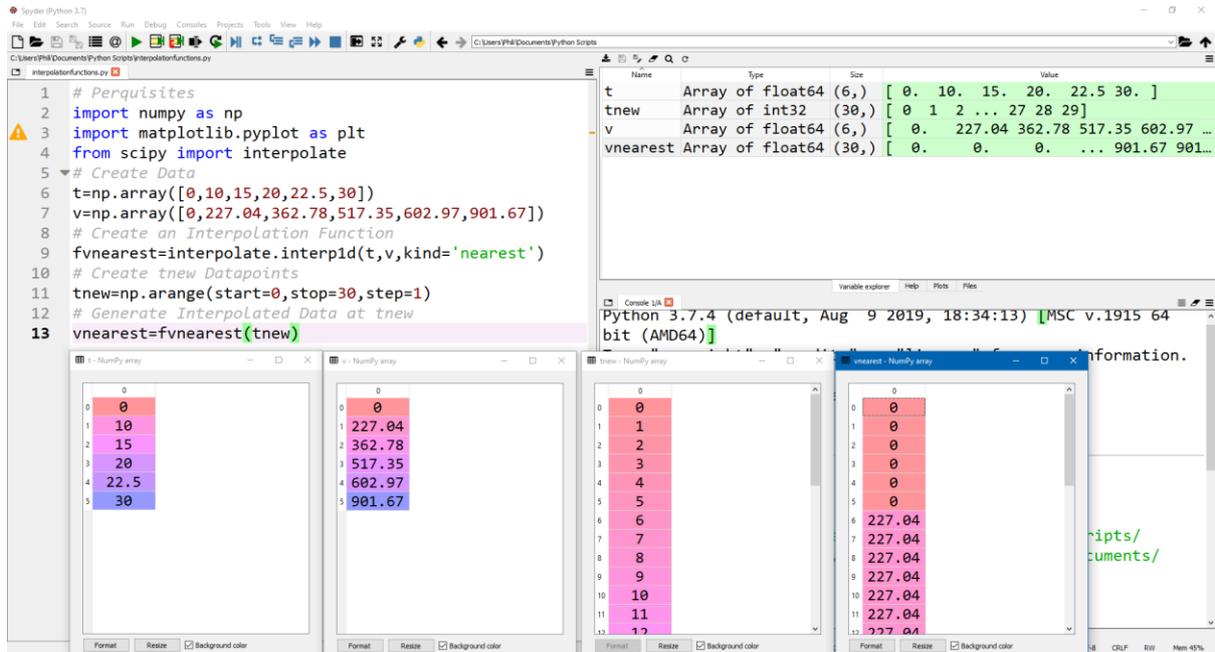


by typing it into the console. This interpolation function can be applied to new time points to create interpolated data.

```

1. # Perquisites
2. import numpy as np
3. import matplotlib.pyplot as plt
4. from scipy import interpolate
5. # Create Data
6. t=np.array([0,10,15,20,22.5,30])
7. v=np.array([0,227.04,362.78,517.35,602.97,901.67])
8. # Create an Interpolation Function
9. fvnearest=interpolate.interp1d(t,v,kind='nearest')
10. # Create tnew Datapoints
11. tnew=np.arange(start=0,stop=30,step=1)
12. # Generate Interpolated Data at tnew
13. vnearest=fvnearest(tnew)

```



Note however that `interp1d` (`scipy.interpolate.interp1d`) is itself a nested subpackage so instead of using:

```
from scipy import interpolate
```

Followed by:

```
fvnearest=interpolate.interp1d(t,v,kind='nearest')
```

It is more commonly imported directly:

```
from scipy.interpolate import interp1d
```

Then called as:

```
fvnearest=interp1d(t,v,kind='nearest')
```

Now supposing we wanted to create a matrix where the 0th column is the time points, the 1st column is nearest interpolated data, the 2nd column is linearly interpolated data and the 3rd column is cubic interpolated data we could type:

```

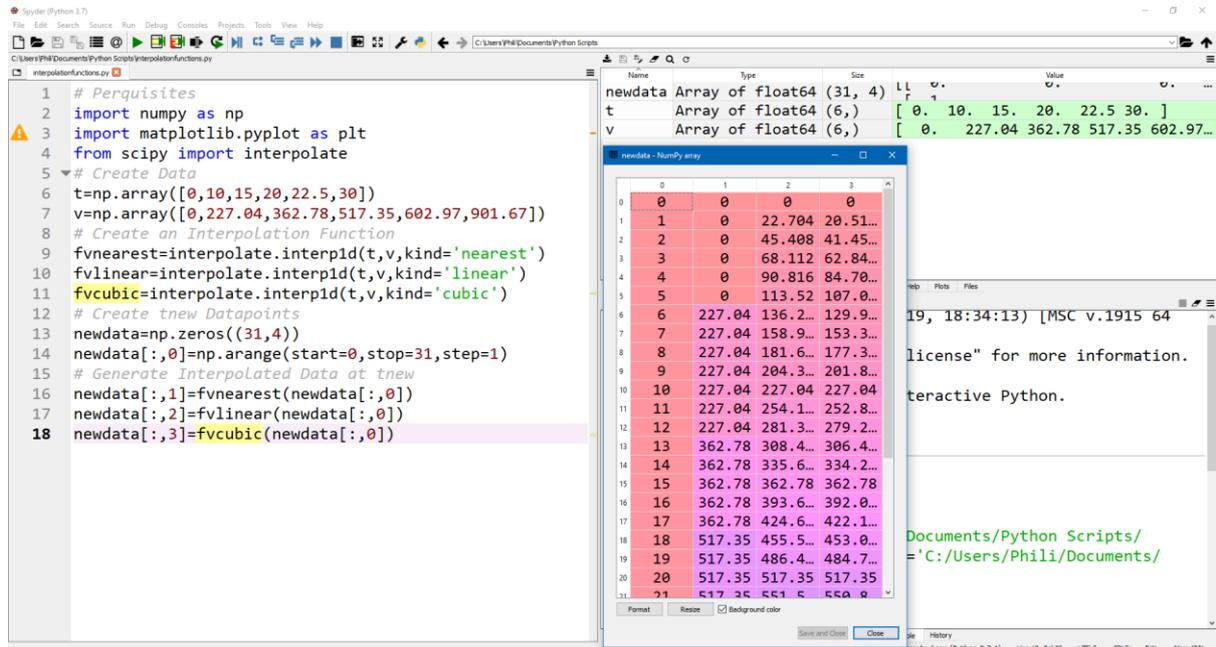
1. # Perquisites
2. import numpy as np
3. import matplotlib.pyplot as plt
4. from scipy import interpolate
5. # Create Data
6. t=np.array([0,10,15,20,22.5,30])
7. v=np.array([0,227.04,362.78,517.35,602.97,901.67])
8. # Create an Interpolation Function
9. fvnearest=interpolate.interp1d(t,v,kind='nearest')
10. fvlinear=interpolate.interp1d(t,v,kind='linear')
11. fvcubic=interpolate.interp1d(t,v,kind='cubic')
12. # Create tnew Datapoints
13. newdata=np.zeros((31,4))

```

```

14. newdata[:,0]=np.arange(start=0,stop=31,step=1)
15. # Generate Interpolated Data at tnew
16. newdata[:,1]=fvnearest(newdata[:,0])
17. newdata[:,2]=fvlinear(newdata[:,0])
18. newdata[:,3]=fvcubic(newdata[:,0])

```

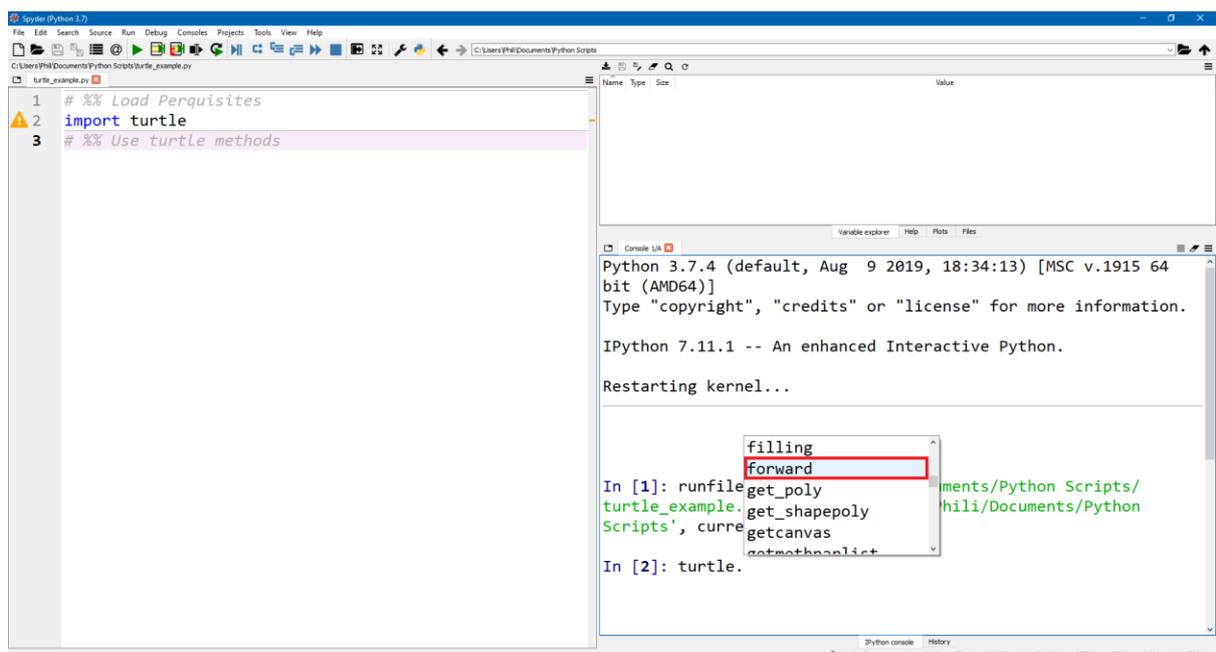


This gives us the matrix `newdata` as expected.

Object Orientated Programming and Classes

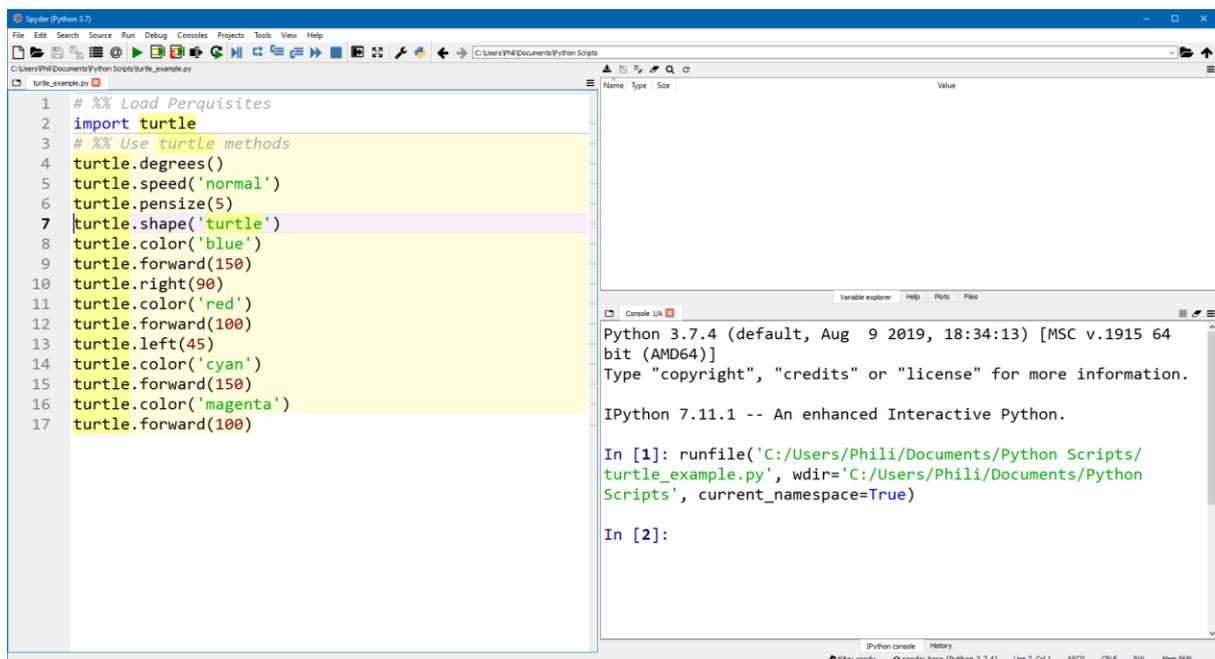
Introduction to the Turtle Library

Let us `import` the `turtle` library to examine some of the basics behind object orientated programming and classes. First, let's just use the library and call the functions in the turtle module.

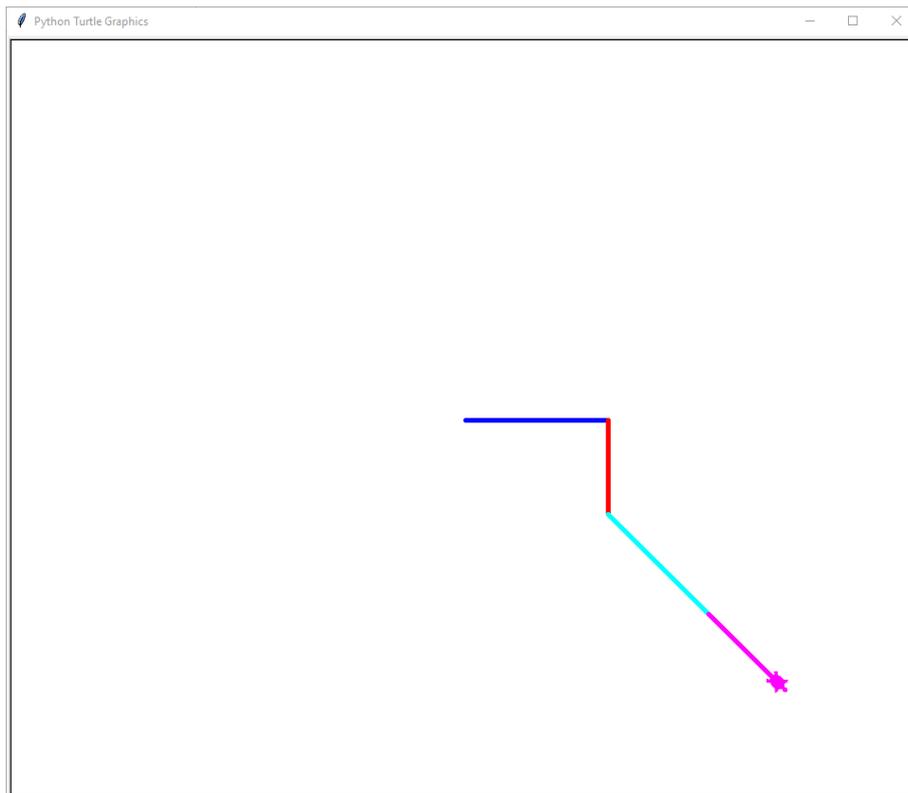


To call up a list of available functions type in the library name `turtle` followed by a dot `.` and then a tab `↵`. The functions `turtle.degrees()`, `turtle.pensize()`, `turtle.speed()` and `turtle.color()` all relate to the properties of the lines being drawn while `turtle.forward()`, `turtle.back()` relate to the length of the line and `turtle.right()` and `turtle.left()` relate to a rotation of the direction.

```
1. # %% Load Perquisites
2. import turtle
3. # %% Use turtle methods
4. turtle.degrees()
5. turtle.speed('normal')
6. turtle.pensize(5)
7. turtle.shape('turtle')
8. turtle.color('blue')
9. turtle.forward(150)
10. turtle.right(90)
11. turtle.color('red')
12. turtle.forward(100)
13. turtle.left(45)
14. turtle.color('cyan')
15. turtle.forward(150)
16. turtle.color('magenta')
17. turtle.forward(100)
```



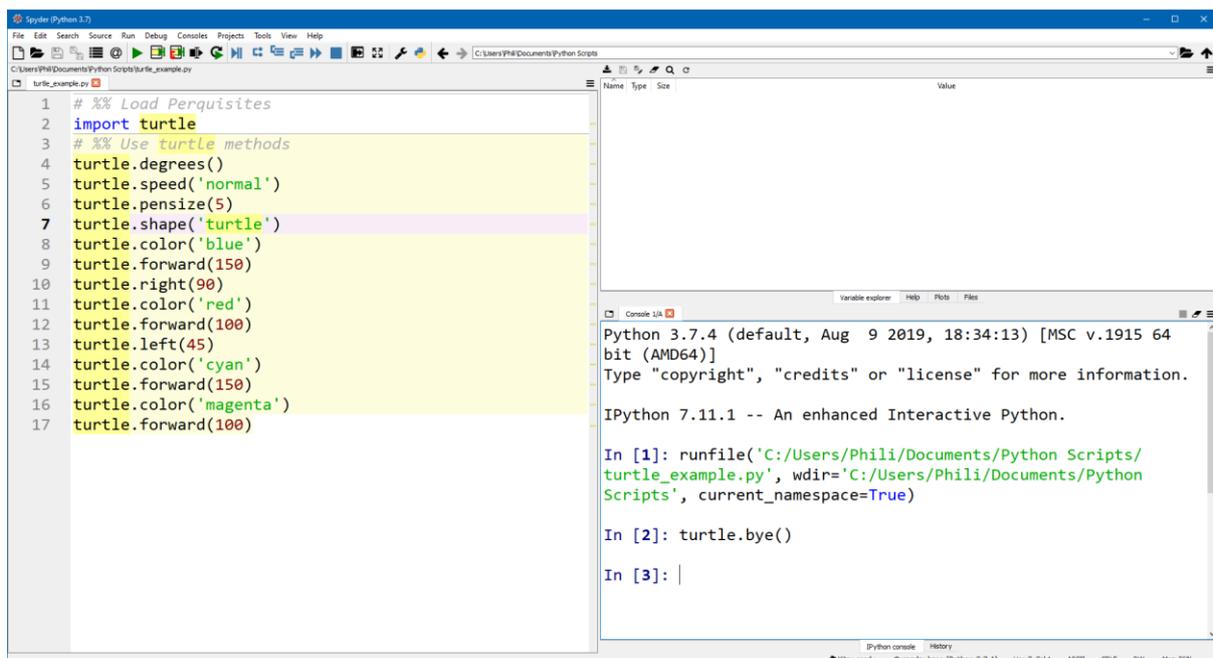
When we launch this, we see a python turtle graphics window with a line that has a width of 5.



It starts from the centre and follows the commands to go forward for 150 where it is colored blue, turn right 90 degrees, then forward for 100 where it is colored red, turn left 45 degrees, then move forwards for 150 where it is colored cyan and then move forward for 100 where it is colored magenta.

Selecting the lose button on the python turtle graphics window may cause spyder to hang. Instead to close the python turtle graphics window, we need to type the following into the console:

```
turtle.bye()
```

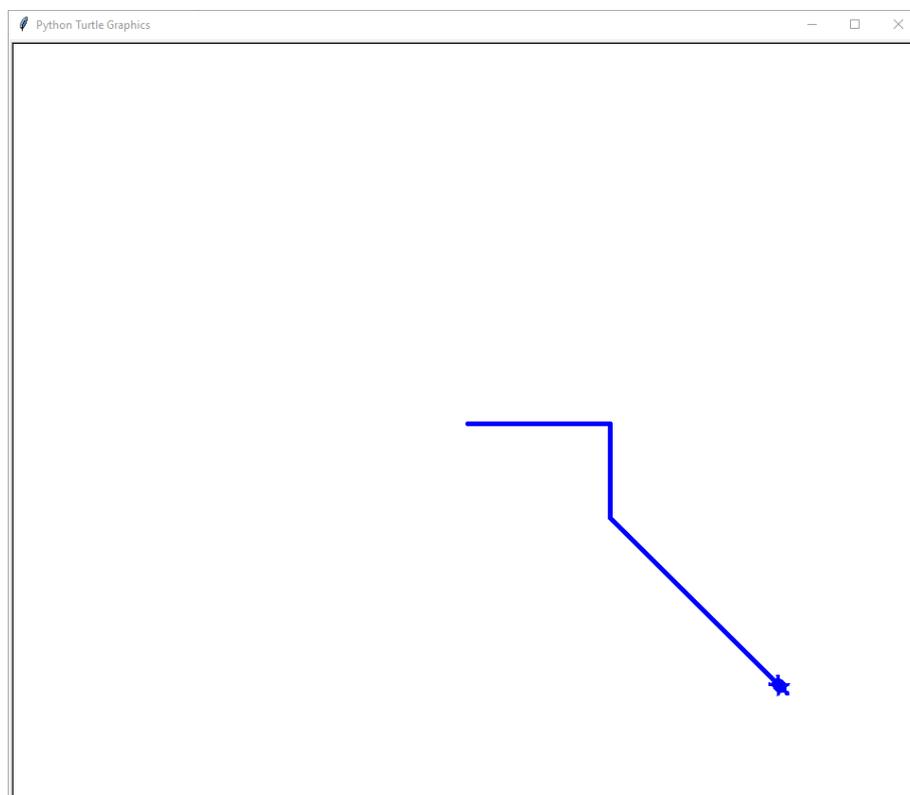


For clarity let's rerun the code just using a single color:

```

1. # %% Load Perquisites
2. import turtle
3. # %% Use turtle methods
4. turtle.degrees()
5. turtle.speed('normal')
6. turtle.pensize(5)
7. turtle.shape('turtle')
8. turtle.color('blue')
9. turtle.forward(150)
10. turtle.right(90)
11. turtle.forward(100)
12. turtle.left(45)
13. turtle.forward(150)
14. turtle.forward(100)

```



Creating an Instance of a Class

Now we can ask ourselves the obvious question, how do we map out the trajectory of another turtle (in the same Python Turtle Graphics Window)? If we keep giving commands as we have done previously, then the original turtle we created will continue to follow them. To command another turtle we need to look at the concept of defining an object of the turtle class and then applying commands or methods to that object which is known as object orientated programming. Let us create four instances of the `Turtle` class which we can independently command and then have a look at the code.

```

1. # %% Load Perquisites
2. import turtle
3. # %% Create and Command Turtle Leonardo

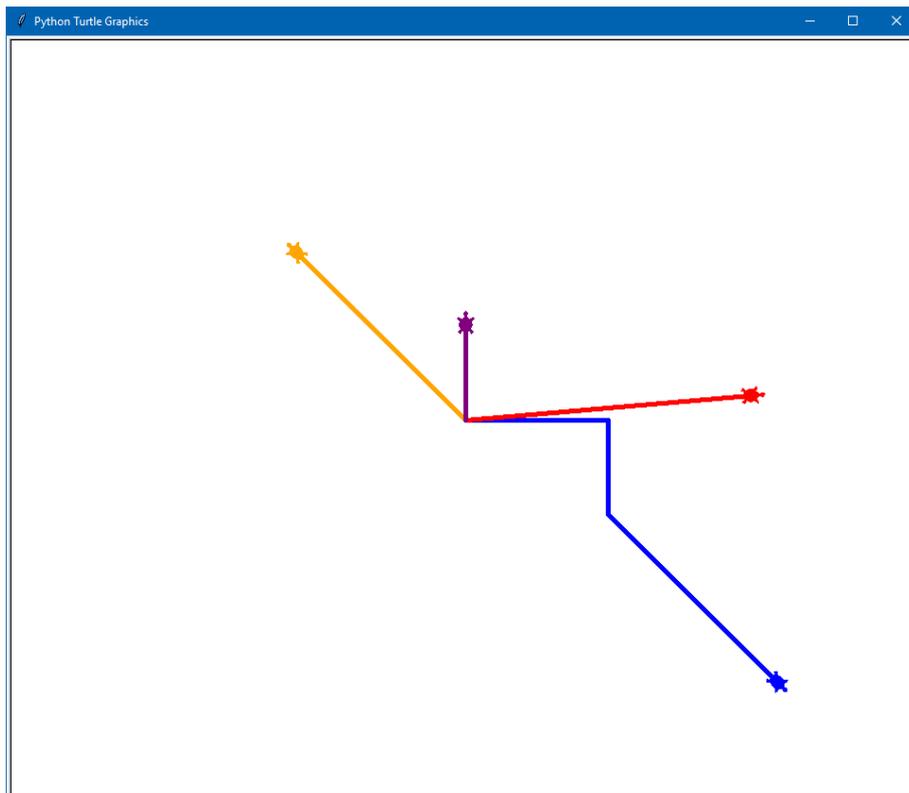
```

```

4. leonardo=turtle.Turtle()
5. leonardo.color('blue')
6. leonardo.pensize(5)
7. leonardo.shape('turtle')
8. leonardo.degrees()
9. leonardo.forward(150)
10. leonardo.right(90)
11. leonardo.forward(100)
12. leonardo.left(45)
13. leonardo.forward(150)
14. leonardo.forward(100)
15.
16. # %% Create and Command Turtle Raphael
17. raphael=turtle.Turtle()
18. raphael.color('red')
19. raphael.pensize(5)
20. raphael.shape('turtle')
21. raphael.degrees()
22. raphael.left(5)
23. raphael.forward(300)
24.
25. # %% Create and Command Turtle Michelangelo
26. michelangelo=turtle.Turtle()
27. michelangelo.color('orange')
28. michelangelo.pensize(5)
29. michelangelo.shape('turtle')
30. michelangelo.degrees()
31. michelangelo.left(135)
32. michelangelo.forward(250)
33.
34. # %% Create and Command Turtle Donatello
35. donatello=turtle.Turtle()
36. donatello.color('purple')
37. donatello.pensize(5)
38. donatello.shape('turtle')
39. donatello.degrees()
40. donatello.left(90)
41. donatello.forward(100)

```

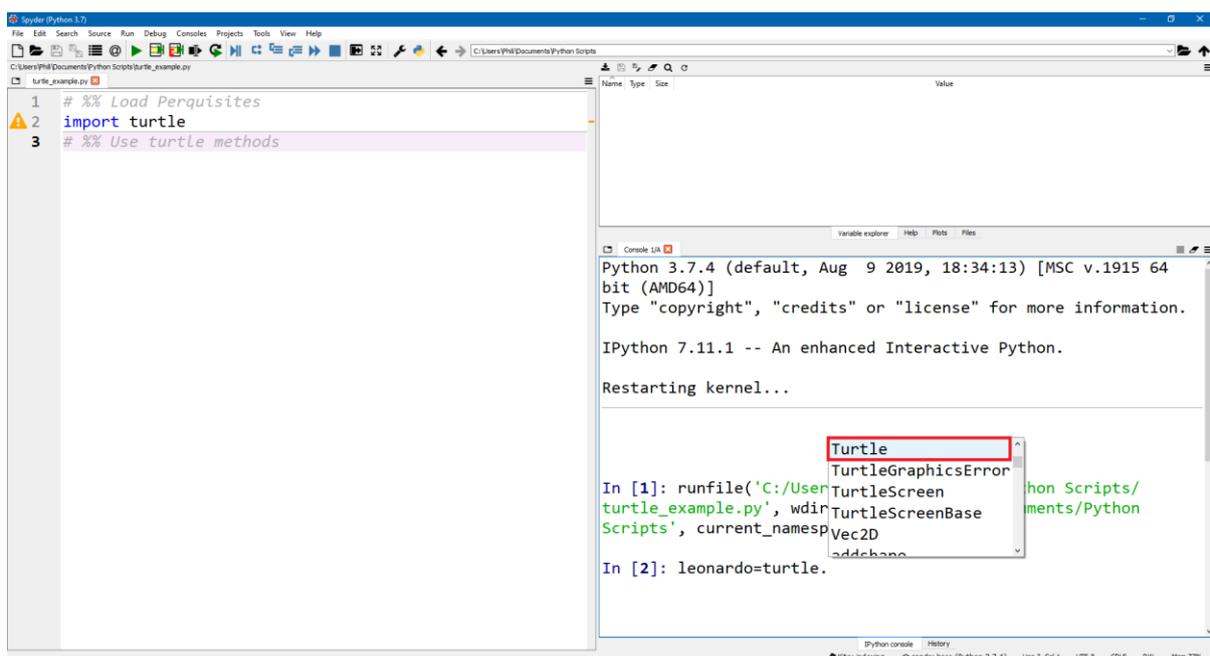
When this script is ran, we see the following:



In this code four instances of the `Turtle` class also known as turtle objects were created (line 6, 19, 28, 37) and display within the variable explorer. These lines of code have the following form:

```
name=turtle.Turtle()
```

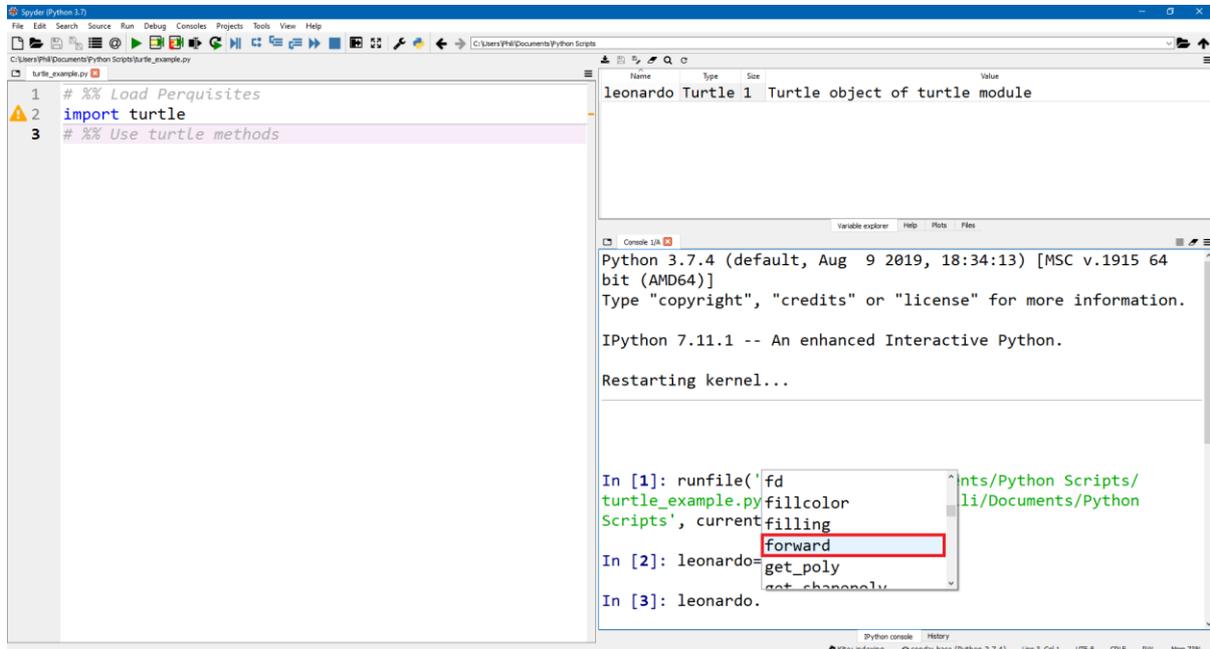
These class instances are created by calling the `Turtle` class (note that third party classes are typically `CamelCaseCapitalized`) from the `turtle` module. Each instance of the `Turtle` class is a turtle object and inherits a variety of methods from the parent class. If we type in dot `.` and then tab `↵` we'll see a list of all the functions available. However, in this list, note that the words at the top are capitalized. This is because they are all classes.



Let's create an object that is an instance of the Turtle class. To do this we use a variable name. For example:

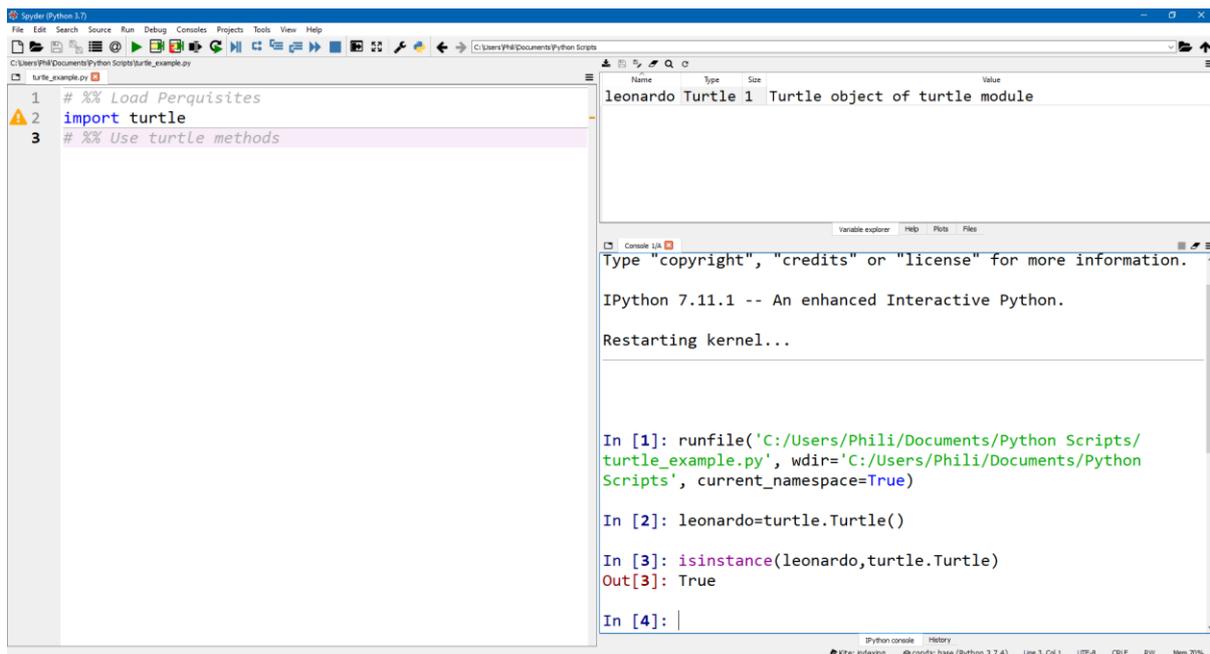
```
leonardo=turtle.Turtle()
```

Once this is created, we can instead type in the name of the instance of the class and then follow this by a dot `.` and then tab `↵`.



To access the classes available to the turtle object, we can type in the name of the turtle object followed by a dot `.` and then a tab `↵`. These methods are inherited from the parent class. We can check whether an object is an instance of a class using the function `isinstance` in this case:

```
isinstance(leonardo,turtle.Turtle)
```



Now let's compare this to something that we have worked before, a list.

```
a1=[1,2,3]
```

Which can also be input as:

```
a2=list([1,2,3])
```

We can check if these are an instance of the class `list` by using:

```
isinstance(a1,list)
```

```
isinstance(a2,list)
```

The screenshot shows the Spyder Python IDE interface. The editor on the left contains the following code:

```
1 # %% Load Perquisites
2 import turtle
3 # %% Use turtle methods
```

The Variable explorer on the right shows the following variables:

Name	Type	Size	Value
a1	list	3	[1, 2, 3]
a2	list	3	[1, 2, 3]
leonardo	Turtle	1	Turtle object of turtle module

The console on the right shows the following execution history:

```
In [2]: leonardo=turtle.Turtle()
Out[2]:
In [3]: isinstance(leonardo,turtle.Turtle)
Out[3]: True
In [4]: a1=[1,2,3]
In [5]: a2=list([1,2,3])
In [6]: isinstance(a1,list)
Out[6]: True
In [7]: isinstance(a2,list)
Out[7]: True
In [8]: |
```

Once the list is created we can type in the name of the list object followed by a dot `.` and then tab `↵`.

The screenshot shows the Spyder Python IDE interface. The editor on the left contains the following code:

```
1 # %% Load Perquisites
2 import turtle
3 # %% Use turtle methods
```

The Variable explorer on the right shows the same variables as in the previous screenshot.

The console on the right shows the following execution history:

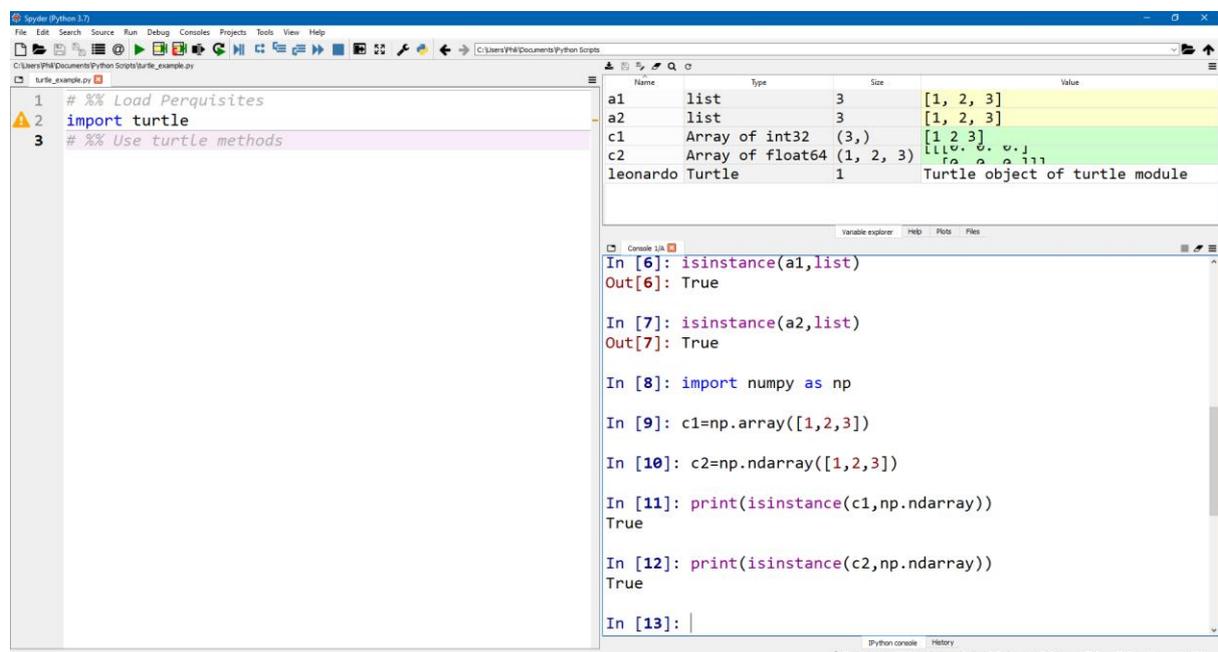
```
In [2]: leonardo=turtle.Turtle()
Out[2]:
In [3]: isinstance(leonardo,turtle.Turtle)
Out[3]: True
In [4]: a1=[1,2,3]
In [5]: a2=list([1,2,3])
In [6]: isiappend 1,list)
Out[6]: True
In [7]: isicount 2,list)
Out[7]: True
In [8]: a1.
```

A dropdown menu is visible below the console, listing the following methods: `clear`, `copy`, `count`, `extend`, and `index`.

We can see all the methods available to this list `a1` which is an instance of the class `list`.

Note that unlike custom classes, core Python classes such as `list` are not typically capitalized. Instead they are lower case and usually colored purple. Other inbuilt classes that we have seen before and created instances of include the `tuple` class, the `dict` class, the `set` class and the `bool` class. Recall that there are a substantially larger number of methods available to the `list` class than there are to the `tuple` class for instance. The library `numpy` is technically a third party library but it is the most commonly used python library and therefore uses a similar syntax to the core python library. `numpy` classes are not capitalized, however unfortunately they aren't color coded in `spyder` either. We have used the `numpy` class `ndarray` (which also has an alias `array`).

```
import numpy as np
c=np.array([1,2,3])
c=np.ndarray([1,2,3])
print(isinstance(c1,np.ndarray))
print(isinstance(c2,np.ndarray))
```



Geometric Shapes, Geometry, Angles and For Loops

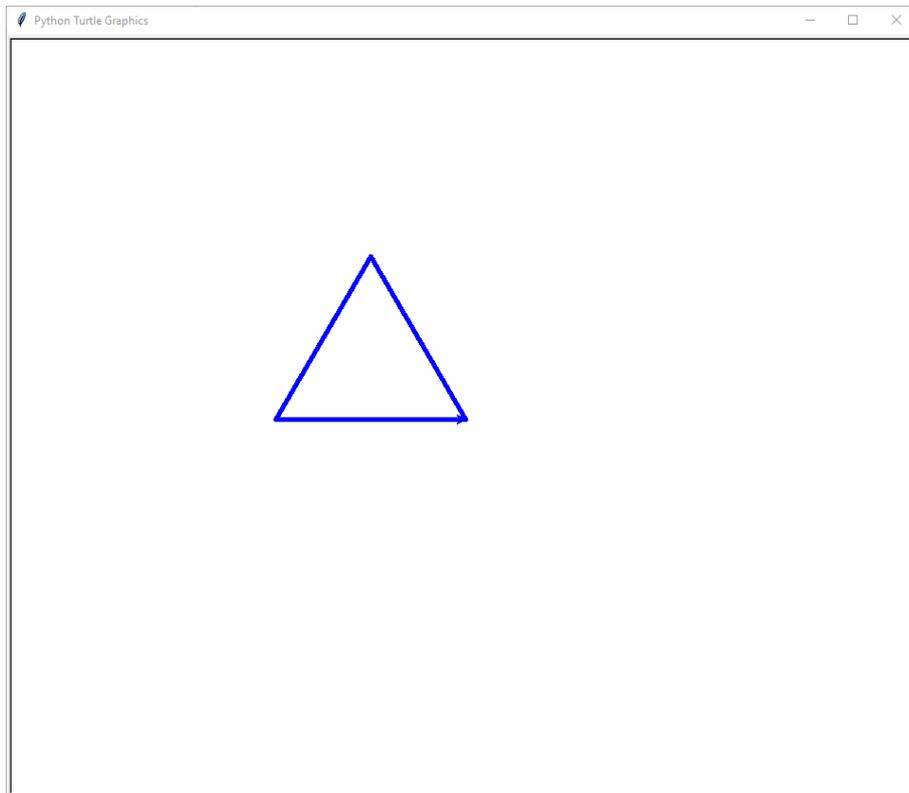
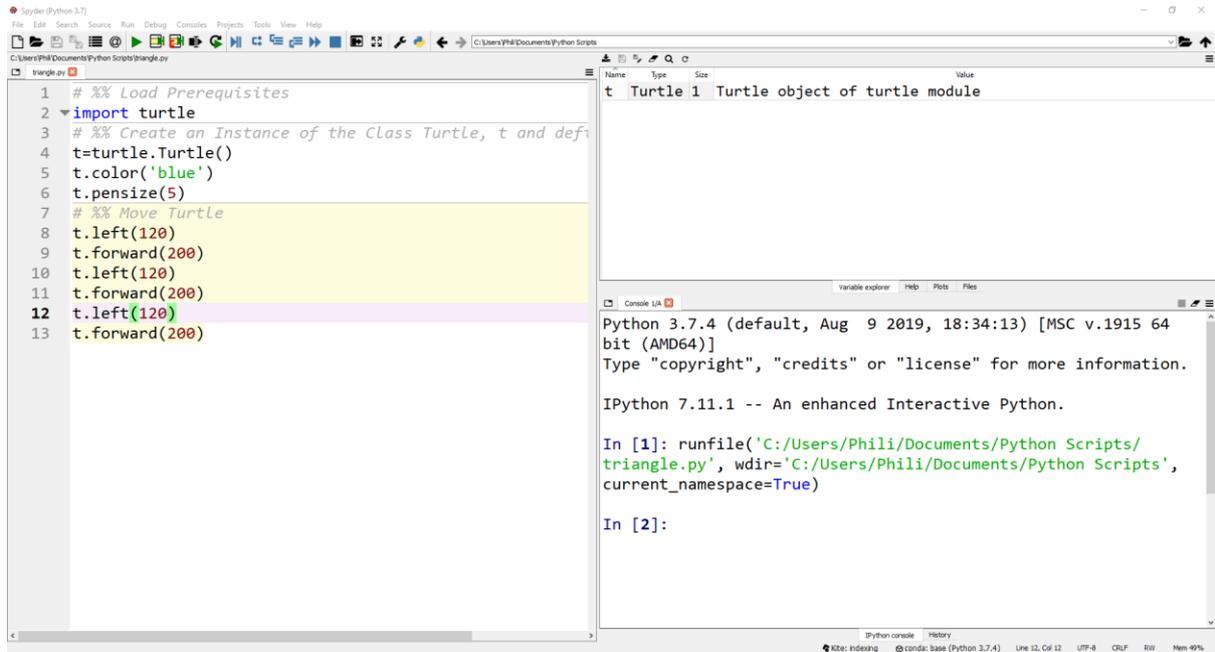
The `turtle` module has `turtle.penup()` and `turtle.pendown()` methods which we can use to move the turtle without leaving a line on the python turtle graphics window.

```
1. # %% Load Perquisites
2. import turtle
3. # %% Create and Command Turtle Leonardo
4. leonardo=turtle.Turtle()
5. leonardo.color('blue')
6. leonardo.pensize(5)
7. leonardo.shape('turtle')
8. leonardo.degrees()
9. leonardo.pendown()
10. leonardo.forward(5)
11. leonardo.penup()
12. leonardo.forward(50)
```



The turtle library is a good library for visually practicing object orientated programming by using geometry to create shapes or patterns using loops. A triangle for example can be made using:

```
1. # %% Load Prerequisites
2. import turtle
3. # %% Create an Instance of the Class Turtle, t and define
   attributes
4. t=turtle.Turtle()
5. t.color('blue')
6. t.pensize(5)
7. # %% Move Turtle
8. t.left(120)
9. t.forward(200)
10. t.left(120)
11. t.forward(200)
12. t.left(120)
13. t.forward(200)
```



Note the repetition in the lines 8-13 which can be simplified using a **for** loop over each of the sides. To complete the shape, we want to get back to the starting position and we know there are 360 degrees in a full rotation, the angle for each rotation is therefore 360 divided by the number of sides 3. We can also normalise the length by the number of sides, so we don't draw off the turtle graphics window.

```

1. # %% Load Prerequisites
2. import turtle
3. # %% Create an Instance of the Class Turtle, t and define
   attributes

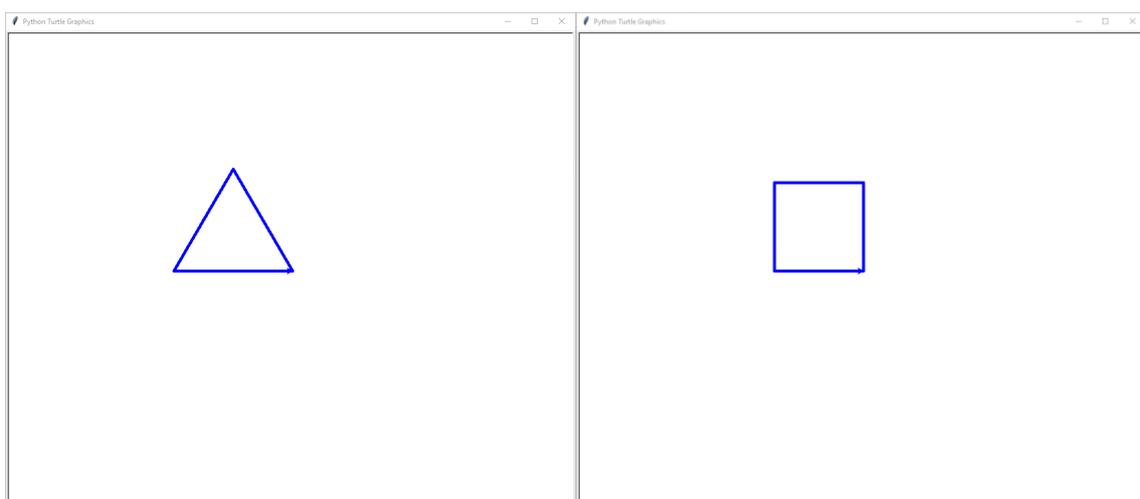
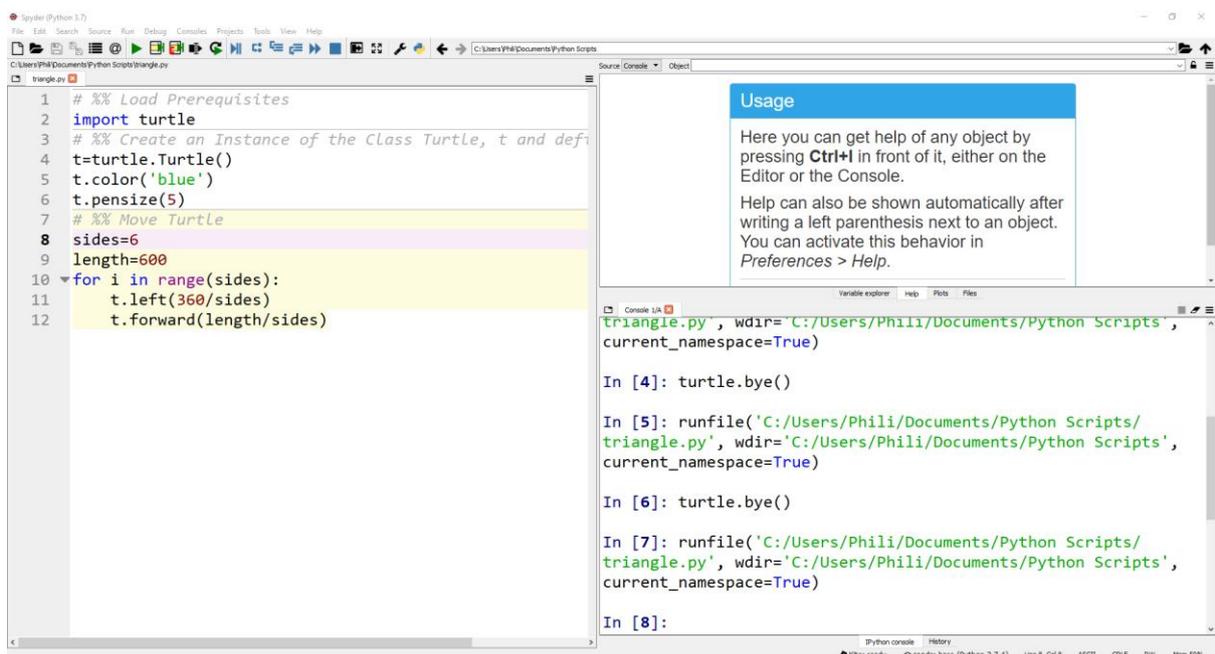
```

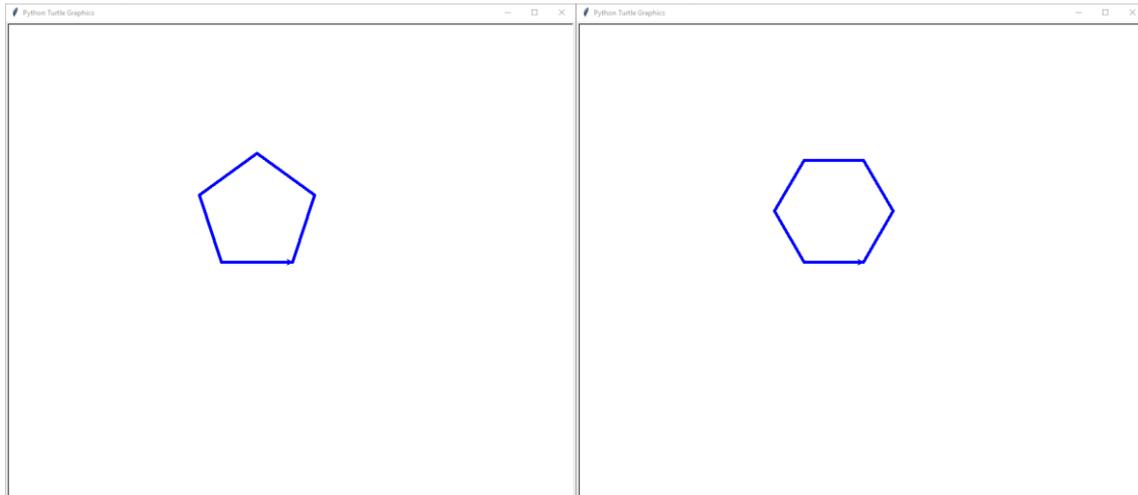
```

4. t=turtle.Turtle()
5. t.color('blue')
6. t.pensize(5)
7. # %% Move Turtle
8. sides=3
9. length=600
10. for i in range(sides):
11.     t.left(360/sides)
12.     t.forward(length/sides)
13.

```

This will draw the exact same triangle, if the variable `sides` is updated to `4`, `5` or `6` (line 8) we will get a square, pentagon or hexagon instead.





Four instances of turtle can be created and set to have different colors and start facing different directions.

```
1. # %% Load Prerequisites
2. import turtle
3. # %% General Attributes
4. color1='red'
5. color2='blue'
6. color3='green'
7. color4='yellow'
8. sides=3
9. length=600
10.
11. # Turtle a
12. a=turtle.Turtle()
13. a.pensize(5)
14. a.color(color1)
15. for i in range(sides):
16.     a.left(360/sides)
17.     a.forward(length/sides)
18.
19. # Turtle b
20. b=turtle.Turtle()
21. b.pensize(5)
22. b.color(color2)
23. b.right(90)
24. for i in range(sides):
25.     b.left(360/sides)
26.     b.forward(length/sides)
27.
28. # Turtle c
29. c=turtle.Turtle()
30. c.pensize(5)
31. c.color(color3)
```

```

32. c.right(180)
33. for i in range(sides):
34.     c.left(360/sides)
35.     c.forward(length/sides)
36.
37. # Turtle d
38. d=turtle.Turtle()
39. d.pensize(5)
40. d.color(color4)
41. d.right(270)
42. for i in range(sides):
43.     d.left(360/sides)
44.     d.forward(length/sides)

```

The screenshot shows the Spyder Python IDE with a file named 'triangles.py'. The code defines four turtles (a, b, c, d) and draws triangles for each. Turtle 'a' is red, 'b' is blue, 'c' is green, and 'd' is yellow. The variable explorer shows the state of these variables: 'a', 'b', 'c', and 'd' are Turtle objects; 'color1', 'color2', 'color3', and 'color4' are strings; 'i' is an integer (1); and 'length' and 'sides' are integers (600 and 3 respectively).

```

1  %% Load Prerequisites
2  import turtle
3  %% General Attributes
4  color1='red'
5  color2='blue'
6  color3='green'
7  color4='yellow'
8  sides=3
9  length=600
10
11 # Turtle a
12 a=turtle.Turtle()
13 a.pensize(5)
14 a.color(color1)
15 for i in range(sides):
16     a.left(360/sides)
17     a.forward(length/sides)
18
19 # Turtle b
20 b=turtle.Turtle()
21 b.pensize(5)
22 b.color(color2)
23 b.right(90)
24 for i in range(sides):
25     b.left(360/sides)
26     b.forward(length/sides)
27
# Turtle c

```

Name	Type	Size	Value
a	Turtle	1	Turtle object of turtle module
b	Turtle	1	Turtle object of turtle module
c	Turtle	1	Turtle object of turtle module
color1	str	1	red
color2	str	1	blue
color3	str	1	green
color4	str	1	yellow
d	Turtle	1	Turtle object of turtle module
i	int	1	2
length	int	1	600
sides	int	1	3

```

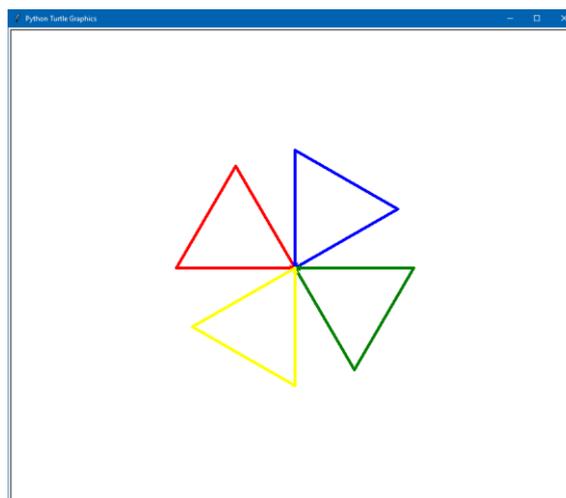
Python 3.7.4 (default, Aug 9 2019, 18:34:13) [MSC v.1915 64
bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.11.1 -- An enhanced Interactive Python.

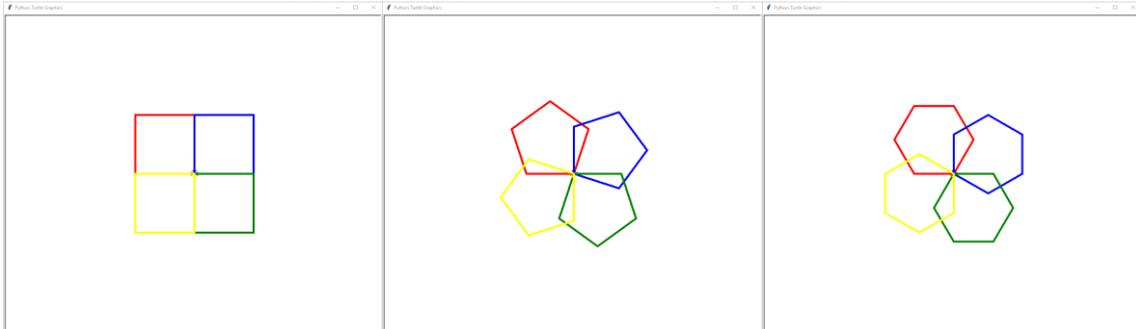
In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/
triangles.py', wdir='C:/Users/Phili/Documents/Python Scripts',
current_namespace=True)

In [2]:

```



Once again other shapes can be made by updating the variable `sides`.

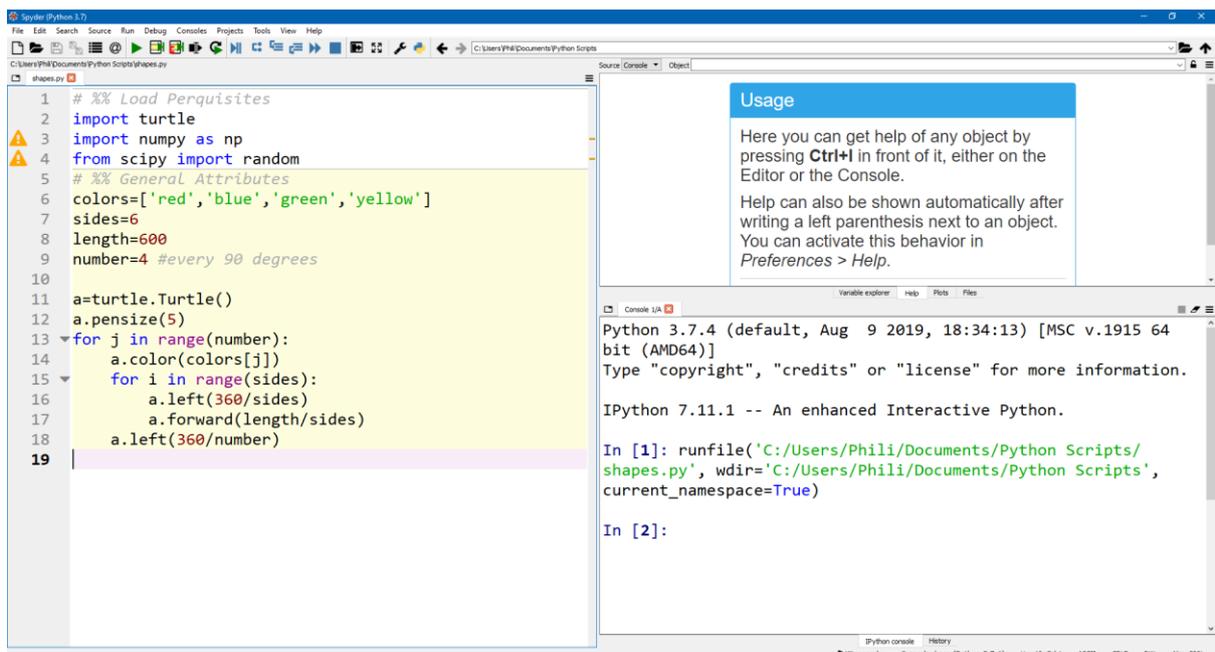


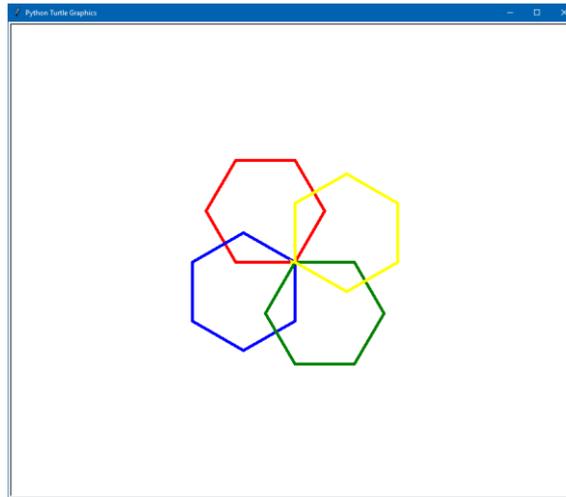
We can take advantage of the fact that when we draw a shape we return to the starting point and instead of creating four separate instances, we can use a single instance in a nested `for` loop.

```

1. # %% Load Prerequisites
2. import turtle
3. # %% General Attributes
4. colors=['red','blue','green','yellow']
5. sides=6
6. length=600
7. number=4 #every 90 degrees
8.
9. a=turtle.Turtle()
10. a.pensize(5)
11. for j in range(number):
12.     a.color(colors[j])
13.     for i in range(sides):
14.         a.left(360/sides)
15.         a.forward(length/sides)
16.     a.left(360/number)

```

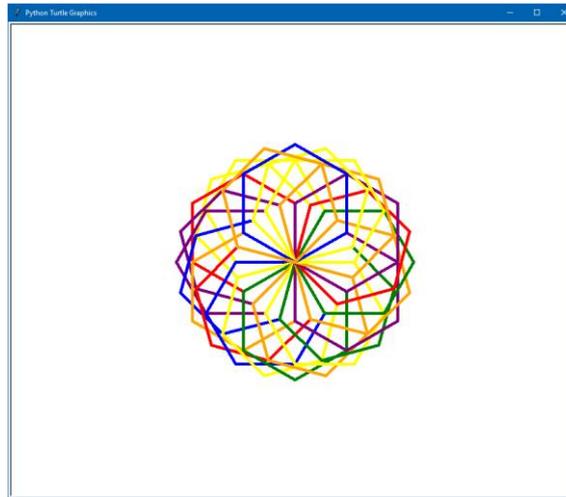




Note however if we want to quickly change the number of shapes, we must also change the color. We can instead specify a larger list of colors and then randomly select a color from the list using `random.choice`.

```
1. # %% Load Perquisites
2. import turtle
3. import numpy as np
4. from scipy import random
5. # %% General Attributes
6. colors=['red', 'blue', 'green', 'yellow', 'purple', 'orange']
7. sides=6
8. length=600
9. number=24 #every 15 degrees
10.
11. a=turtle.Turtle()
12. a.pensize(5)
13. for j in range(number):
14.     a.color(random.choice(colors))
15.     for i in range(sides):
16.         a.left(360/sides)
17.         a.forward(length/sides)
18.     a.left(360/number)
```

This will give a shape like this:



As we have only 6 colors however we see the colors repeat. We can instead create totally random floats for each color channel, and this will give us a much larger selection of random colors. We can also change the drawing speed of a to 0 (line 17) which will draw the shape faster.

```
1. # %% Load Perquisites
2. import turtle
3. import numpy as np
4. from scipy import random
5. # %% Set General Attributes
6. sides=6
7. length=600
8. number=24 #every 15 degrees
9. # %% Initialise seed and colors for loop
10. random.seed(0)
11. r=0
12. g=0
13. b=0
14. # Initialise instance of class
15. a=turtle.Turtle()
16. a.pensize(5)
17. a.speed(0)
18. # Nested For Loop
19. for j in range(number):
20.     r=random.rand()
21.     g=random.rand()
22.     b=random.rand()
23.     a.color((r,g,b))
24.     for i in range(sides):
25.         a.left(360/sides)
26.         a.forward(length/sides)
27.     a.left(360/number)
```

```

1  # %% Load Perquisites
2  import turtle
3  import numpy as np
4  from scipy import random
5  # %% Set General Attributes
6  sides=6
7  length=600
8  number=24 #every 15 degrees
9  # %% Initialise seed and colors for loop
10 random.seed(0)
11 r=0
12 g=0
13 b=0
14 # Initialise instance of class
15 a=turtle.Turtle()
16 a.pensize(5)
17 a.speed(0)
18 # Nested For Loop
19 for j in range(number):
20     r=random.rand()
21     g=random.rand()
22     b=random.rand()
23     a.color((r,g,b))
24     for i in range(sides):
25         a.left(360/sides)
26         a.forward(length/sides)
27         a.left(360/number)

```

```

Console IPython
for i in range(sides):
^
SyntaxError: invalid syntax

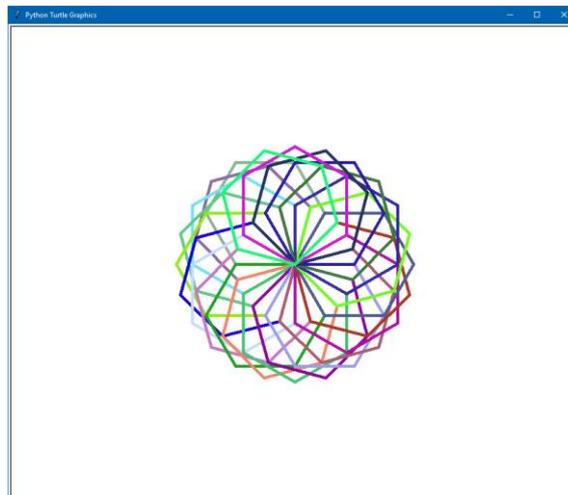
In [4]: runfile('C:/Users/Phili/Documents/Python Scripts/
shapes2.py', wdir='C:/Users/Phili/Documents/Python Scripts',
current_namespace=True)

In [5]: turtle.bye()

In [6]: runfile('C:/Users/Phili/Documents/Python Scripts/
shapes2.py', wdir='C:/Users/Phili/Documents/Python Scripts',
current_namespace=True)

In [7]:

```



In the above case, all the shapes start in the centre. We can move away from the centre by using `penup()` and then when we are in the location we wish to draw, we can use `pendown()`. We can change the orientation of the shape by rotating right by 45 degrees. Instead of just turning left back to the starting position we can rotate left by 145 degrees which will take us to the other side of the circle.

```

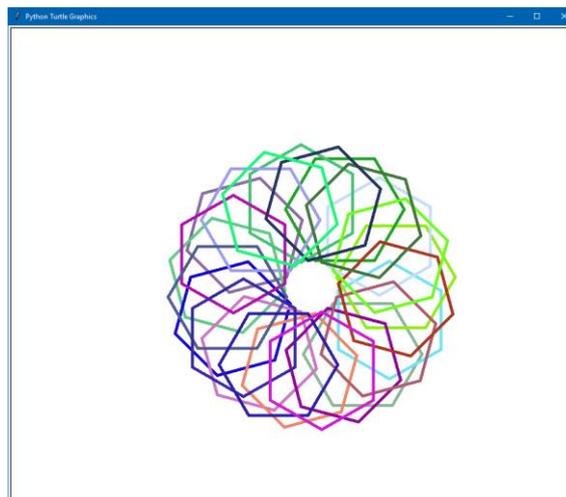
1. # %% Load Perquisites
2. import turtle
3. import numpy as np
4. from scipy import random
5. # %% Set General Attributes
6. sides=6
7. length=600
8. number=24 #every 15 degrees
9. # %% Initialise seed and colors for loop
10. random.seed(0)
11. r=0

```

```

12. g=0
13. b=0
14. # Initialise instance of class
15. a=turtle.Turtle()
16. a.pensize(5)
17. a.speed(0)
18. # Nested For Loop
19. for j in range(number):
20.     r=random.rand()
21.     g=random.rand()
22.     b=random.rand()
23.     a.color((r,g,b))
24.     a.penup()
25.     a.right(45)
26.     a.forward(90)
27.     a.right(135)
28.     a.pendown()
29.     for i in range(sides):
30.         a.left(360/sides)
31.         a.forward(length/sides)
32.     a.left(360/number)

```



A **while** loop can also be used opposed to a **for** loop and during each iteration the distance can be incremented followed by a rotation to create a spiral.

```

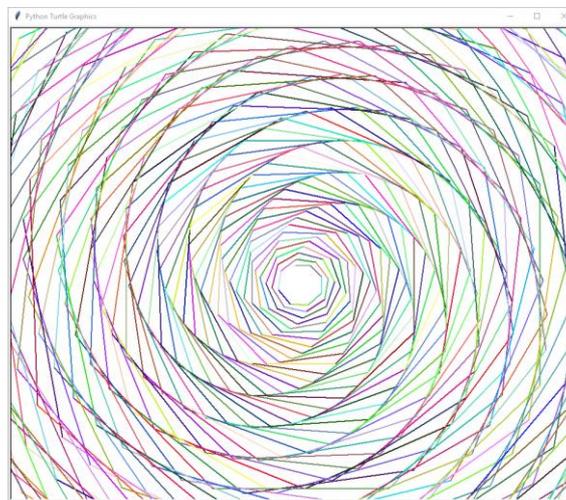
1. # %% Load Perquisites
2. import turtle
3. import numpy as np
4. from scipy import random
5. # %% Initialise seed and colors for loop
6. random.seed(0)
7. r=0
8. g=0
9. b=0
10. x=1

```

```

11. # Initialise instance of class
12. a=turtle.Turtle()
13. a.pensize(2)
14. a.speed(0)
15. # Nested For Loop
16. while x < 500:
17.     r=random.rand()
18.     g=random.rand()
19.     b=random.rand()
20.     a.color((r,g,b))
21.     a.forward(25+x)
22.     a.right(46)
23.     x+=1

```



It is worthwhile modifying the above scripts to create additional shapes and patterns to get more practice in using `for` loops and `while` loops.

Pandas – Python and Data Analysis Library (PANDAS)

In the last two sections we looked at initialising a matrix to having 4 by 30 points, generating the time data in the 0th column using the `np.arange` function and then populating the remaining columns using the interpolation functions for nearest, linear and cubic data. When we look at the matrix however it is not obvious what each column corresponds to. We also looked at the concept behind object orientated programming. We can address this earlier problem by converting the matrix to an instance of the `DataFrame` class found within the `pandas` library. A dataframe is in essence a spreadsheet. Note the use of CamelCaseCapitalization of the word `DataFrame` because we are calling a third library class. For convenience, we will create a dataframe of randomly generated numbers. We will use the matrix of randomly generated integers as the 0th input argument and the list of columns as the 1st input argument of the pandas class `DataFrame`.

```

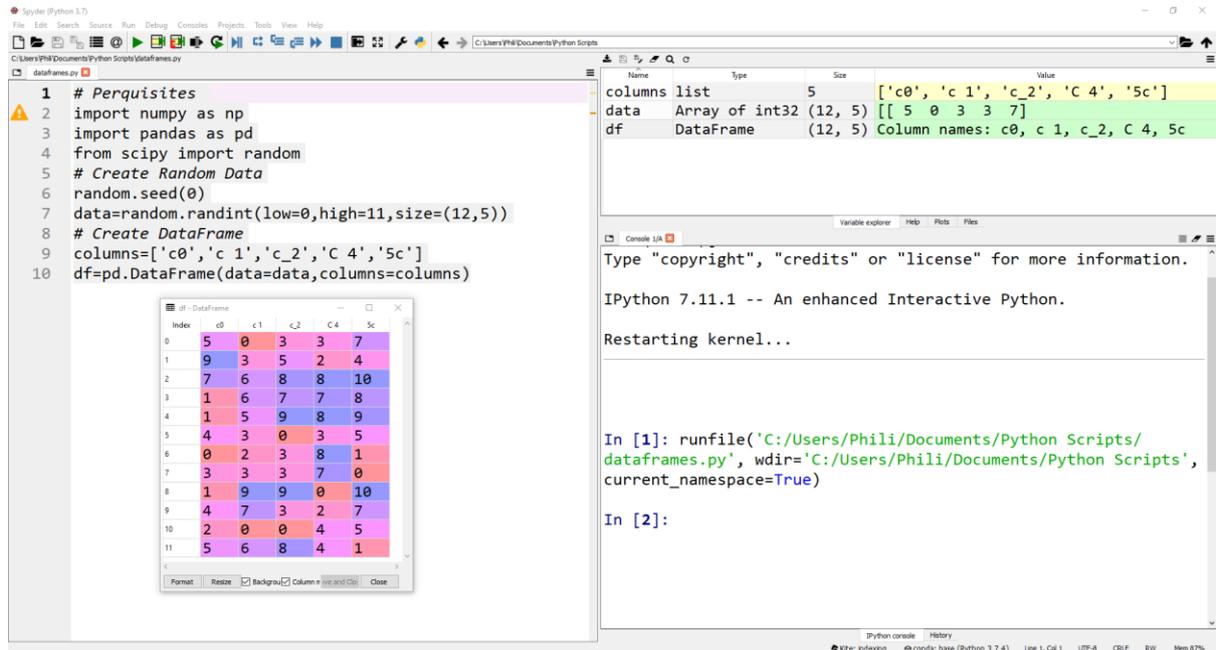
1. # Perquisites
2. import numpy as np
3. import pandas as pd
4. from scipy import random
5. # Create Random Data

```

```

6. random.seed(0)
7. matrix=random.randint(low=0,high=11,size=(12,5))
8. # Create DataFrame
9. cnames=['c0','c 1','c_2','C 4','5c']
10. df=pd.DataFrame(data=matrix,columns=cnames)

```



We can open this dataframe up in the variable explorer. The dataframe created contains five panda series 'c0', 'c 1', 'c_2', 'C 4' and '5 c' which we can also refer to as columns.

The image shows a close-up of the 'df - DataFrame' window in the variable explorer. It displays the same data as the heatmap in the previous image:

Index	c0	c 1	c_2	C 4	5c
0	5	0	3	3	7
1	9	3	5	2	4
2	7	6	8	8	10
3	1	6	7	7	8
4	1	5	9	8	9
5	4	3	0	3	5
6	0	2	3	8	1
7	3	3	3	7	0
8	1	9	9	0	10
9	4	7	3	2	7
10	2	0	0	4	5
11	5	6	8	4	1

Each column is equally sized and has an index of length 12. By default, the index names are numeric integers using zero-order indexing.

Attributes

Dataframes are objects which contain several attributes. Attributes can be thought of as a variable that is looked up from another object using the dot `.` syntax. Attributes can be accessed by typing in the name of the dataframe followed by a `.` and then a `<tab>`.

The screenshot shows the Spyder Python IDE with a script named `dataframes.py` containing the following code:

```
1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c 1','c_2','C 4','5c']
10 df=pd.DataFrame(data=data,columns=columns)
```

The variable explorer on the right shows the following variables:

Name	Type	Size	Value
columns	list	5	['c0', 'c 1', 'c_2', 'C 4', '5c']
data	Array of int32	(12, 5)	[[5 0 3 3 7]
df	DataFrame	(12, 5)	Column names: c0, c 1, c_2, C 4, 5c

The console shows the output of the script, including the IPython version and the DataFrame structure.

For example, we may access the attribute `columns` using:

```
df.columns
```

The screenshot shows the Spyder Python IDE with the same script as above. The console output shows the following:

```
In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/
dataframes.py', wdir='C:/Users/Phili/Documents/Python Scripts',
current_namespace=True)

In [2]: df.columns
Out[2]: Index(['c0', 'c 1', 'c_2', 'C 4', '5c'],
dtype='object')
```

The variable explorer on the right shows the following variables:

Name	Type	Size	Value
columns	list	5	['c0', 'c 1', 'c_2', 'C 4', '5c']
data	Array of int32	(12, 5)	[[5 0 3 3 7]
df	DataFrame	(12, 5)	Column names: c0, c 1, c_2, C 4, 5c

Let's access the attribute `columns` and save the output to a variable name:

```
data2=df.columns
```

Here we see that the new variable `'columns2'` is listed in the variable explorer and the variable explorer shows that the type is `Index`.

The screenshot shows the Spyder Python IDE with a script named `dataframes.py` and its interactive console. The script defines a DataFrame with 12 rows and 5 columns. The console shows the execution of the script and subsequent queries for column names and DataFrame shape.

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c 1','c_2','C 4','5c']
10 df=pd.DataFrame(data=data,columns=columns)

```

The DataFrame's columns are: `c0`, `c 1`, `c_2`, `C 4`, `5c`. The console output shows:

```

IPython 7.11.1 -- An enhanced Interactive Python.
Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/
dataframes.py', wdir='C:/Users/Phili/Documents/Python Scripts',
current_namespace=True)

In [2]: df.columns
Out[2]: Index(['c0', 'c 1', 'c_2', 'C 4', '5c'],
dtype='object')

In [3]: columns2=df.columns

In [4]:

```

We can also lookup the attribute `shape` which gives the dimensions of the dataframe in the form of a tuple with 2 values representing the number of indexes (rows) and the number of series (columns) respectively.

```
df.shape()
```

This screenshot is similar to the first one but includes the execution of `df.shape()` in the console. The output shows the dimensions of the DataFrame as a tuple `(12, 5)`.

```

In [4]: df.shape
Out[4]: (12, 5)

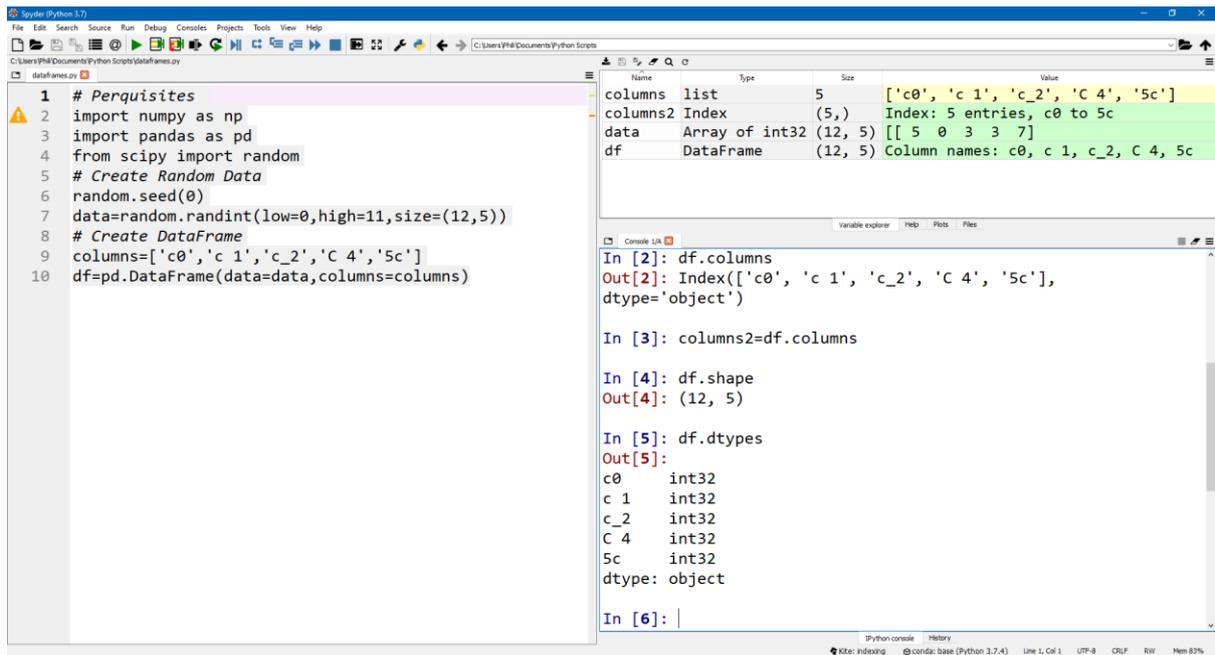
In [5]:

```

In addition, we can use the attribute `dtypes` to get the type of data present in each panda series (column).

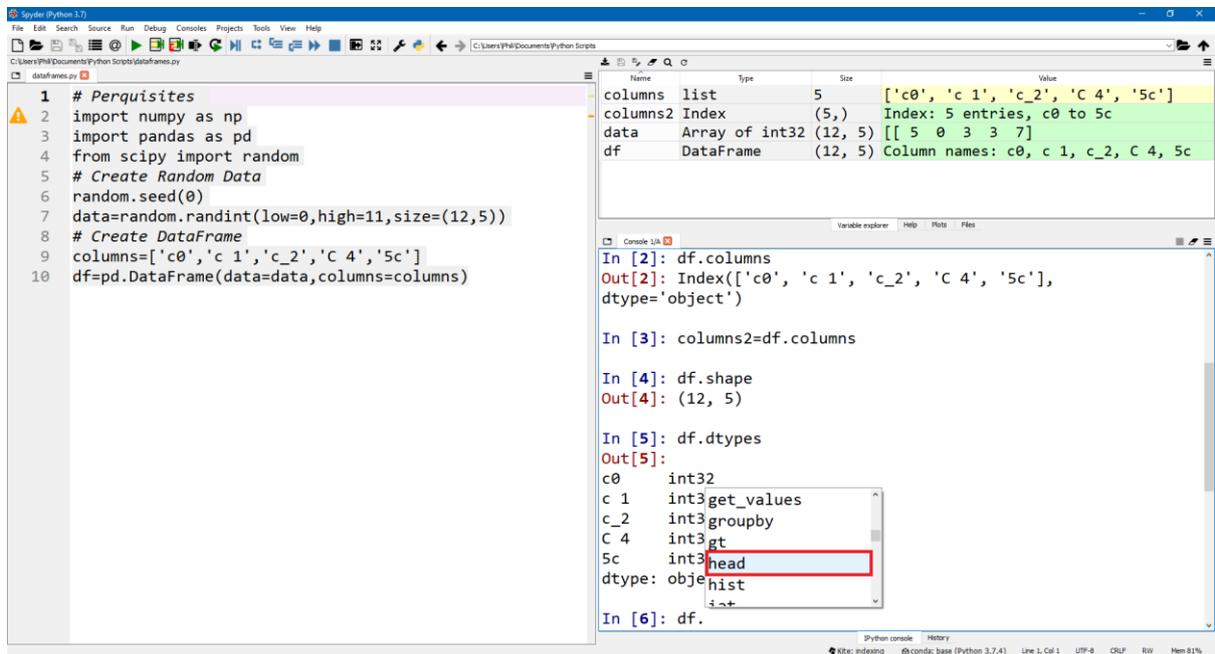
```
pd.dtypes
```

In this case they are all `int32` as expected.



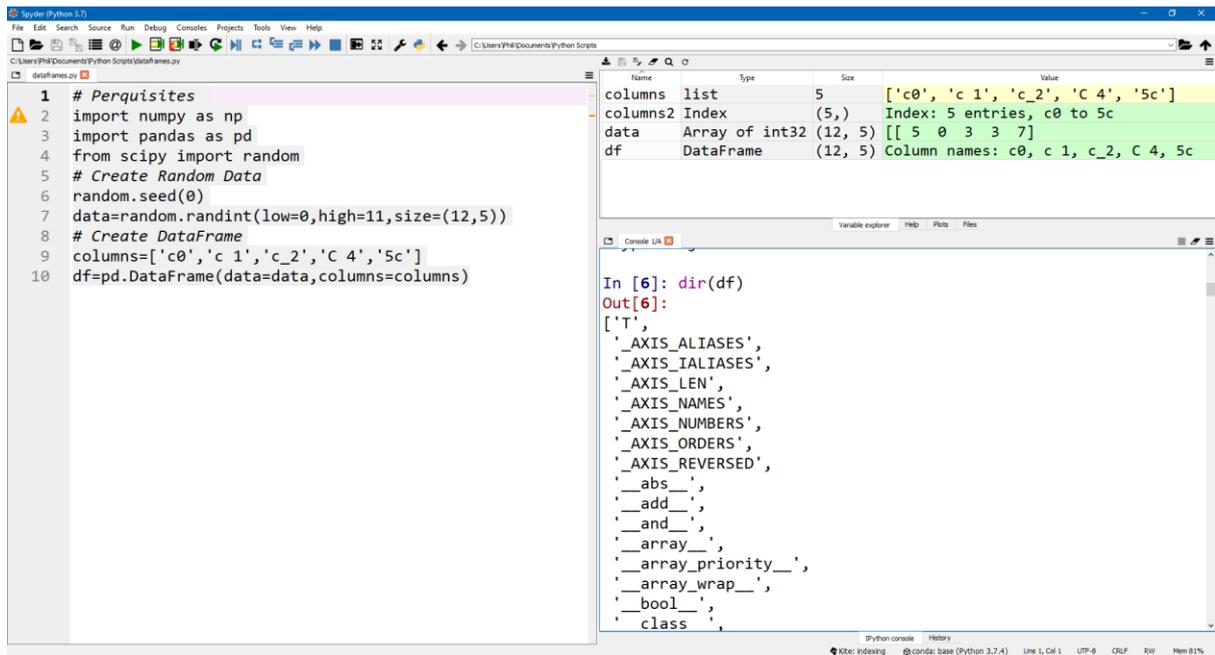
Methods

Notice that the above attributes are called without use of parenthesis. Attributes should not be confused with methods which are also called up using the dot `.` notation but require parenthesis usually to place input arguments. To view the list of methods available for a dataframe once again we can type the dataframe name followed by a dot `.` and then tab `↵`. The list generated however is of both attributes and methods, with no clear indicator between them.

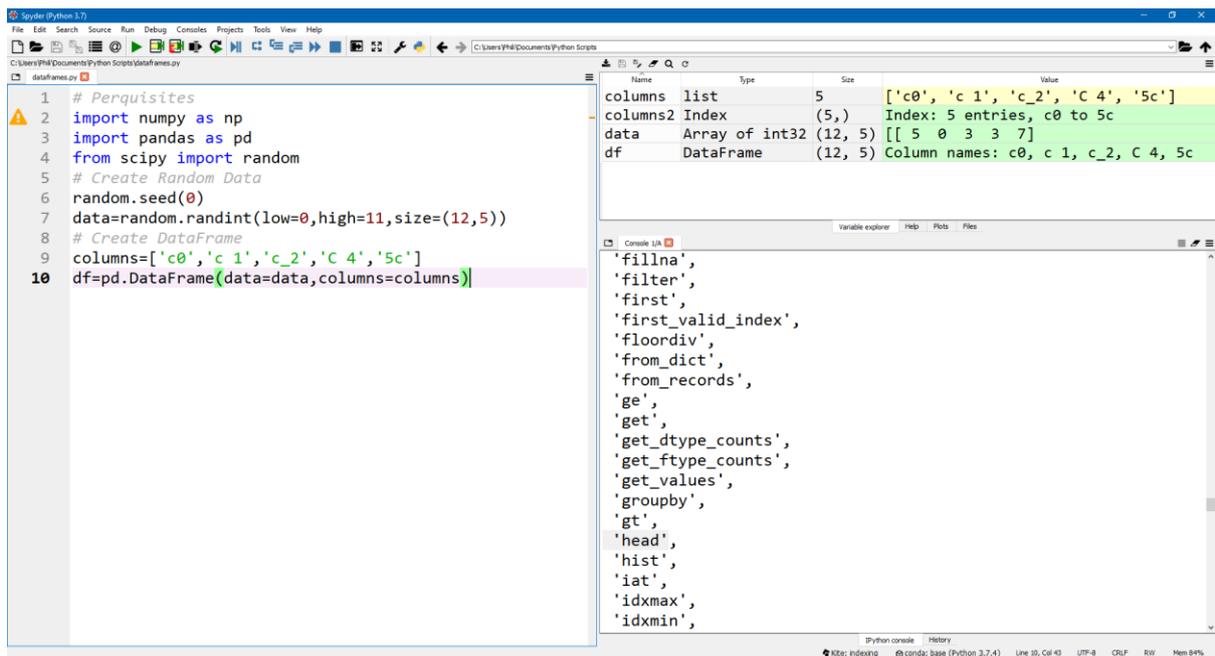


This list can be printed to the console by using the function

```
dir(df)
```

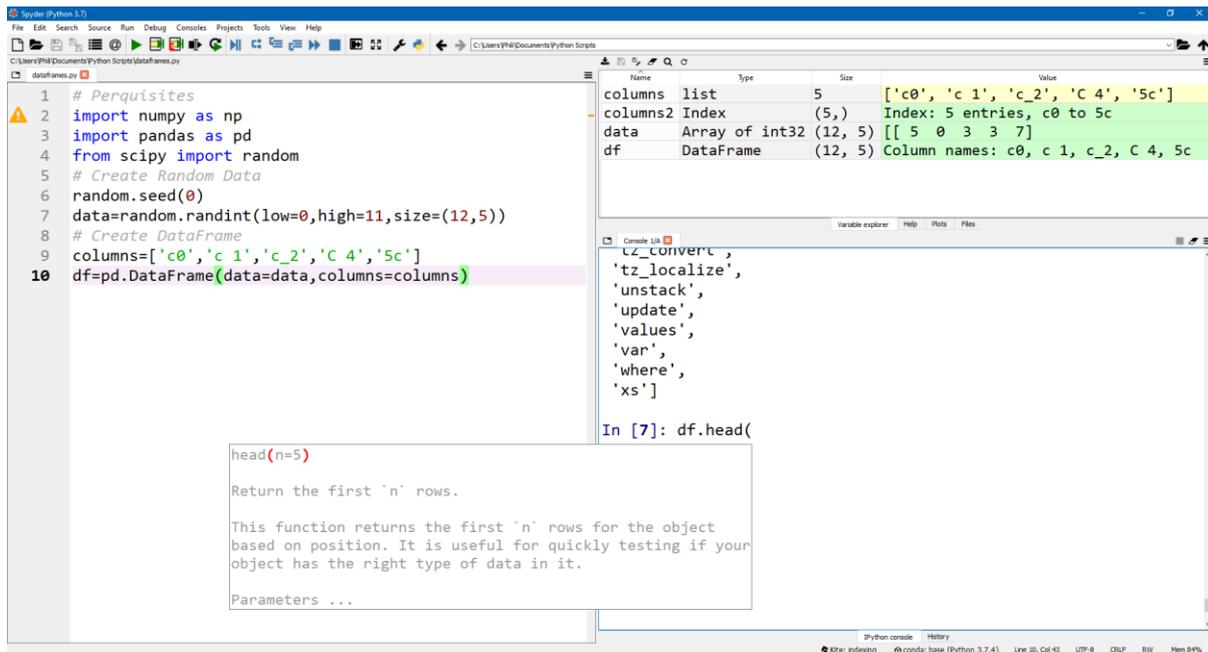


The items beginning with the double underscore can be ignored for just now. Again, looking through the list is not obviously clear which is a function and which is a method.

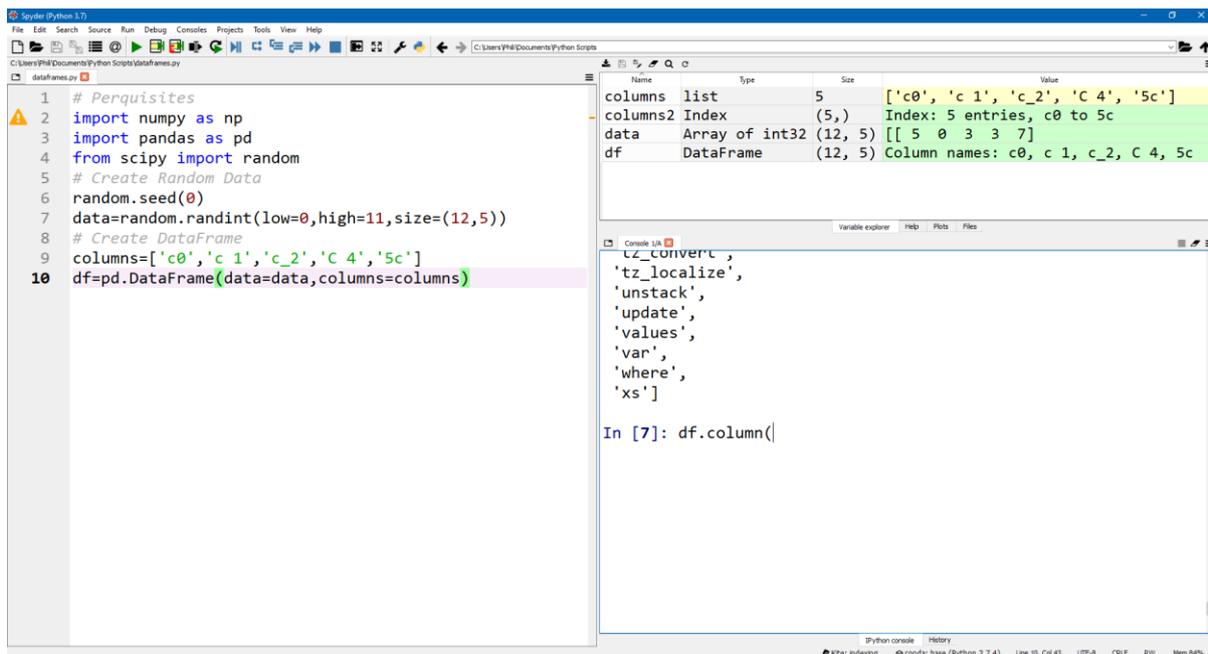


Recall if you are unsure, you can type in a dataframe method with an open parenthesis and this will give you details about the positional and keyword input arguments. For example, if we type in

```
df.head(
```



In contrast if we attempt this with an attribute:



Nothing will show as the attribute doesn't have input arguments.

Let's have a look at the dataframe method `head`, as we can see this method has a default keyword argument of 5. This method will create a new dataframe that previews the first 5 indexes (or `n` indexes if the keyword input argument is modified).

```
df.head()
```

The screenshot shows the Spyder Python IDE with a script named `dataframes.py`. The script creates a DataFrame with 12 rows and 5 columns. The console output shows the DataFrame's structure and the first five rows of data.

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c 1','c_2','C 4','5c']
10 df=pd.DataFrame(data=data,columns=columns)

```

Index	c0	c 1	c_2	C 4	5c
0	5	0	3	3	7
1	9	3	5	2	4
2	7	6	8	8	10
3	1	6	7	7	8
4	1	5	9	8	9
5	4	3	0	3	5
6	0	2	3	8	1
7	3	3	3	7	0
8	1	9	9	0	10
9	4	7	3	2	7
10	2	0	0	4	5
11	5	6	8	4	1

Alternatively, the method `tail` can be used to look at the last 5 indexes (or `n` indexes if the keyword argument is modified).

```
df.tail()
```

The screenshot shows the same Spyder Python IDE environment. The script is identical to the previous one. The console output now shows the last five rows of the DataFrame.

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c 1','c_2','C 4','5c']
10 df=pd.DataFrame(data=data,columns=columns)

```

Index	c0	c 1	c_2	C 4	5c
7	3	3	3	7	0
8	1	9	9	0	10
9	4	7	3	2	7
10	2	0	0	4	5
11	5	6	8	4	1

These could be assigned to new variable names and displayed within the variable explorer if desired by using the assignment operator to a variable name on the left-hand side. Basic statistics about each dataframe series (columns) can be obtained by use of the method `describe`. Once again if we type in this method with open parenthesis, we can see it has keyword input arguments.

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c 1','c_2','C 4','5c']
10 df=pd.DataFrame(data=data,columns=columns)

```

Variable Explorer:

Name	Type	Size	Value
columns	list	5	['c0', 'c 1', 'c_2', 'C 4', '5c']
columns2	Index	(5,)	Index: 5 entries, c0 to 5c
data	Array of int32	(12, 5)	[[5 0 3 3 7]
df	DataFrame	(12, 5)	Column names: c0, c 1, c_2, C 4, 5c

Console I/O:

```

In [7]: df.head()
Out[7]:
   c0  c 1  c_2  C 4  5c
0  5  0  3  3  7
1  9  3  5  2  4
2  7  6  8  8  10
3  1  6  7  7  8
4  1  5  9  8  9

In [8]: df.tail()
Out[8]:
   c0  c 1  c_2  C 4  5c
7  3  3  3  7  0
8  1  9  9  0  10
9  4  7  3  2  7
10 2  0  0  4  5
11 5  6  8  4  1

In [9]: df.describe()
Out[9]:
           c0         c 1         c_2         C 4         5c
count  12.000000  12.000000  12.000000  12.000000  12.000000
mean     3.500000   4.166667   4.833333   4.666667   5.583333
std     2.713602   2.790677   3.298301   2.806918   3.528026
min      0.000000   0.000000   0.000000   0.000000   0.000000
25%     1.000000   2.750000   3.000000   2.750000   3.250000
50%     3.500000   4.000000   4.000000   4.000000   6.000000
75%     5.000000   6.000000   8.000000   7.250000   8.250000
max      9.000000   9.000000   9.000000   8.000000  10.000000

```

We can call the method `describe` without modification of any keyword input arguments:

```
df.describe()
```

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c 1','c_2','C 4','5c']
10 df=pd.DataFrame(data=data,columns=columns)

```

Variable Explorer:

Name	Type	Size	Value
columns	list	5	['c0', 'c 1', 'c_2', 'C 4', '5c']
columns2	Index	(5,)	Index: 5 entries, c0 to 5c
data	Array of int32	(12, 5)	[[5 0 3 3 7]
df	DataFrame	(12, 5)	Column names: c0, c 1, c_2, C 4, 5c

Console I/O:

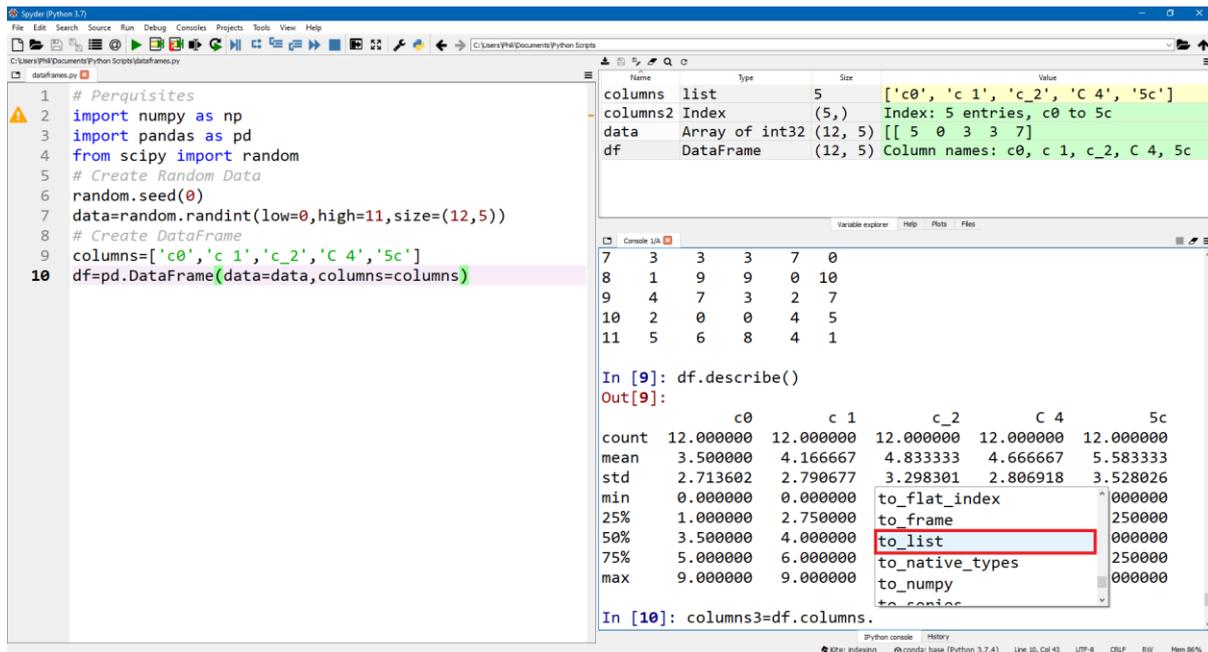
```

In [9]: df.describe()
Out[9]:
           c0         c 1         c_2         C 4         5c
count  12.000000  12.000000  12.000000  12.000000  12.000000
mean     3.500000   4.166667   4.833333   4.666667   5.583333
std     2.713602   2.790677   3.298301   2.806918   3.528026
min      0.000000   0.000000   0.000000   0.000000   0.000000
25%     1.000000   2.750000   3.000000   2.750000   3.250000
50%     3.500000   4.000000   4.000000   4.000000   6.000000
75%     5.000000   6.000000   8.000000   7.250000   8.250000
max      9.000000   9.000000   9.000000   8.000000  10.000000

In [10]:

```

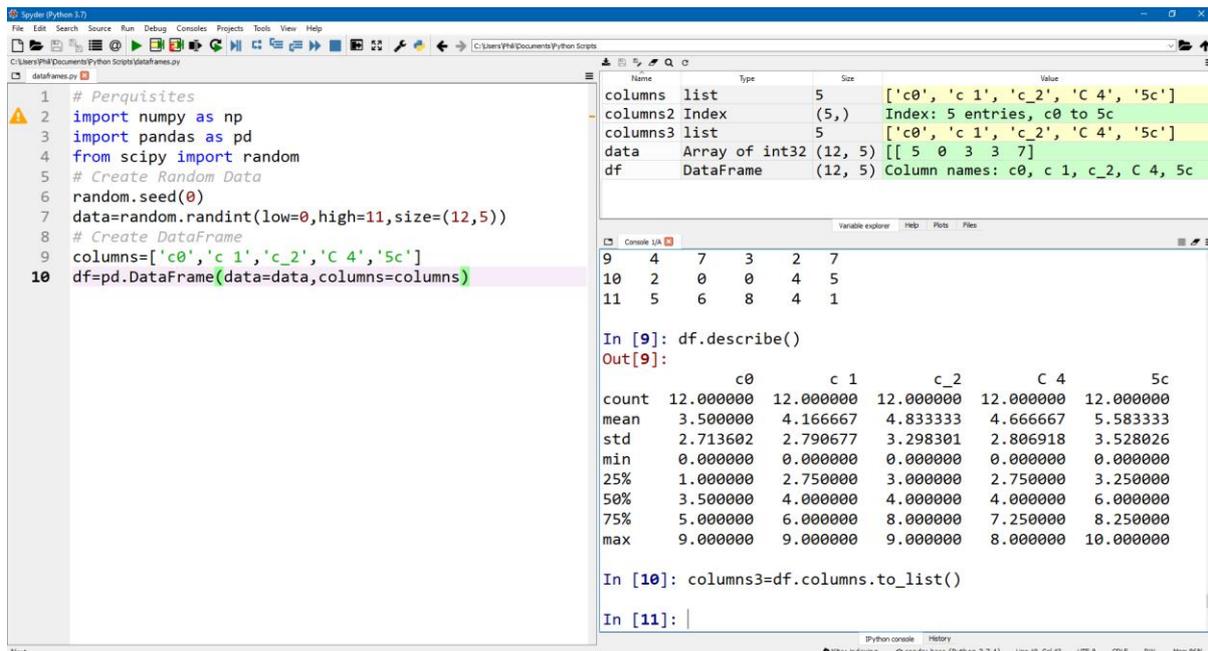
Attributes as mentioned earlier can be thought of as a variable that is looked up from another object using the dot `.` syntax. Attributes, themselves can have nested attributes and methods. Once again we can access these using a dot `.` and a tab `⌘`.



In this case we may use the method `to_list` to convert the column of type index to a list.

```
columns3=df.columns.to_list()
```

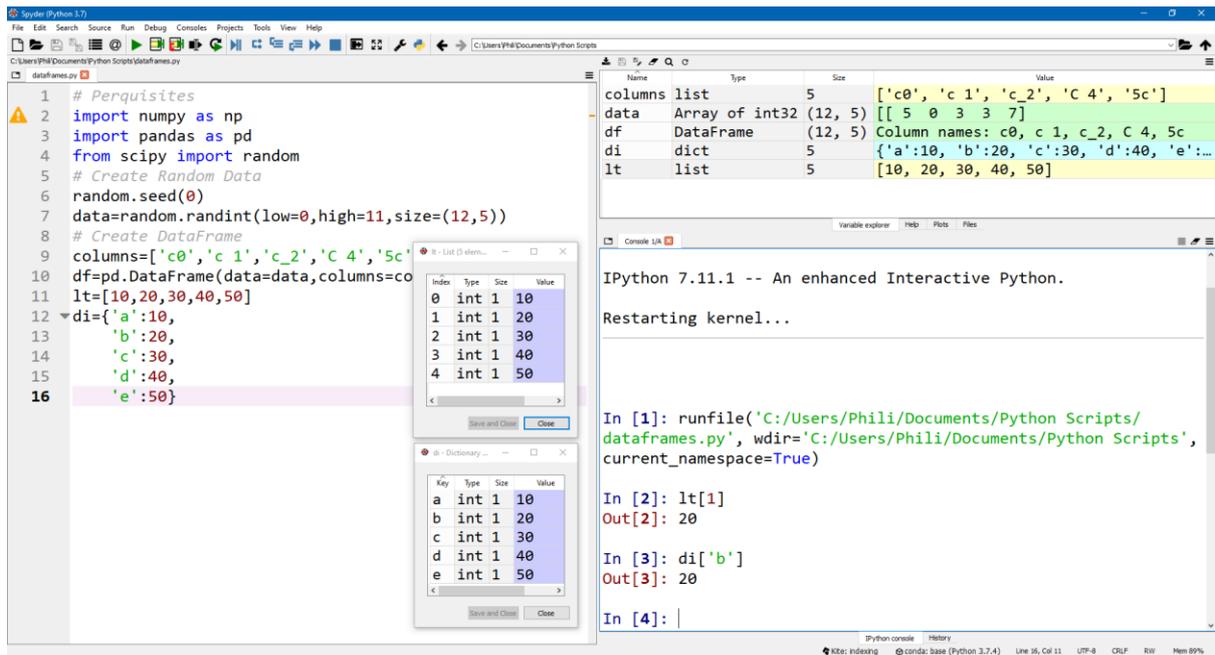
Compare columns2 to columns 3 in the variable editor.



Indexing a Pandas Series (Column) – Square Bracket Indexing vs Dot Attribute Lookup

Recall that when we index into a list, we type in the list name followed by the numerical index we want to index which is enclosed in square brackets. Likewise, for a dictionary we use the dictionary name followed by the key as a string which is also enclosed in square brackets. For example:

```
lt[1]
dt['b']
```



We can also follow this convention in a pandas dataframe by use of the pandas series (column) name.

Index	c0	c1	c2	c4	5c
0	5	0	3	3	7
1	9	3	5	2	4
2	7	6	8	8	10
3	1	6	7	7	8
4	1	5	9	8	9
5	4	3	0	3	5
6	0	2	3	8	1
7	3	3	3	7	0
8	1	9	9	0	10
9	4	7	3	2	7
10	2	0	0	4	5
11	5	6	8	4	1

Like a key in the dictionary these names are input as strings and are case sensitive. Let's have a look at the pandas series 'c0':

```
df['c0']
```

The screenshot shows the Spyder Python IDE with a script named `dataframes.py`. The script creates a DataFrame with 12 rows and 5 columns. The code is as follows:

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c 1','c_2','C 4','5c']
10 df=pd.DataFrame(data=data,columns=columns)
11 lt=[10,20,30,40,50]
12 di={'a':10,
13     'b':20,
14     'c':30,
15     'd':40,
16     'e':50}

```

The Variable explorer on the right shows the following variables:

Name	Type	Size	Value
columns	list	5	['c0', 'c 1', 'c_2', 'C 4', '5c']
data	Array of int32	(12, 5)	[[5 0 3 3 7]
df	DataFrame	(12, 5)	Column names: c0, c 1, c_2, C 4, 5c
di	dict	5	{'a':10, 'b':20, 'c':30, 'd':40, 'e':...
lt	list	5	[10, 20, 30, 40, 50]

The console shows the following output:

```

Out[3]: 20
In [4]: df['c0']
Out[4]:
0    5
1    9
2    7
3    1
4    1
5    4
6    0
7    3
8    1
9    4
10   2
11   5
Name: c0, dtype: int32
In [5]:

```

When we assign this to a variable name, we see that it shows up as a pandas series within the variable explorer.

```
col0=df['c0']
```

Recall we could use the method `to_list` to convert it to a list.

The screenshot shows the Spyder Python IDE with the same script as above, but with an additional line of code:

```

14 col0=df['c0'].to_list()

```

The Variable explorer now shows an additional variable:

Name	Type	Size	Value
col0	Series	(12,)	Series object of pandas.core.series m...
columns	list	5	['c0', 'c 1', 'c_2', 'C 4', '5c']
data	Array of int32	(12, 5)	[[5 0 3 3 7]
df	DataFrame	(12, 5)	Column names: c0, c 1, c_2, C 4, 5c
di	dict	5	{'a':10, 'b':20, 'c':30, 'd':40, 'e':...
lt	list	5	[10, 20, 30, 40, 50]

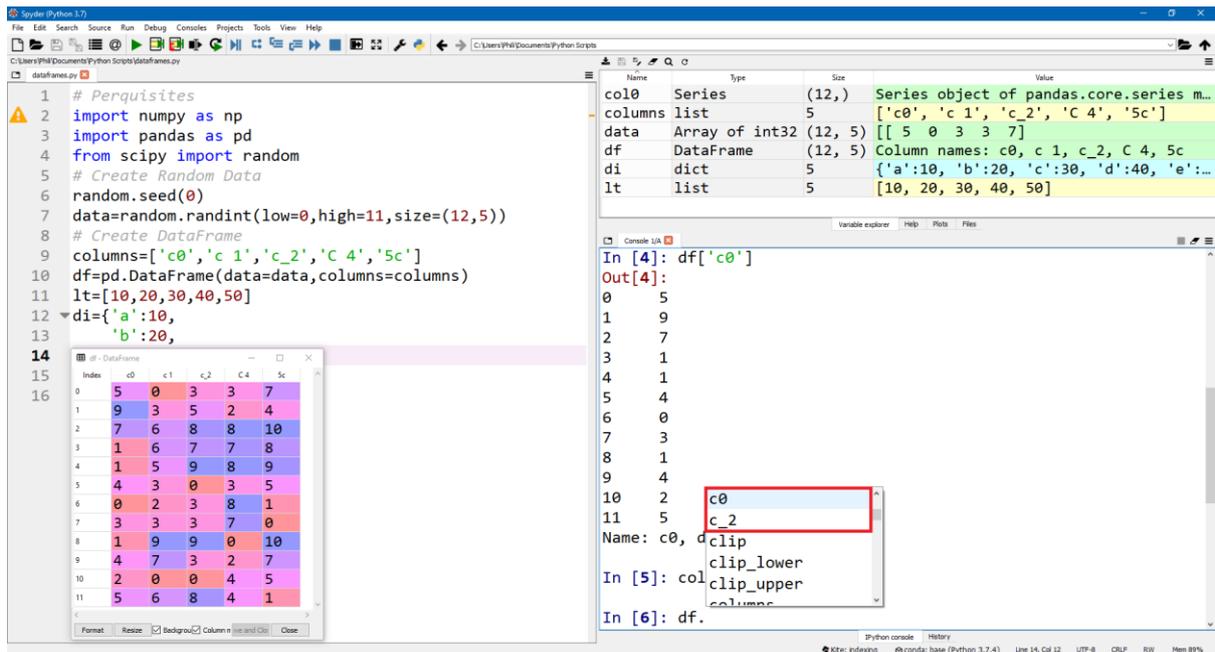
The console shows the following output:

```

In [4]: df['c0']
Out[4]:
0    5
1    9
2    7
3    1
4    1
5    4
6    0
7    3
8    1
9    4
10   2
11   5
Name: c0, dtype: int32
In [5]: col0=df['c0']
In [6]:

```

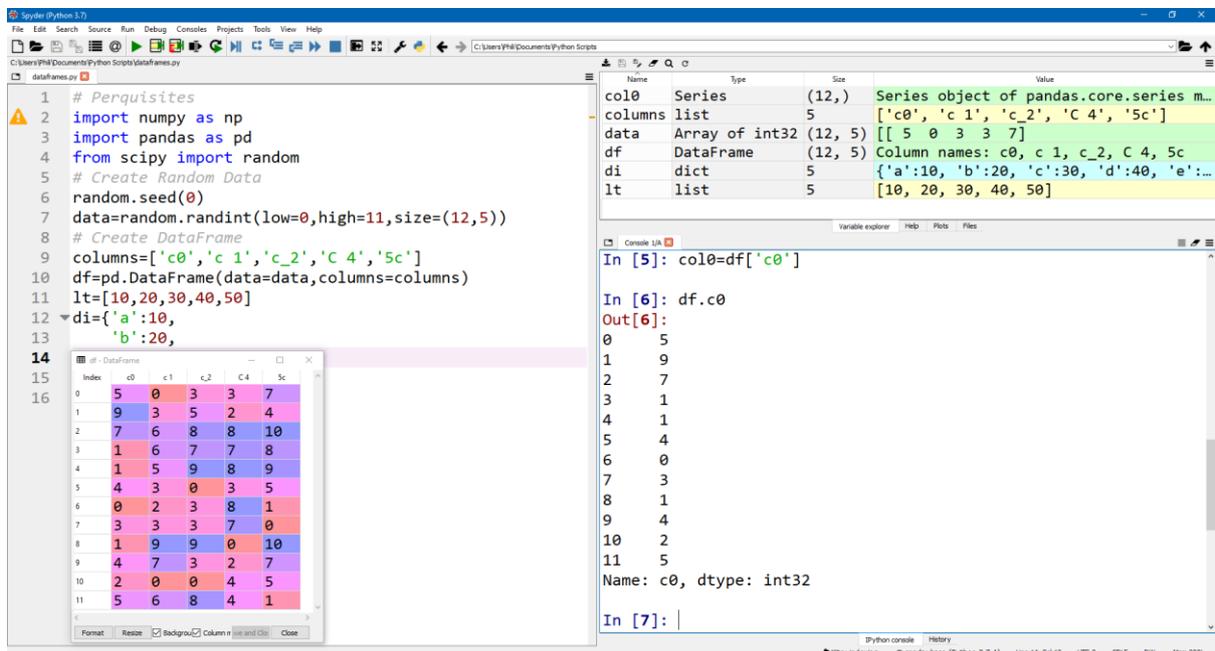
If the series (column) name follows the same rules as variable names, it is automatically added as an attribute to the dataframe. Recall that the attributes can be viewed by typing in the dataframe name followed by a dot `.` and then a tab `↵`.



In this case only the series (columns) 'c0' and 'c_2' follow the rules. These can be selected as an attribute of the dataframe:

```
df.c0
df.c_2
```

Note no quotations are used when using attribute dot notation.



Attributes can also be indexed. For example, we can select the 5th index of the pandas series (column) by use of square brackets:

```
df.c0[5]
```

The screenshot shows the Spyder Python IDE with a script named 'dataframes.py'. The script creates a DataFrame with 12 rows and 5 columns. The variable explorer on the right shows the DataFrame's structure, and the console shows the output of a command to access a specific element in the DataFrame.

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c 1','c_2','C 4','5c']
10 df=pd.DataFrame(data=data,columns=columns)
11 lt=[10,20,30,40,50]
12 di={'a':10,
13     'b':20,
14
15
16

```

Index	c0	c 1	c_2	C 4	5c
0	5	0	3	3	7
1	9	3	5	2	4
2	7	6	8	8	10
3	1	6	7	7	8
4	1	5	9	8	9
5	4	3	0	3	5
6	0	2	3	8	1
7	3	3	3	7	0
8	1	9	9	0	10
9	4	7	3	2	7
10	2	0	0	4	5
11	5	6	8	4	1

```

Out[6]:
0 5
1 9
2 7
3 1
4 1
5 4
6 0
7 3
8 1
9 4
10 2
11 5
Name: c0, dtype: int32

In [7]: df.c0[5]
Out[7]: 4

In [8]:

```

Other attributes can be used on them for example:

```
df.c0.dtypes
```

This screenshot is similar to the first one but shows the execution of the `df.c0.dtypes` command in the console, which returns the data type of the 'c0' column.

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c 1','c_2','C 4','5c']
10 df=pd.DataFrame(data=data,columns=columns)
11 lt=[10,20,30,40,50]
12 di={'a':10,
13     'b':20,
14
15
16

```

Index	c0	c 1	c_2	C 4	5c
0	5	0	3	3	7
1	9	3	5	2	4
2	7	6	8	8	10
3	1	6	7	7	8
4	1	5	9	8	9
5	4	3	0	3	5
6	0	2	3	8	1
7	3	3	3	7	0
8	1	9	9	0	10
9	4	7	3	2	7
10	2	0	0	4	5
11	5	6	8	4	1

```

Out[6]:
0 5
1 9
2 7
3 1
4 1
5 4
6 0
7 3
8 1
9 4
10 2
11 5
Name: c0, dtype: int32

In [7]: df.c0[5]
Out[7]: 4

In [8]: df.c0.dtypes
Out[8]: dtype('int32')

In [9]:

```

Or methods:

```
df.c0.max()
```

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c 1','c_2','C 4','5c']
10 df=pd.DataFrame(data=data,columns=columns)
11 lt=[10,20,30,40,50]
12 di={'a':10,
13     'b':20,
14     }
15
16

```

Name	Type	Size	Value
col0	Series	(12,)	Series object of pandas.core.series m...
columns	list	5	['c0', 'c 1', 'c_2', 'C 4', '5c']
data	Array of int32	(12, 5)	[[5 0 3 3 7]
df	DataFrame	(12, 5)	Column names: c0, c 1, c_2, C 4, 5c
di	dict	5	{'a':10, 'b':20, 'c':30, 'd':40, 'e':...
lt	list	5	[10, 20, 30, 40, 50]

```

5 4
6 0
7 3
8 1
9 4
10 2
11 5
Name: c0, dtype: int32

In [7]: df.c0[5]
Out[7]: 4

In [8]: df.c0.dtypes
Out[8]: dtype('int32')

In [9]: df.c0.max()
Out[9]: 9

In [10]:

```

Series named without following the rules behind variable names will not have an attribute and so cannot be indexed in this way but can be indexed with square brackets. For example `'c 1'` which has a space.

```
df['c 1']
```

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c 1','c_2','C 4','5c']
10 df=pd.DataFrame(data=data,columns=columns)
11 lt=[10,20,30,40,50]
12 di={'a':10,
13     'b':20,
14     }
15
16

```

Name	Type	Size	Value
col0	Series	(12,)	Series object of pandas.core.series m...
columns	list	5	['c0', 'c 1', 'c_2', 'C 4', '5c']
data	Array of int32	(12, 5)	[[5 0 3 3 7]
df	DataFrame	(12, 5)	Column names: c0, c 1, c_2, C 4, 5c
di	dict	5	{'a':10, 'b':20, 'c':30, 'd':40, 'e':...
lt	list	5	[10, 20, 30, 40, 50]

```

Out[9]: 9

In [10]: df['c 1']
Out[10]:
0 0
1 3
2 6
3 6
4 5
5 3
6 2
7 3
8 9
9 7
10 0
11 6
Name: c 1, dtype: int32

In [11]:

```

Creating a New Series (Column)

One other nuance to keep in mind when comparing series selection using square brackets and attribute dot notation is that attribute names can only be used on attributes that exist. They cannot be used to create a new attribute. We can see this if we try and generate a new series (column) in the dataframe:

```
df['c6']=random.randint(low=0,high=11,size=12)
```

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c 1','c_2','C 4','5c']
10 df=pd.DataFrame(data=data,columns=columns)
11

```

Index	c0	c 1	c_2	C 4	5c
0	5	0	3	3	7
1	9	3	5	2	4
2	7	6	8	8	10
3	1	6	7	7	8
4	1	5	9	8	9
5	4	3	0	3	5
6	0	2	3	8	1
7	3	3	3	7	0
8	1	9	9	0	10
9	4	7	3	2	7
10	2	0	0	4	5
11	5	6	8	4	1

If we try:

```
df.c7=random.randint(low=0,high=11,size=12)
```

We get the pandas user error `UserWarning: Pandas doesn't allow columns to be created via a new attribute name.`

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c 1','c_2','C 4','5c']
10 df=pd.DataFrame(data=data,columns=columns)
11
12 df.c7=random.randint(low=0,high=11,size=12)

```

Index	c0	c 1	c_2	C 4	5c
0	5	0	3	3	7
1	9	3	5	2	4
2	7	6	8	8	10
3	1	6	7	7	8
4	1	5	9	8	9
5	4	3	0	3	5
6	0	2	3	8	1
7	3	3	3	7	0
8	1	9	9	0	10
9	4	7	3	2	7
10	2	0	0	4	5
11	5	6	8	4	1

Renaming Series (Column)

We can rename individual columns by use the rename method. Its keyword input argument expects the form of a dictionary where the original name is specified as the key and the new value is specified as the value.

```
df.rename(columns = {'oldcol0': 'newcol0', 'oldcol1': 'newcol1'})
```

Supposing we wish to rename `'c 1'` to `'c1'`, `'c_2'` to `'c2'` and `'C 4'` to `'c4'` we can use:

```
df.rename(columns={'c 1':'c1','c_2':'c2','C 4':'c4'})
```

The screenshot shows a Python script in a Jupyter Notebook environment. The script imports numpy and pandas, and uses scipy's random module to generate random data. It creates a DataFrame with 12 rows and 5 columns. The columns are named 'c0', 'c 1', 'c_2', 'C 4', and '5c'. The DataFrame is displayed in a table view. The console output shows the DataFrame structure and the result of the rename operation, which creates a new DataFrame with 6 columns.

Notice however that a new dataframe is printed to the console and the original dataframe remains unchanged. This can be addressed by manually reassigning the dataframe created to the original dataframe name:

```
df=df.rename(columns={'c 1':'c1','c_2':'c2','C 4':'c4'})
```

However this method has a keyword argument `inplace` which has a default value of `False`.

The screenshot shows the same Python script as above, but with the `inplace` parameter added to the `rename` method. A tooltip for the `inplace` parameter is visible, explaining its function and default value.

We can simply reassign it to `True` to perform an inplace update of our original dataframe.

```
df.rename(columns={'c 1':'c1','c_2':'c2','C 4':'c4'},inplace=True)
```

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c 1','c_2','C 4','5c']
10 df=pd.DataFrame(data=data,columns=columns)
11 df['c6']=random.randint(low=0,high=11,size=12)

```

Index	c0	c1	c2	c4	5c	
0	5	0	3	3	7	4
1	9	3	5	2	4	9
2	7	6	8	8	10	10
3	1	6	7	7	8	10
4	1	5	9	8	9	8
5	4	3	0	3	5	1
6	0	2	3	8	1	1
7	3	3	3	7	0	7
8	1	9	9	0	10	9
9	4	7	3	2	7	9
10	2	0	0	4	5	3
11	5	6	8	4	1	6

Note if this keyword input argument `inplace=True`, there is no output argument for this method and a `NoneType` object is returned, we have seen this earlier when we looked at custom functions.

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c 1','c_2','C 4','5c']
10 df=pd.DataFrame(data=data,columns=columns)
11 df['c6']=random.randint(low=0,high=11,size=12)

```

Index	c0	c1	c2	c4	5c	c6
0	5	0	3	3	7	4
1	9	3	5	2	4	9
2	7	6	8	8	10	10
3	1	6	7	7	8	10
4	1	5	9	8	9	8
5	4	3	0	3	5	1
6	0	2	3	8	1	1
7	3	3	3	7	0	7
8	1	9	9	0	10	9
9	4	7	3	2	7	9
10	2	0	0	4	5	3
11	5	6	8	4	1	6

Once these columns have been renamed. They will show up as attributes because they now follow the rules behind variable names.

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c 1','c_2','C 4','5c']
10 df=pd.DataFrame(data=data,columns=columns)
11 df['c6']=random.randint(low=0,high=11,size=12)

```

Index	c0	c1	c2	c4	5c	c6
0	5	0	3	3	7	4
1	9	3	5	2	4	9
2	7	6	8	8	10	10
3	1	6	7	7	8	10
4	1	5	9	8	9	8
5	4	3	0	3	5	1
6	0	2	3	8	1	1
7	3	3	3	7	0	7
8	1	9	9	0	10	9
9	4	7	3	2	7	9
10	2	0	0	4	5	3
11	5	6	8	4	1	6

Supposing series (column) '5c' was actually meant to be 'c3' we can reorder the series names by creating a list of the new order of the series names and indexing to the dataframe with it. We can assign the output to a new dataframe or reassign it to the original dataframe:

```

neworder=['c0','c1','c2','5c','c4','c6']
df2=df[neworder]

```

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c 1','c_2','C 4','5c']
10 df=pd.DataFrame(data=data,columns=columns)
11 df['c6']=random.randint(low=0,high=11,size=12)

```

Index	c0	c1	c2	5c	c4	c6
0	5	0	3	7	3	4
1	9	3	5	4	2	9
2	7	6	8	10	8	10
3	1	6	7	8	7	10
4	1	5	9	9	8	8
5	4	3	0	5	3	1
6	0	2	3	1	8	1
7	3	3	3	0	7	7
8	1	9	9	10	0	9
9	4	7	3	7	2	9
10	2	0	0	5	4	3
11	5	6	8	1	4	6

Now supposing we want to rename all the pandas series (columns). We can use the dictionary method as earlier. However, for convenience we may want to use a for loop to create a list of string names. We can then reassign the attribute `df.columns` to this new list.

```

1. # Perquisites
2. import numpy as np
3. import pandas as pd

```

```

4. from scipy import random
5. # Create Random Data
6. random.seed(0)
7. data=random.randint(low=0,high=11,size=(12,5))
8. # Create DataFrame
9. columns=['c0','c 1','c_2','C 4','5c']
10. df=pd.DataFrame(data=data,columns=columns)
11. df['c6']=random.randint(low=0,high=11,size=12)
12. df.rename(columns={'c 1':'c1','c_2':'c2','C 4':'c4'},
13.           inplace=True)
14. neworder=['c0','c1','c2','5c','c4','c6']
15. df=df[neworder]
16. # Create column and index names
17. (nrows,ncols)=np.shape(df)
18. col=np.arange(ncols)
19. col=['c' + i for i in col.astype(np.str)]
20. df.columns=col

```

The screenshot shows the Spyder Python IDE interface. The left pane contains the Python code from the previous block. The right pane is split into three sections: Variable explorer, Console, and IPython console.

Variable explorer: Shows variables defined in the code:

Name	Type	Size	Value
col	list	6	['c0', 'c1', 'c2', 'c3', 'c4', 'c5']
columns	list	5	['c0', 'c 1', 'c_2', 'C 4', '5c']
data	Array of int32	(12, 5)	[[5 0 3 3 7]
df	DataFrame	(12, 6)	Column names: c0, c1, c2, c3, c4, c5
ncols	int	1	6
neworder	list	6	['c0', 'c1', 'c2', '5c', 'c4', 'c6']
nrows	int	1	12

Console: Shows the execution of the code and the output of the IPython console.

```

Python 3.7.4 (default, Aug 9 2019, 18:34:13) [MSC v.1915 64
bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.11.1 -- An enhanced Interactive Python
Restoring kernel...

In [1]: runfile('C:/Users/Phili/Do
dataframes.py', wdir='C:/Users/Phi
current_namespace=True)

In [2]:

```

IPython console: Shows a DataFrame with 12 rows and 6 columns (c0 to c5). The values are integers ranging from 0 to 10.

Index	c0	c1	c2	c3	c4	c5
0	5	0	3	7	3	4
1	9	3	5	4	2	9
2	7	6	8	10	8	10
3	1	6	7	8	7	10
4	1	5	9	9	8	8
5	4	3	0	5	3	1
6	0	2	3	1	8	1
7	3	3	3	0	7	7
8	1	9	9	10	0	9
9	4	7	3	7	2	9
10	2	0	0	5	4	3
11	5	6	8	1	4	6

The for loop here carries out several steps. It uses the method `astype` to convert from integer values to strings. It takes advantage of the fact that string concatenation combines strings and is also enclosed in square brackets to make a list. We can see what is going on in more detail below.

The screenshot shows the Spyder Python IDE with a script named 'dataframes.py' and its execution output in the console. The script creates a DataFrame with 12 rows and 5 columns, renames some columns, and generates a list of column names with spaces.

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c 1','c_2','C 4','5c']
10 df=pd.DataFrame(data=data,columns=columns)
11 df['c6']=random.randint(low=0,high=11,size=12)
12 df.rename(columns={'c 1':'c1','c_2':'c2','C 4':'c4'},
13           inplace=True)
14 neworder=['c0','c1','c2','5c','c4','c6']
15 df=df[neworder]
16 # Create column and index names
17 (nrows,ncols)=np.shape(df)
18 col=np.arange(ncols)
19 col=['c ' + i for i in col.astype(np.str)]
20 df.columns=col

```

The console output shows the following steps:

```

In [2]: (nrows,ncols)=np.shape(df)
In [3]: print(ncols)
6
In [4]: col=np.arange(ncols)
In [5]: print(col)
[0 1 2 3 4 5]
In [6]: col=col.astype(np.str)
In [7]: print(col)
['0' '1' '2' '3' '4' '5']
In [8]:
col=['c'+col[0],'c'+col[1],'c'+col[2],'c'+col[3],'c'+col[4],'c'+col[5]]
In [9]: print(col)
['c0', 'c1', 'c2', 'c3', 'c4', 'c5']

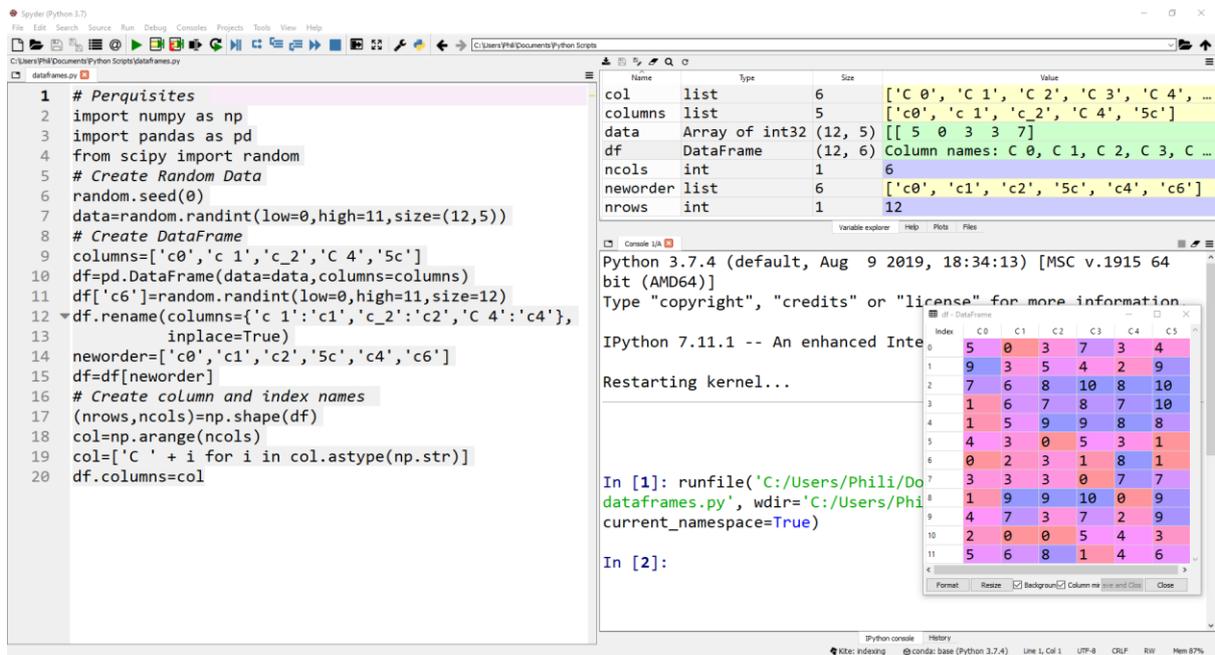
```

In many cases we will come across data which contains a space or upper-case letter in its column name. Let's amend the code above to emulate this.

```

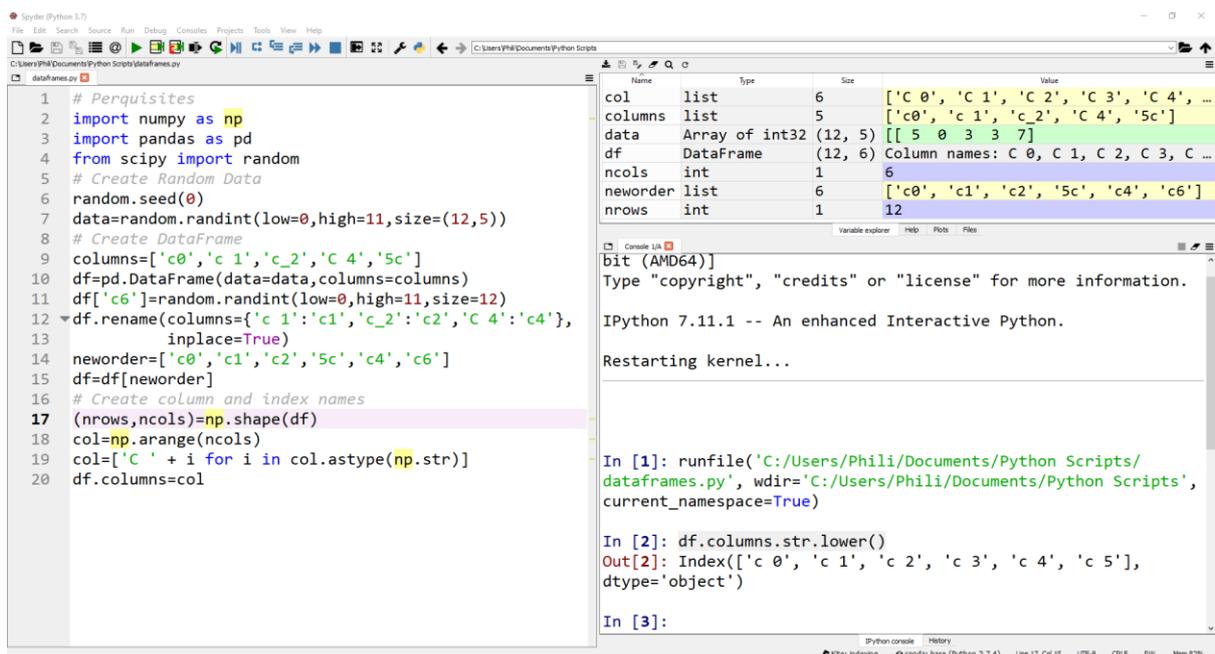
1. # Perquisites
2. import numpy as np
3. import pandas as pd
4. from scipy import random
5. # Create Random Data
6. random.seed(0)
7. data=random.randint(low=0,high=11,size=(12,5))
8. # Create DataFrame
9. columns=['c0','c 1','c_2','C 4','5c']
10. df=pd.DataFrame(data=data,columns=columns)
11. df['c6']=random.randint(low=0,high=11,size=12)
12. df.rename(columns={'c 1':'c1','c_2':'c2','C 4':'c4'},
13.           inplace=True)
14. neworder=['c0','c1','c2','5c','c4','c6']
15. df=df[neworder]
16. # Create column and index names
17. (nrows,ncols)=np.shape(df)
18. col=np.arange(ncols)
19. col=['C ' + i for i in col.astype(np.str)]
20. df.columns=col

```



We can correct this by first typing in the DataFrame name, accessing the attribute `columns`, then accessing the nested attribute `str` and then using the method `lower`.

```
df.columns.str.lower()
```



However, we will need to reassign the value of `df.columns` to this new value:

```
df.columns=df.columns.str.lower()
```

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c 1','c_2','C 4','5c']
10 df=pd.DataFrame(data=data,columns=columns)
11 df['c6']=random.randint(low=0,high=11,size=12)
12 df.rename(columns={'c 1':'c1','c_2':'c2','C 4':'c4'},
13           inplace=True)
14 neworder=['c
15 df=df[neworder]
16 # Create col
17 (nrows,ncols)
18 col=np.arange(ncols)
19 col=['c' + i fo
20 df.columns=col

```

Index	c0	c1	c2	c3	c4	c5
0	5	0	3	7	3	4
1	9	3	5	4	2	9
2	7	6	8	10	8	10
3	1	6	7	8	7	10
4	1	5	9	9	8	8
5	4	3	0	5	3	1
6	0	2	3	1	8	1
7	3	3	3	0	7	7
8	1	9	9	10	0	9
9	4	7	3	7	2	9
10	2	0	0	5	4	3
11	5	6	8	1	4	6

```

Name      Type      Size      Value
col       list       6         ['c 0', 'c 1', 'c 2', 'C 3', 'C 4', ...]
columns   list       5         ['c0', 'c 1', 'c_2', 'C 4', '5c']
data      Array of int32 (12, 5) [[ 5  0  3  3  7]]
df        DataFrame (12, 6) Column names: c 0, c 1, c 2, c 3, c ...
ncols     int        1         6
neworder  list       6         ['c0', 'c1', 'c2', '5c', 'c4', 'c6']
nrows     int        1         12

```

```

IPython 7.11.1 -- An enhanced Interactive Python.
Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/
dataframes.py', wdir='C:/Users/Phili/Documents/Python Scripts',
current_namespace=True)

In [2]: df.columns.str.lower()
Out[2]: Index(['c 0', 'c 1', 'c 2', 'c 3', 'c 4', 'c 5'],
dtype='object')

In [3]: df.columns=df.columns.str.lower()

In [4]:

```

We can also select the `str` attribute and then use the method `replace` to replace a repeating pattern in this case a space with an underscore replacement.

```
df.columns=df.columns.str.replace(pat=' ', repl='_')
```

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c 1','c_2','C 4','5c']
10 df=pd.DataFrame(data=data,columns=columns)
11 df['c6']=random.randint(low=0,high=11,size=12)
12 df.rename(columns={'c 1':'c1','c_2':'c2','C 4':'c4'},
13           inplace=True)
14 neworder=['c0','c1','c2','5c','c4','c6']
15 df=df[neworder]
16 # Create column names
17 (nrows,ncols)=df.shape
18 col=np.arange(ncols)
19 col=['c' + i for i in range(ncols)]
20 df.columns=col

```

Index	c0	c1	c2	c3	c4	c5
0	5	0	3	7	3	4
1	9	3	5	4	2	9
2	7	6	8	10	8	10
3	1	6	7	8	7	10
4	1	5	9	9	8	8
5	4	3	0	5	3	1
6	0	2	3	1	8	1
7	3	3	3	0	7	7
8	1	9	9	10	0	9
9	4	7	3	7	2	9
10	2	0	0	5	4	3
11	5	6	8	1	4	6

```

Name      Type      Size      Value
col       list       6         ['c0', 'c1', 'c2', '5c', 'c4', 'c6']
columns   list       5         ['c0', 'c 1', 'c_2', 'C 4', '5c']
data      Array of int32 (12, 5) [[ 5  0  3  3  7]]
df        DataFrame (12, 6) Column names: c_0, c_1, c_2, c_3, c_...
ncols     int        1         6
neworder  list       6         ['c0', 'c1', 'c2', '5c', 'c4', 'c6']
nrows     int        1         12

```

```

IPython 7.11.1 -- An enhanced Interactive Python.
Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/
dataframes.py', wdir='C:/Users/Phili/Documents/Python Scripts',
current_namespace=True)

In [2]: df.columns=df.columns.str.lower()

In [3]: df.columns=df.columns.str.replace(pat=' ', repl='_')

In [4]:

```

Deleting a Series (Column) or Index (Row)

We can delete a column by using the method `drop`. This method has two additional keyword arguments `axis` similar to many of the numpy functions we seen earlier where `axis=0` acts on indexes (rows) and `axis=1` acts on series (columns). This method also has an `inplace` keyword argument which is set to `False` by default similar to the dataframe method `rename` which we used earlier.

```
df.drop('c_0', axis=1, inplace=True)
```

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c1','c2','c4','5c']
10 df=pd.DataFrame(data=data,columns=columns)
11 df['c6']=random.randint(low=0,high=11,size=12)
12 df.rename(columns={'c1':'c1','c2':'c2','c4':'c4'},
13           inplace=True)
14 neworder=['c0',
15           'c6']
16 df=df[neworder]
17 # Create column
18 (nrows,ncols)=np
19 col=np.arange(nc
20 df.columns=col
21 df.columns=df.co
22 df.columns=df.co

```

Index	c-1	c-2	c-3	c-4	c-5
0	0	3	7	3	4
1	3	5	4	2	9
2	6	8	10	8	10
3	6	7	8	7	10
4	5	9	9	8	8
5	3	0	5	3	1
6	2	3	1	8	1
7	3	3	0	7	7
8	9	9	10	0	9
9	7	3	7	2	9
10	0	0	5	4	3
11	6	8	1	4	6

```

Python 3.7.4 (default, Aug 9 2019, 18:34:13) [MSC v.1915 64
bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.11.1 -- An enhanced Interactive Python.

Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/
dataframes.py', wdir='C:/Users/Phili/Documents/Python Scripts',
current_namespace=True)

In [2]: df.drop('c_0',axis=1,inplace=True)

In [3]:

```

Multiple series (columns) can be dropped by inputting a list:

```
df.drop(['c_3','c_5'],axis=1,inplace=True)
```

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c1','c2','c4','5c']
10 df=pd.DataFrame(data=data,columns=columns)
11 df['c6']=random.randint(low=0,high=11,size=12)
12 df.rename(columns={'c1':'c1','c2':'c2','c4':'c4'},
13           inplace=True)
14 neworder=['c0',
15           'c6']
16 df=df[neworder]
17 # Create column
18 (nrows,ncols)=np
19 col=np.arange(nc
20 df.columns=col
21 df.columns=df.co
22 df.columns=df.co

```

Index	c-1	c-2	c-4
0	0	3	3
1	3	5	2
2	6	8	8
3	6	7	7
4	5	9	8
5	3	0	3
6	2	3	8
7	3	3	7
8	9	9	0
9	7	3	2
10	0	0	4
11	6	8	4

```

Python 3.7.4 (default, Aug 9 2019, 18:34:13) [MSC v.1915 64
bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.11.1 -- An enhanced Interactive Python.

Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/
dataframes.py', wdir='C:/Users/Phili/Documents/Python Scripts',
current_namespace=True)

In [2]: df.drop('c_0',axis=1,inplace=True)

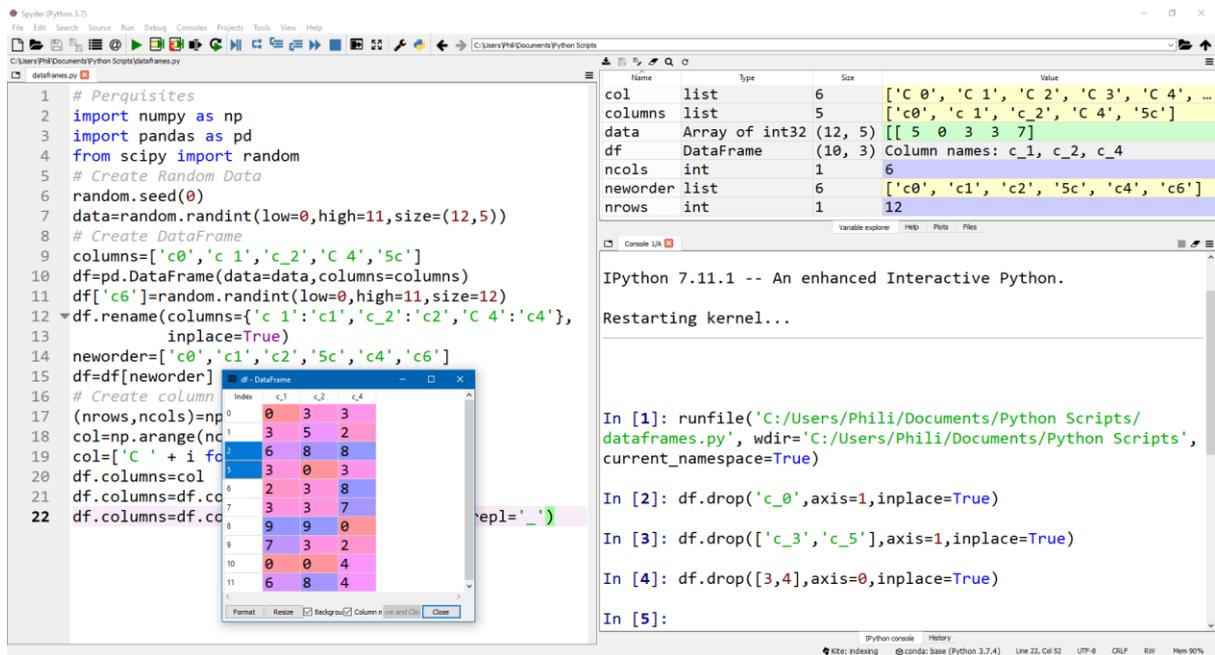
In [3]: df.drop(['c_3','c_5'],axis=1,inplace=True)

In [4]:

```

Indexes (rows) can be deleted by using the numerical value of the rows and setting `axis=0`.

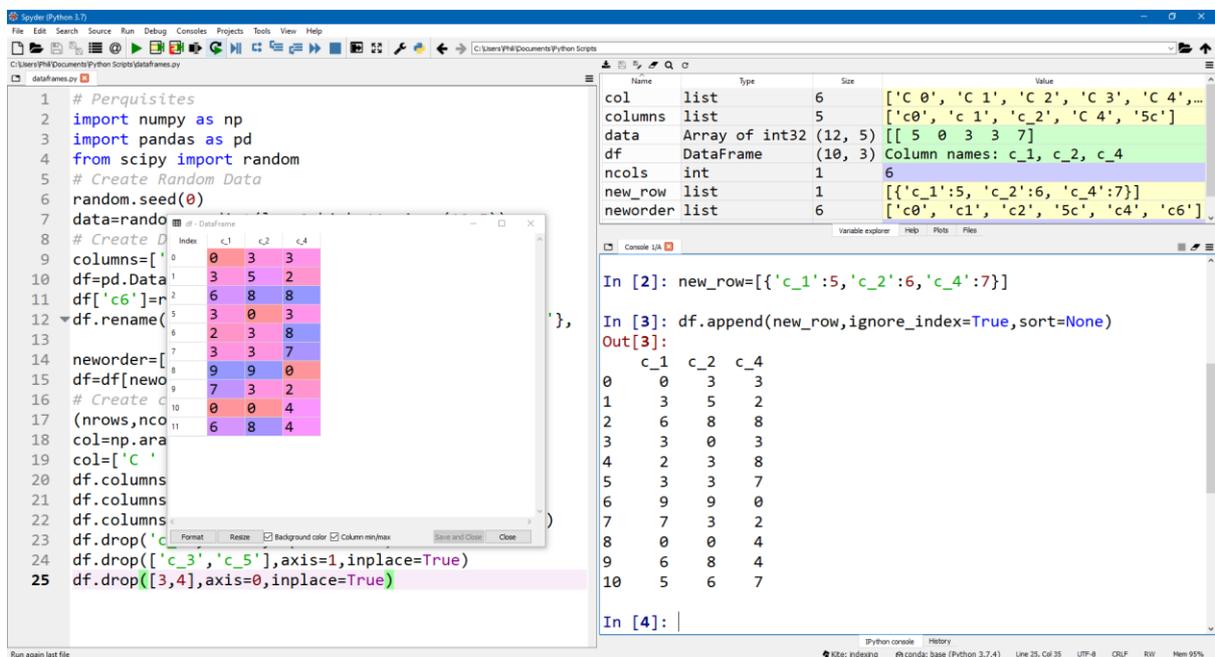
```
df.drop([3,4],axis=0,inplace=True)
```



Adding an Index (Row)

An index (row) can be added using the method `append`. In this case the first input argument to `append`, the new index (row) must be a dictionary nested as a list. Each of its keywords should correspond to the names of the series (columns) in the dataframe and the values should be the values to be updated.

```
new_row=[{'c_1':5, 'c_2':6, 'c_4':7}]
df.append(new_row, ignore_index=True, sort=None)
```



Note this method doesn't update the dataframe but prints the output to the console. Unlike the methods `drop` and `rename`, this method does not have the keyword input argument `inplace` and therefore one must manually reassign the output to the original dataframe for an inplace update.

```
df=df.append(new_row,ignore_index=True,sort=None)
```

The screenshot shows the Spyder Python IDE with a script named 'dataframes.py'. The code defines a DataFrame with columns 'c_0', 'c_1', 'c_2', 'c_4', and '5c'. It then renames columns 'c_1', 'c_2', and 'c_4' to 'c1', 'c2', and 'c4' respectively. A new row is added to the DataFrame. The Variable Explorer on the right shows the state of the DataFrame after these operations. The console output shows the Python version and IPython version.

Name	Type	Size	Value
col	list	6	['c_0', 'c_1', 'c_2', 'c_3', 'c_4', ...]
columns	list	5	['c0', 'c_1', 'c_2', 'c_4', '5c']
data	Array of int32	(12, 5)	[[5 0 3 3 7]]
df	DataFrame	(11, 3)	Column names: c_1, c_2, c_4
ncols	int	1	6
new_row	list	1	[{'c_1':5, 'c_2':6, 'c_4':7}]
neworder	list	6	['c0', 'c1', 'c2', '5c', 'c4', 'c6']

Sorting Data

We can sort a pandas series (column) by using the method `sort_values()`. We can use this method on a column by inputting the dataframe name, then using the column attribute and then call the method `sort_values()` without an input argument:

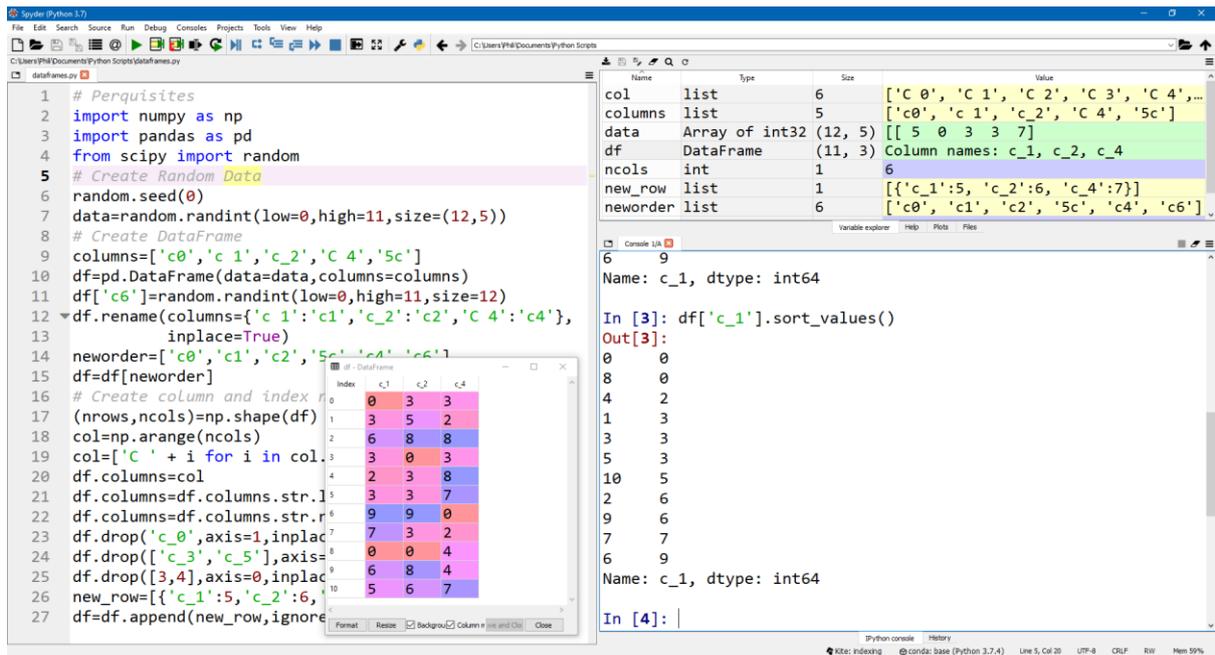
```
df.c_1.sort_values()
```

The screenshot shows the Spyder Python IDE with the same script as above. The code now includes `df.c_1.sort_values()` in the console. The Variable Explorer shows the DataFrame after sorting. The console output shows the result of the sorting operation.

Index	c_1	c_2	c_4
0	0	3	3
1	3	5	2
2	6	8	8
3	3	0	3
4	2	3	8
5	3	3	7
6	9	9	0
7	7	3	2
8	0	0	4
9	6	8	4
10	5	6	7

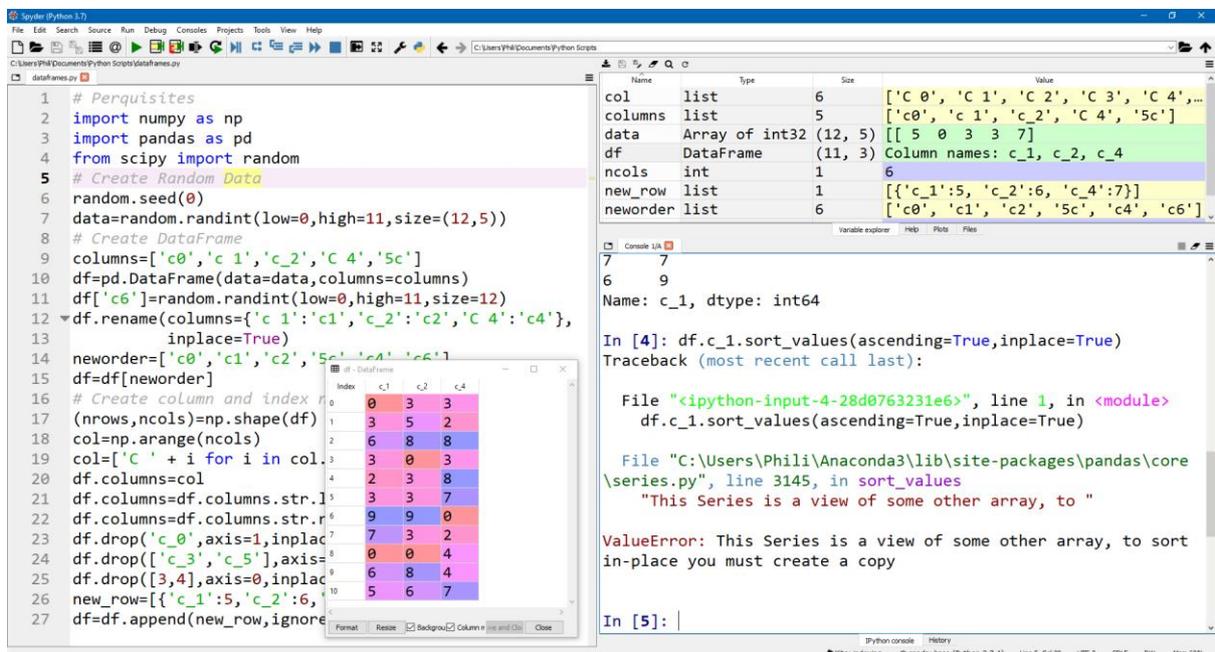
This can also be done by selecting the column using square bracket notation:

```
df['c_1'].sort_values()
```



The keyword input argument `ascending` can be reassigned from the default value of `True` to `False` to sort the data in descending order. The dataframe method `sort_values` also has the keyword input arguments `axis` and `inplace` which we have seen before. When `axis=0` the method `sort_values` works on all the rows of the selected column. `inplace` should be set to `True` when replacing the original column within the dataframe and set to `False` when previewing the changes for example in the console.

```
df.c_1.sort_values(ascending=True,inplace=True)
```



Note because we are sorting the values of an attribute of the dataframe we get the following error if we attempt an inplace update. `ValueError: This Series is a view of some other array`, to sort in-place you must create a copy. To get around this we should instead use the method `sort_values` directly on the dataframe.

```
df.sort_values(['c_1'])
```

The screenshot shows the Spyder Python IDE with a script named 'dataframes.py'. The script performs the following steps:

- Imports numpy, pandas, and scipy.
- Creates random data with 12 rows and 5 columns.
- Creates a DataFrame with columns 'c_0', 'c_1', 'c_2', 'c_4', and '5c'.
- Generates a new order for the rows.
- Sorts the DataFrame by column 'c_1' using `df.sort_values(['c_1'])`.

The Variable explorer shows the DataFrame `df` with 11 rows and 3 columns (`c_1`, `c_2`, `c_4`). The console output shows the sorted DataFrame:

	c_1	c_2	c_4
0	0	3	3
8	0	0	4
4	2	3	8
1	3	5	2
3	3	0	3
5	3	3	7
10	5	6	7
2	6	8	8
9	6	8	4
7	7	3	2
6	9	9	0

```
df.sort_values(['c_1'], inplace=True)
```

This screenshot is similar to the previous one, but the script uses `df.sort_values(['c_1'], inplace=True)` to sort the DataFrame in-place. The console output shows the sorted DataFrame after the in-place operation:

	c_1	c_2	c_4
0	0	3	3
8	0	0	4
4	2	3	8
1	3	5	2
3	3	0	3
5	3	3	7
10	5	6	7
2	6	8	8
9	6	8	4
7	7	3	2
6	9	9	0

Note that although the dataframe `df` is sorted, that the original indexes remain.

We can sort using multiple pandas series (columns) by passing in a list of the pandas series (column) names. In this case let's create a list `sort_order` which contains the pandas series (column) names we wish to sort by called `sort_order`. The pandas series (column) name that is the 0th index of the vector `sort_order` is used to sort the data in `df` in this case `'c_2'`. When two indexes in `'c_2'` are the same value, the values in `'c_4'` are then sorted in order.

```
sort_order=['c_2', 'c_4']  
df.sort_values(sort_order, inplace=True)
```

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c 1','c_2','C 4','5c']
10 df=pd.DataFrame(data=data,columns=columns)
11 df['c6']=random.randint(low=0,high=11,size=12)
12 df.rename(columns={'c 1':'c1','c_2':'c2','C 4':'c4'},
13           inplace=True)
14 neworder=['c0','c1','c2','5c','c4','c6']
15 df=df[neworder]
16 # Create column and index
17 (nrows,ncols)=np.shape(df)
18 col=np.arange(ncols)
19 col=['C ' + i for i in col]
20 df.columns=col
21 df.columns=df.columns.str.lstrip()
22 df.columns=df.columns.str.rstrip()
23 df.drop('c_0',axis=1,inplace=True)
24 df.drop(['c_3','c_5'],axis=1,inplace=True)
25 df.drop([3,4],axis=0,inplace=True)
26 new_row=[{'c_1':5,'c_2':6,'c_4':7}]
27 df=df.append(new_row,ignore_index=True)

```

Name	Type	Size	Value
col	list	6	['C 0', 'C 1', 'C 2', 'C 3', 'C 4', ...]
columns	list	5	['c0', 'c 1', 'c_2', 'C 4', '5c']
data	Array of int32	(12, 5)	[[5 0 3 3 7]]
df	DataFrame	(11, 3)	Column names: c_1, c_2, c_4
ncols	int	1	6
new_row	list	1	[{'c_1':5, 'c_2':6, 'c_4':7}]
neworder	list	6	['c0', 'c1', 'c2', '5c', 'c4', 'c6']

```

In [6]: df.sort_values(['c_1'],inplace=True)
In [7]: sort_order=['c_2','c_4']
In [8]: df.sort_values(sort_order,inplace=True)
In [9]:

```

This of course could be combined into one line:

```
df.sort_values(['c_2','c_4'],inplace=True)
```

If we restart the kernel and then relaunch the script, we can try this:

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c 1','c_2','C 4','5c']
10 df=pd.DataFrame(data=data,columns=columns)
11 df['c6']=random.randint(low=0,high=11,size=12)
12 df.rename(columns={'c 1':'c1','c_2':'c2','C 4':'c4'},
13           inplace=True)
14 neworder=['c0','c1','c2','5c','c4','c6']
15 df=df[neworder]
16 # Create column and index
17 (nrows,ncols)=np.shape(df)
18 col=np.arange(ncols)
19 col=['C ' + i for i in col]
20 df.columns=col
21 df.columns=df.columns.str.lstrip()
22 df.columns=df.columns.str.rstrip()
23 df.drop('c_0',axis=1,inplace=True)
24 df.drop(['c_3','c_5'],axis=1,inplace=True)
25 df.drop([3,4],axis=0,inplace=True)
26 new_row=[{'c_1':5,'c_2':6,'c_4':7}]
27 df=df.append(new_row,ignore_index=True)

```

Name	Type	Size	Value
col	list	6	['C 0', 'C 1', 'C 2', 'C 3', 'C 4', ...]
columns	list	5	['c0', 'c 1', 'c_2', 'C 4', '5c']
data	Array of int32	(12, 5)	[[5 0 3 3 7]]
df	DataFrame	(11, 3)	Column names: c_1, c_2, c_4
ncols	int	1	6
new_row	list	1	[{'c_1':5, 'c_2':6, 'c_4':7}]
neworder	list	6	['c0', 'c1', 'c2', '5c', 'c4', 'c6']

```

Python 3.7.4 (default, Aug 9 2019, 18:34:13) [MSC v.1915 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.
IPython 7.11.1 -- An enhanced Interactive Python.
Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/
dataframes.py', wdir='C:/Users/Phili/Documents/Python Scripts',
current_namespace=True)

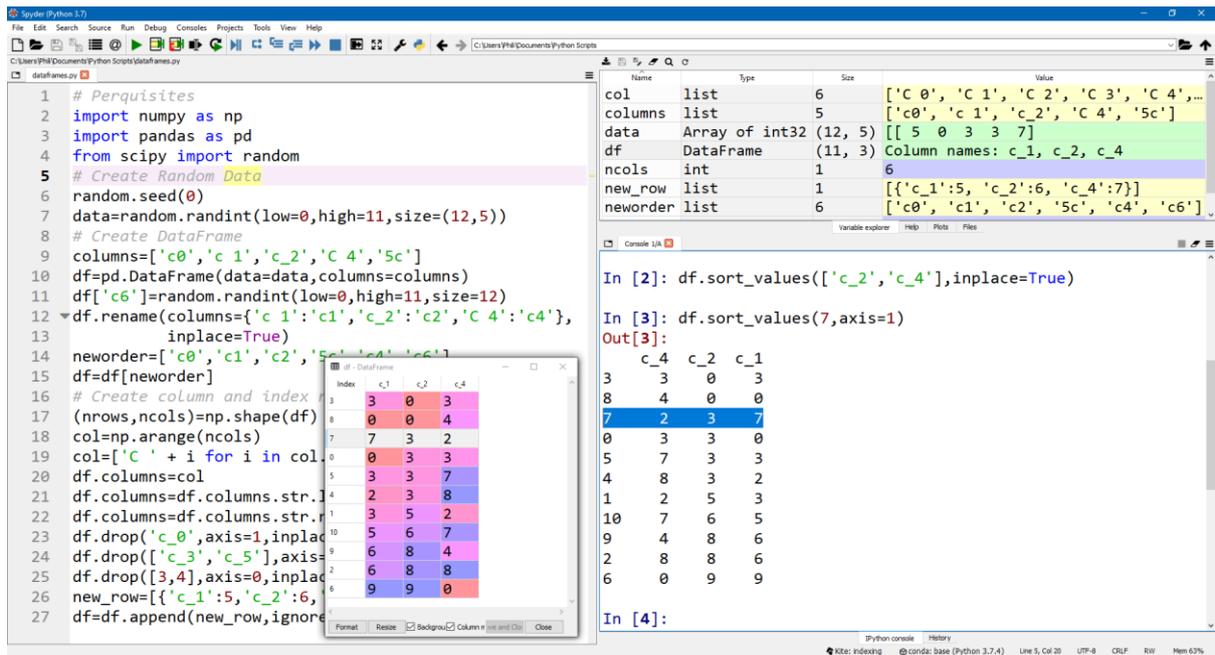
In [2]: df.sort_values(['c_2','c_4'],inplace=True)

In [3]:

```

We can also select the data by an index and then reorder the data in the columns. For the dataframe `df` we must select the index by its numeric value. Once again `axis=1` means the operation is carried out on the columns (of the rows).

```
df.sort_values(7,axis=1)
```



Filtering

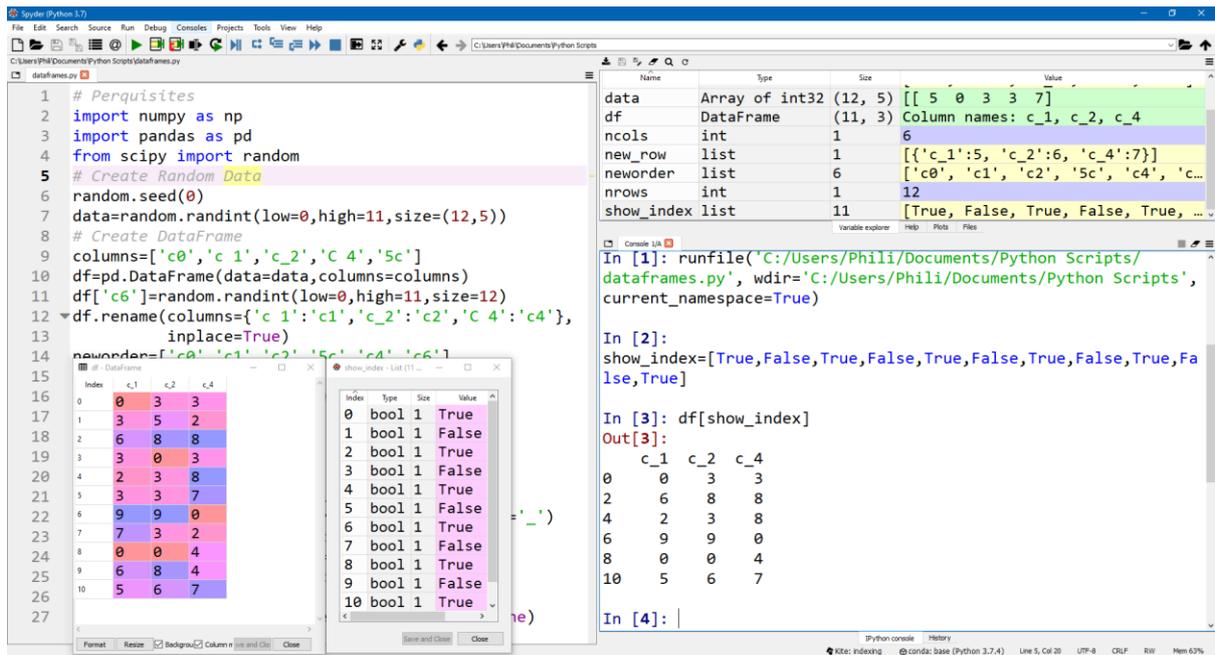
Filtering is typically carried out by use of a Boolean. Let's create a Boolean array which has the same length as a series in the dataframe `df`.

```
show_index=[True, False, True, False, True, False, True, False, True, False, True, False, True]
```

For convenience let's restart the kernel and run the kernel so the indexes are in numerical order. Now we can open up the dataframe `df` and the Boolean list `show_index` in the variable explorer side by side. Now we can index into the dataframe `df` by use of the Boolean.

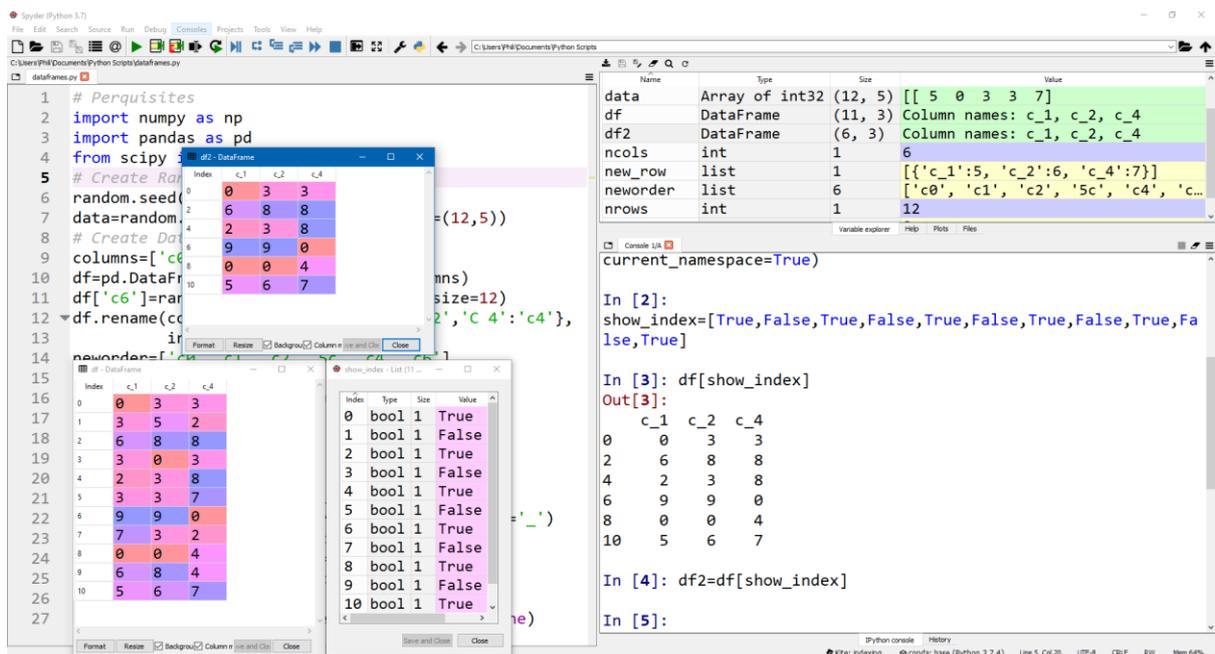
```
df[show_index]
```

Indexing into a dataframe by the Boolean list will create a new dataframe that contains only the indexes which were `True` in the Boolean list.



This new dataframe is not assigned to an output variable name and only previewed in the console. We can save it to a new variable name, in this case `df2`:

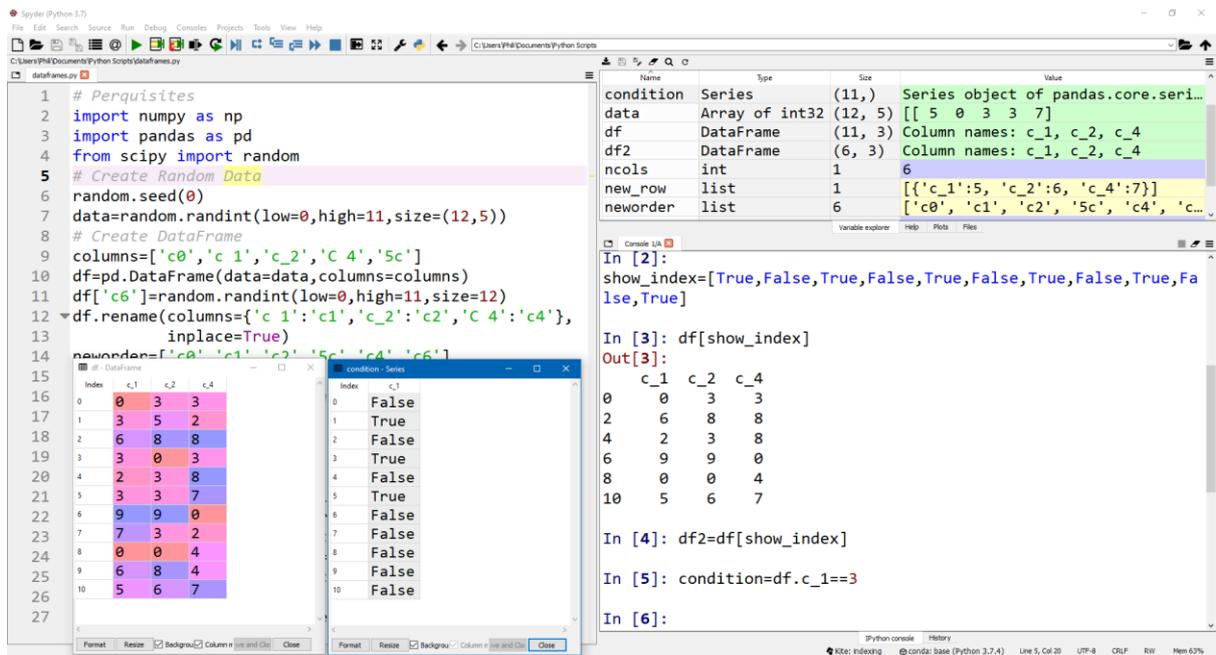
```
df2=df[show_index]
```



In the above case we have made a filter selection by manually creating a Boolean vector. Quite often this Boolean vector is created by examining a condition applicable for example to one of the pandas series (columns). For example, we can check each value of the series `'c_1'` and assign a `True` statement only if it is equal to the value of `3`.

```
condition=df.c_1==3
```

Note that `condition` shows up on the Variable explorer as a pandas series.

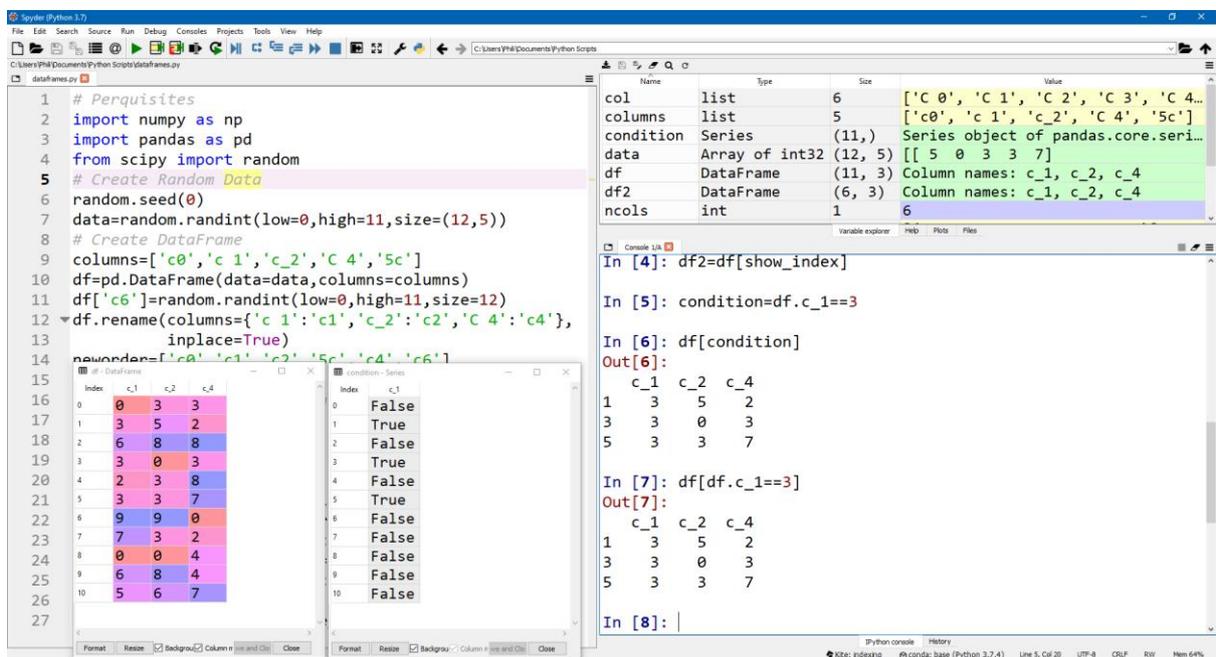


This can then be used to index into the DataFrame `df` to filter by this condition.

```
df[condition]
```

This can be combined in one step by using:

```
df[df.c_1==3]
```



Multiple logical statements can be indexed in this way. Each condition has to be enclosed in parenthesis and the symbols `&` or `|` can be used to combine the confitions.

```
df[(df.c_1==3) & (df.c_2<5)]
```

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c 1','c_2','C 4','5c']
10 df=pd.DataFrame(data=data,columns=columns)
11 df['c6']=random.randint(low=0,high=11,size=12)
12 df.rename(columns={'c 1':'c1','c_2':'c2','C 4':'c4'},
13           inplace=True)
14 new_row={'c0':'c1','c2':'5c','c4':'c4','c6'}
15
16
17
18
19
20
21
22
23
24
25
26
27

```

Name	Type	Size	Value
col	list	6	['C 0', 'C 1', 'C 2', 'C 3', 'C 4'...
columns	list	5	['c0', 'c 1', 'c_2', 'C 4', '5c']
condition	Series	(11,)	Series object of pandas.core.serie...
data	Array of int32	(12, 5)	[[5 0 3 3 7]
df	DataFrame	(11, 3)	Column names: c_1, c_2, c_4
df2	DataFrame	(6, 3)	Column names: c_1, c_2, c_4
ncols	int	1	6

```

In [7]: df[df.c_1==3]
Out[7]:
   c_1  c_2  c_4
1    3    5    2
3    3    0    3
5    3    3    7

In [8]: df[(df.c_1==3) & (df.c_2<5)]
Out[8]:
   c_1  c_2  c_4
3    3    0    3
5    3    3    7

In [9]:

```

So far, we have only looked at dataframes containing a panda series (column) of numerical values. However, we can see that the variable condition is a panda series of Boolean values. We can add this to the dataframe `df`. Recall that when assigning we need to use square bracket indexing to create a new panda series within an existing dataframe.

```
df['condition']=condition
```

We can then use the attribute `dtypes` to see the datatypes of each pandas series.

```
df.dtypes
```

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c 1','c_2','C 4','5c']
10 df=pd.DataFrame(data=data,columns=columns)
11 df['c6']=random.randint(low=0,high=11,size=12)
12 df.rename(columns={'c 1':'c1','c_2':'c2','C 4':'c4'},
13           inplace=True)
14 new_row={'c0':'c1','c2':'5c','c4':'c4','c6'}
15
16
17
18
19
20
21
22
23
24
25
26
27

```

Name	Type	Size	Value
col	list	6	['C 0', 'C 1', 'C 2', 'C 3', 'C 4'...
columns	list	5	['c0', 'c 1', 'c_2', 'C 4', '5c']
condition	Series	(11,)	Series object of pandas.core.serie...
data	Array of int32	(12, 5)	[[5 0 3 3 7]
df	DataFrame	(11, 4)	Column names: c_1, c_2, c_4, condi...
ncols	int	1	6
new_row	list	1	[{'c_1':5, 'c_2':6, 'c_4':7}]

```

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/
dataframes.py', wdir='C:/Users/Phili/Documents/Python Scripts',
current_namespace=True)

In [2]: condition=df.c_1==3

In [3]: df['condition']=condition

In [4]: df.dtypes
Out[4]:
c_1          int64
c_2          int64
c_4          int64
condition    bool
dtype: object

In [5]:

```

Here we see that the pandas series `'condition'` added to the dataframe is of type bool (Boolean).

Categorical

We can prescribe the numeric value in 'c_1' to a grade where a value less than 4 is low grade, a value of greater than or equal to 4 to greater than of equal to 6 is of medium grade and a value of greater than 6 is of high grade.

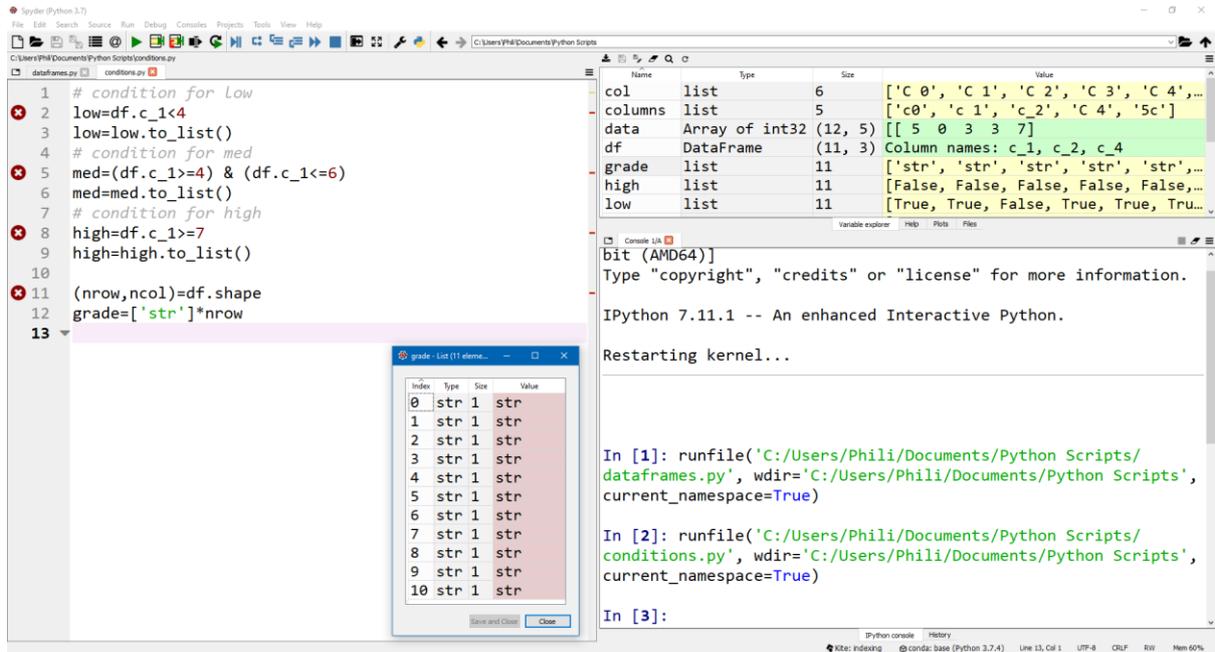
```
1. # condition for low
2. low=df.c_1<4
3. # condition for med
4. med=(df.c_1>=4) & (df.c_1<=6)
5. # condition for high
6. high=df.c_1>6
```

In order to use these as conditions within an `if`, `elif` and `if` statement nested within a `for` loop we need to convert the panda series (columns) generated to lists.

```
1. # condition for low
2. low=df.c_1<4
3. low=low.to_list()
4. # condition for med
5. med=(df.c_1>=4) & (df.c_1<=6)
6. med=med.to_list()
7. # condition for high
8. high=df.c_1>6
9. high=high.to_list()
```

Now we can use the method `df.shape` to get the number of Indexes (rows) `nrow` and number of pandas series (columns) `ncol`. We can then initialise a list of strings called grades by taking advantage of string concatenation.

```
1. # condition for low
2. low=df.c_1<4
3. low=low.to_list()
4. # condition for med
5. med=(df.c_1>=4) & (df.c_1<=6)
6. med=med.to_list()
7. # condition for high
8. high=df.c_1>=7
9. high=high.to_list()
10.
11. (nrow,ncol)=df.shape
12. grade=['str']*nrow
```



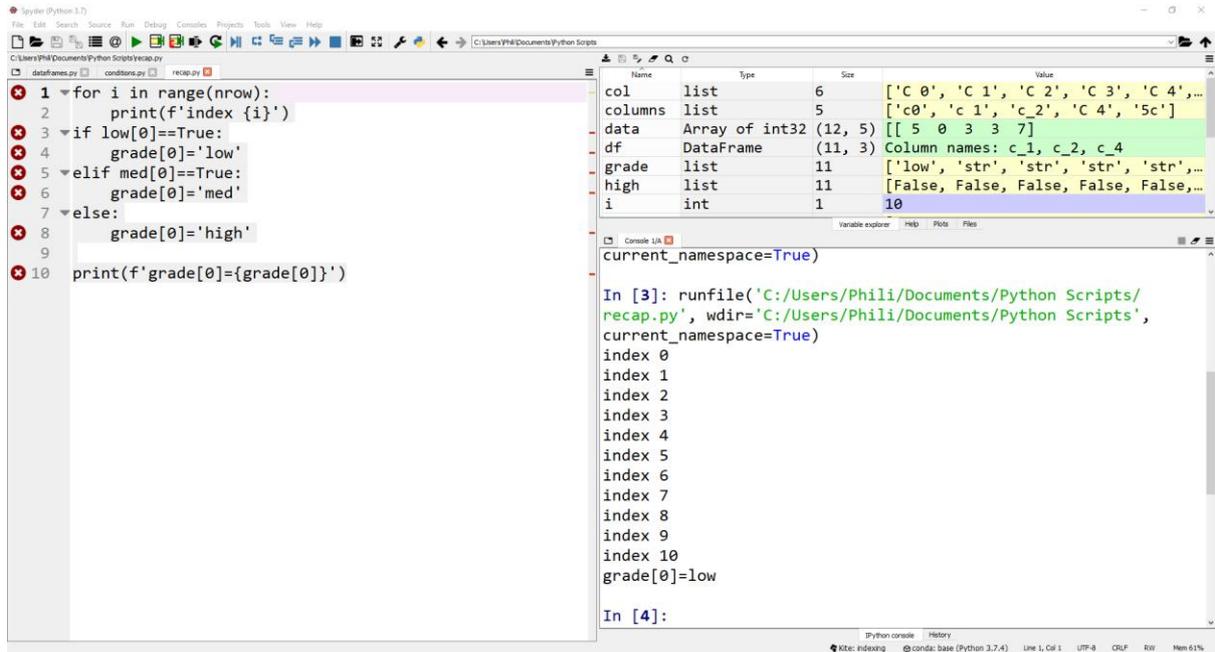
Note the script `conditions.py` has an error every time the DataFrame `df` is called. In this case the script `conditions.py` requires the separate script `dataframes.py` to be ran in advance to generate the dataframe `df`. In our case `df` is in the console namespace and spyder's general settings are set to run scripts in the consoles mainspace opposed to an empty one so this script will execute. For convenience we will restart the kernel, then run `dataframes.py`.

Now we can recreate the `for` loop with nested `if`, `elif` and `if` statement. To recap we can look at the `for` loop and the `if`, `elif` and `if` statement individually:

```

1. for i in range(nrow):
2.     print(f'index {i}')
3.     if low[0]==True:
4.         grade[0]='low'
5.     elif med[0]==True:
6.         grade[0]='med'
7.     else:
8.         grade[0]='high'
9.
10. print(f'grade[0]={grade[0]}')

```

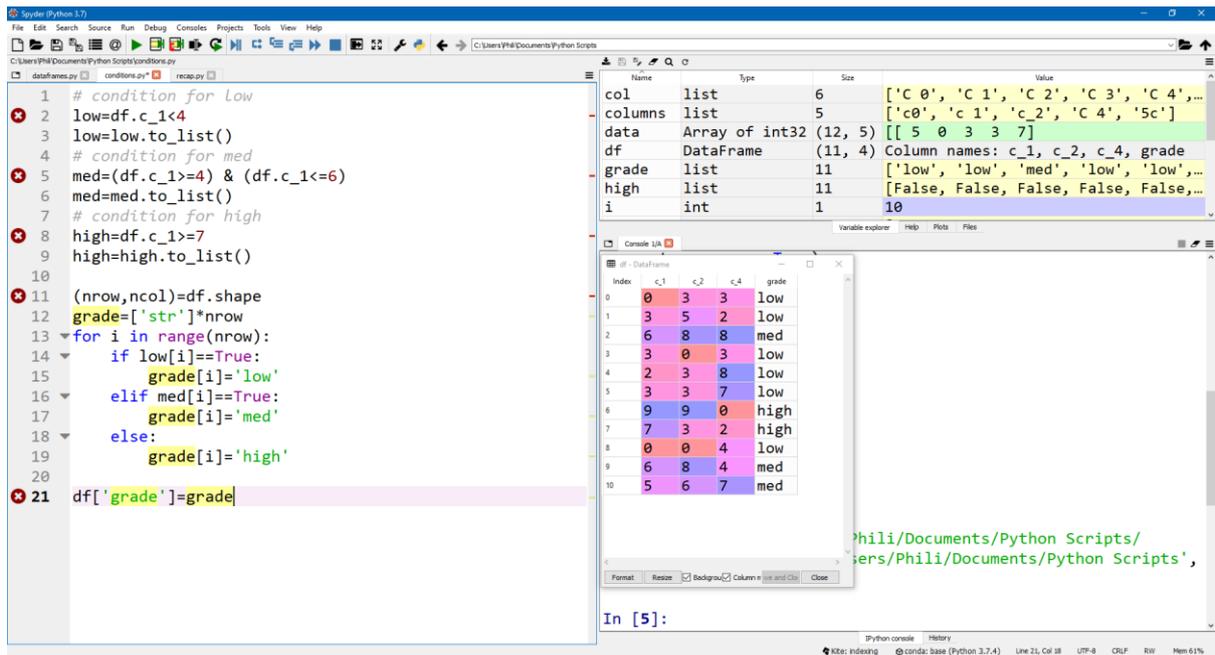


Finally, we can create the nest the **if, elif, else** statement within the **for** loop to create the list grade which we can add as a new series to the dataframe **df**.

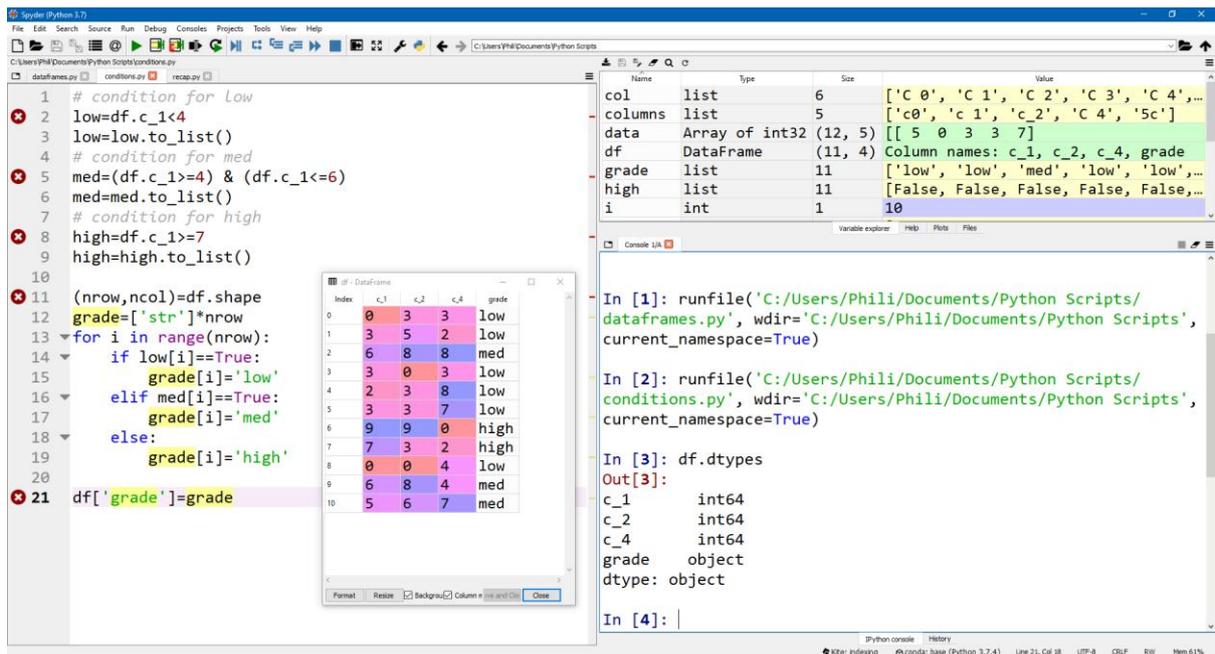
```

1. # condition for low
2. low=df.c_1<4
3. low=low.to_list()
4. # condition for med
5. med=(df.c_1>=4) & (df.c_1<=6)
6. med=med.to_list()
7. # condition for high
8. high=df.c_1>=7
9. high=high.to_list()
10.
11. (nrow,ncol)=df.shape
12. grade=['str']*nrow
13. for i in range(nrow):
14.     if low[i]==True:
15.         grade[i]='low'
16.     elif med[i]==True:
17.         grade[i]='med'
18.     else:
19.         grade[i]='high'
20.
21. df['grade']=grade

```



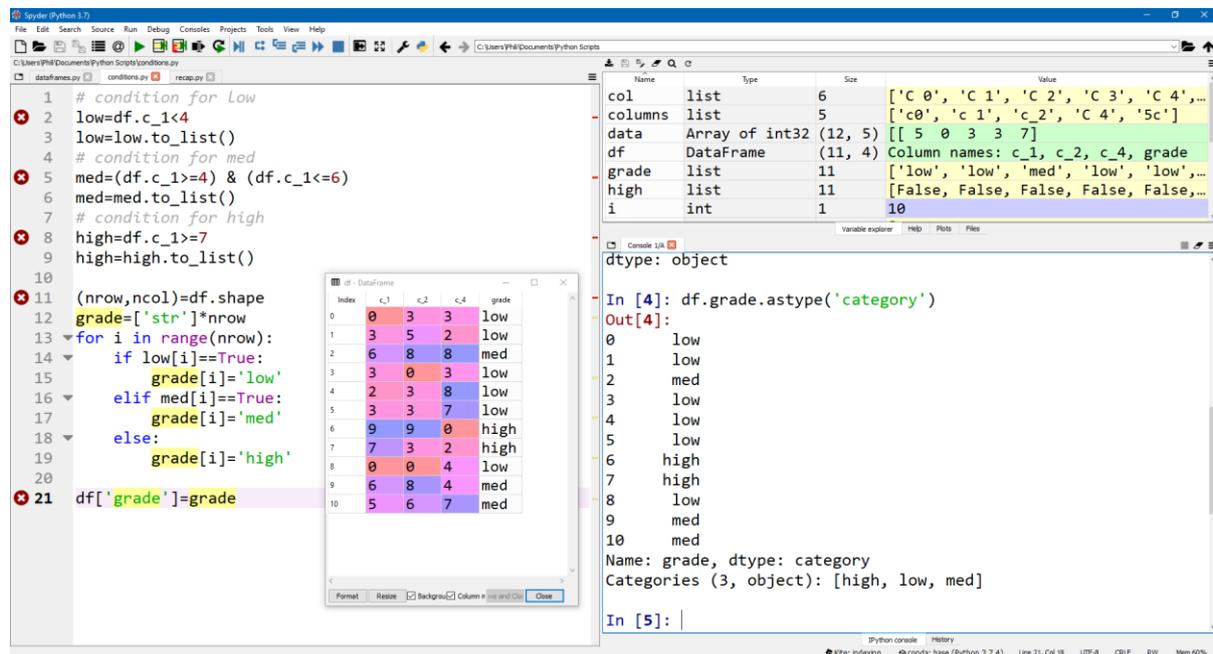
If we look at the attribute dtypes we can see the pandas series (column) 'grade' has a dtype of object.



This can be converted into a category using the method `astype` with the input argument being the string 'category'.

```
df.grade.astype('category')
```

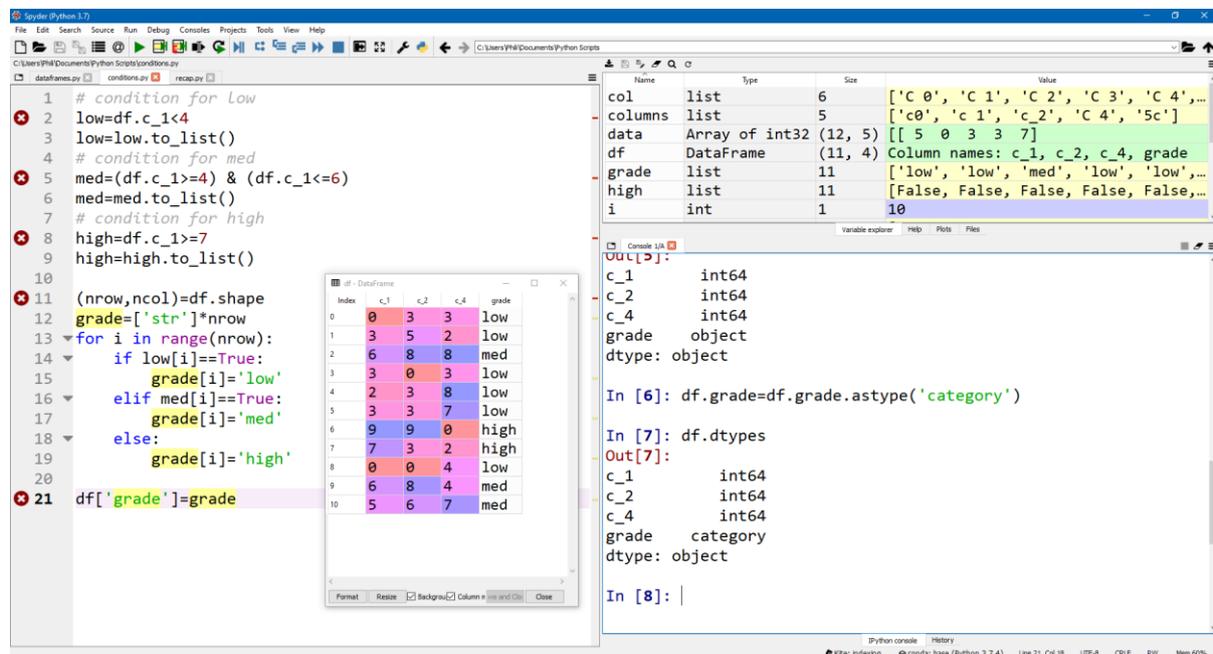
Notice that this outputs a panda series (column) and doesn't update the original panda series (column). In addition, the categories are listed in alphabetical order and not by 'low', 'med', 'high' like we might expect.



We need to perform an inplace upgrade of the pandas series (column):

```
df.grade=df.grade.astype('category')
df.dtypes
```

Now we will see the datatype of the pandas series 'grade' is updated to category.



Ordering Categories

Once the datatype is a category we type in the DataFrame `df`, select the categorical column `grade` as an attribute, and then select the `cat` attribute and then dot `.` and tab `⌘` to access a number of categorical attributes and methods.

The screenshot shows the Spyder Python IDE with a script named `conditions.py`. The code defines a DataFrame `df` with columns `c_1`, `c_2`, `c_4`, and `grade`. The `grade` column is initially of type `object`. The code then converts it to a `category` type. A variable explorer window shows the `df` object with its attributes and methods. A tooltip for the `cat` attribute lists several methods, with `reorder_categories` highlighted in red. The console shows the following interactions:

```

Out[5]:
c_1      int64
c_2      int64
c_4      int64
grade    object
dtype: object

In [6]: df.grade=df.grade.astype('category')

In [7]: df.dtypes
Out[7]:
c_1      int64
c_2      int64
c_4      int64
grade    category
dtype: object

In [8]: df.grade.cat.
  
```

We can use the method `reorder_categories` to specify the new order:

```
df.grade.cat.reorder_categories(['low', 'med', 'high'])
```

The screenshot shows the same Spyder Python IDE environment. The code in `conditions.py` is identical to the previous screenshot. The console shows the execution of the `reorder_categories` method:

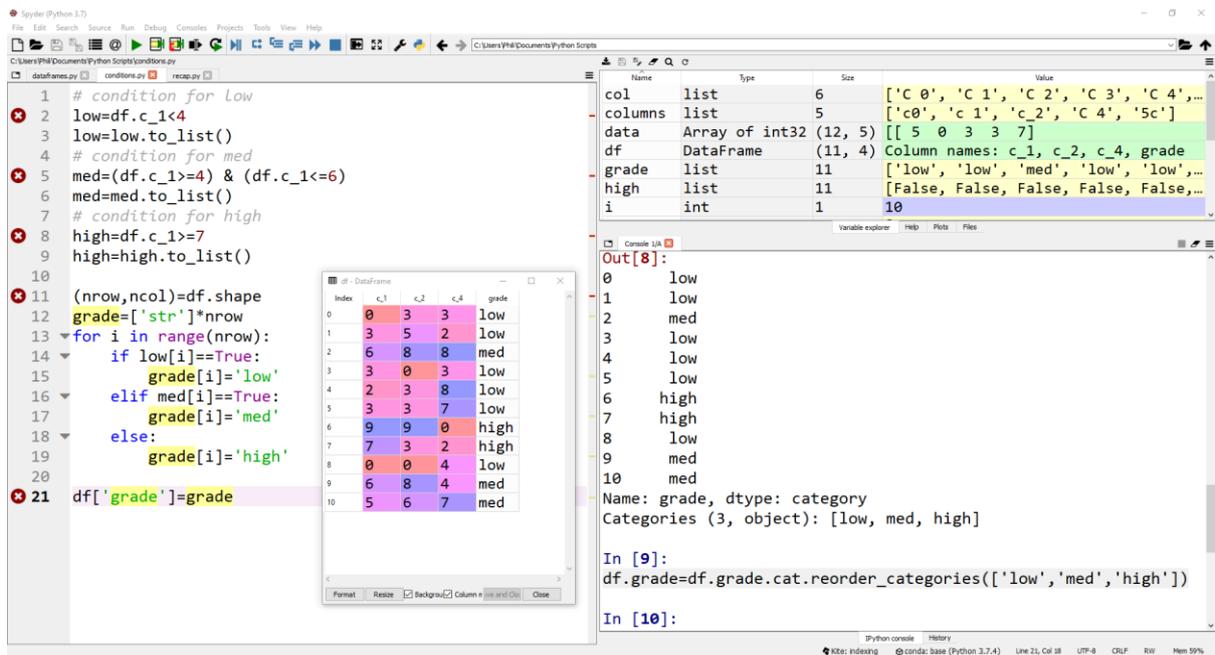
```

In [8]: df.grade.cat.reorder_categories(['low', 'med', 'high'])
Out[8]:
0    low
1    low
2    med
3    low
4    low
5    low
6    high
7    high
8    low
9    med
10   med
Name: grade, dtype: category
Categories (3, object): [low, med, high]

In [9]:
  
```

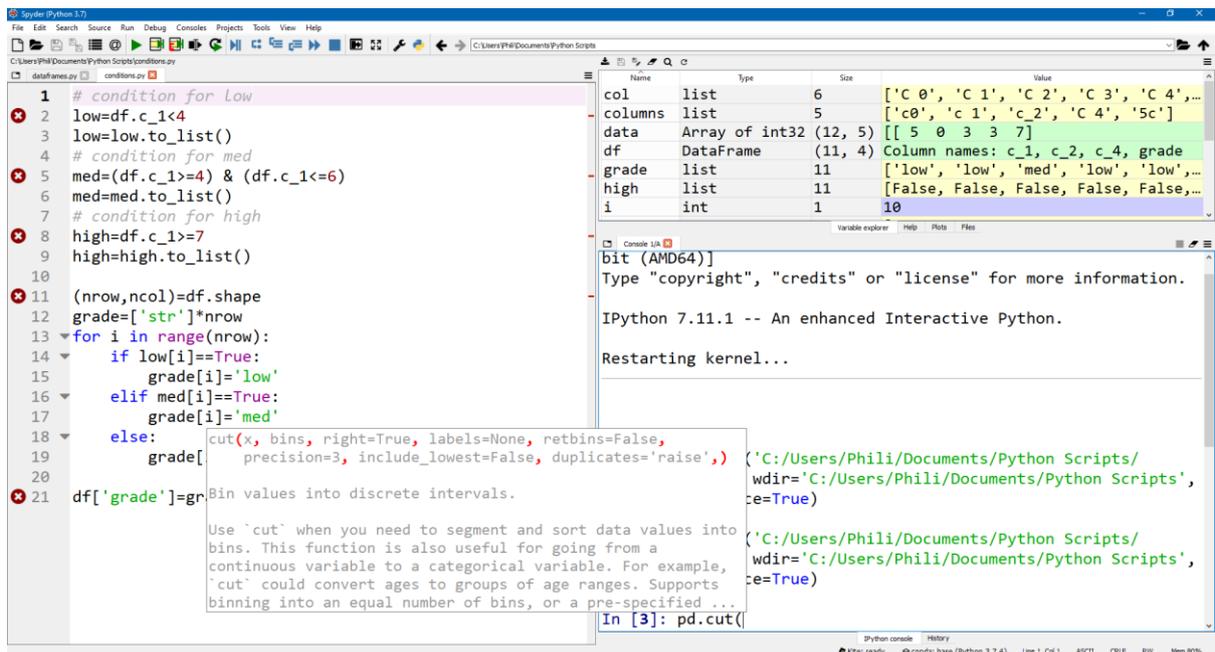
Notice once again that this outputs a pandas series (column) and doesn't update the original pandas series (column). We need to perform an in-place update:

```
df.grade=df.grade.cat.reorder_categories(['low', 'med', 'high'])
```



The Cut Function

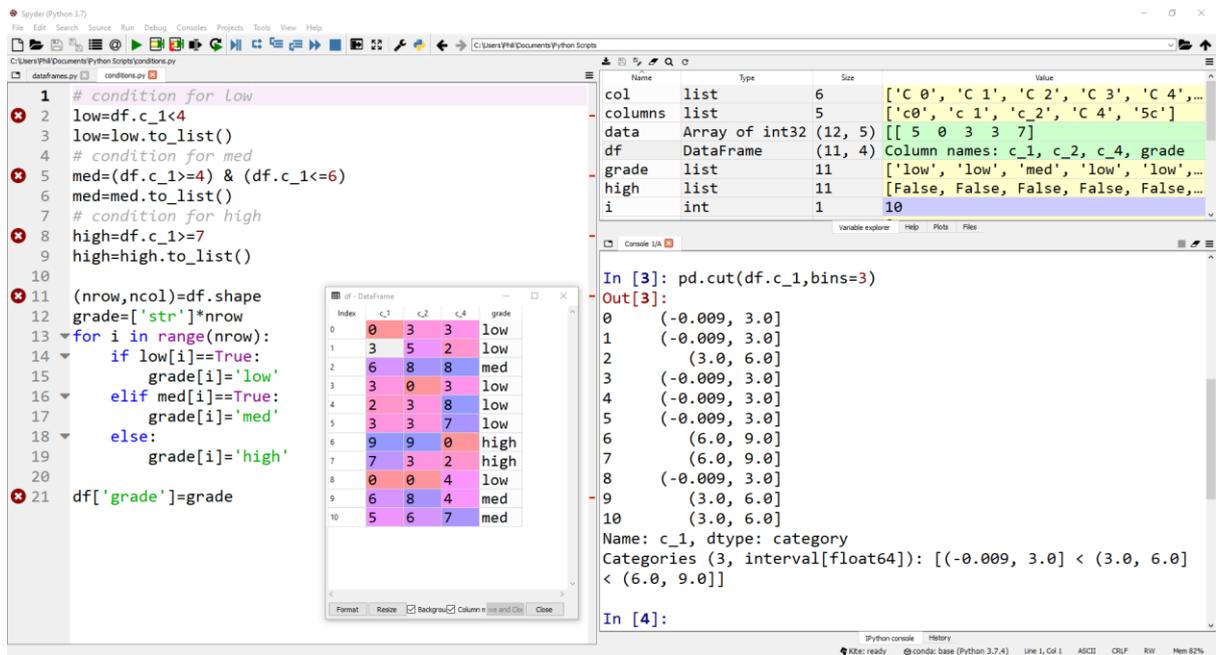
The procedure used above was quite a cumbersome way to create ordered categories and can be created with the function `cut`. This function has 2 positional input arguments, `x` which is the column to be cut and `bins` which is the number of bins we want to cut the column up into.



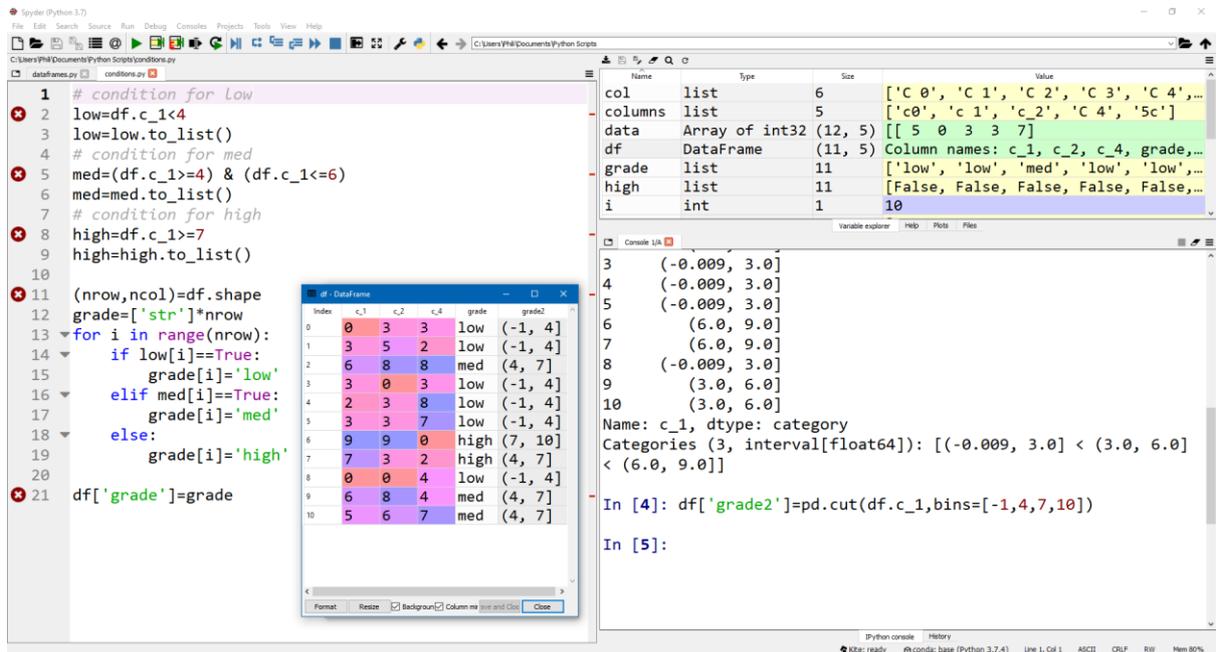
Let's use it to create 3 bins:

```
pd.cut(df.c_1,bins=3)
```

The three slices `(-0.009, 3.0]`, `(3.0, 6.0]` and `(6.0, 9.0]` are created. Note the `(` corresponds to greater than the lower bound and the `]` corresponds to less than or equal to the top bound.



The function will return a pandas series (column) which in this case isn't assigned to a new name so just displays in the console. It isn't quite what we want and instead of using a scalar to define the number of bins we can instead supply a vector. For example, if we supply the vector `[-1, 4, 6, 10]` then the boundaries will be `(-1, 4]`, `(4, 7]`, `(7, 10]` and `(10, inf]` respectively corresponding to the values of the boundaries we created earlier when we used the `if`, `elif` and `else` statement nested within a `for` loop. We can also assign this to a new panda series within the dataframe.



Notice that these all line up with our `'low'`, `'med'` and `'high'` categorical strings.

```
df['grade2']=pd.cut(df.c_1,bins=[-1,4,7,10])
```

Rename Categories

Note the default names are of the upper and lower boundaries. We can rename them to correspond to 'low', 'med' and 'high' by creating a list of the new category names and the category method `rename_categories`:

```
new_cat_names=['low', 'med', 'high']  
df.grade2.cat.rename_categories(new_cat_names)
```

Once again, this function will return a pandas series (column) which in this case isn't assigned to a new name so just displays in the console.

The screenshot shows the Spyder Python IDE with a script named 'conditions.py'. The code defines a DataFrame with columns 'c_1', 'c_2', 'c_4', 'grade', and 'grade2'. It then uses `df.grade2.cat.rename_categories(new_cat_names)` to rename the categories of the 'grade2' column. The console output shows the resulting series for 'grade2' with categories 'low', 'med', and 'high'.

```
1 # condition for low  
2 low=df.c_1<4  
3 low=low.to_list()  
4 # condition for med  
5 med=(df.c_1>=4) & (df.c_1<=6)  
6 med=med.to_list()  
7 # condition for high  
8 high=df.c_1>7  
9 high=high.to_list()  
10  
11 (nrow,ncol)=df.shape  
12 grade=['str']*nrow  
13 for i in range(nrow):  
14     if low[i]==True:  
15         grade[i]='low'  
16     elif med[i]==True:  
17         grade[i]='med'  
18     else:  
19         grade[i]='high'  
20  
21 df['grade2']=grade
```

Index	c_1	c_2	c_4	grade	grade2
0	0	3	3	low	(-1, 4]
1	3	5	2	low	(-1, 4]
2	6	8	8	med	(4, 6]
3	3	0	3	low	(-1, 4]
4	2	3	8	low	(-1, 4]
5	3	3	7	low	(-1, 4]
6	9	9	0	high	(6, 10]
7	7	3	2	high	(6, 10]
8	0	0	4	low	(-1, 4]
9	6	8	4	med	(4, 6]
10	5	6	7	med	(4, 6]

```
col      list      6      ['C 0', 'C 1', 'C 2', 'C 3', '...' ]  
columns  list      5      ['c0', 'c 1', 'c_2', 'C 4', '5...' ]  
data     Array of int32 (12, 5) [[ 5  0  3  3  7] ]  
df       DataFrame (11, 5) Column names: c_1, c_2, c_4, g...  
grade    list      11     ['low', 'low', 'med', 'low', '...' ]  
high     list      11     [False, False, False, False, F...]  
i        int       1      10
```

```
In [5]: new_cat_names=['low', 'med', 'high']  
In [6]: df.grade2.cat.rename_categories(new_cat_names)  
Out[6]:  
0 low  
1 low  
2 med  
3 low  
4 low  
5 low  
6 high  
7 high  
8 low  
9 med  
10 med  
Name: grade2, dtype: category  
Categories (3, object): [low < med < high]
```

We can reassign the original column to these new values for an inplace update:

```
df.grade2=df.grade2.cat.rename_categories(new_cat_names)
```

The screenshot shows the same Spyder Python IDE with the same script. The code now includes `df.grade2=df.grade2.cat.rename_categories(new_cat_names)` to perform an inplace update. The console output shows the updated series for 'grade2' with categories 'low', 'med', and 'high'.

```
1 # condition for low  
2 low=df.c_1<4  
3 low=low.to_list()  
4 # condition for med  
5 med=(df.c_1>=4) & (df.c_1<=6)  
6 med=med.to_list()  
7 # condition for high  
8 high=df.c_1>7  
9 high=high.to_list()  
10  
11 (nrow,ncol)=df.shape  
12 grade=['str']*nrow  
13 for i in range(nrow):  
14     if low[i]==True:  
15         grade[i]='low'  
16     elif med[i]==True:  
17         grade[i]='med'  
18     else:  
19         grade[i]='high'  
20  
21 df['grade2']=grade
```

Index	c_1	c_2	c_4	grade	grade2
0	0	3	3	low	low
1	3	5	2	low	low
2	6	8	8	med	med
3	3	0	3	low	low
4	2	3	8	low	low
5	3	3	7	low	low
6	9	9	0	high	high
7	7	3	2	high	high
8	0	0	4	low	low
9	6	8	4	med	med
10	5	6	7	med	med

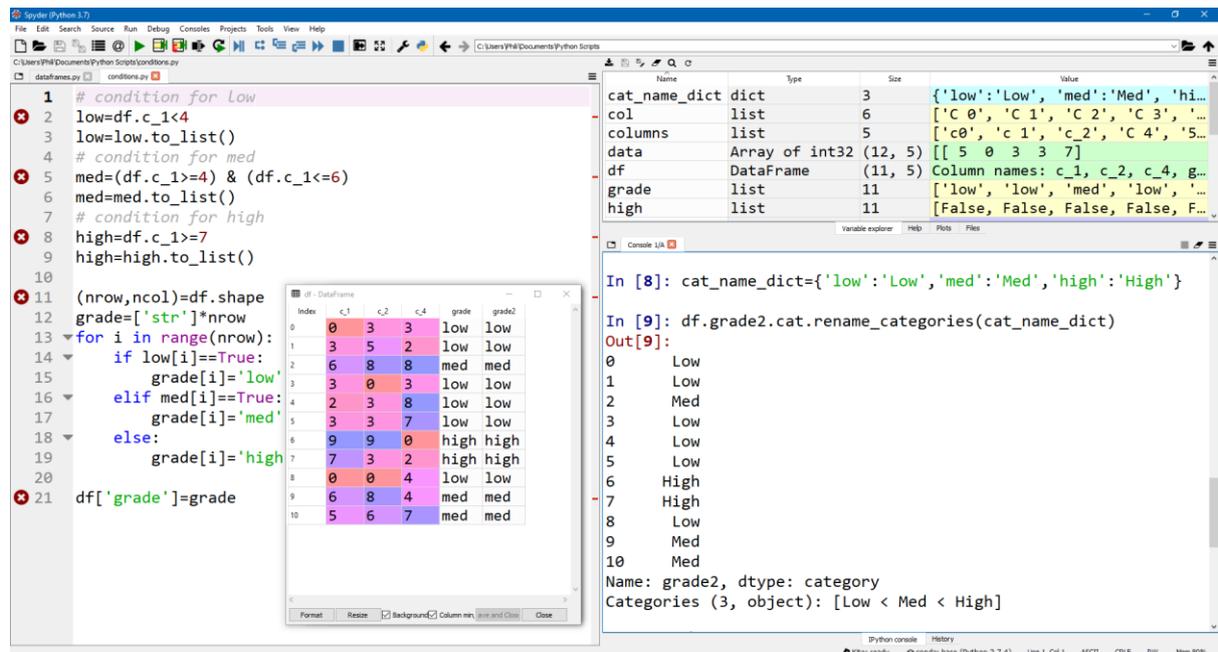
```
col      list      6      ['C 0', 'C 1', 'C 2', 'C 3', '...' ]  
columns  list      5      ['c0', 'c 1', 'c_2', 'C 4', '5...' ]  
data     Array of int32 (12, 5) [[ 5  0  3  3  7] ]  
df       DataFrame (11, 5) Column names: c_1, c_2, c_4, g...  
grade    list      11     ['low', 'low', 'med', 'low', '...' ]  
high     list      11     [False, False, False, False, F...]  
i        int       1      10
```

```
In [6]:  
0 low  
1 low  
2 med  
3 low  
4 low  
5 low  
6 high  
7 high  
8 low  
9 med  
10 med  
Name: grade2, dtype: category  
Categories (3, object): [low < med < high]
```

```
In [7]:  
df.grade2=df.grade2.cat.rename_categories(new_cat_names)  
In [8]:
```

The categories being renamed above were ordered categories as they were created from numerical values, so it was more convenient to rename them by use of a list. For unordered categories or in scenarios where one or two categories are to be renamed it may be more convenient to use a dictionary where the key is the old name and the value is the new name. In this case we can demonstrate this by merely just changing the case of the category.

```
cat_name_dict={'low':'Low','med':'Med','high':'High'}
df.grade2.cat.rename_categories(cat_name_dict)
```

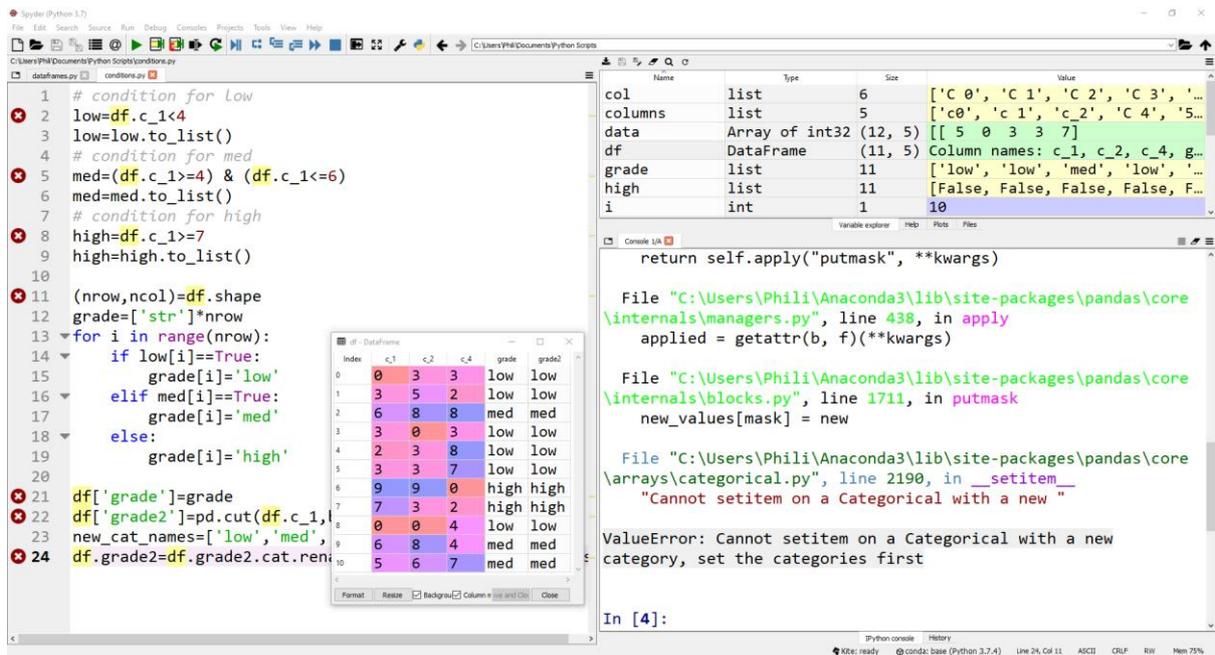


Adding Categories

We may wish to scrap all the indexes (rows) in the series (column) 'c_1' which correspond to 0 and update the corresponding indexes (rows) in the series (column) 'grade2' to a new category 'scrap'.

```
df.grade2[df.c_1==0]='scrap'
```

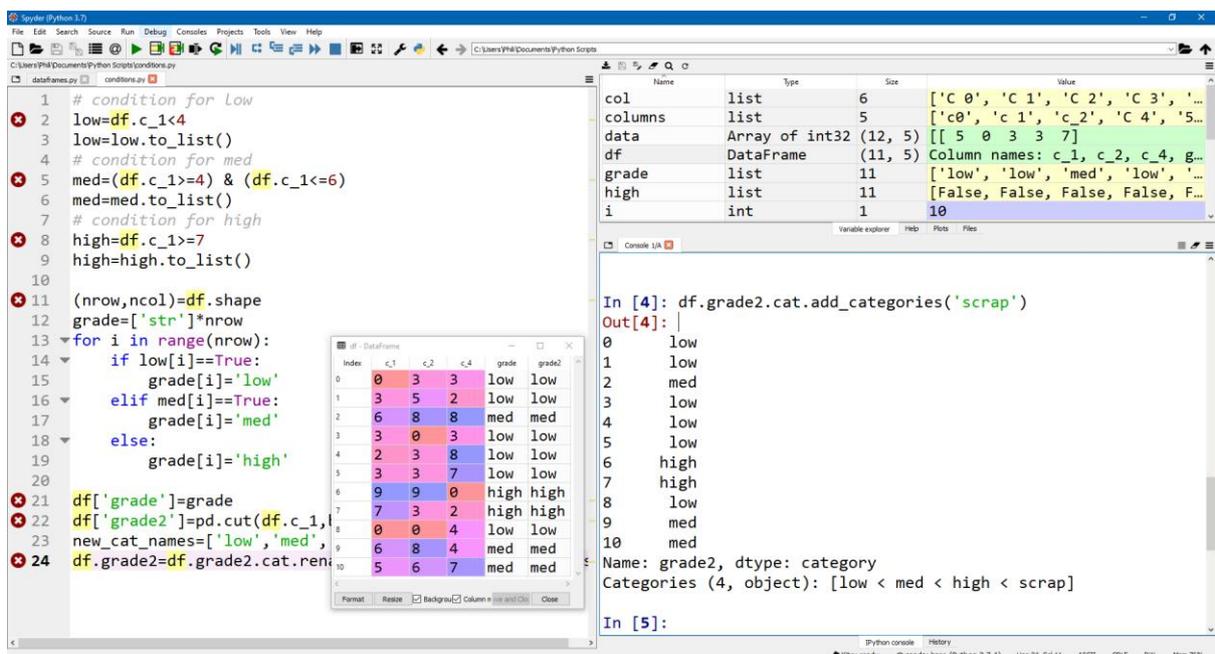
Attempting to do so directly will result in a `ValueError: Cannot setitem on a Categorical with a new category, set the categories first.`



This is because we must create the new category before we can reassign it. Things are done in this manner usually to prevent one from creating an accidental category e.g. 'Low' and 'low' by using capitalization or a punctuation error such as a typo or addition of a space. To create a new category we can select

```
df.grade2.cat.add_categories('scrap')
```

Once again, this function will return a pandas series (column) which in this case isn't assigned to a new name so just displays in the console. However we also note here the order of the categories, the new category 'scrap' is seen as a higher category of the series 'grade2' then the grade high which is clearly wrong.



We can reassign this to the original pandas series (column) 'scrap2'. Then we can reorder the categories. Finally, we can select the indexes where the panda series 'c_1' is 0 in the panda series

'grade2' and set them to the category 'scrap' which works in this case because the category 'scrap' exists.

```
df.grade2=df.grade2.cat.add_categories('scrap')
```

```
df.grade2=df.grade2.cat.reorder_categories(['scrap','low','med','high'])
```

```
df.grade2[df.c_1==0]='scrap'
```

The screenshot shows the Spyder Python IDE with a script named 'dataframes.py' open. The code defines a DataFrame with columns 'c_1', 'c_2', 'c_4', 'grade', and 'grade2'. It uses conditional logic to assign 'low', 'med', or 'high' grades based on 'c_1' values. The variable explorer on the right shows the state of the DataFrame after execution. The 'grade2' variable is a categorical type with categories ['low', 'med', 'high', 'scrap']. The console output shows the execution of the code and the resulting DataFrame structure.

Index	c_1	c_2	c_4	grade	grade2
0	0	3	3	low	scrap
1	3	5	2	low	low
2	6	8	8	med	med
3	3	0	3	low	low
4	2	3	8	low	low
5	3	3	7	low	low
6	9	9	0	high	high
7	7	3	2	high	high
8	0	0	4	low	scrap
9	6	8	4	med	med
10	5	6	7	med	med

Selecting by Category

Once we have our categories, we can use conditional logic to index by them. For example:

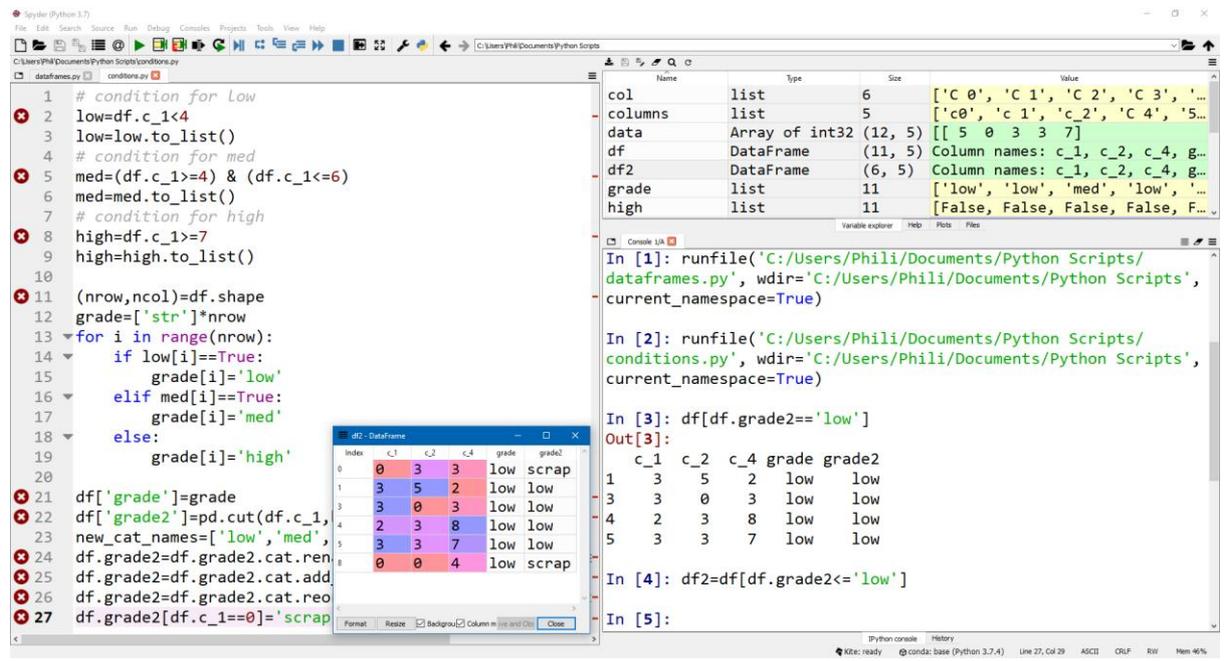
```
df[df.grade2=='low']
```

The screenshot shows the Spyder Python IDE with a script named 'dataframes.py' open. The code defines a DataFrame with columns 'c_1', 'c_2', 'c_4', 'grade', and 'grade2'. It uses conditional logic to assign 'low', 'med', or 'high' grades based on 'c_1' values. The variable explorer on the right shows the state of the DataFrame after execution. The console output shows the execution of the code and the resulting DataFrame structure.

c_1	c_2	c_4	grade	grade2
1	3	5	2	low
3	3	0	3	low
4	2	3	8	low
5	3	3	7	low

Once again, the output dataframe isn't assigned to a variable name and just displays in the console. Because the categories are ordered we can also index using less than `<`, less than or equal to `<=`, greater than or equal to `>=` and greater than `>`.

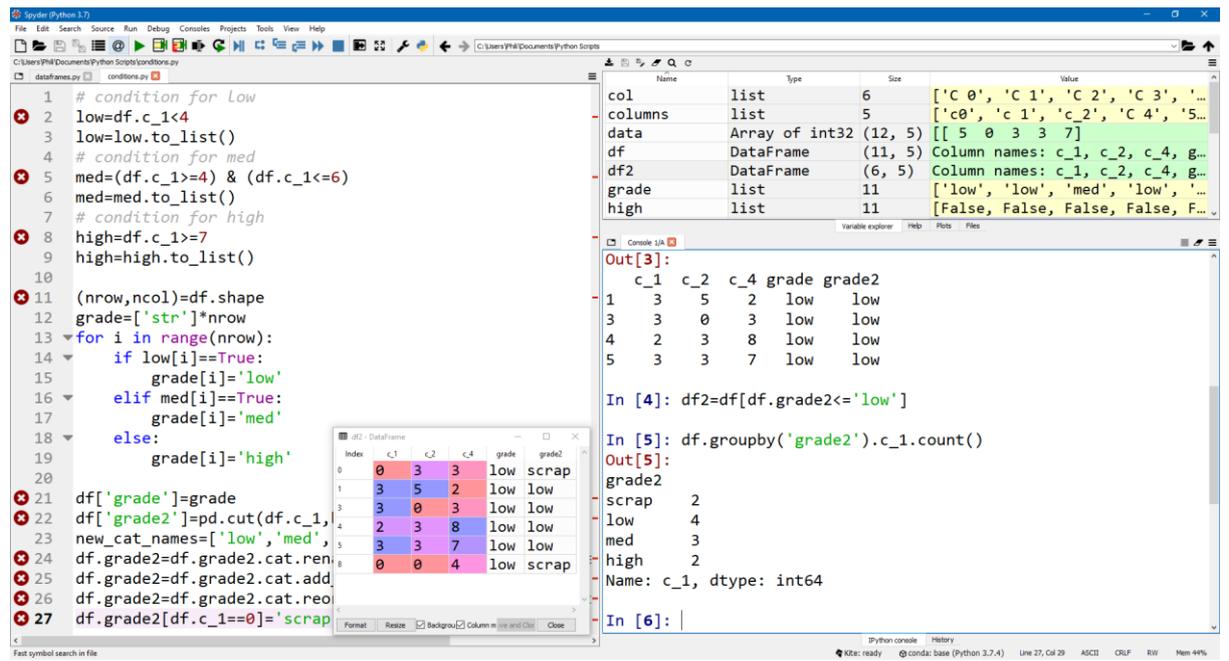
```
df2=df[df.grade2<='low']
```



Grouping by Category

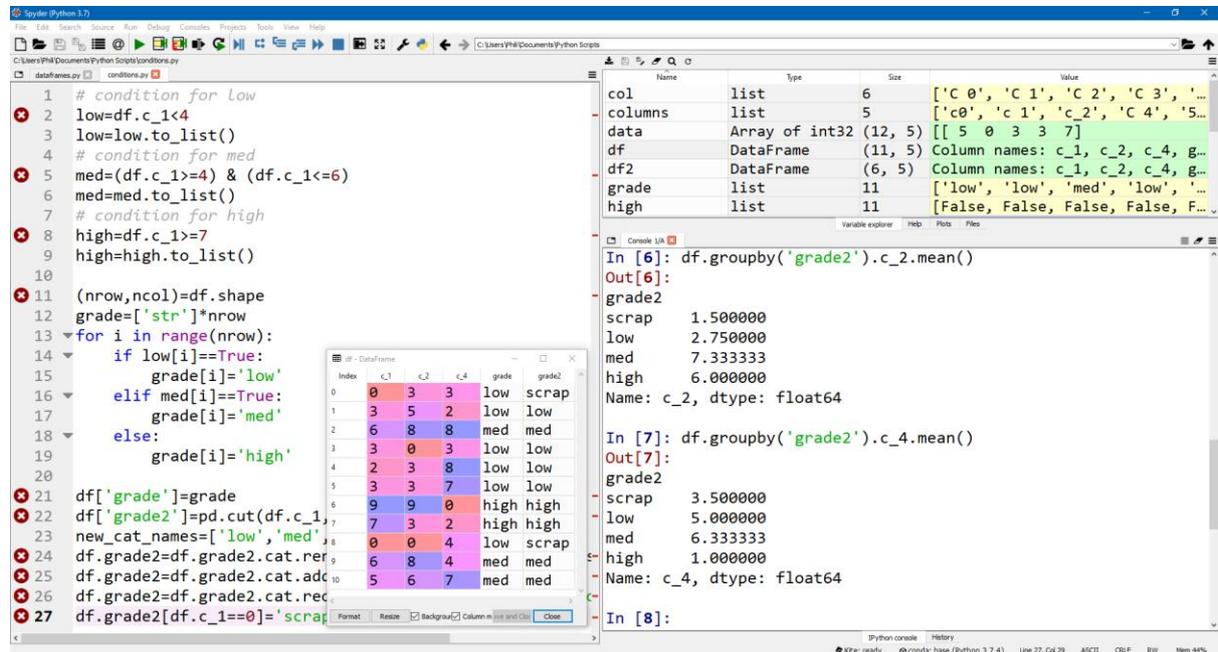
It is also possible to use the method `groupby` to group data by the categories in a selected column. A column can then be selected as an attribute to apply the `groupby` to and an additional method such as `count`, `mean`, `std` can be used to look at the properties of each group. For example, we can count the number of values in the series 'c_1' where 'grade2' corresponds to each category.

```
df.groupby('grade2').c_1.count()
```



We might also want to see if there is a relation between the categories in 'grade2' and the mean value of each category in the series 'c_2' and 'c_4':

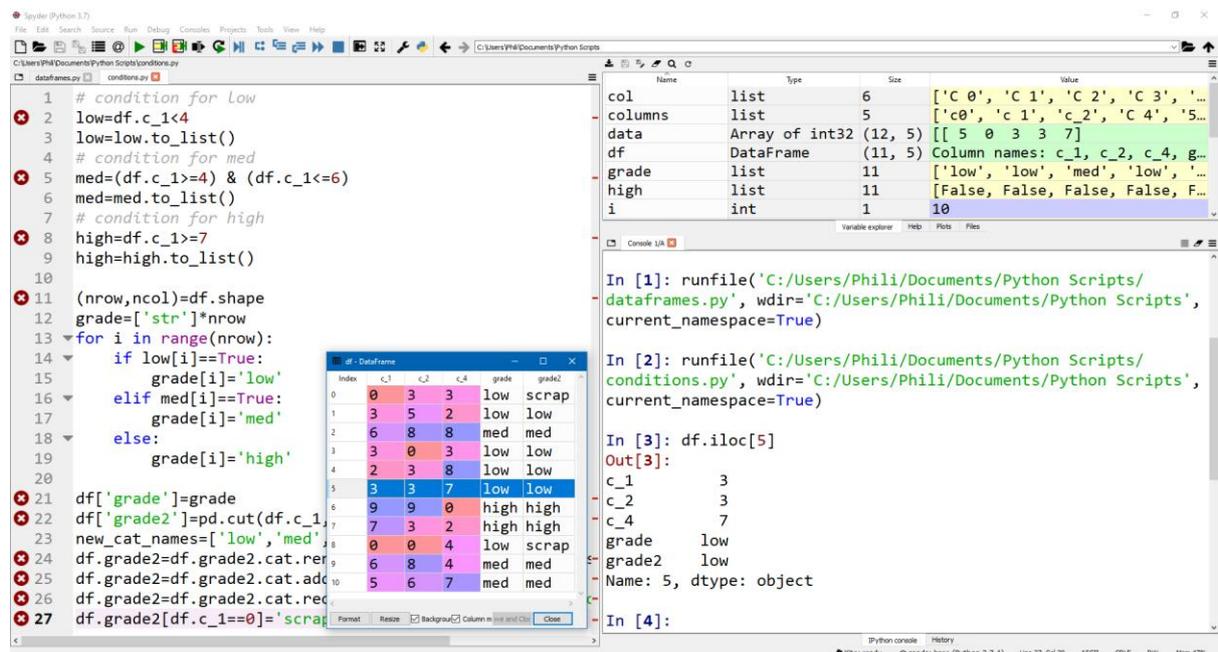
```
df.groupby('grade2').c_2.mean()
df.groupby('grade2').c_4.mean()
```



Selecting an Index

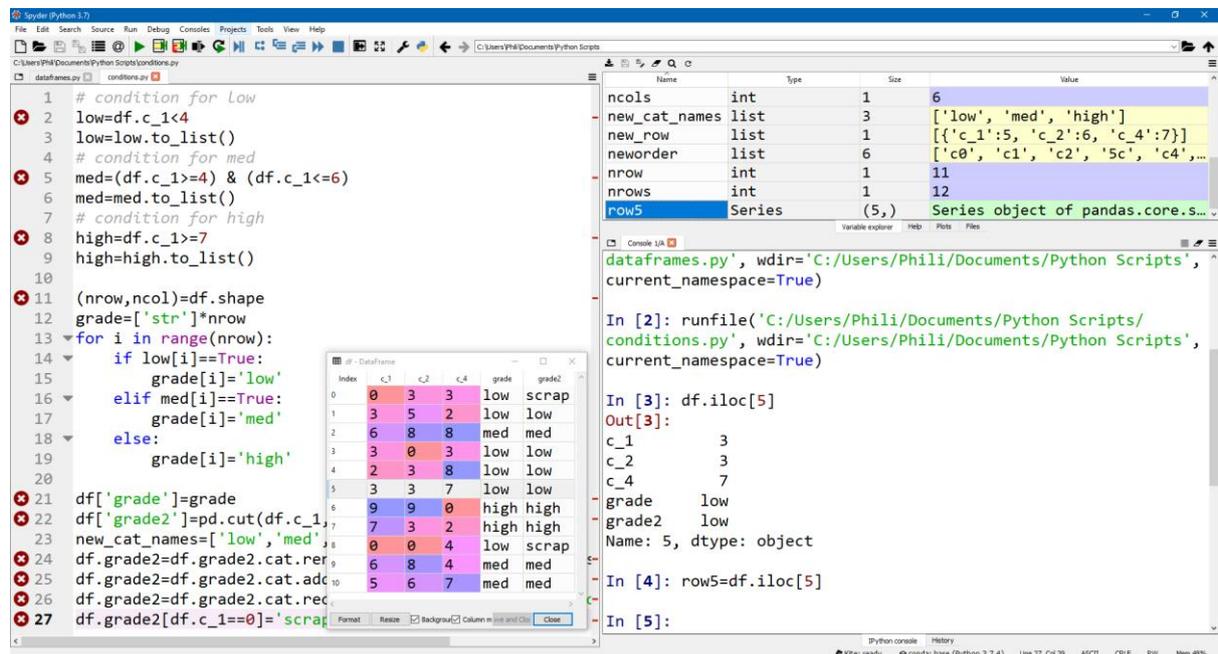
We have looked at selecting pandas series (columns). It is also possible to select a numerically labelled index (row) by use of the method integer location `iloc` which is followed by the desired index (row) enclosed in square brackets.

```
df.iloc[5]
```



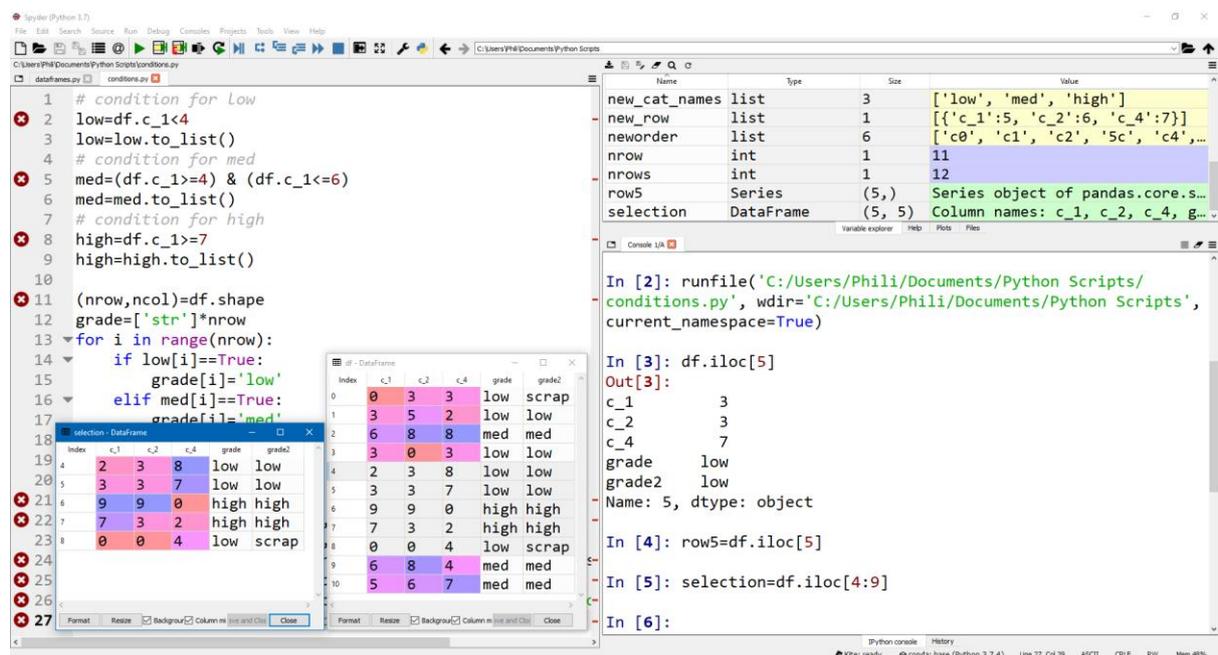
In this case we have not assigned the selection to an output variable so only see the results in the console. If instead we assign it to a variable name, the Type will be a pandas series.

```
row5=df.iloc[5]
```



Multiple indexes can be selected by indexing a slice. For example:

```
selection=df.iloc[4:9]
```



Which will give the output dataframe `selection`. Recall that zero-order indexing is applied so we go up to the end value but don't include it when slicing.

Selecting a Cell

Although it is technically possible to select a cell by using either:

```
df.c_4[2]
df.iloc[2].c_4
```

It is not recommended to use the expressions above for reassignment of the value of a cell. For instance, if we type in the following expression:

```
df.iloc[2].c_4=16
```

Although the desired changes appears to work we get a `SettingWithCopyWarning: A value is trying to be set on a copy of a slice from a DataFrame.`

The screenshot shows the Spyder Python IDE interface. On the left, a script named 'conditions.py' contains the following code:

```
1 # condition for low
2 low=df.c_1<4
3 low=low.to_list()
4 # condition for med
5 med=(df.c_1>=4) & (df.c_1<=6)
6 med=med.to_list()
7 # condition for high
8 high=df.c_1>=7
9 high=high.to_list()
10
11 (nrow,ncol)=df.shape
12 grade=['str']*nrow
13 for i in range(nrow):
14
15
16
17
18
19
20
21
22
23
24
25
26
27
```

Two DataFrames are displayed in the middle. The first DataFrame has columns 'c_1', 'c_2', 'c_4', 'grade', and 'grade2'. The second DataFrame is a copy of the first. The console on the right shows the following output:

```
In [3]: df.c_4[2]
Out[3]: 8

In [4]: df.iloc[2].c_4
Out[4]: 8

In [5]: df.iloc[2].c_4=2
C:\Users\Phili\Anaconda3\lib\site-packages\pandas\core\generic.py:5208: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  self[name] = value

In [6]:
```

Instead it is recommended to use the method `at` where both the index (row) and then series (columns) can be specified when enclosed in square brackets. For example we can reassign the value at the 5th index (row) and series `'c_1'` (column):

```
df.at[5, 'c_1']=8
```

The screenshot shows a Python IDE with a script named 'conditions.py'. The script filters a DataFrame based on three conditions: 'low' (c_1 < 4), 'med' (4 <= c_1 <= 6), and 'high' (c_1 > 7). It then iterates over the rows to assign a 'grade' based on these conditions. Two preview windows show the resulting DataFrame with columns 'c_1', 'c_2', 'c_4', 'grade', and 'grade2'. The console shows the following interactions:

```

Out[4]: 8
In [5]: df.iloc[2].c_4=2
C:\Users\Phili\Anaconda3\lib\site-packages\pandas\core\generic.py:5208: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame
See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  self[name] = value
In [6]: df.at[5,'c_1']
Out[6]: 3
In [7]: df.at[5,'c_1']=7
In [8]:

```

Renaming Indexes

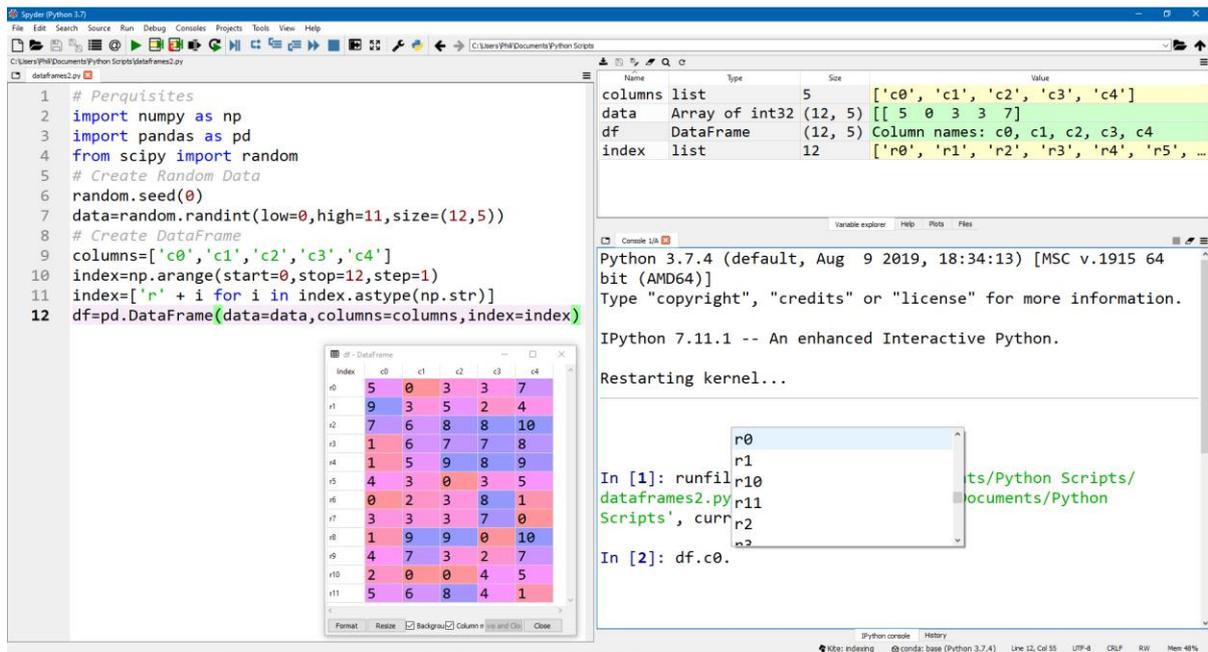
Earlier we seen that we can rename the pandas series (columns) and if we use names that follow the rules of variable names, we can access them using attributes. It is also possible to rename the indexes although it is more common to leave them unnamed and access them using their numeric values like we have already seen. In this case we will use a `for` loop to create `index` names.

```

1. # Perquisites
2. import numpy as np
3. import pandas as pd
4. from scipy import random
5. # Create Random Data
6. random.seed(0)
7. data=random.randint(low=0,high=11,size=(12,5))
8. # Create DataFrame
9. columns=['c0','c1','c2','c3','c4']
10. index=np.arange(start=0,stop=12,step=1)
11. index=['r' + i for i in index.astype(np.str)]
12. df=pd.DataFrame(data=data,columns=columns,index=index)

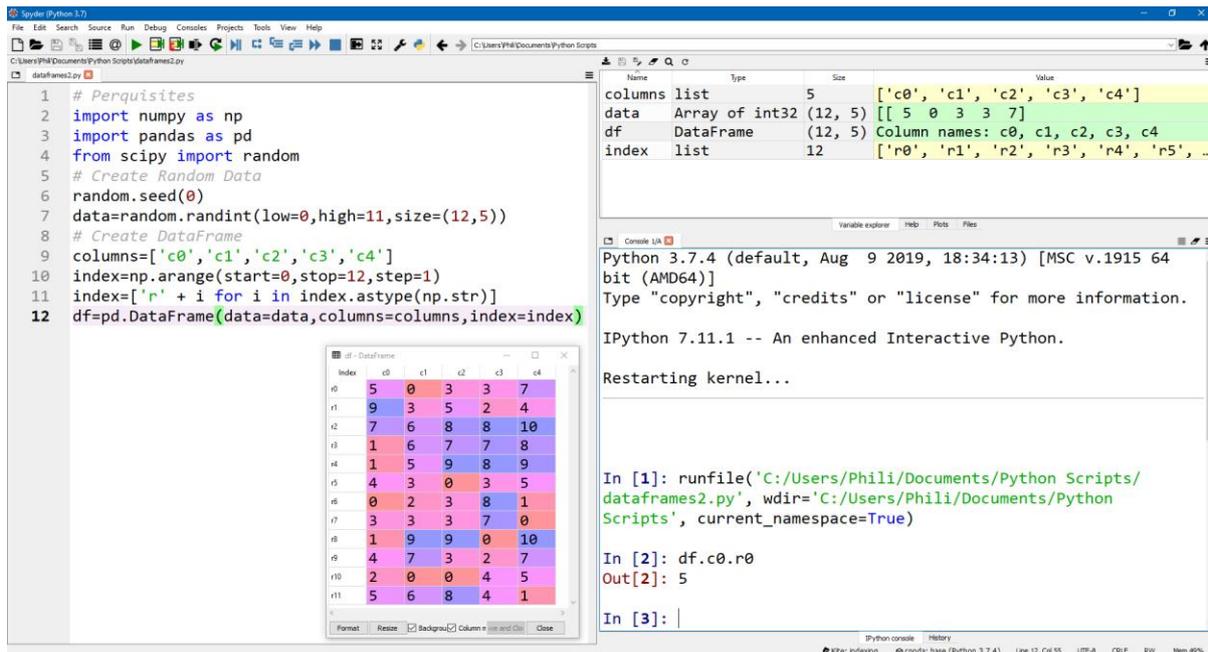
```

Once index (row) names are created, provided that they are named following the rules behind variable names, they too will show up as attributes to the dataframe or more specifically as attributes to each column in the dataframe. If we type in a dataframe followed by a dot `.` then a column name as an attribute and then type a another dot `.` followed by a tab `↵` we will see these indexes as attributes.



For example:

```
df.r0.c0
```



This also allows us to use both the attributes integer location `iloc` and location `loc` to select an index of the dataframe using its numerical value an index (row) name respectively.

```
df.iloc[4]
df.loc['r4']
```

Note for dataframes where the indexes are not named, the name of the index will be the numeric value so the methods `loc` and `iloc` will appear to be the same. However if the indexes are named `loc` will not accept the numeric value as a name as it has been changed.

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c1','c2','c3','c4']
10 index=np.arange(start=0,stop=12,step=1)
11 index=['r'+ i for i in index.astype(np.str)]
12 df=pd.DataFrame(data=data,columns=columns,index=index)

```

Name	Type	Size	Value
columns	list	5	['c0', 'c1', 'c2', 'c3', 'c4']
data	Array of int32	(12, 5)	[[5 0 3 3 7]
df	DataFrame	(12, 5)	Column names: c0, c1, c2, c3, c4
index	list	12	['r0', 'r1', 'r2', 'r3', 'r4', 'r5', ...]

```

In [3]: df.iloc[4]
Out[3]:
c0    1
c1    5
c2    9
c3    8
c4    9
Name: r4, dtype: int32

In [4]: df.loc['r4']
Out[4]:
c0    1
c1    5
c2    9
c3    8
c4    9
Name: r4, dtype: int32

```

The method `at` by will also use the name of the index. For example:

```
df.at['r2', 'c1']
```

```

1 # Perquisites
2 import numpy as np
3 import pandas as pd
4 from scipy import random
5 # Create Random Data
6 random.seed(0)
7 data=random.randint(low=0,high=11,size=(12,5))
8 # Create DataFrame
9 columns=['c0','c1','c2','c3','c4']
10 index=np.arange(start=0,stop=12,step=1)
11 index=['r'+ i for i in index.astype(np.str)]
12 df=pd.DataFrame(data=data,columns=columns,index=index)

```

Name	Type	Size	Value
columns	list	5	['c0', 'c1', 'c2', 'c3', 'c4']
data	Array of int32	(12, 5)	[[5 0 3 3 7]
df	DataFrame	(12, 5)	Column names: c0, c1, c2, c3, c4
index	list	12	['r0', 'r1', 'r2', 'r3', 'r4', 'r5', ...]

```

In [4]: df.loc['r4']
Out[4]:
c0    1
c1    5
c2    9
c3    8
c4    9
Name: r4, dtype: int32

In [5]: df.at['r2', 'c1']
Out[5]: 6

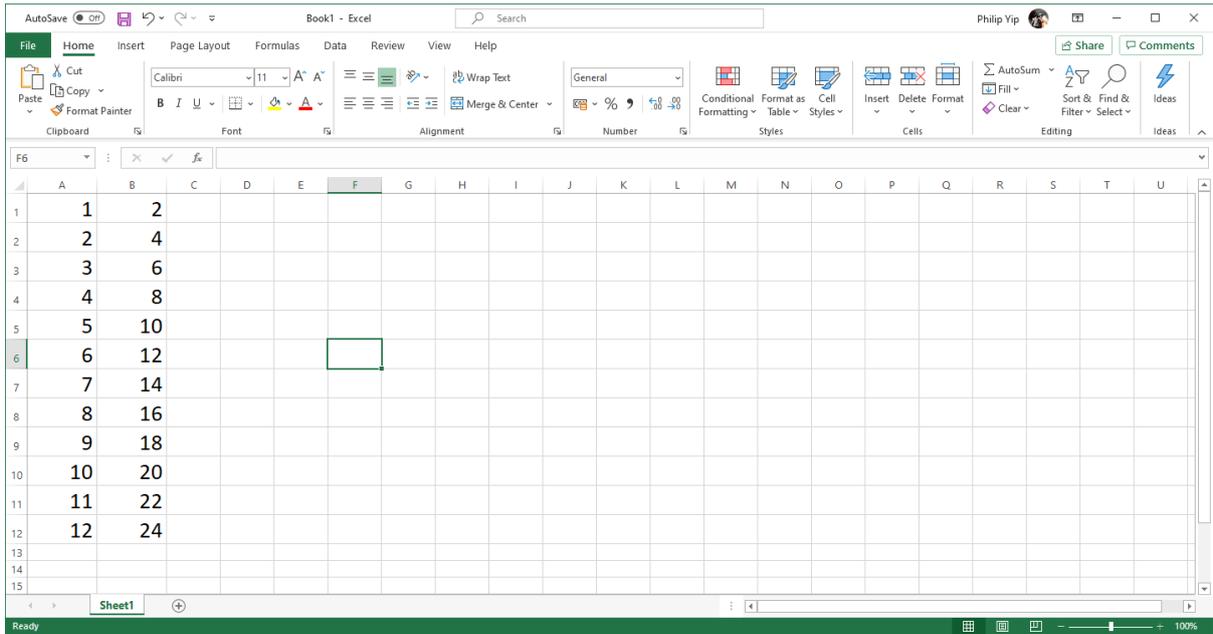
In [6]:

```

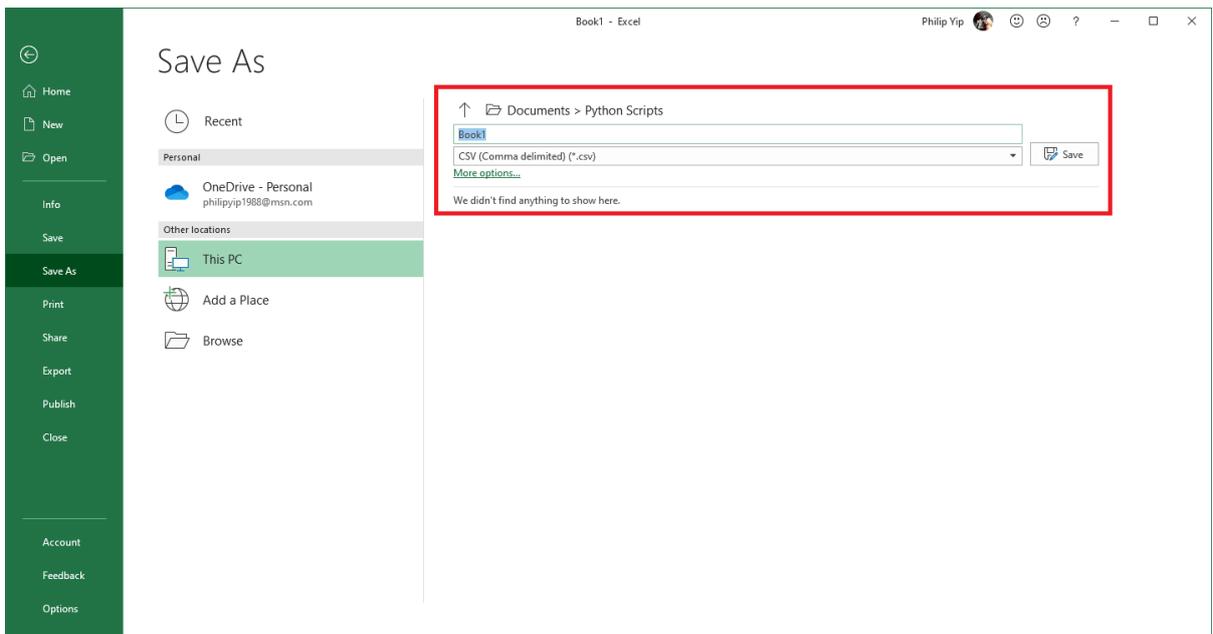
Comma Separated Values (CSV) and Tab Delimited (TXT) Files

So far, every single array we have examined has been manually input. In this section we will instead look at how we can open data from a file, manipulate the data and then save it to a new file or overwrite the original.

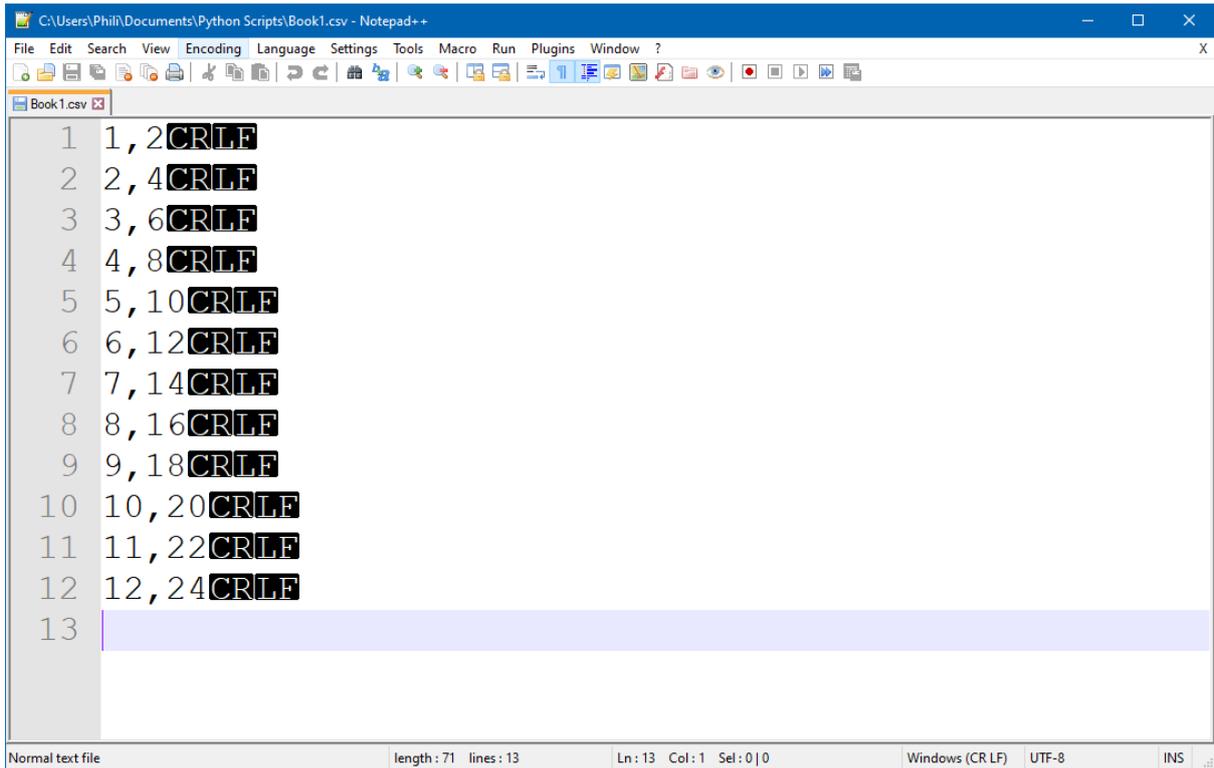
We will start off with a spreadsheet created in Microsoft Excel. In this file we have two columns "A" and "B" which contain numeric values.



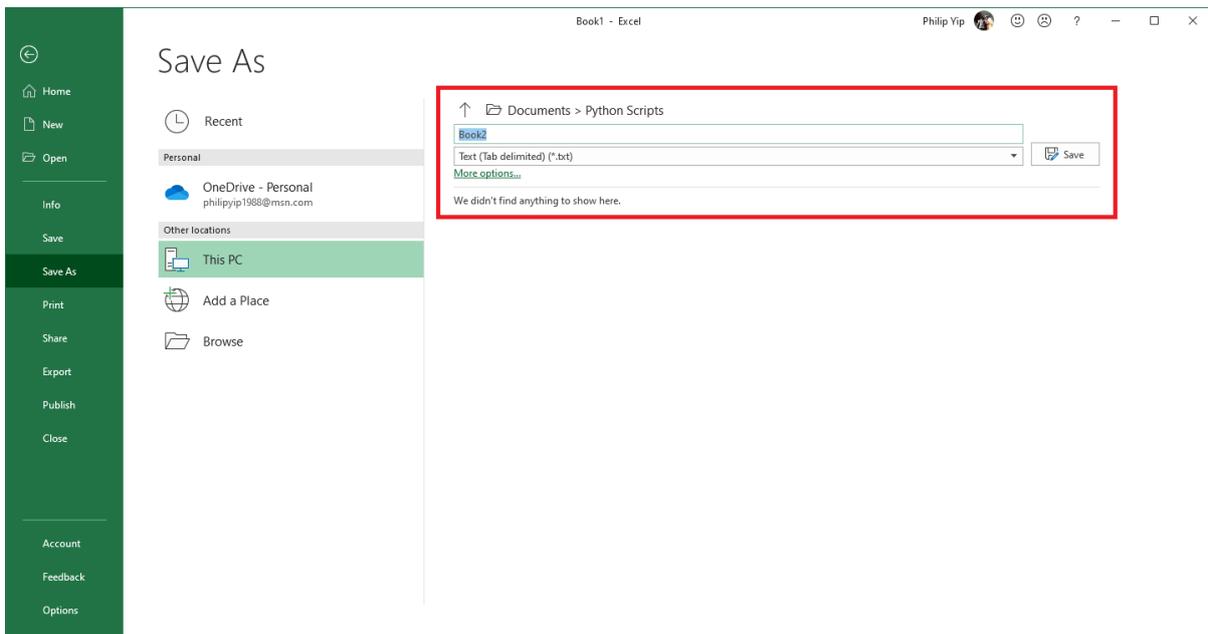
Let us save this to a comma separated values (.csv) file:



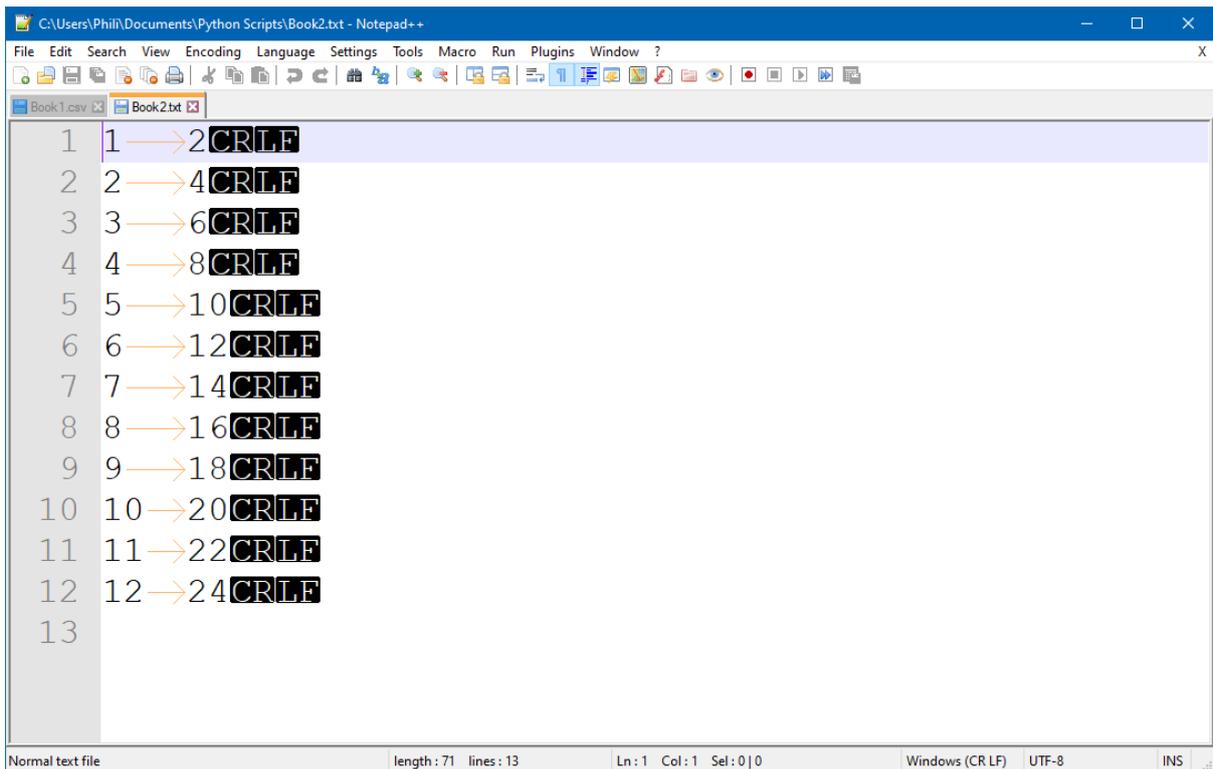
Now let's use an additional program Notepad++ <https://notepad-plus-plus.org/downloads/> to open this saved csv file. In this file we will select view→show symbol→show all characters to show what are usually hidden formatting/punctuation characters in order to help us better understand how data in the file is stored. Here we can see that a comma , acts a delimiter (separating each value in Column A from the corresponding value in Column B). At the end of each line is **CR** and **LF**. These stand for Carriage Return and Line Feed respectively. We need both as **CR** returns us to the left-hand side of the file and **LF** brings us onto a new line. The name of these punctuation marks originates from typewriters. To use a typewriter after each line, the carriage had to be returned to the left-hand side of the piece of paper (carriage return) and the piece of paper then had to be slid up by the space of a line (line feed).



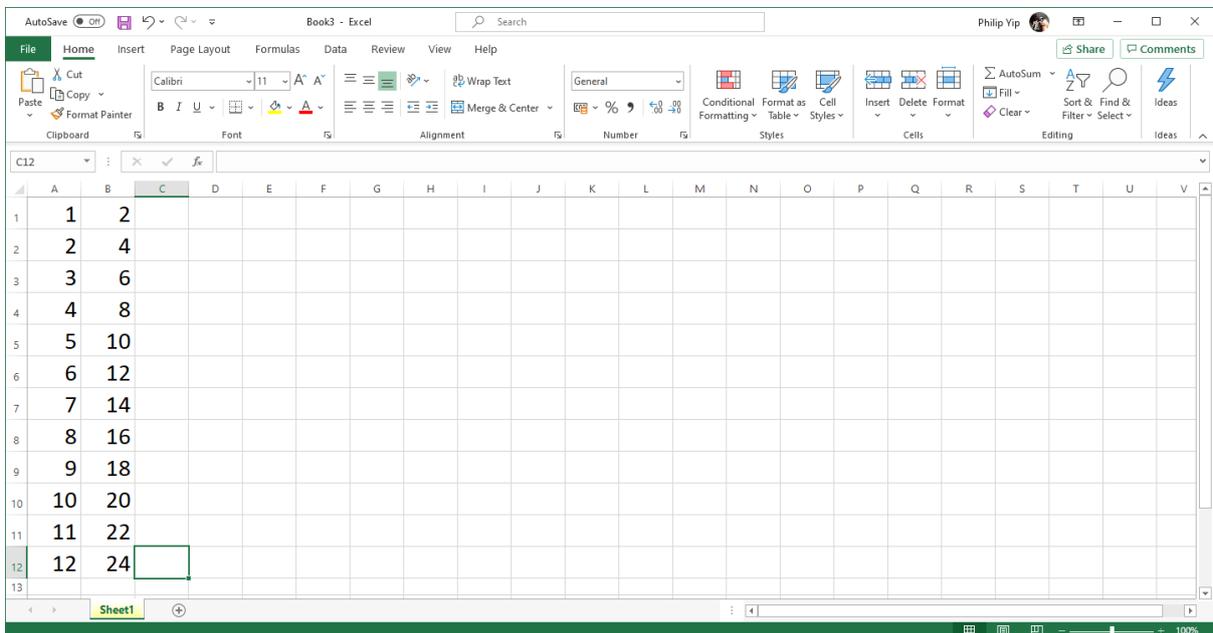
We can alternatively save the file as a **Text (Tab delimited) .txt** file.



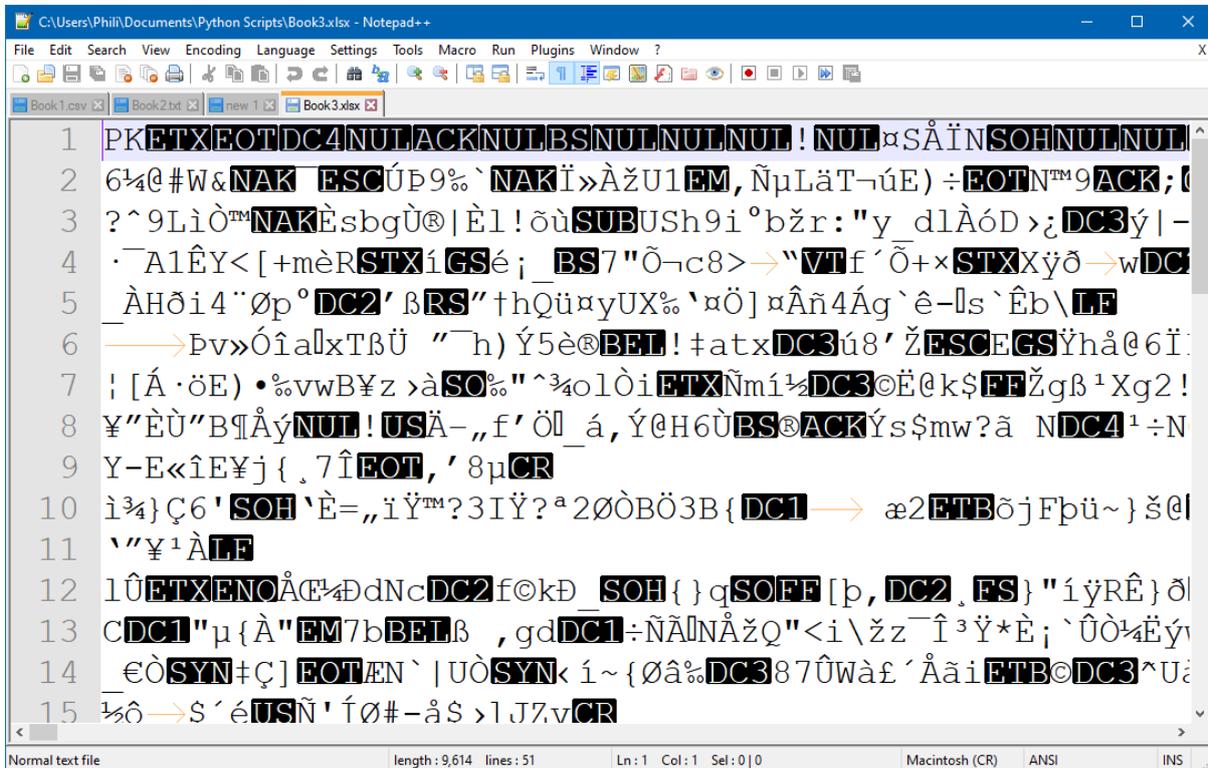
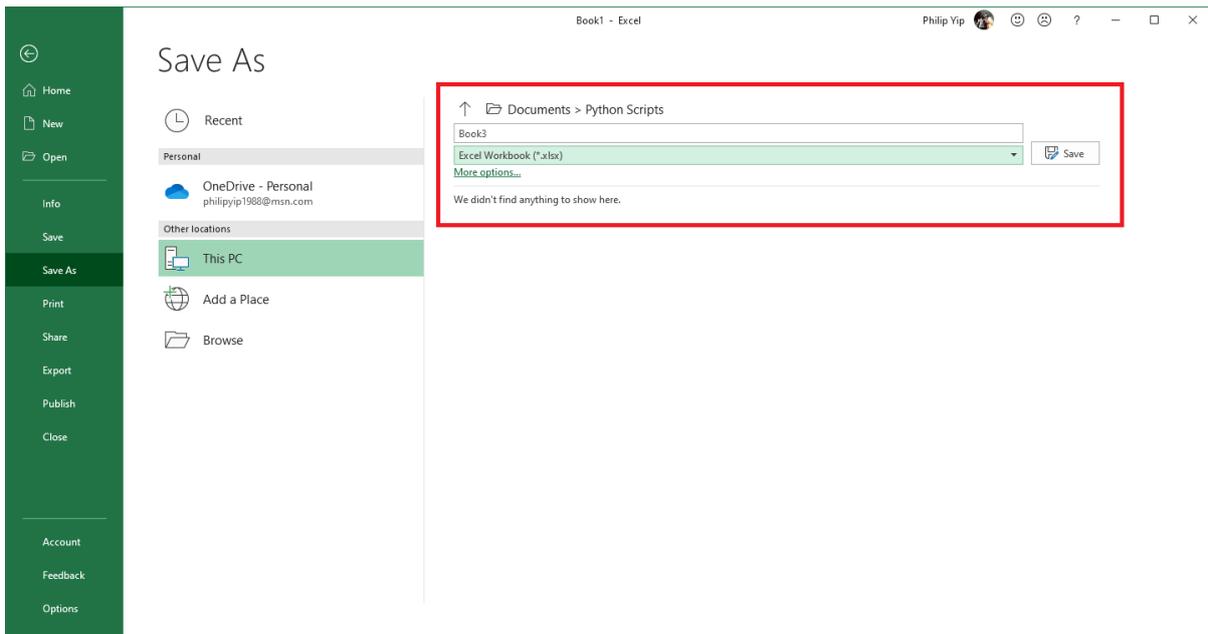
We can now open this up in Notepad++ and the only difference we will see between the two files is the delimiter. In the.csv file the comma , is used as the delimiter and in the text file, the \rightarrow is used as the delimiter.



Again, punctuation marks like the `\r` are usually hidden but we have enabled them in this case in order to better understand what is going on. We can also save the file as Microsoft's own .xlsx format. Note in this format that our data is contained in an Excel workbook that has Sheet1.



When we try and open this file up in Notepad++ we won't be able to intuitively understand what is going on and it will look like gibberish. This is because this file format has a larger number of features and each character in the workbook will have its size, font, font color etc. In this file format many more formatting characters are used and like `CR` and `LF` these are likewise shown using white characters on a white background.

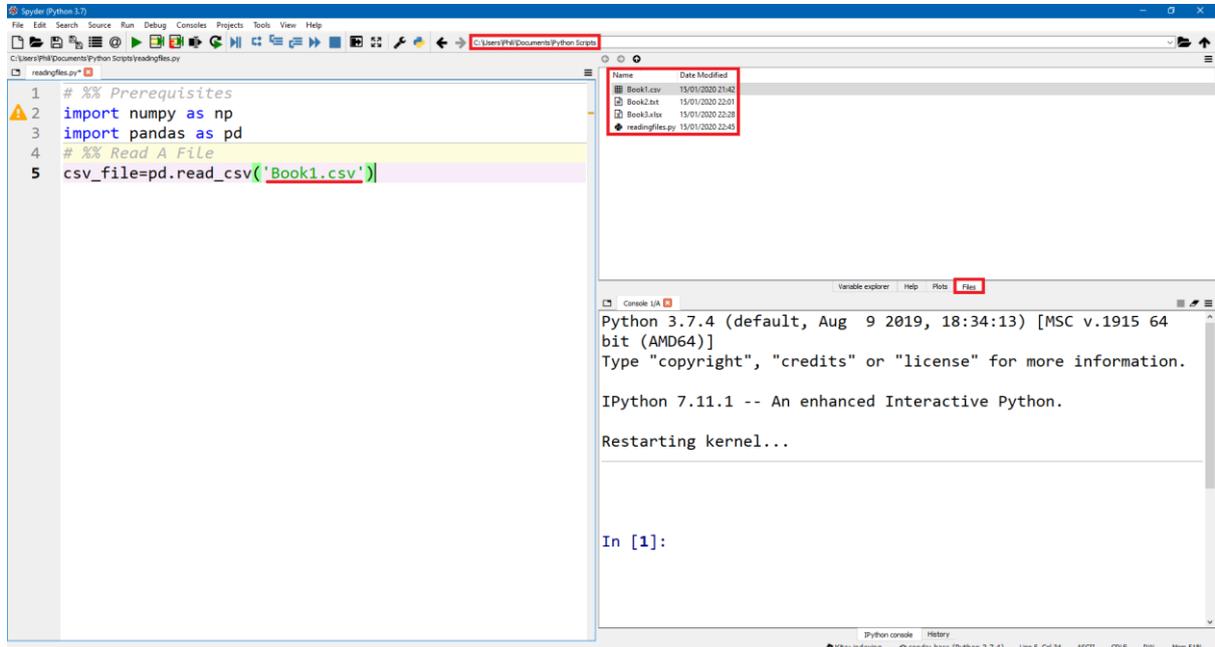


The pandas library has a function called `pd.read_csv()` which can be used to read in `csv` and `txt` files. And although we cannot understand the `xlsx` file by use of notepad++, the pandas library also contains a similar function `pd.read_excel()` with more under the hood which can be used with the `xlsx` file format.

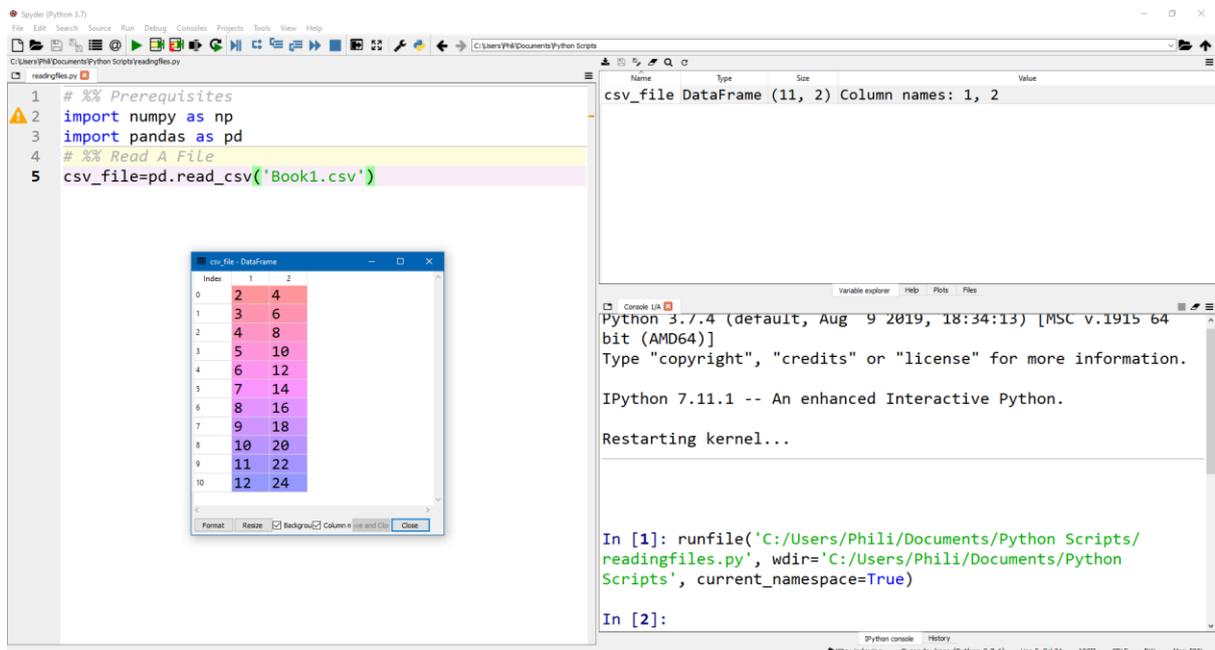
Reading and Writing to CSV, TXT or XLSX File

Let's first use `pd.read_csv()` to create a dataframe from an csv file when the csv file is in the same folder as the python script. We can verify that are in the same folder by use of the files tab. Because they are in the same folder, the full file path of the file doesn't need to be specified and we can instead use the file name and extension.

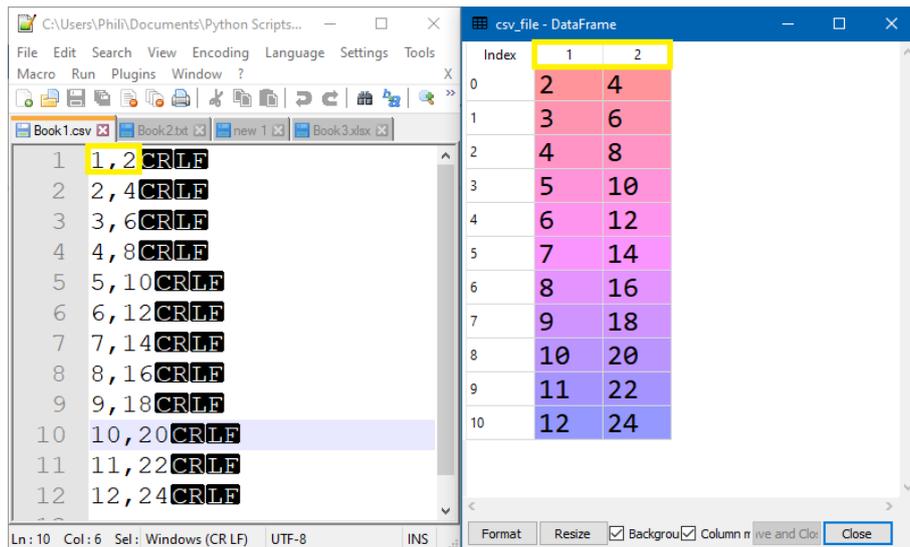
1. # %% Prerequisites
2. `import numpy as np`
3. `import pandas as pd`
4. # %% Read A File
5. `csv_file=pd.read_csv('Book1.csv')`



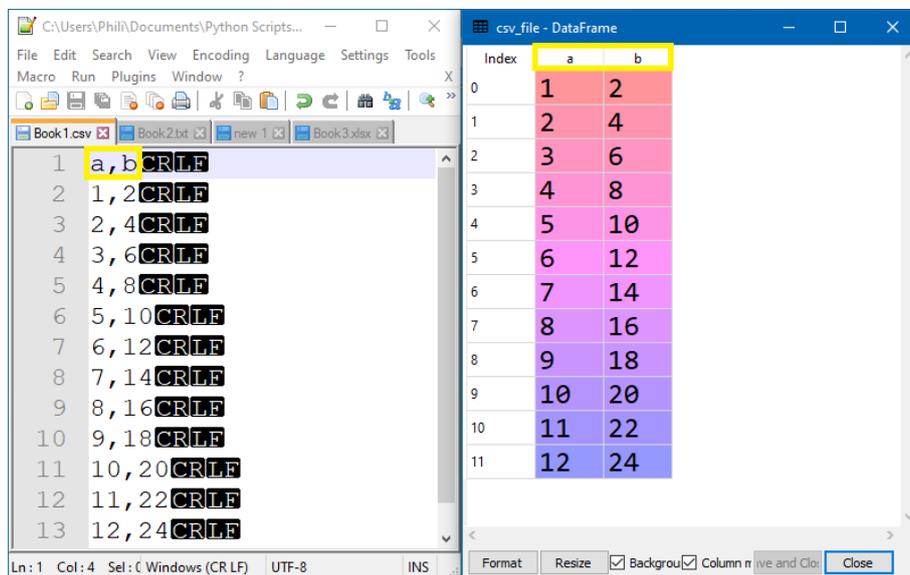
Launching the script will give us the new dataframe `csv_file`.



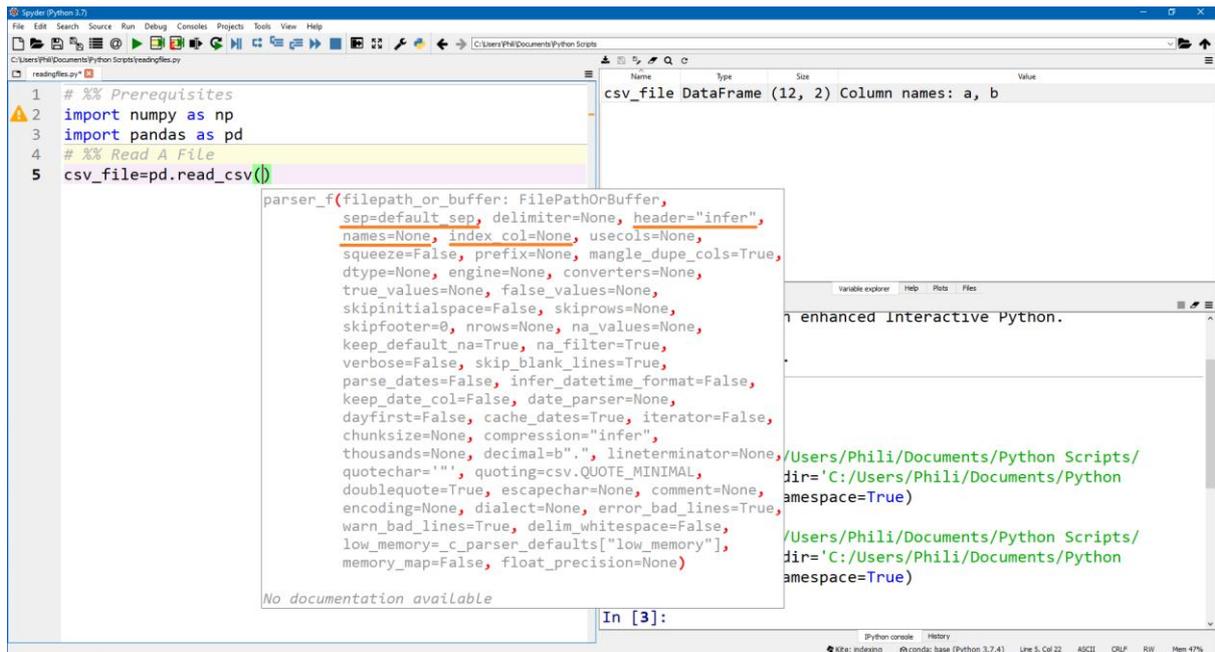
If we compare the `csv_file` dataframe to the original csv file opened in notepad++ we'll see that notepad++ uses 1st order indexing and python uses 0th order indexing. That aside the top row, is missing in the `csv_file` because the values have been taken as the column names.



If we modify the csv file to have column names and reimport, we can see that these are instead used.



The function `read_csv` has a position argument, the file to be read in and several keyword input arguments. Recall that we can have a look at the input arguments for the function by typing in the function without any input arguments and hovering over it. Looking at the first 5 keyword input arguments. We can specify a separator however the function uses a comma `,` by default. The delimiter is just an alias for the separator. The next keyword argument `header='infer'` will mean the first non-blank line is taken as the column names by default. It can be overridden by changing its value to `None` and assigning a list to the next keyword value `names` which has a default value of `None`. An index column can also be specified by using `index_col` which by default is set to `None`.

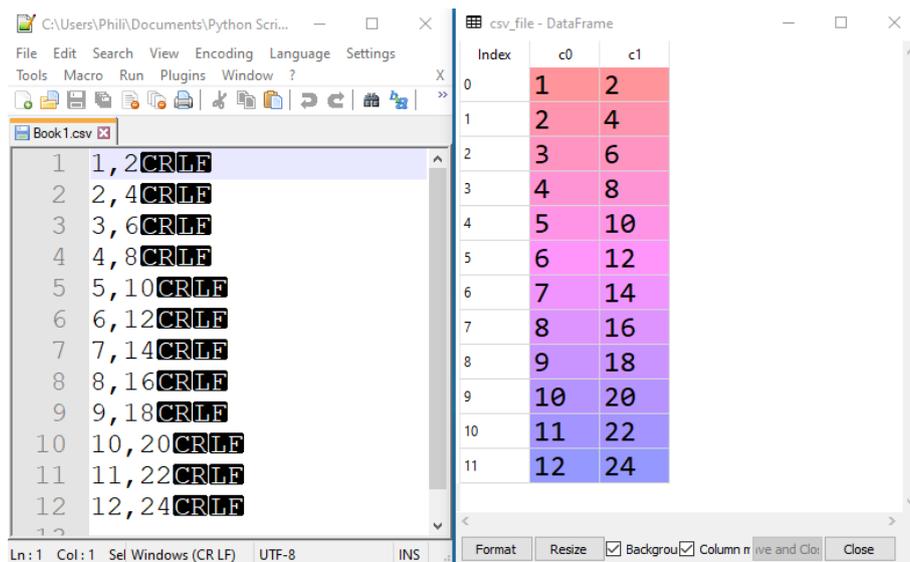


Removing the headings from the csv file we can import the file as a dataframe by selecting `header=None` and assigning `names` to a vector of the desired names.

```

1. # %% Prerequisites
2. import numpy as np
3. import pandas as pd
4. # %% Read A File
5. csv_file=pd.read_csv('Book1.csv',
6.                       header=None,
7.                       names=['c0', 'c1'],
8.                       index_col=None)

```



We may use the keyword input argument `skiprows` for a file with non-data comment lines, for example in the file below three lines are non-data comment lines and there is no header.

```

1. # %% Prerequisites

```

```

2. import numpy as np
3. import pandas as pd
4. # %% Read A File
5. csv_file=pd.read_csv('Book1.csv',
6.                       skiprows=3,
7.                       header=None,
8.                       names=['c0','c1'],
9.                       index_col=None)

```

The screenshot shows a Python IDE window with a CSV file named 'Book1.csv' open. The file content is as follows:

```

1 This is a csv file
2 This contains
3 data
4 1,2
5 2,4
6 3,6
7 4,8
8 5,10
9 6,12
10 7,14
11 8,16
12 9,18
13 10,20
14 11,22
15 12,24

```

On the right, a 'DataFrame' window displays the data loaded from the CSV file:

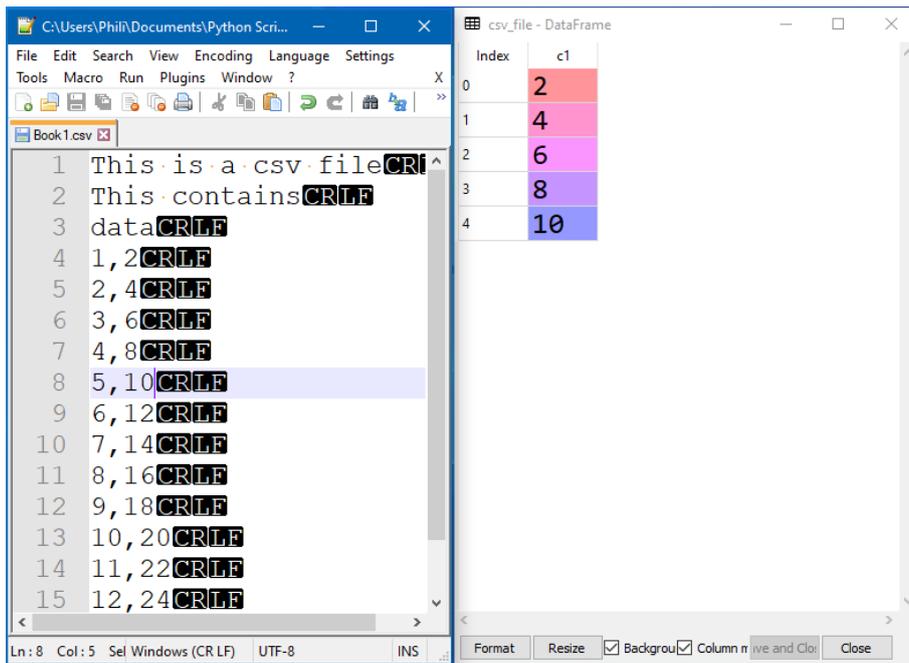
Index	c0	c1
0	1	2
1	2	4
2	3	6
3	4	8
4	5	10
5	6	12
6	7	14
7	8	16
8	9	18
9	10	20
10	11	22
11	12	24

It is also possible to read in only a selection of rows which can sometimes save time when previewing an excel file with a large number of rows for example using the keyword input argument `nrows`. Columns may also be selected by using the keyword input argument `usecols` and selecting the rows to use as a vector of the original row names. We can use the following if we only want to import the data from the first 5 data rows in column 1 of the csv file.

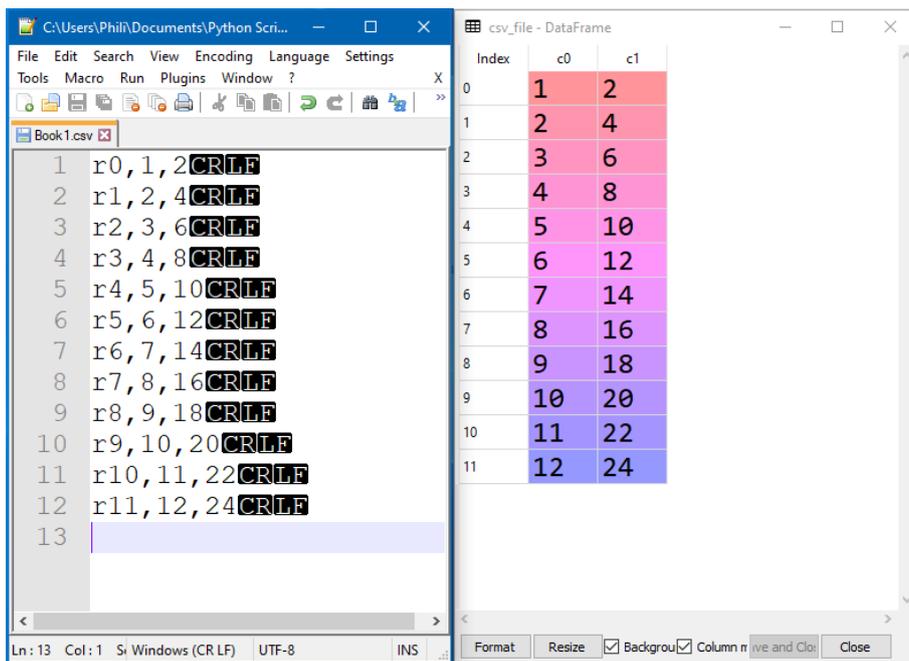
```

1. # %% Prerequisites
2. import numpy as np
3. import pandas as pd
4. # %% Read A File
5. csv_file=pd.read_csv('Book1.csv',
6.                       skiprows=3,
7.                       header=None,
8.                       names=['c1'],
9.                       index_col=None,
10.                      nrows=5,
11.                      usecols=[1])

```



Supposing we have the following `csv` file. Here the 0th column of the raw data contains the index names. We do not have the column names specified in the file and instead we want to create them when reading the file and assigning it to a dataframe. We can therefore use:

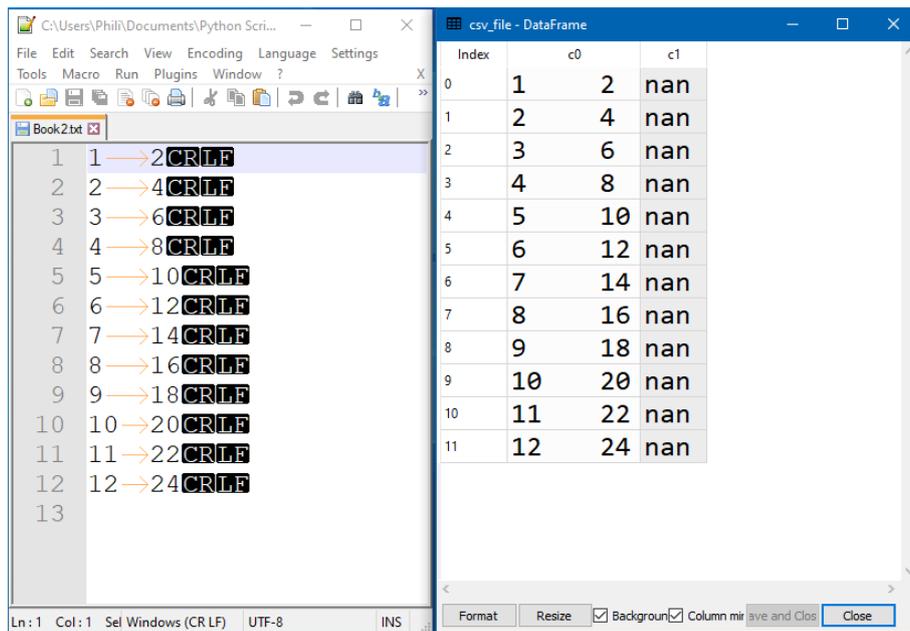


```

1. # %% Prerequisites
2. import numpy as np
3. import pandas as pd
4. # %% Read A File
5. csv_file=pd.read_csv('Book1.csv',
6.                       header=None,
7.                       names=['c1', 'c2'],
8.                       index_col=0)

```

If we now return to Book2.txt and try to read it specifying column names and no index names.



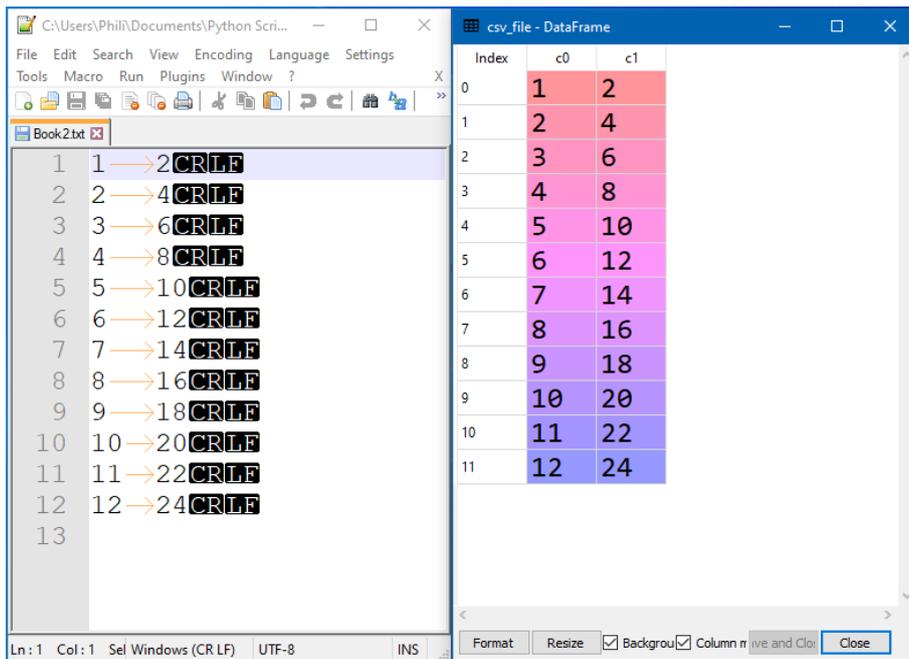
Index	c0	c1
0	1	2 nan
1	2	4 nan
2	3	6 nan
3	4	8 nan
4	5	10 nan
5	6	12 nan
6	7	14 nan
7	8	16 nan
8	9	18 nan
9	10	20 nan
10	11	22 nan
11	12	24 nan

```
1. # %% Prerequisites
2. import numpy as np
3. import pandas as pd
4. # %% Read A File
5. txt_file=pd.read_csv('Book2.txt',
6.                       header=None,
7.                       names=['C1', 'C2'],
8.                       index_col=None)
```

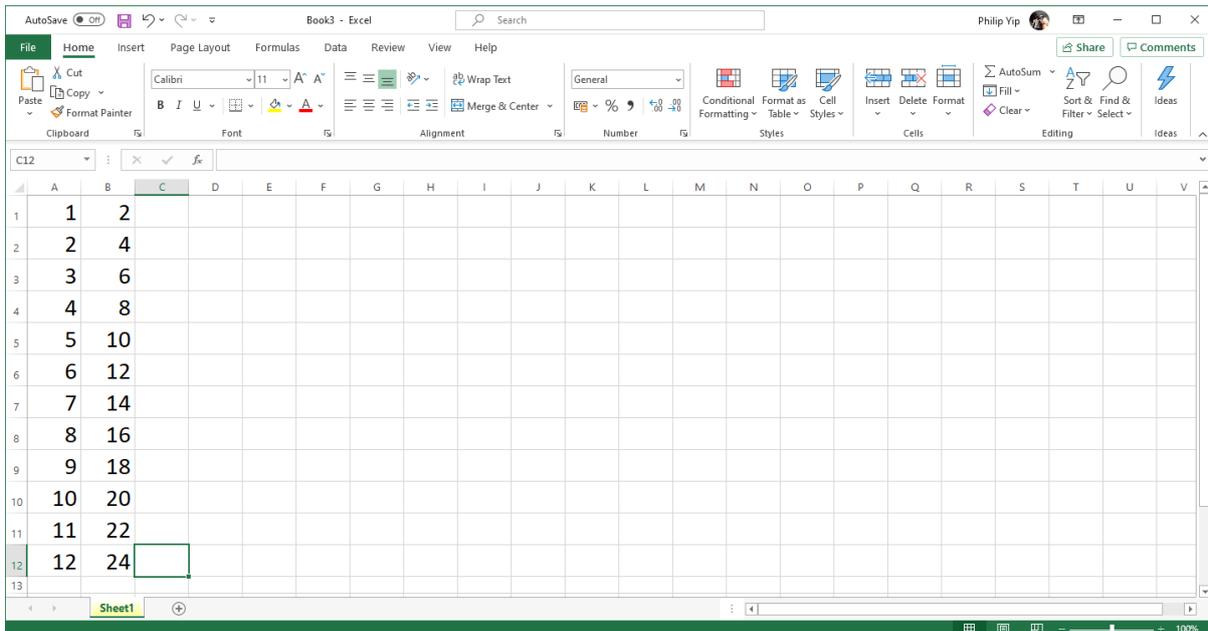
We get the situation above. This is because the separator (delimiter) is taken to be a comma , when it is a tab `\t`, and because we have specified 2 columns via the use of 2 column names, the second column is full of `nan` not a number as no numeric values were found. This can of course be amended by specifying the separator or delimiter as a tab `\t` to do this we use `\t`. We can use either `sep` or its alias `delimiter` to do this. So far, we have assumed the file to be read is in the same folder. If it is in a different folder or if we want to manually specify the full path, we can type it in. Recall that we can use a relative string if we copy and paste the filepath from Windows explorer:

```
1. # %% Prerequisites
2. import numpy as np
3. import pandas as pd
4. # %% Read A File
5. txt_file=pd.read_csv(r'C:\Users\Phili\Documents\Python
6. Scripts\Book2.txt',
7.                       sep='\t',
8.                       header=None,
9.                       names=['C1', 'C2'],
10.                      index_col=None)
```

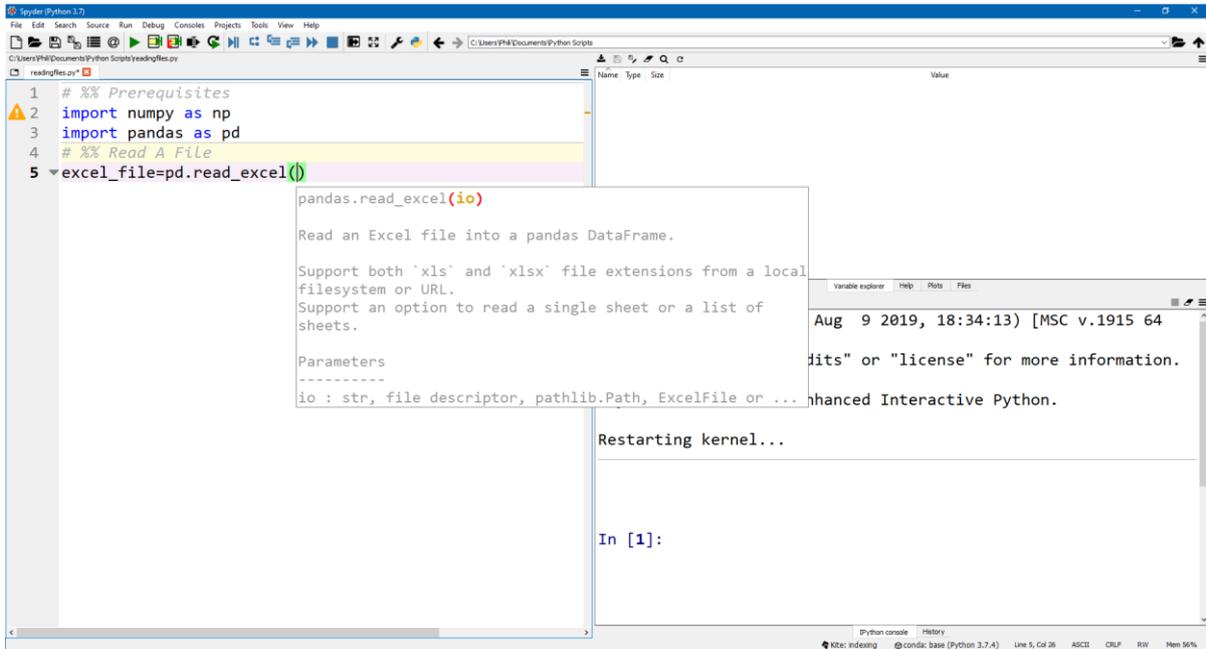
This reads the file in correctly:



Now let's use `pd.read_excel()` to read an Excel xlsx file and save it as a dataframe called `excel_file`. When reading in an Excel File we can specify both the name of the xlsx file itself and the sheet within the xlsx file. In this case `'Book3.xlsx'` and `'Sheet1'` respectively.



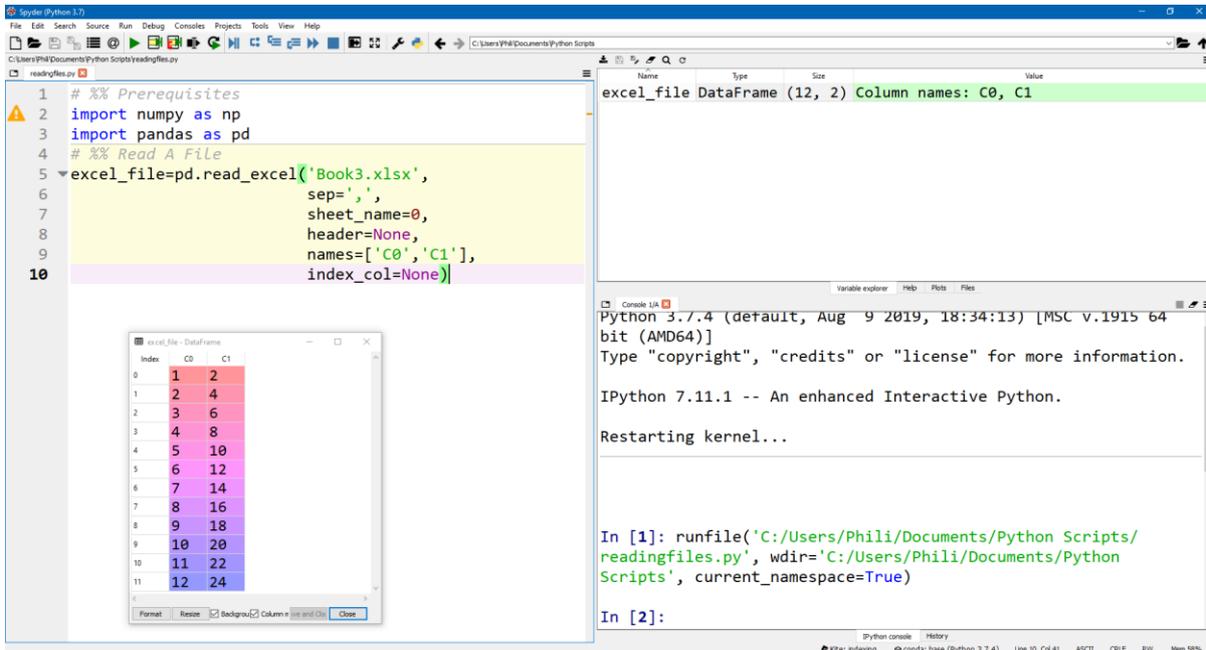
By default, the keyword argument `sheet_name` is assigned to a numerical value of `0` meaning it will read in the 0th Sheet created in the excel workbook. This can be changed to another sheet using a numerical value corresponding to the order of the sheets in the Excel workbook or alternatively the sheet name can be input as a string. The information about the keyword arguments for `read_excel()` isn't as in depth as `read_csv()` however it shares most of the keyword arguments and has the additional keyword argument `sheet_name`.



```

1. # %% Prerequisites
2. import numpy as np
3. import pandas as pd
4. # %% Read A File
5. excel_file=pd.read_excel('Book3.xlsx',
6.                           sheet_name=0,
7.                           header=None,
8.                           names=['C0', 'C1'],
9.                           index_col=None)

```



We have seen before how to index, individual columns and individual indexes. In addition we have also seen how to use the `.index` and `.columns` attributes to access the index and column names.

```
excel_file.index
excel_file.columns
```

The screenshot shows a Jupyter Notebook interface. The code editor on the left contains the following code:

```
1 # %% Prerequisites
2 import numpy as np
3 import pandas as pd
4 # %% Read A File
5 excel_file=pd.read_excel('Book3.xlsx',
6                           sheet_name=0,
7                           header=None,
8                           names=['C0', 'C1'],
9                           index_col=None)
10
```

The variable explorer on the right shows the variable `excel_file` as a `DataFrame (12, 2)` with `Column names: C0, C1`.

The console output shows the following interactions:

```
Restarting kernel...

In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/
readingfiles.py', wdir='C:/Users/Phili/Documents/Python
Scripts', current_namespace=True)

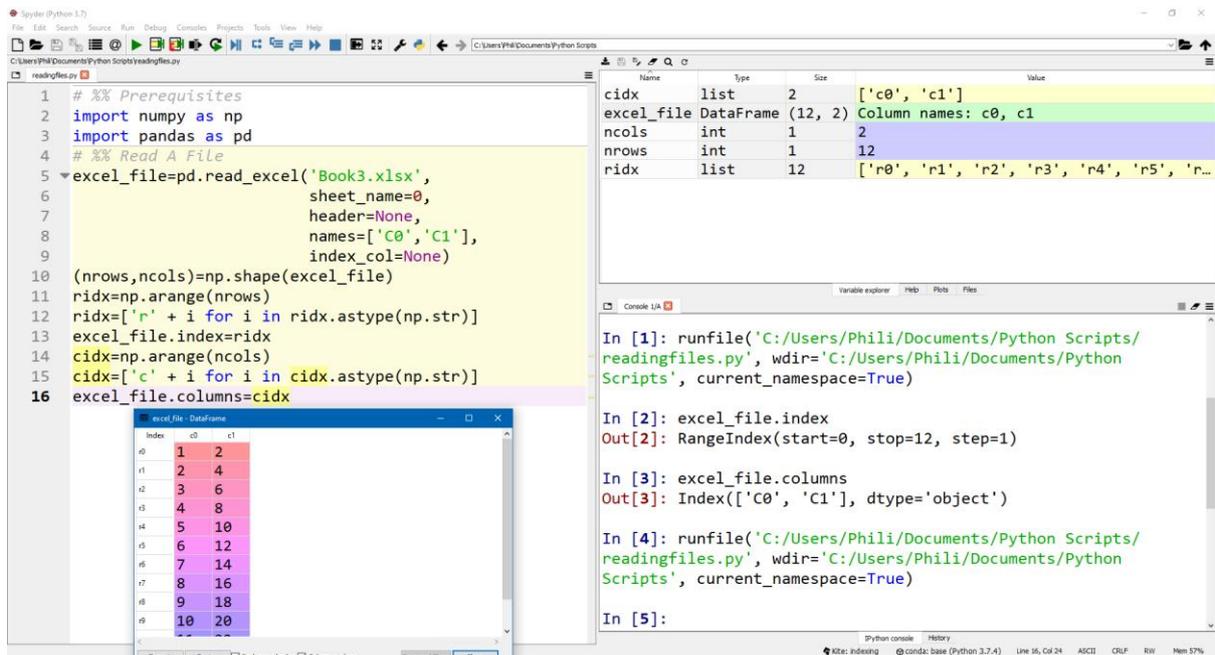
In [2]: excel_file.index
Out[2]: RangeIndex(start=0, stop=12, step=1)

In [3]: excel_file.columns
Out[3]: Index(['C0', 'C1'], dtype='object')

In [4]:
```

We can rename the columns and indexes using the `shape` of the imported dataframe and associated `for` loops.

```
1. # %% Prerequisites
2. import numpy as np
3. import pandas as pd
4. # %% Read A File
5. excel_file=pd.read_excel('Book3.xlsx',
6.                           sheet_name=0,
7.                           header=None,
8.                           names=['C0', 'C1'],
9.                           index_col=None)
10. (nrows,ncols)=np.shape(excel_file)
11. ridx=np.arange(nrows)
12. ridx=['r' + i for i in ridx.astype(np.str)]
13. excel_file.index=ridx
14. cidx=np.arange(ncols)
15. cidx=['c' + i for i in cidx.astype(np.str)]
16. excel_file.columns=cidx
```



Now that we have a modified dataframe, we might want to save it back to a file. To do this we can type in the dataframes name followed by its method `.to_excel()`. We need to specify the file name as an input argument. We can add a `sheet_name` as a keyword argument. This has a default value of `'Sheet1'`. Note if the excel file is already present, it will be overwritten without warning.

```

1. # %% Prerequisites
2. import numpy as np
3. import pandas as pd
4. # %% Read A File
5. excel_file=pd.read_excel('Book3.xlsx',
6.                             sheet_name=0,
7.                             header=None,
8.                             names=['c0', 'c1'],
9.                             index_col=None)
10. (nrows,ncols)=np.shape(excel_file)
11. ridx=np.arange(nrows)
12. ridx=['r' + i for i in ridx.astype(np.str)]
13. excel_file.index=ridx
14. cidx=np.arange(ncols)
15. cidx=['c' + i for i in cidx.astype(np.str)]
16. excel_file.columns=cidx
17. excel_file.to_excel('Book3_Updated.xlsx',
18.                       sheet_name='Updated_Sheet')

```

Once again if a full file path is not specified, the generated excel file will be in the same folder as the script. A full path or relative path can be input to save the file elsewhere.

The screenshot displays the Spyder Python IDE interface. The top-left pane shows a Python script named `readingfiles.py` with the following code:

```

1 # %% Prerequisites
2 import numpy as np
3 import pandas as pd
4 # %% Read A File
5 excel_file=pd.read_excel('Book3.xlsx',
6                           sheet_name=0,
7                           header=None,
8                           names=['C0','C1'],
9                           index_col=None)
10 (nrows,ncols)=np.shape(excel_file)
11 ridx=np.arange(nrows)
12 ridx=['r' + i for i in ridx.astype(np.str)]
13 excel_file.index=ridx
14 cidx=np.arange(ncols)
15 cidx=['c' + i for i in cidx.astype(np.str)]
16 excel_file.columns=cidx
17 excel_file.to_excel('Book3_Updated.xlsx',
18                     sheet_name='Updated_Sheet')

```

The top-right pane shows a file explorer window with the following files:

Name	Date Modified
Book1.csv	15/01/2020 23:26
Book2.txt	15/01/2020 22:01
Book3_Updated.xlsx	16/01/2020 23:10
Book3.xlsx	15/01/2020 22:28
readingfiles.py	16/01/2020 23:10

The bottom-right pane shows the Python console with the following output:

```

In [2]: excel_file.index
Out[2]: RangeIndex(start=0, stop=12, step=1)

In [3]: excel_file.columns
Out[3]: Index(['C0', 'C1'], dtype='object')

In [4]: runfile('C:/Users/Phili/Documents/Python Scripts/
readingfiles.py', wdir='C:/Users/Phili/Documents/Python
Scripts', current_namespace=True)

In [5]: runfile('C:/Users/Phili/Documents/Python Scripts/
readingfiles.py', wdir='C:/Users/Phili/Documents/Python
Scripts', current_namespace=True)

In [6]:

```

The bottom pane shows an Excel spreadsheet titled `Book3_Updated - Excel`. The spreadsheet has a single sheet named `Updated_Sheet`. The data is as follows:

	C0	C1
r0	1	2
r1	2	4
r2	3	6
r3	4	8
r4	5	10
r5	6	12
r6	7	14
r7	8	16
r8	9	18
r9	10	20
r10	11	22
r11	12	24

There is a similar method `to_csv()` and like `read_csv()` doesn't have a keyword argument `sheet_name` because `csv` or `txt` files don't have sheets. `to_csv` can be used to write both `csv` and `txt` files.

```

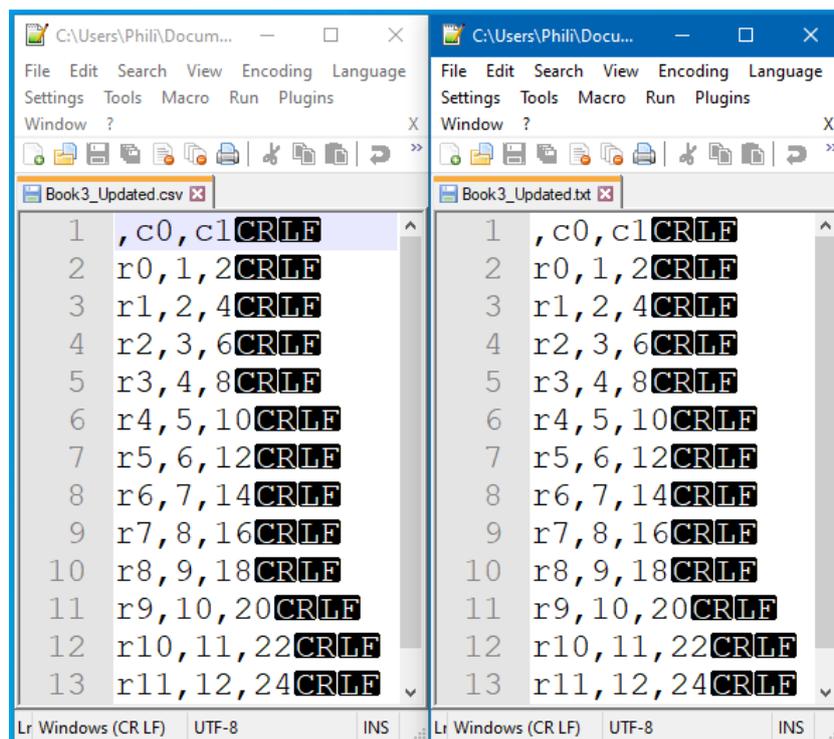
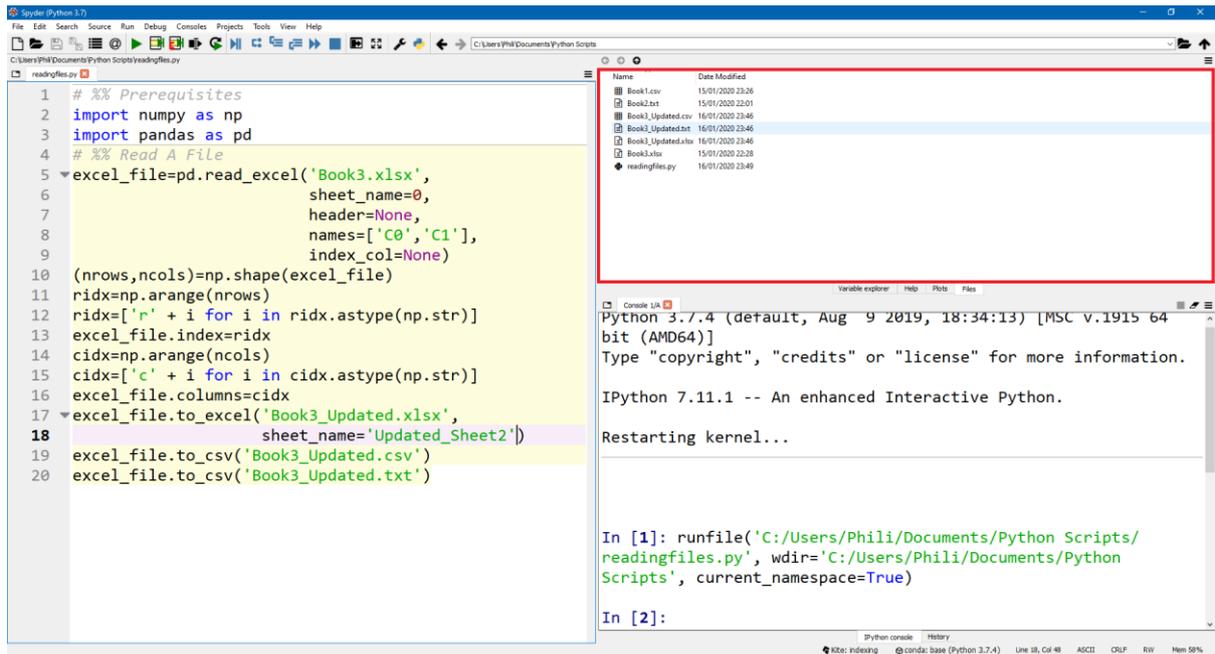
1. # %% Prerequisites
2. import numpy as np
3. import pandas as pd
4. # %% Read A File
5. excel_file=pd.read_excel('Book3.xlsx',
6.                             sheet_name=0,
7.                             header=None,
8.                             names=['C0', 'C1'],
9.                             index_col=None)
10. (nrows,ncols)=np.shape(excel_file)

```

```

11. ridx=np.arange(nrows)
12. ridx=['r' + i for i in ridx.astype(np.str)]
13. excel_file.index=ridx
14. cidx=np.arange(ncols)
15. cidx=['c' + i for i in cidx.astype(np.str)]
16. excel_file.columns=cidx
17. excel_file.to_excel('Book3_Updated.xlsx',
18.                    sheet_name='Updated_Sheet')
19. excel_file.to_csv('Book3_Updated.csv')
20. excel_file.to_csv('Book3_Updated.txt')

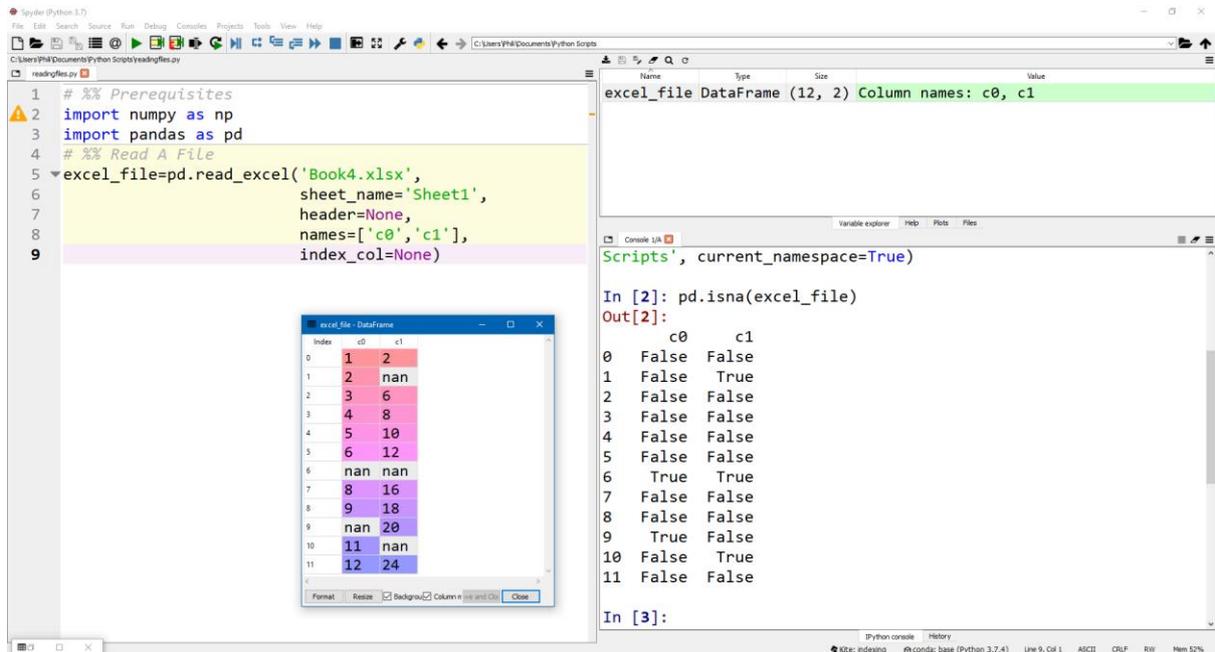
```



We can use the pandas functions `isna()` or `notna()` to check if a value is or isn't a nan.

```
pd.isna(excel_file)
pd.notna(excel_file)
```

This returns an output Boolean corresponding to the input.



```
1 # %% Prerequisites
2 import numpy as np
3 import pandas as pd
4 # %% Read A File
5 excel_file=pd.read_excel('Book4.xlsx',
6                          sheet_name='Sheet1',
7                          header=None,
8                          names=['c0','c1'],
9                          index_col=None)
```

Index	c0	c1
0	1	2
1	2	nan
2	3	6
3	4	8
4	5	10
5	6	12
6	nan	nan
7	8	16
8	9	18
9	nan	20
10	11	nan
11	12	24

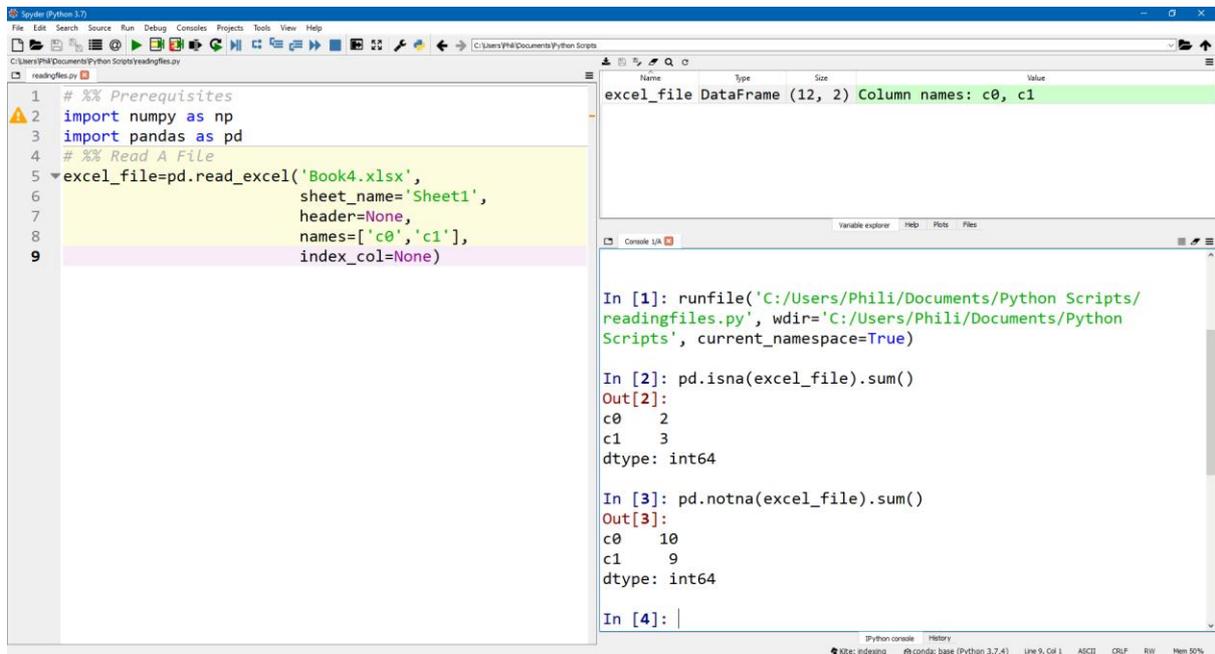
```
excel_file DataFrame (12, 2) Column names: c0, c1

In [2]: pd.isna(excel_file)
Out[2]:
c0    c1
0  False  False
1  False  True
2  False  False
3  False  False
4  False  False
5  False  False
6  True   True
7  False  False
8  False  False
9  True   False
10 False  True
11 False  False

In [3]:
```

We can also use the method `sum()`, so sum the number of nan values and not nan values.

```
pd.isna(excel_file).sum()
pd.notna(excel_file).sum()
```



```
1 # %% Prerequisites
2 import numpy as np
3 import pandas as pd
4 # %% Read A File
5 excel_file=pd.read_excel('Book4.xlsx',
6                          sheet_name='Sheet1',
7                          header=None,
8                          names=['c0','c1'],
9                          index_col=None)
```

```
In [1]: runfile('C:/Users/Phili/Documents/Python Scripts/
readingfiles.py', wdir='C:/Users/Phili/Documents/Python
Scripts', current_namespace=True)

In [2]: pd.isna(excel_file).sum()
Out[2]:
c0    2
c1    3
dtype: int64

In [3]: pd.notna(excel_file).sum()
Out[3]:
c0    10
c1    9
dtype: int64

In [4]:
```

The method `dropna` can also be used on a dataframe. We can type it with open parenthesis to see the keyword input arguments:

```
excel_file.dropna(
```

The screenshot shows the Spyder Python IDE with a script named 'readfiles.py'. The script reads an Excel file 'Book4.xlsx' into a DataFrame named 'excel_file'. The DataFrame has 12 rows and 2 columns, 'c0' and 'c1'. The data is as follows:

Index	c0	c1
0	1	2
1	2	nan
2	3	6
3	4	8
4	5	10
5	6	12
6	nan	nan
7	8	16
8	9	18
9	nan	20
10	11	nan
11	12	24

The console shows the following code and output:

```

In [2]: pd.isna(excel_file)
Out[2]:
      c0  c1
0  False False
1  False  True
2  False False
3  False False

```

The variable explorer shows 'excel_file' as a DataFrame (12, 2) with column names 'c0', 'c1'. A tooltip for the `dropna` function is also visible, showing its signature: `dropna(axis=0, how='any', thresh=None, subset=None, inplace=False,)`.

We can see that by default `axis=0`, `how='any'` and `inplace=False`. This means it will act on indexes (rows) and remove any indexes (rows) that have any nan value and this will be saved as an output matrix.

```
excel_file.dropna(how='any')
```

The screenshot shows the same Spyder Python IDE environment. The script is the same, but the console shows the result of `excel_file.dropna(how='any')`:

```

In [3]: excel_file.dropna(how='any')
Out[3]:
      c0  c1
0  1.0  2.0
2  3.0  6.0
3  4.0  8.0
4  5.0 10.0
5  6.0 12.0
7  8.0 16.0
8  9.0 18.0
11 12.0 24.0

```

The variable explorer shows 'excel_file' as a DataFrame (12, 2) with column names 'c0', 'c1'. The DataFrame viewer shows the updated data, where rows 1, 6, 9, and 10 have been removed because they contained any NaN values.

On the other hand, it can be changed to `how='all'` which will only remove indexes (rows) where all values are missing.

```
excel_file.dropna(how='all')
```

The screenshot shows the Spyder Python IDE with the following code in the editor:

```

1 # %% Prerequisites
2 import numpy as np
3 import pandas as pd
4 # %% Read A File
5 excel_file=pd.read_excel('Book4.xlsx',
6                           sheet_name='Sheet1',
7                           header=None,
8                           names=['c0','c1'],
9                           index_col=None)

```

The Variable explorer shows the variable `excel_file` as a DataFrame with 12 rows and 2 columns (`c0`, `c1`).

The Console shows the output of `excel_file.dropna(how='all')`:

```

In [4]: excel_file.dropna(how='all')
Out[4]:
0    1.0    2.0
1    2.0   NaN
2    3.0    6.0
3    4.0    8.0
4    5.0   10.0
5    6.0   12.0
6    nan    nan
7    8.0   16.0
8    9.0   18.0
9   NaN   20.0
10  11.0   NaN
11  12.0   24.0

```

A small DataFrame viewer window shows the data for `excel_file`:

Index	c0	c1
0	1	2
1	2	nan
2	3	6
3	4	8
4	5	10
5	6	12
6	nan	nan
7	8	16
8	9	18
9	nan	20
10	11	nan
11	12	24

It is also possible to specify a subset of pandas series (columns) to remove rows from which have 'any' or 'all' nan values. For instance, if we were only interested in removing indexes that had a nan value in the panda series 'c1' we would assign the keyword `subset` to a single element vector.

```
excel_file.dropna(how='any', subset=['c1'])
```

The screenshot shows the Spyder Python IDE with the same code as above, but the console output shows the result of `excel_file.dropna(how='any', subset=['c1'])`:

```

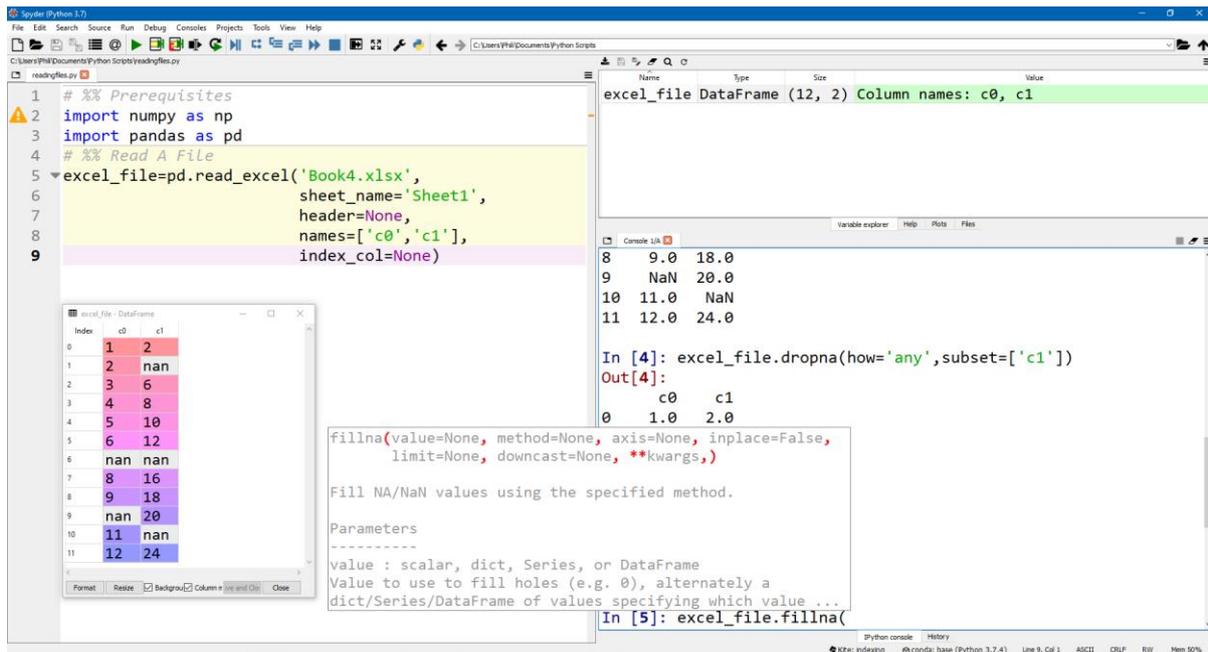
In [4]: excel_file.dropna(how='any', subset=['c1'])
Out[4]:
0    1.0    2.0
2    3.0    6.0
3    4.0    8.0
4    5.0   10.0
5    6.0   12.0
7    8.0   16.0
8    9.0   18.0
11  12.0   24.0

```

The DataFrame viewer window shows the data for `excel_file` after dropping rows with any NaN values in column 'c1':

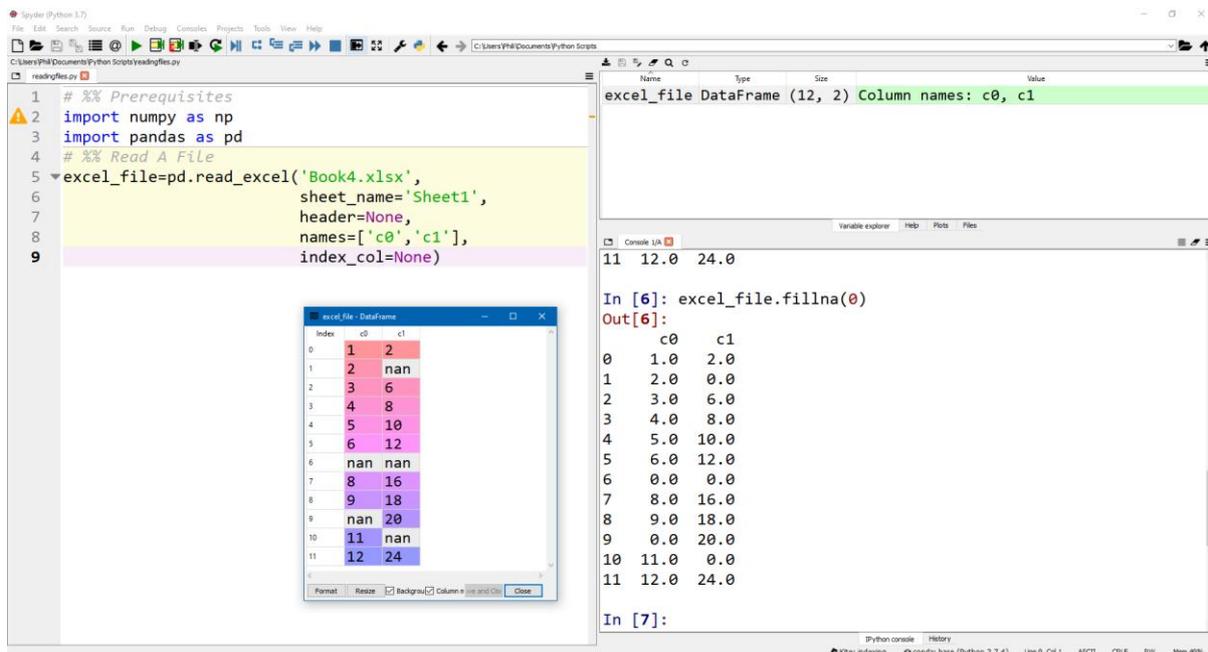
Index	c0	c1
0	1	2
1	2	nan
2	3	6
3	4	8
4	5	10
5	6	12
6	nan	nan
7	8	16
8	9	18
9	nan	20
10	11	nan
11	12	24

The method `fillna` can be used to fill any missing values, we can once again input it with open parenthesis to see the keyword input arguments.



We can fill in the data with a number, say for example the number 0.

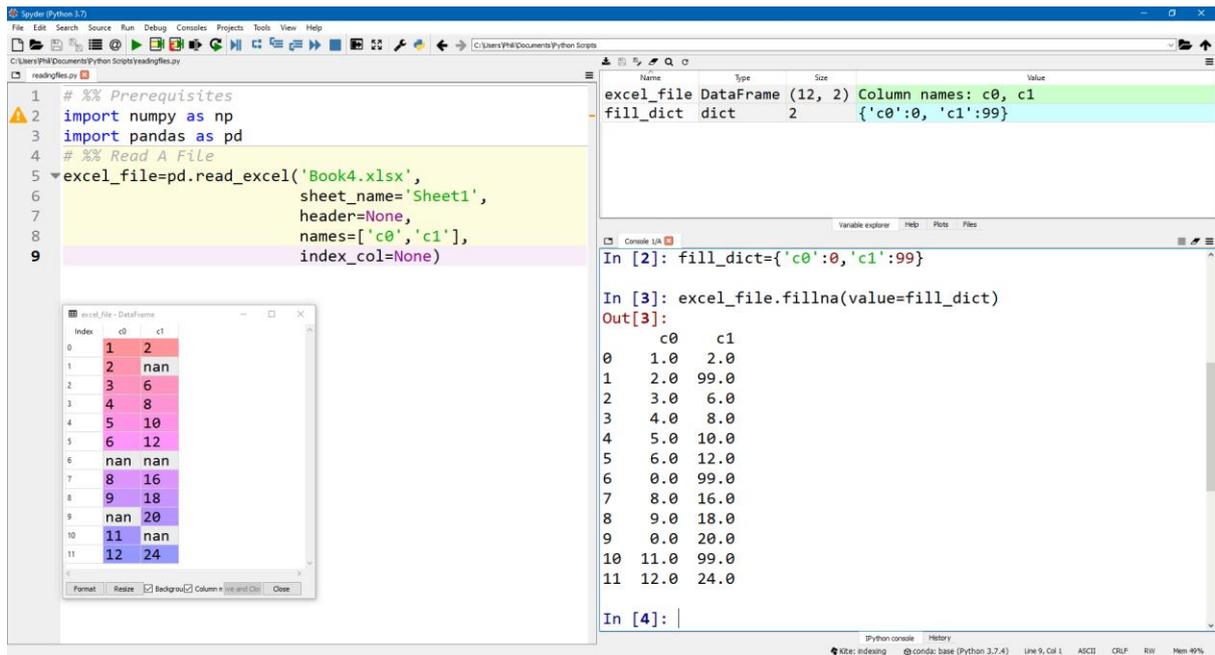
```
excel_file.fillna(0)
```



Alternatively, we can use a dictionary to fill each column with a specific value for instance replacing every missing value in pandas series 'c0' with 0 and panda series 'c1' with 99.

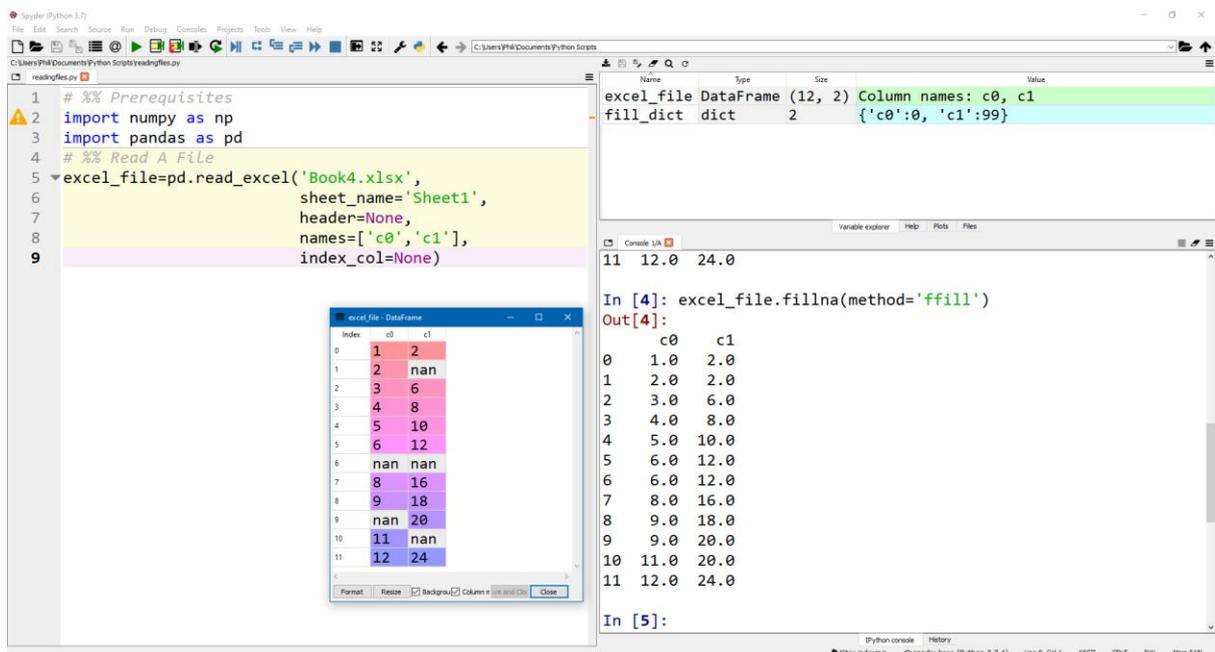
```
fill_dict={'c0':0, 'c1':99}
excel_file.fillna(value=fill_dict)
```

Which may be particularly useful if each pandas series is a different dtype for example if you have a dataframe where a series that is numeric, a series that is Boolean and a series that consists of strings it will make more sense to have a default value for the nan values is required to suit each series.



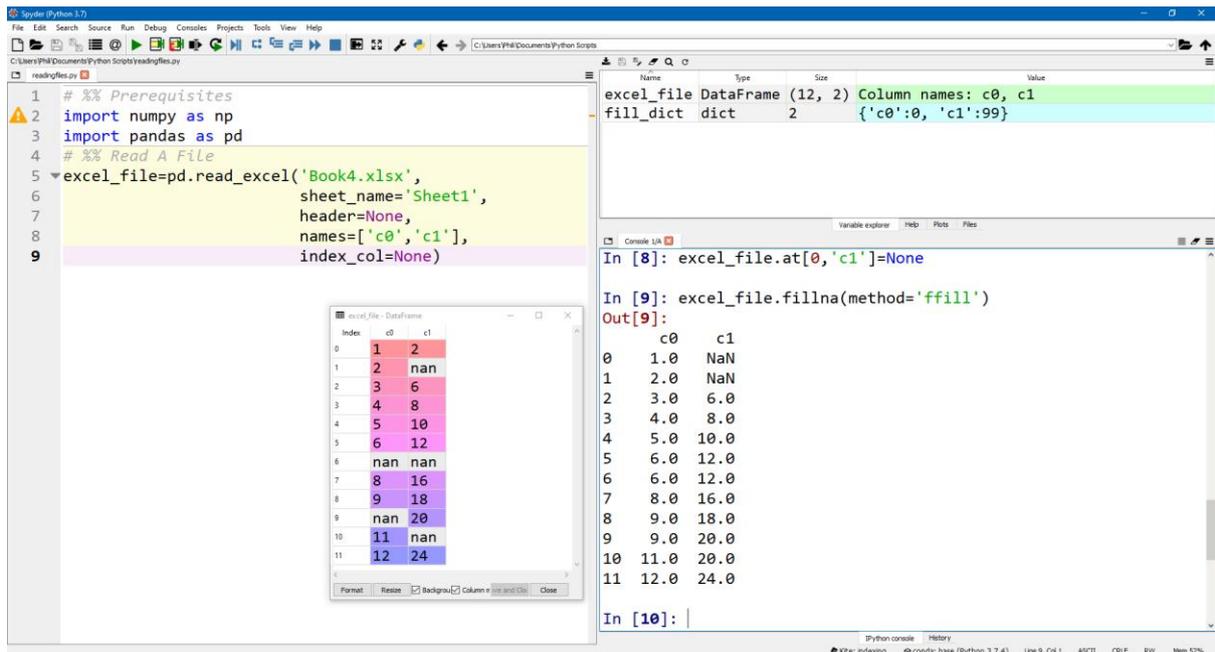
The keyword input argument `method` can be set to forward fill `'ffill'` or backfill `'bfill'` where each missing value will take the previous non-missing value or next non-missing value respectively.

```
excel_file.fillna(method='ffill')
```

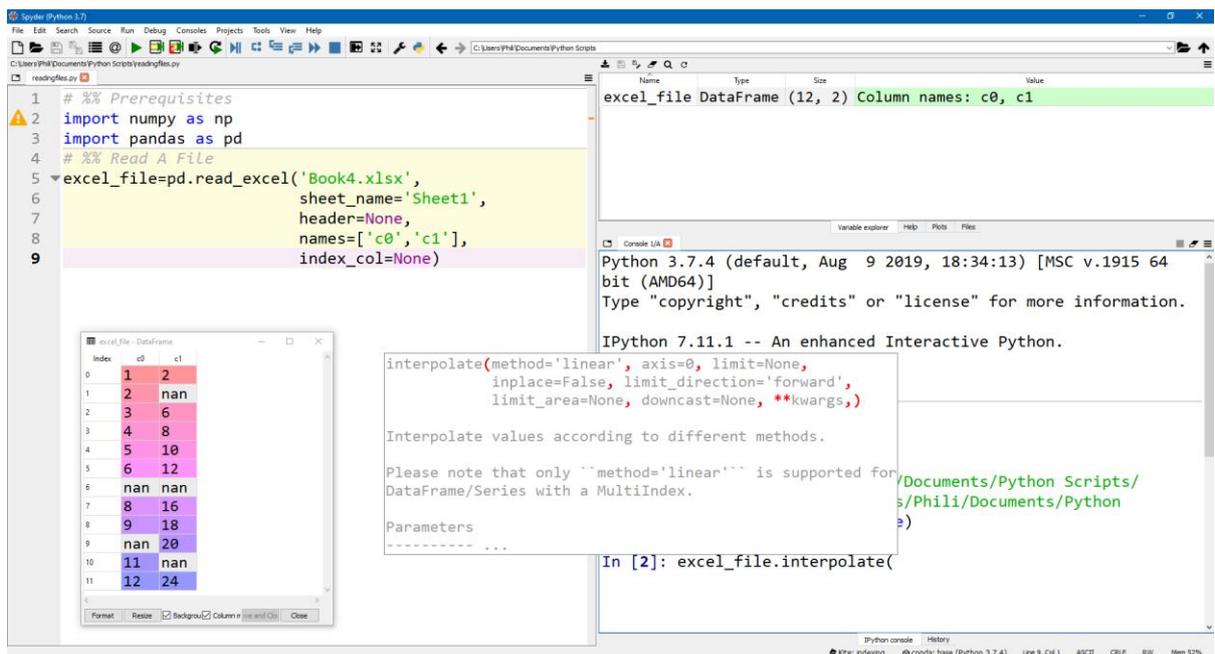


Forward fill requires a previous number. If we replace the value on the zeroth index of series `'c1'` with `None` we will see it is unable to replace the nan values on the first two indexes of `'c1'`.

```
excel_file.at[0, 'c1']=None
excel_file.fillna(method='ffill')
```



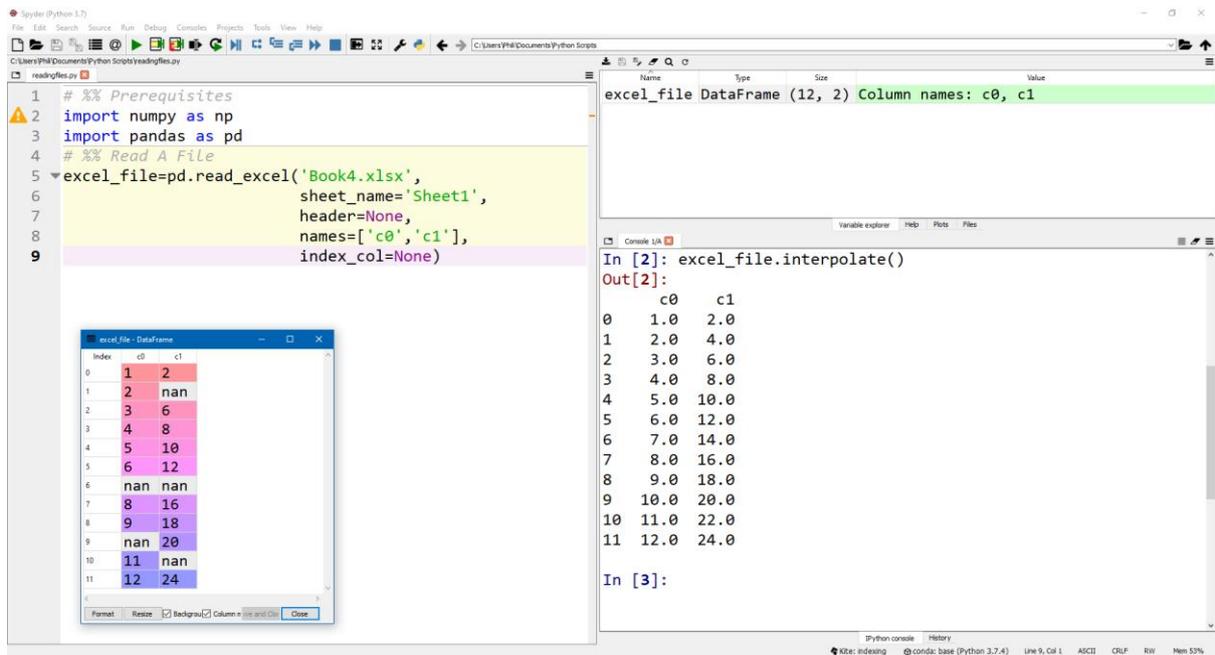
The pandas dataframe also has the method `interpolate` which will only work on numeric values and uses the same code as the scipy interpolate under the hood. We can type it in with open parenthesis to view the keyword input arguments:



We can try uses default linear interpolation where all keyword input arguments are left at their default values:

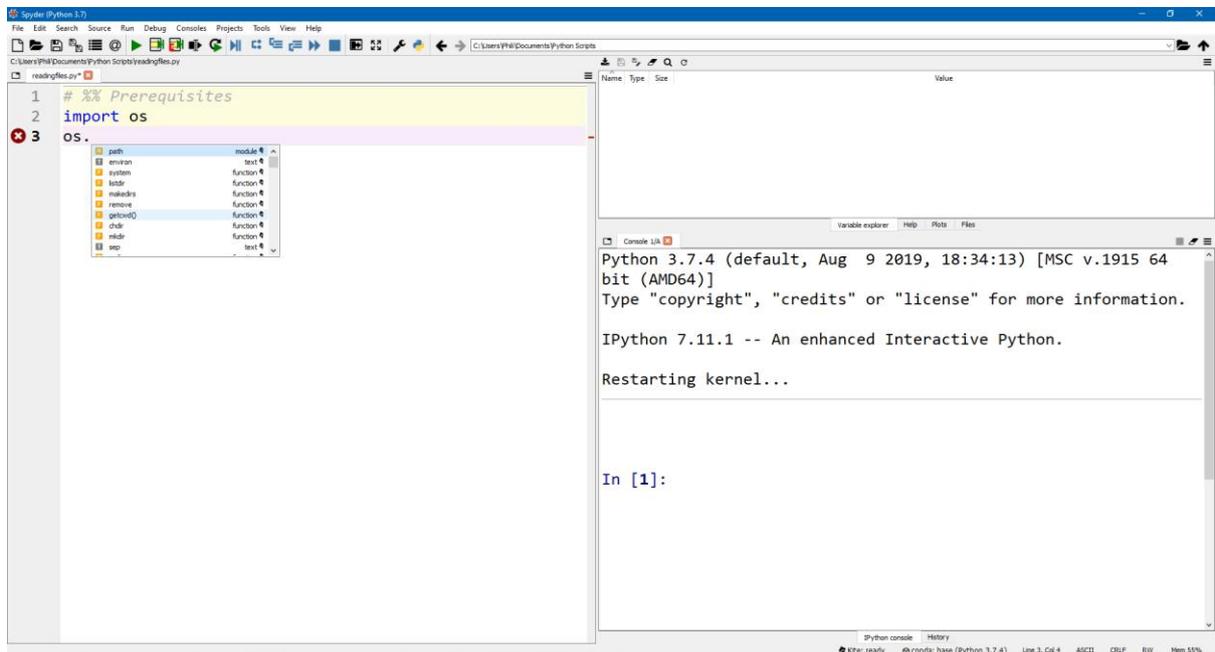
```
excel_file.interpolate()
```

And in the case of our very simple dataset the values are interpolated correctly.



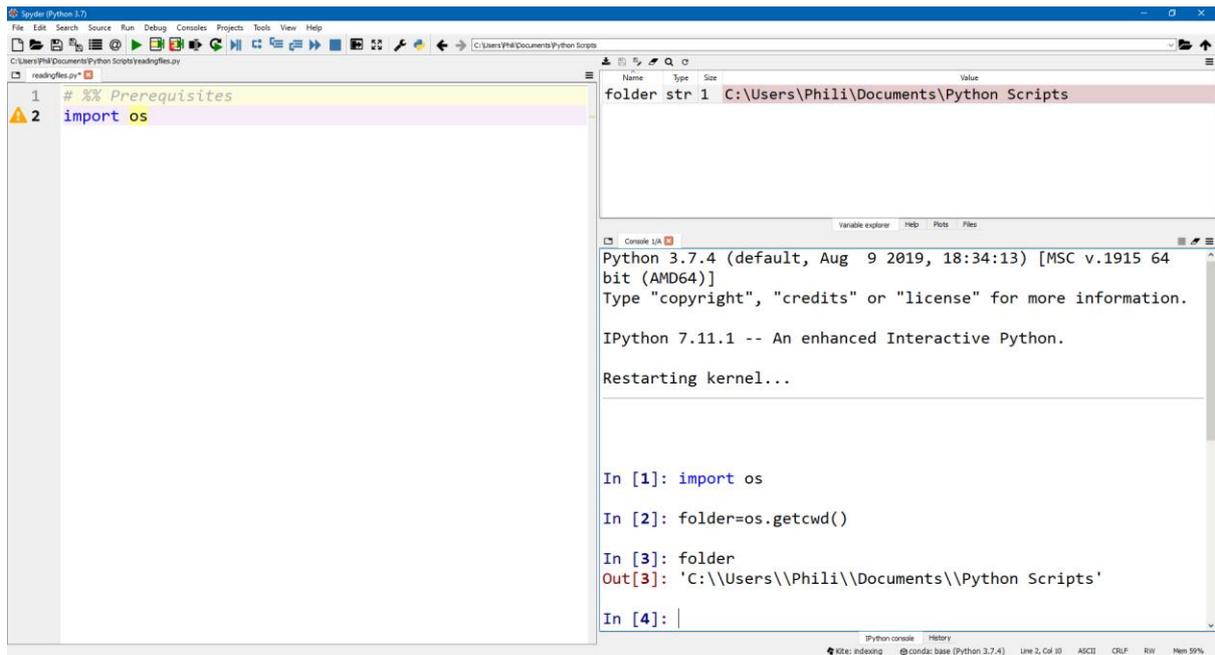
Operating System Module

The operating system `os` module is useful for finding the current folder, to move to a folder, to make a folder, to rename a folder and to delete a folder. Another word for folder is directory and the functions within the `os` module reference the word directory. It is useful to `import` the `os` module and then type in `os` followed by a dot `.` then a tab `↵` to see the number of functions available.



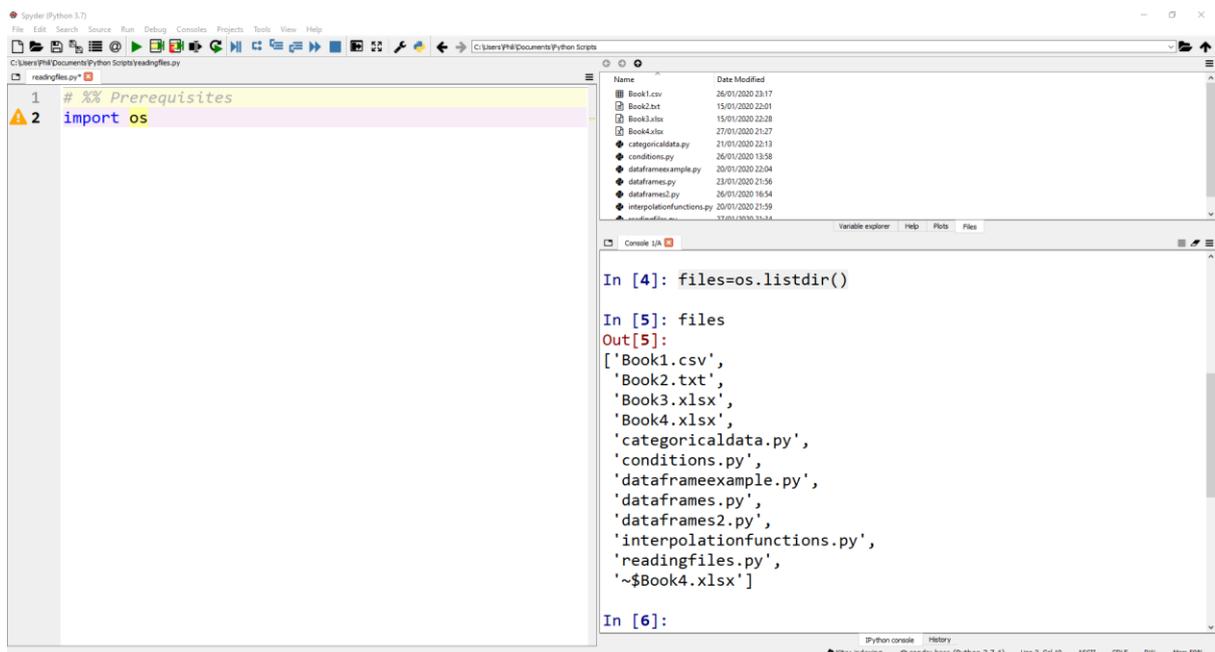
The function get current working directory `getcwd` can be used to quickly obtain the working directory. We can save this to a variable.

```
folder=os.getcwd()
```



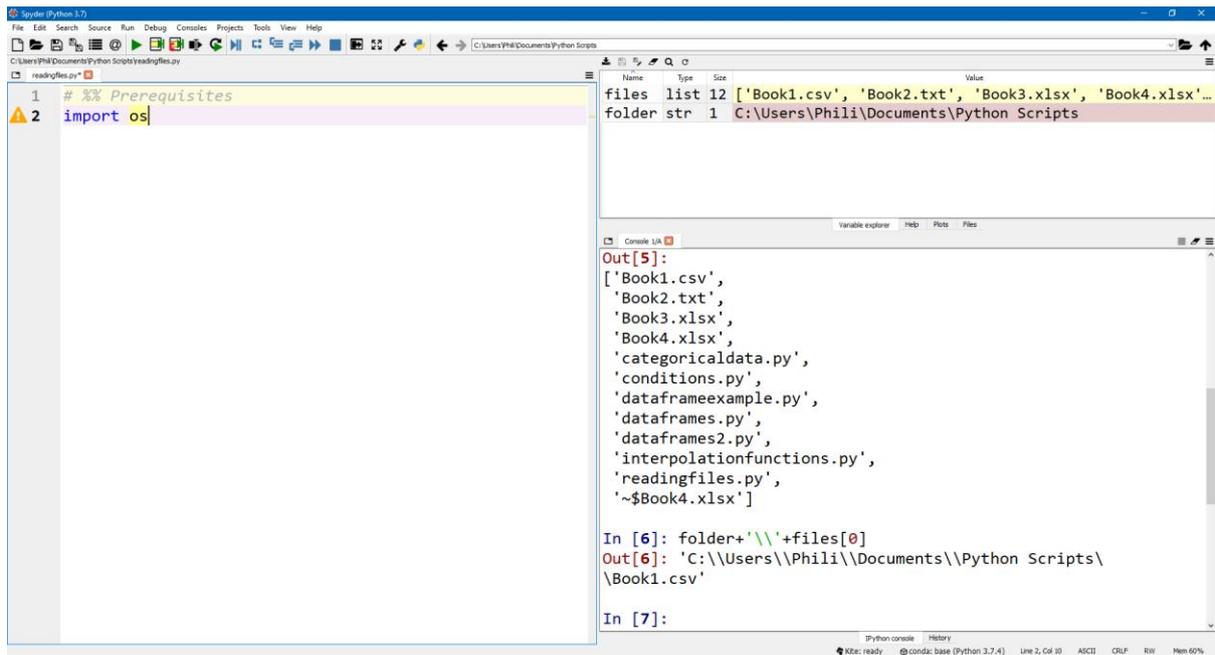
The function list directory `listdir` can be used to list all the files available in a directory. This can be saved to a list:

```
files=os.listdir()
```



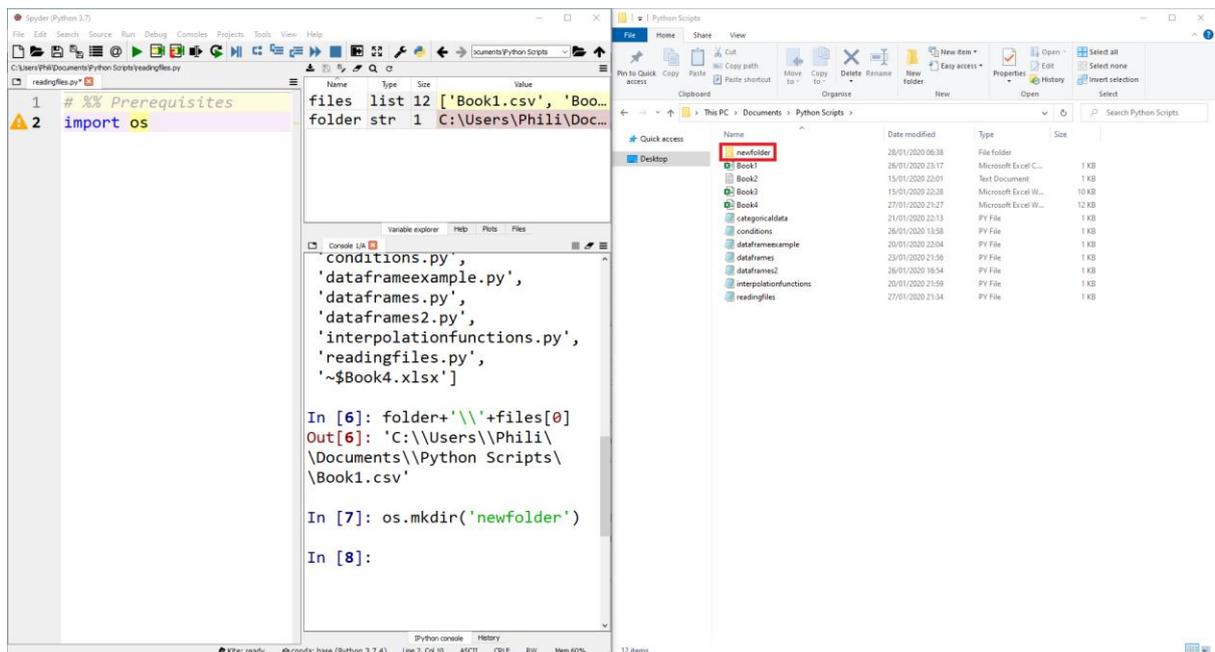
Recall that we can use string concatenation to get the full path of a file including the directory. In this case:

```
folder+'\\'+files[0]
```



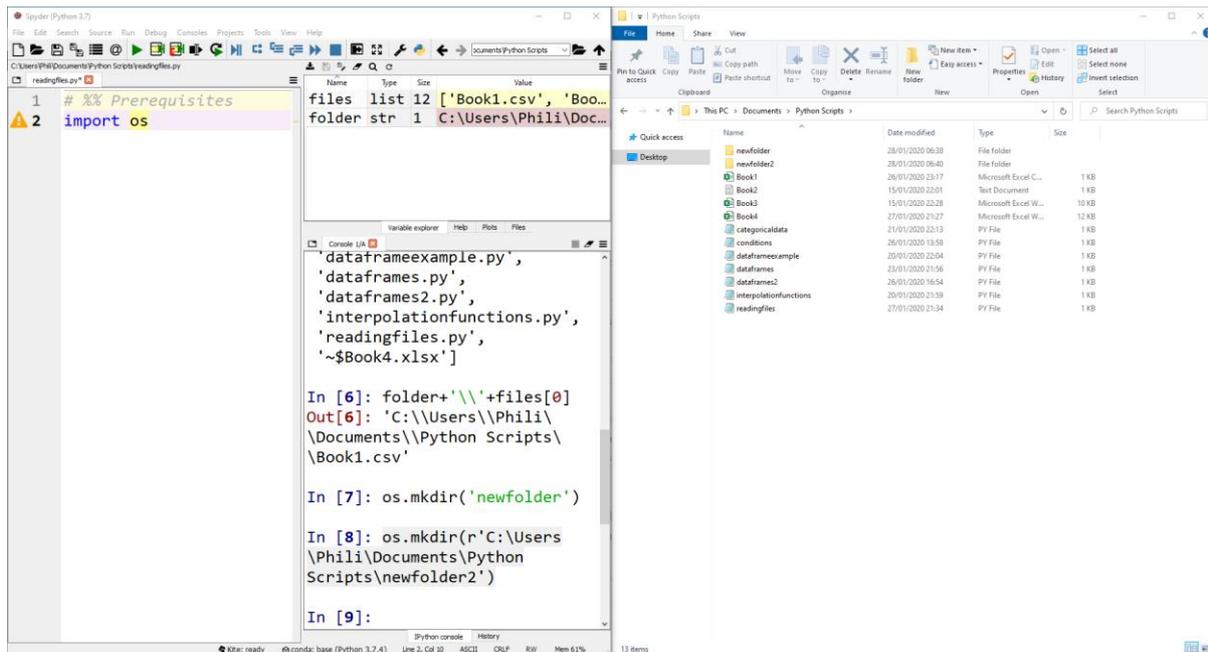
The function `make directory mkdir` can be used to make folders. If the input is a string, the directory will be created as a subfolder within the current working directory:

```
os.mkdir('newfolder')
```



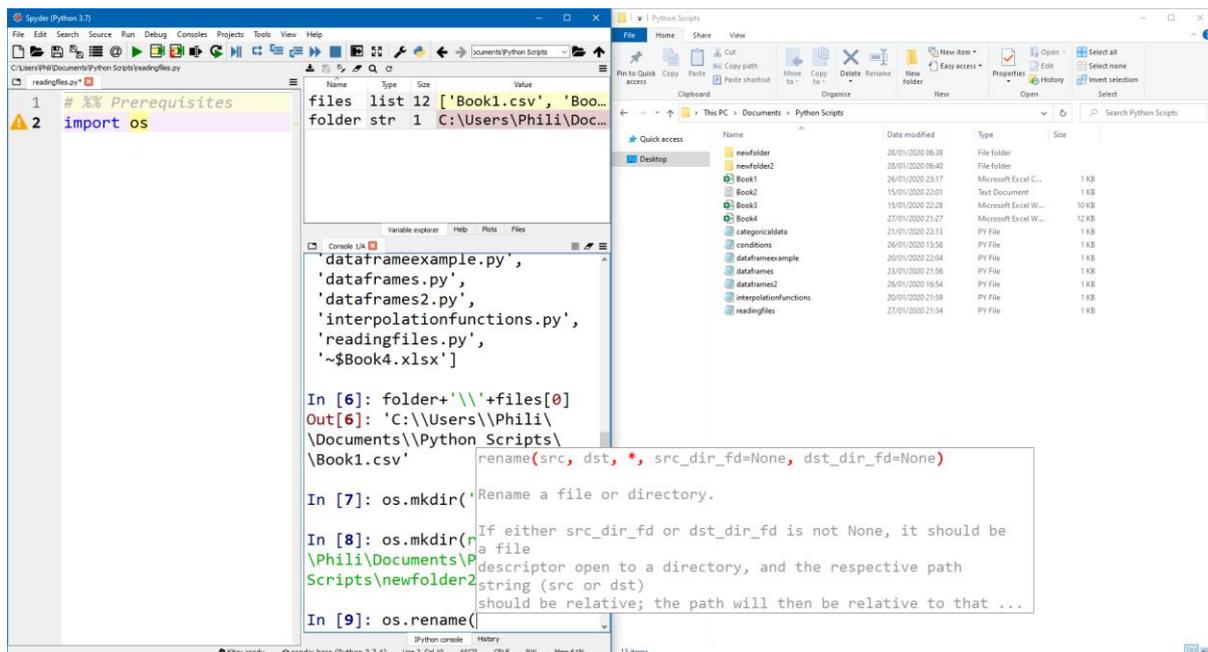
However, a relative string may instead be input specifying the full path:

```
os.mkdir(r'C:\Users\Phili\Documents\Python Scripts\newfolder2')
```



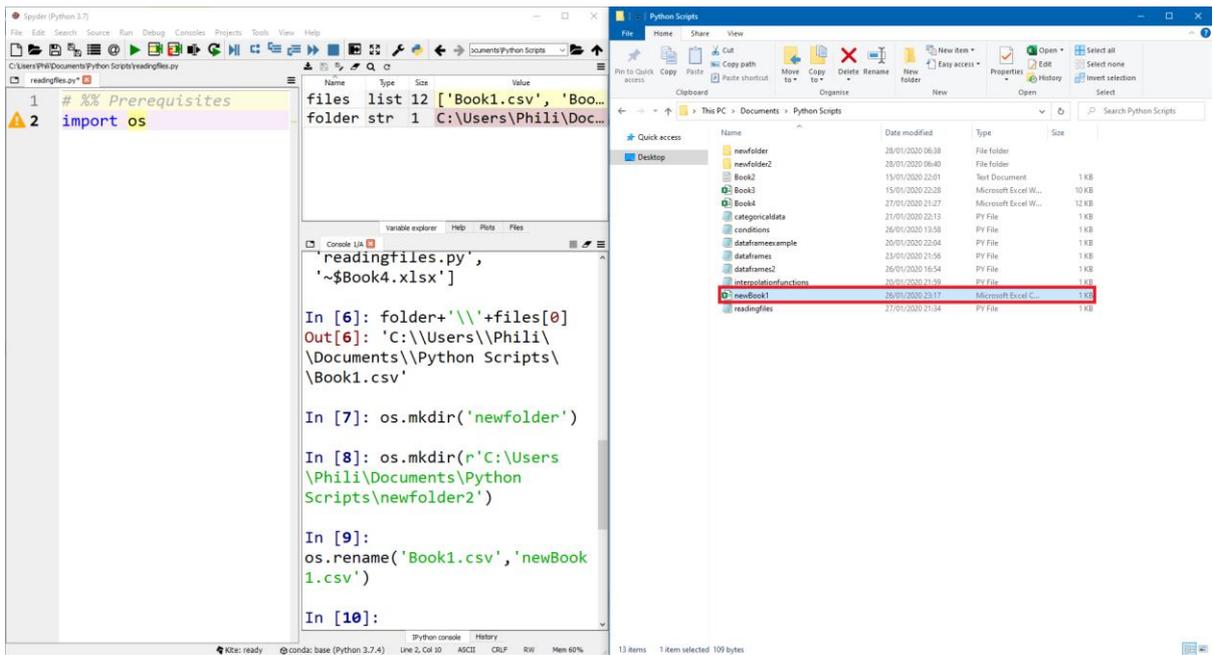
We can rename files or directories using the function `rename`. Let's type it in with open parenthesis to view the keyword input arguments:

```
os.rename (
```



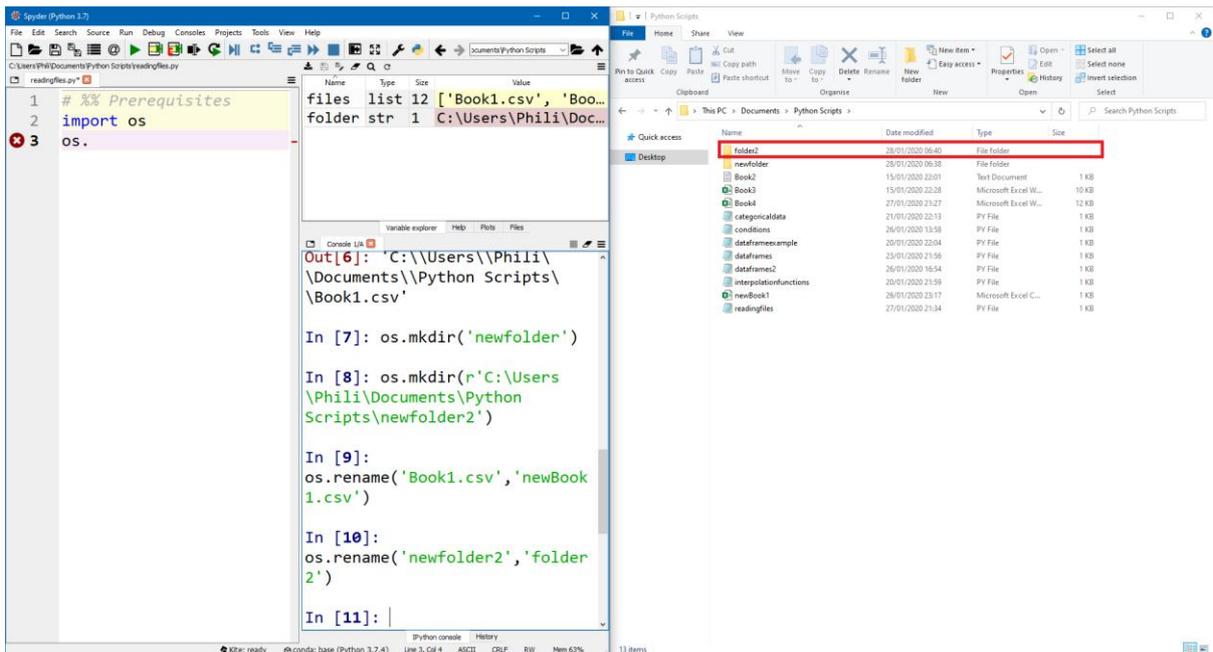
The keyword arguments are the source `src`, destination `dst`. The source directory folder `src_dir_fd` and destination directory folder `dst_dir_fd` keyword arguments are set to `None` and unavailable on the Windows platform. Instead full paths can be used to move location of a file. We can rename the `'Book1.csv'` file to `'newBook1.csv'`.

```
os.rename('Book1.csv', 'newBook1.csv')
```



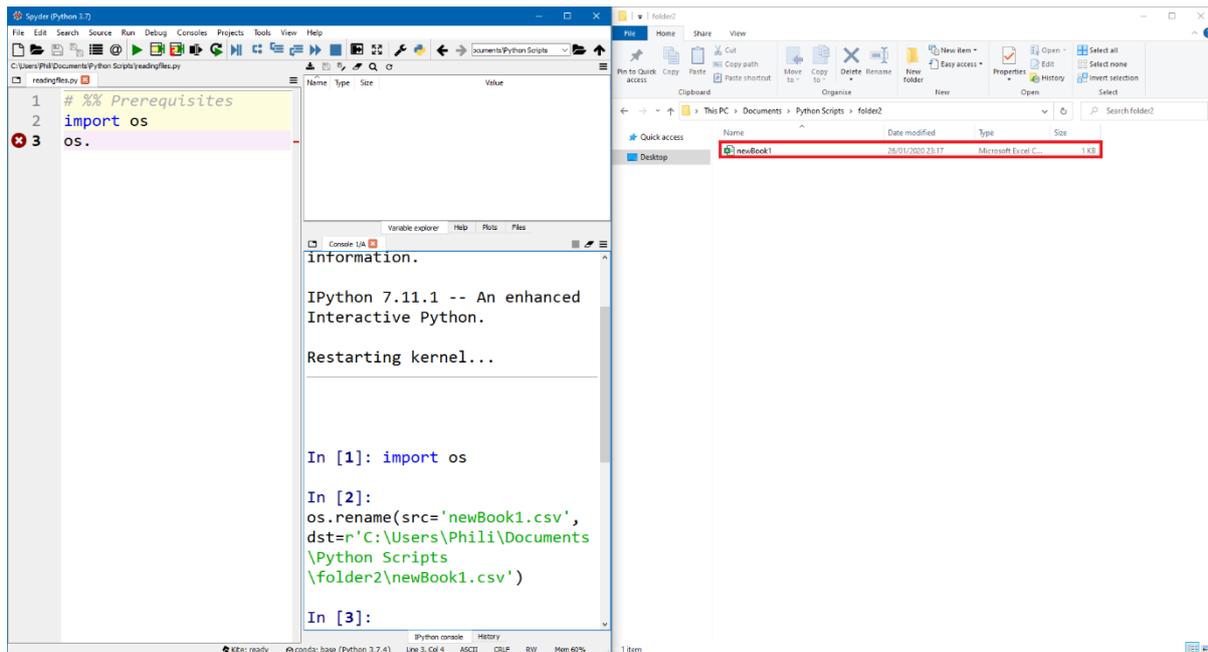
The function `rename` can also be used to rename folders:

```
os.rename('newfolder2', 'folder2')
```



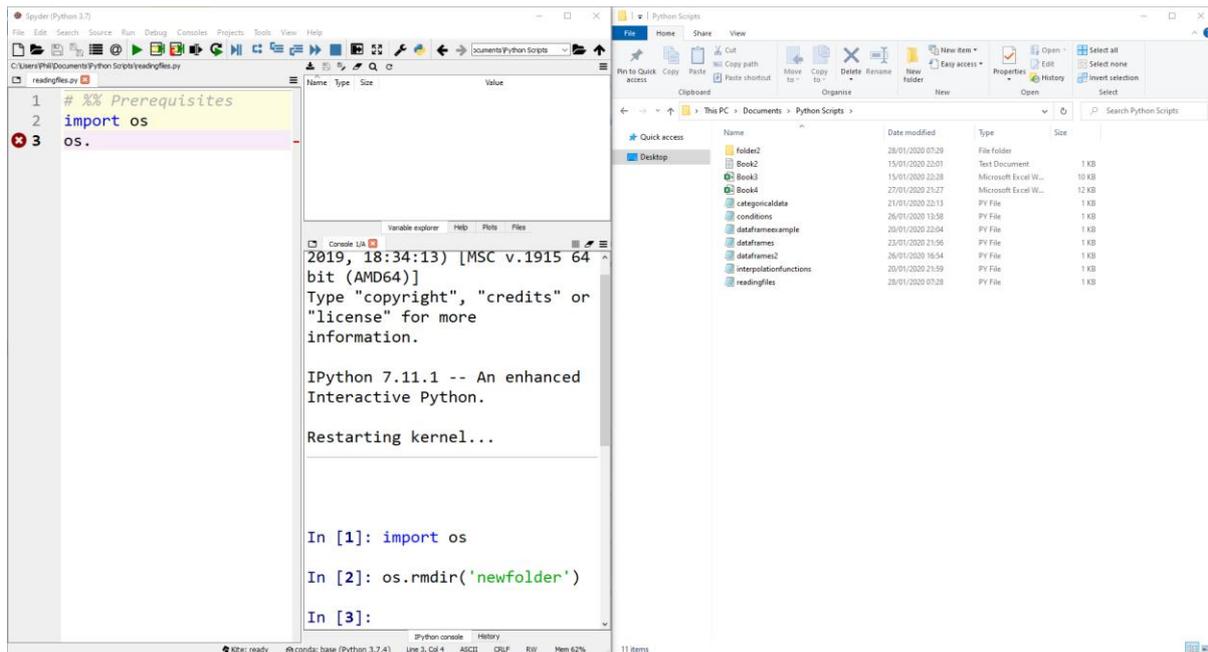
We can use `rename` to move a file by specifying a relative string which includes the new destination directory:

```
os.rename(src='newBook1.csv', dst=r'C:\Users\Phili\Documents\Python
Scripts\folder2\newBook1.csv')
```



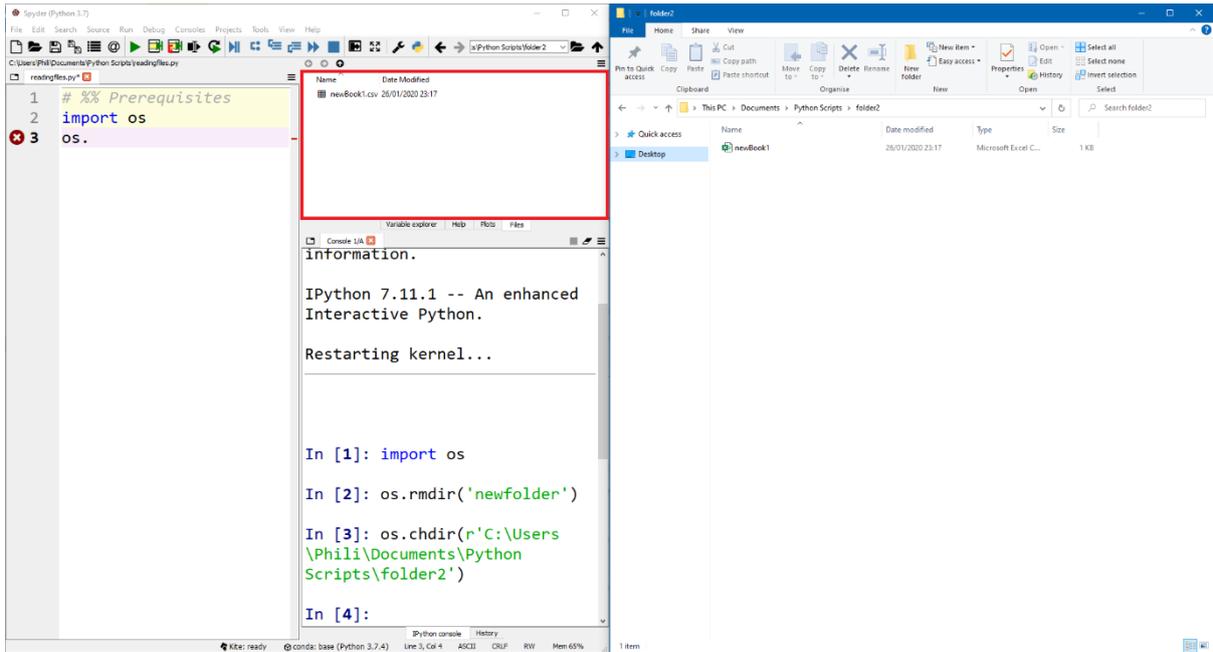
We can also remove a directory using:

```
os.rmdir('newfolder')
```



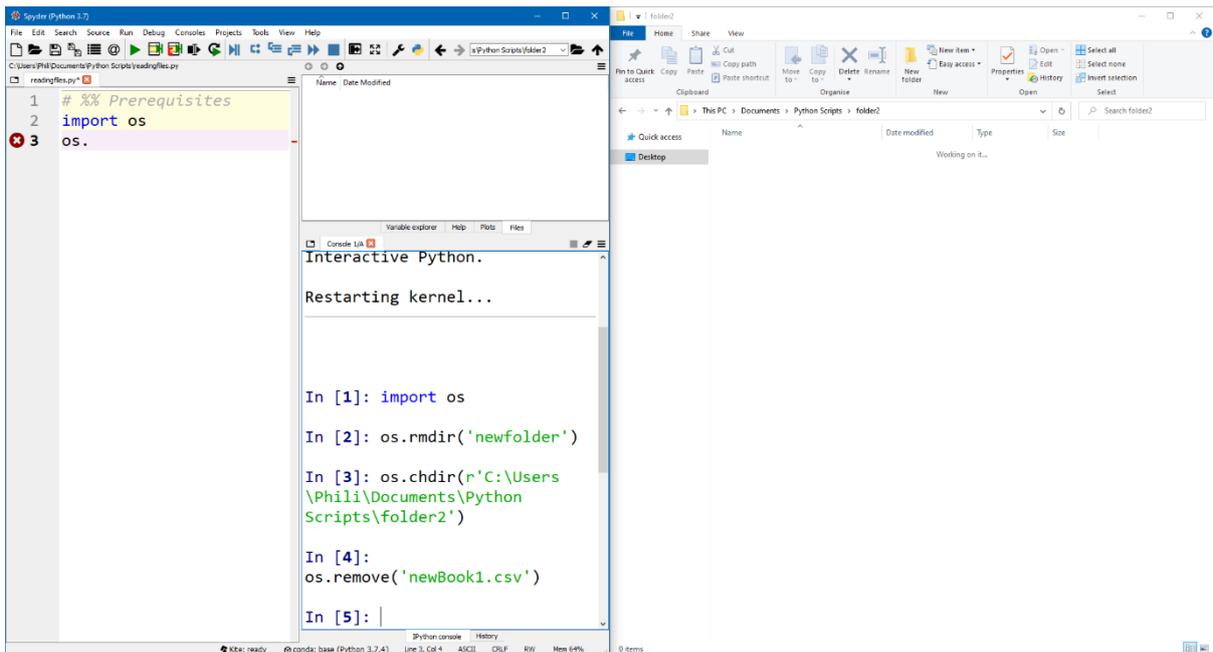
The function change directory `chdir` changes the current working directory (folder):

```
os.chdir(r'C:\Users\Phili\Documents\Python Scripts\folder2')
```



We can then delete a file using the function remove:

```
os.remove('newBook1.csv')
```



To quickly change move up a directory (folder) we can use `'..'` as an input argument for the function `chdir`:

```
os.chdir('..')
```

