Python Coding Club

This series is to introduce Python as a programming language to be used for data analysis, scientific computing and plotting. It is by **no means comprehensive** but will provide a basis for further investigation and exploration into this powerful language. These notes are **best visualised** in a **Jupyter Notebook** and I encourage you to **follow along** in your **preferred IDE** (this will all make sense after Part 1).

Part 1: Introduction and getting started

Part 1 in this series will introduce what Python is and show you the ways in which you can write and execute Python scripts on your machine.

Part 1.1: Introduction to Python

What is Python?

Python is what is known as an ***object-oriented*** **'high-level' programming language** and was invented by Guido van Rossum in 1991. Other high-level languages include **C++, R and Java**.

A 'high-level' language (HLL) is a language that is more **similar to human language** than machine language. HLL's are typically **machine independent** and can therefore run on a variety of hardware.

Once written, a Python script is **compiled and interpreted** into a low-level language which is machine specific and finally into machine language **(1's and 0's – or bits)** which then executes the program.

What is object-oriented programming?

Object-oriented programming languages (like Python) consider every variable or function as a **specific object**. This means that every object in a Python script has **methods** (functions) and **attributes** (variables) associated with it.

This type of programming is very powerful as it is **very logical** and can lead to **modular readable code** that can be reused without having to retype code.

New **types** of object can be made using the `class` keyword. More on objects and classes later.

Why Python?

Python is **very powerful** as it is **easily interpreted and heavily supported** via open source libraries and modules (more about this later).

Python is particularly useful for **scientific computing** as it is easy to learn due to its **clear syntax** as well as offering many **mathematical libraries for data analysis** that are **well documented** on their respective websites.

Python itself comes with many useful **'modules'** for computing but where its usefulness really shines is through the **open source packages/modules** that have been written by software developers for use in **other programs**. The full library of these modules is called the **Python Package Index** or **PyPI** for short and can be found here. Modules and packages will be covered more in a later section.

What are Python modules, packages and libraries?

One of Python's main appeals is how well supported the language is. This includes **open source well documented** Python scripts called **modules**. **Modules** are simply Python scripts containing **functions and/or classes** for use in programs (more on functions and classes later).

Groups of modules are called **libraries** or **packages**. Many common libraries include NumPy for numerical Python programming, SciPy for scientific computing, Pandas for data analysis and matplotlib for publication quality data plotting.

To gain access to a particular module or package, an `import` statement is required in your Python program file, more on this later.

Python comes pre-packaged with base modules such as `math`, `statistics`, `sys`, `os`, `collections`, `datetime` in Python's 'standard library. It is worth having a look through Python's documentation on what modules the standard library contains.

Apart from modules, Python comes preloaded with built-in functions that can be used without importing any new modules. More on this later.

Python guiding principles

Python's design philosophy encourages **code readability** and therefore a list of *guiding design principles* when writing Python code have been established called *'The Zen of Python'*:

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.

Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

This essentially means that ***simple code is better than complex code but complex code is better than complicated code*** and if the implementation of how your code works is difficult to explain **it is probably too complex**.

The purpose of Python is to be easily readable and easily understandable by other programmers.

These guiding principles of programming in Python can also be easily retrieved by running the following cell (**select the cell in a Jupyter notebook and press Shift+Enter**):

In [1]:

```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Part 1.2: Setting up

Installing Python

In order to begin writing and executing Python scripts, first you will need to install Python on your computer by following this link.

Choose the correct download for your machine and make sure to **download Python 3.x (current version)** as Python 2.7 is no longer supported.

Once downloaded, follow the installer on your machine to install Python.

Running Python in the Interactive Development and Learning Environment (IDLE)

Once installed, navigate to the **installation location on your machine** and launch **'IDLE'** or the Interactive Development and Learning Environment. This will open a **Python 'shell'** which simply means an interactive Python interpreter that will **execute each line as it is typed**.

Try typing in the following code into IDLE to print **'Hello World'** underneath.

```
In [2]:
```

```
print('Hello World')
Hello World
```

Congratulations on running your first Python program!

Printing 'Hello World' is typically a **first step** when using a new programming language simply to check that the **installation has been successful**.

Running Python in the command prompt (Windows) or terminal (Mac OSX)

Python can also be accessed by opening either Command Prompt for Windows or Terminal for Mac OSX by typing **python3 or python**.

This will open a Python interpreter in the terminal through which the same `print('Hello World')` should run.

Interactive Developer Environments (IDEs)

Running Python code line by line is useful to test the installation is successful, but typically programs are hundreds or thousands of lines long, this is where Interactive Developer Environments (IDEs) come in.

IDEs are applications where Python scripts can be written, complied and executed. There are many benefits to IDEs including **line numbering**, **syntax highlighting** and **code completion**.

- **Line numbering** means each line of code is numbered; this can help with debugging when Errors occur as the line number causing the issue will be shown.
- **Syntax highlighting** shows different Python objects/methods in seperate colours, making it easier to visualise how your script will be interpreted. This also shows syntax errors can occur if a line of code is not interpretable by the Python interpreter.
- **Code completion** provides suggestions on what you are going to type (a variable or function name etc.) automatically and can also show you what type of object the name is referring to (class, function, variable etc. more on this later...).

IDE's also have the ability of code 'traceback' when an `Error` is invoked in a program (i.e. it does not run correctly) the line number where the error occurred and what error it is which can be very useful when debugging your programs.

There are **many different IDEs** out there and **different programmers will have different preferences** on which IDE is the best, therefore it may be useful to **try a couple and see which you prefer**.

A few common IDEs include:

- PyCharm
- Microsoft Visual Studio
- Eclipse
- Spyder (part of Anaconda package)

Personally, I prefer **PyCharm as my IDE of choice** as it contains a **variety of background themes, is customisable and provides the above features**. PyCharm can also come with an **educational introductory course on Python by installing the plugin 'EduTools'**.

In this series of notes I will explain how to **install PyCharm** and also the **Anaconda Package** which includes the **IDE Spyder and a link to Jupyter notebooks** (where you may be reading this currently).

Jupyter Notebooks

Jupyter notebooks are **interactive Python notebooks** (of which some of this series may be written).

They can be used to show a **workflow of data analysis** as well as markdown text such as the text shown here. Jupyter notebooks as well as **Spyder IDE come prepacked in the Anaconda distribution** mentioned above.

Jupyter notebooks are a useful tool when it comes to **showing specific reports of programs**. Each cell is executed by pressing **Shift+Enter** when the cursor is **selected on the cell**. Jupyter notebooks therefore have the ability to act as a **Python interpreter like IDLE** but also as an **IDE as the code written is saved and can be referenced later**, for instance:

```
In [3]:
x = 5+8

In [4]:
x

Out[4]:
13
```

Running the cell containing `x` before the cell `x = 5+8` will cause a `ValueError` as `x` has not been assigned any value yet. This is due to Python interpreting the code line-by-line before compiling and executing therefore variables **must** be assigned before being called. Therefore the cell `x = 5+8` must be executed first, then the cell containing `x`. More on variable assignment and Errors later...

A **jupyter notebook** can be started through loading the **anaconda navigator** up after installation and launching the **jupyter notebook** from there.

Alternatively, a **jupyter notebook** can be started from the **command prompt** (Windows) or **terminal** (Mac OSX) after installing **anaconda** by typing:

```
jupyter notebook
```

This will load a webpage that will show your local filesystem.

From here you can either start a **new** notebook or **navigate** to where the '**.ipynb'** files are located on your machine. The extension '.ipynb' stands for 'iPython notebook'.

Installing and setting up PyCharm

First navigate to the PyCharm downloads link (linked above) and download the correct package for your machine. Make sure to pick the **community version** of PyCharm.

Secondly install PyCharm by clicking on the **installer** and following the dialogues.

Once PyCharm has installed open the application. Make a **new folder** somewhere for your scripts to go and then click **File > Open** and navigate to that folder and a new project should start.

Now click on PyCharm's **Preferences/Settings** tab. Navigate to **"Project: _Project_name_"** and click on **Project Interpreter**.

Click on the Python 3.x note that should appear if you installed Python correctly before.

Test that everything is working by going **File > New**, clicking on Python File and entering the name Hello, World.

In the first line of the file type `print('Hello, World')`.

Either right-click and press Run or press the green 'Run' or 'Play' triangle in the top right of the screen.

If everything is working correctly, `Hello, World` should have printed to the Python console at the bottom of the screen.

Anaconda Distribution

The **Anaconda distribution** is a distribution of Python that contains pre-packaged with many useful libraries for data science and scientific computing including **SciPy, NumPy, Pandas, Scikit-Learn, TensorFlow, matplotlib** and many others as well as **Spyder IDE**, the statistical **programming language 'R'** and **Jupyter notebooks**.

Therefore, **Anaconda** is a very useful tool to have even if your preferred IDE is not Spyder. Downloading and installing the **Anaconda** pacakge is very easy and simply requires navigating to this link and following the **installation instructions**.

Once **Anaconda** is installed, open Anaconda and then launch Spyder. Write `print('Hello, World')` to test that everything was successful.

Part 1.3: Support and documentation

Python documentation

Part of the appeal of Python is that it is an **open source language** and the usefulness of it is well documented. The documentation for **some base functions and uses** of Python can be found here.

I strongly advise taking note of where this documentation is and **having it to hand in future whilst programming**.

Package documentation

While using the many **packages/modules** that have been written for use in Python programs (NumPy, Pandas, SciPy, matplotlib etc.) it is important to take note of the **documentation for any packages** you are using. Hopefully these should be your **first go-to** if you find something in your code not working the way you expect it to when using a package.

Navigate to the respective packages website and search for the specific function/class you are trying to use and try to figure out what is not correct. The 'traceback' on your IDE should also provide some clues as to why your code is not working.

Stack Overflow

If all else fails and you cannot understand why your code is not working OR if you have a specific question about the best way to implement your code, Stack Overflow is your best friend.

Stack Overflow is a **computing forum** where various questions are asked regarding how to **implement** all sorts of code into programs. Before asking a question, it is **highly** likely that your question may have already been answered, a simple search on Stack Overflow may provide the answer you are looking for.

Be aware when reading answers on Stack Overflow, the *exact* implementation shown in the answer **will** almost certainly need to be altered to fit your own **use case**.

Alternatively you can ask your own question, when doing so make sure your query is a minimal working example or MWE. This means that your question is as direct as possible about the exact problem you are facing and what you expect the code to be doing without including any extra complexities that make it difficult to reach the correct solution.

In [ ]:

```
print("Time to start coding!")
```