1. An Introduction to Python and Jupyter notebooks

This is the first Jupyter notebook associated with the PHYS10008 unit. The purpose of this notebook is to introduce you to Jupyter notebooks and how they work. When you have completed this notebook you should be familiar with:

- the basic concept of a Jupyter notebook,
- the way in which text and code fragments are written in cells in the notebook,
- how to enter simple text into 'text cells',
- how to enter python code into 'code cells',
- how to run, clear and restart Jupyter notebooks,
- how to recognise coding errors and simple strategies to resolve them,
- how to introduce python packages into your code,
- how to execute simple 'calculator like' snippets of python code in a notebook,
- how to use the math package.

1.1 Structure of a Jupyter Notebook

If you are reading this, then you have already managed to start a Jupyter notebook. The first thing to note is a Jupyter notebook is divided into cells. There are two main types of cell that we will be concerned with:

- 'Markdown' cells that contain text and notes (this cell is an example of a Markdown cell),
- 'Code' cells that, in this unit, contain the python computer code.

We will see how they are used in the following sections.

1.1.1 The Menu bar

At the top of the notebook you will find two menu bars which you use to edit and control the function of the notebook. The top menu bar consists of several tabs that contain all the commands that can be used to modify the notebook. The second line, consisting of clickable icons provides shortcuts to some of the common notebook commands, including saving the notebook ('floppy disk symbol'), adding cells ('+') and cut, copy and paste functions. The other icos allow cells to be moved within the notebook and control the execution of the code in the cells or the whole notebook. If you hover, with the mouse, over each icon its function will be shown.

1.1.2 Creating Markdown and Code cells

New cells may be added to the notebook using the '+' button on the menu bar. This will by default create a code cell immediately below the current cell. A code cell may be recognised as it has the form

In [n]:

where n represents the current sequence or execution number of the python code.

IMPORTANT. This sequence number does not refer to a line number in the underlying code. If you execute the same cell again the sequence number will increment by one. The order in which the cells are executed in a notebook does not have to be sequential (top to bottom), so you could execute a cell at the end of a notebook, followed by one at the top. Hence, this number shows you the order in which *you* have executed the cells. This can be very confusing and we will discuss in detail later.

To make a code cell into a markdown cell, go to the menu bar and find the button marked 'code' with a pull down list. You then select the 'Markdown' option and you will see the code prompt disappear. You can type text into this box and as you get more experienced you will learn more sophisticated ways to format it.

For example, as you will see later, with a little knowledge of the $L_A T_E X$ text processing language you can even write equations such as,

$f(x) = \int x^2 dx.$

We will leave describing how to do this until later.

IMPORTANT. Markdown cells may in in two states, *editable* or *viewable*. In order to change the contents of a markdown cell you need to be in the editable state. Double-clicking a markdown cell in the viewable state will change it to the editable state. If the cell is in the editable state then it can be changed to the viewable state by either, clicking the 'Run' icon on the menu bar, or by, \< *CTRL*>\< *ENTER*> on the keyboard. When the changed to, viewable, the text you have typed is processed to required, more readable form in a way similar to the *HyperTextMarkupLanguage (HTML)* used for web pages.

When you run the notebook you may also find that the notebook creates, for single line commands, additional, non-editable cells that contain the single line output from the code. These may be recognised as they have the form:

Out [n]:

1.1.3 Running the notebook

When you load a Jupyter Notebook for editing it immediately starts an *instance* of the python *kernel* . The kernel will execute any python commands that it receives, in the order it receives them. (This is also true of standalone python consoles such as Anaconda or Idle). When the notebook is first loaded _no_ python commands have been executed and no variables will have been set. However, unnless cleared, the notebook may show the results of previous calculations.

Any cell in a notebook may be 'Run' at any time using the 'Run' button on the menu bar or on the cell itself. There is no specific order in which they need to be executed.

As we have noted (1.1.2), when run, a markdown cell changes its state from 'editable' to 'Viewable'.

For a Code cell the python code in the cell is executed.

IMPORTANT. There are some very important points you should note when running code in your notebooks:

- the code cells in your worksheet may be executed in any order with the Run button,
- if a cell has not been executed, no sequence number will be shown,
- when a code cell is executed its sequence number, in the square brackets is shown,
- the notebook remembers the results from previously executed cells in the order they were executed.

To use successfully the Jupyter Notebook, you need to understand how to control the kernel. The 'Kernel' tab on the menu gives you several options

- Interrupt - this will stop any code currently executing, but nothing else is changed. You can carry on executing cells afterwards. You may use this option if your code gets stuck in a loop (we'll discuss this later),
- Restart - A new kernel is started. All previous variables, functions, etc are cleared. The notebook can run code again but the output (Out[] cells, error messages etc.) from the previous kernel remains in the notebook until overwritten.
- Restart & Clear Output - as above, the kernel is restarted but in this case all previous output is cleared,

- Restart & Run All - the kernel is restarted and the code blocks in the notebook are executed in order.

  The last option is the best way to finally verify that your code is working so you should work on ordering your code blocks to be sequentially executed.

IMPORTANT. You will be asked to complete assignments as Jupyter Notebooks. These will be autograded using a special format Jupyter Notebook that can be used with *nbgrader* system that you will hear about later. It is essential for these assignments that the notebook runs properly using the 'Restart & Run All" option.

You may also quickly validate your notebook with the 'Validate' button to pick up basic mistakes (but not runtime errors).

1.2 Editing Jupyter Notebooks

1.2.1 Reordering cells in the notebook.

You can re-arrange code and text in the notebooks cells using the cut & paste function on the menu bar. It is also very easy to re-order the cells in the notebook using the 'up' and 'down' buttons on the menu bar. These will move the current cell (code or markdown) up or down on step in the notebook respectively.

1.2.2 Editing Markdown cells

You have seen that when you open your Jupyter notebook you may find the markdown cells in either their 'editable' or 'Viewable' state. To make a viewable editable you should double-click somehere in the cell. To make an editable cell viewable you should execute the cell using the 'Run' button or \< CTRL>\< ENTER>.

You should try editing the following markdown cell by entering you name at the point indicated. You can also see how headers and sub-headers are created using the '#', '##', '###' symbols etc. If you look at the markdown cells in this and other notebooks you will see other ways in which text may be formated, i.e. as **bold** or *italics* or as equation in the $L_A T_E X$ format using $ LATEX FORMAT $. We will look at some of these options later, but you will learn a lot by looking at the markdown cells when they are in their editable state.

Try editing the cell below:

Please enter your name here:

Enter some text here:

Once you've entered your text click on the 'Run' button on the toolbar at the top of the notebook.

Hopefully, if all has worked you will see your name in the text and the equation appearing in proper mathematical format.

1.2.3 Editing and executing python code cells.

Now let's run our first bit of python code in a Jupyter code cell. We'll start with the traditional exercise, "Hello World". To do this click on the cell below and press the 'Run this cell' button (the arrow and vertical bar) at the beginning of the cell. Remember the format of a code cell has 'In [ ]:' the left hand margin. If there is a number in the square bracket it indicates the cell has already been executed and where it is in the sequence of cells you have already executed. Note, if you restart the kernel this cell will have no sequence number.

In [ ]:

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

Error messages

You should see that the code does not seem to have done what we wanted. This will certainly not be the last time you find an error in your code and as you can see the error messages are not always as informative as you might hope! Error messages can be very frustrating and it can be easy to go round and round in circles trying to fix them. If you are not sure what is happening then copying the error message into a web search may help explain the error. This is what I found with a quick google search on "SyntaxError: EOL while scanning string literal" message:

"In Python, the error SyntaxError: EOL while scanning literal string occurs when the python interpreter reaches the end of the line while searching for a string literal or a character within the line. The syntax error eol while scanning literal string in python is due to the missing quotation in the string, typing mistakes generally."

This is more helpful and maybe we can see that we didn't close the quotes for "Hello World". Before we do anything further note the number, _n_ in the In [n] for the code cell. Now go back to this cell and edit the line to add the extra " that you need after World. Hopefully you will see the error disappear when you run the cell. Note also that number In [n] has increased by one indicating that we have executed this cell again.

Discussion boards are also a good way of finding out what error messages mean. There is a Blackboard discussion board associated with this unit. If you find an error message you can't resolve you may find the answer on the discussion board already, if not, may post a query that I or maybe one of your fellow students may be able to answer to explain how to resolve your problem.

1.2.4 Saving Jupyter notebooks

Before going any further you should save this notebook using the 'floppy disk' symbol in the menu bar or by usinf the 'File' tab. When you save the notebook all the changes you have made in the text and code cells will be saved, for example, your name will be saved where you typed it and the quotation mark you added to the code cell will saved as well. It will also save the results that you obtained when running a cell - including any error messages and the execution number for each cell.

Note. Jupyter notebooks have an extension *.ipynb* if you need to search for or recognise them on your computer.

If you wish to save a 'clean' version of your notebook you may wish to restart the kernel & clear output before you save it. Alternatively, you may wish to restart the kernel and Run All if you wish to save all the results (perhaps if the code takes some time to run).

Note. When loading a Jupyter notebook a new kernel is started, So, you might find on reloading, the notebook that you thought was working gives different results or errors if your code cells are not in the sequence you thought!

Finally, a Jupyter Notebook is a live document, as you change it the way it works will change. You may periodically wish to save separate versions that you know work so you can 'roll-back' to a working version if you can't fix your working version. It is also a good idea to keep regular backups on a different computer.

The code cell below shows the working code if you were unable to correct the cell above.

In [ ]:

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

1.3 Getting started with maths in python

You have already run some python code earlier (Hellow World) - congratulations. For the rest of this notebook you will try some simple, one line, python examples.

Execute the code cell below and make sure that the answer is as you expect.

In [ ]:

```
6.5*9+3
```

It is very easy to run python code in this way. Indeed it is like using a pocket calculator. You may use the simple operators + (add), - (minus) *(multiply) and / (divide). Powers may be taking with the ** operator, i.e. 2** 3 is equivalent to 2^3 . The operator preference is the same as you should have learnt at school. You can control the operator preference by the suitable use of brackets i.e. in the expression* 2(3+2) the addition is done before multiplication.

As an exercise you should type in some simple calculations using these operators in the code cell below. Verify, with a caclulator or otherwise that the answers are as you expect. Note, how the sequence number increases after each calculation.

In [ ]:

```

```

1.3.1 Maths libaries, packages and modules

Hopefully, you will not have found any errors in the exercise, or of you did, that you have been able to resolve them on the internet or through the discussion board.

For physics we clearly need to move beyond the basic operators and to use, at least, the common functions (for example trigonometric functions) that you will find on a scientific calculator. You may be surprised to know that these additional functions are not found in the basic python kernel. In most computer languages this would be something like *sin()* . Try running the code cell below that tries to use a trigonometric sine function.

In [ ]:

```
3+2**5+sin(0.2)
```

The error message clearly states that 'sin' is not recognised. We might try sine( ) or other combinations but none will work. Although basic python is very limited its strength lies with the ease in which more many more, sometimes very advanced, functions can be included in the code.

This is achieved by importing extra "packages" and "modules", sometimes generically called 'libraries' at the beginning of the programme. Hence, at the beginning of all but the simplest python programmes you will see a series of lines in which specialist packages and modules are imported into the programme. The NameError that you see above is a common sign that you have not imported the package or module correctly, or that the function does not exist in the package.

There are many packages available for python and the simplicity in which they can be imported into your code is one of the reasons why the python language is so popular. The difficulty is normally making sure that they are installed on your python implementation. We will not cover installing packages on python systems (such as Anaconda) but this too is normally quite straightforward. We will introduce the most common scientific packages in this unit.

1.3.2 Importing packages and modules

The syntax for importing packages can often be confusing so we will look carefully at several examples, using the 'maths' package as an example.

Try the following

In [ ]:

```
import math
3+2**5+math.sin(0.2)
```

Hopefully this should have worked and given you an answer - you might like to check on your calculator. Especially note that in computer programming languages the argument to trigonometric functions is in radians and not degrees unless specified otherwise. Try changing sin(0.2) to cos(0.2) or tan(0.2) and you should see the code still works.

IMPORTANT. Always remember to use radians as the argument for trigonometric functions.

It may seem a bit odd that we need to explicitly write math.sin for what seems a fairly obvious and common function we would want to use. For example why can't we use sin() as we tried originally?

The reason is that it is perfectly possible to have a program that uses different packages, written by different people, who happen to choose the same name for a function that carries out two quite different calculations. If we don't specify which package contains the function we want to use python cannot decide for itself! For example, suppose we have two packages, pack1 and pack2, that both contain a 'sin' function. This code fragment,

import pack1
import pack2
pack1.sin(0.2)+pack2.sin(0.2)

makes sense, but if we didn't specify the 'pack1' or 'pack2' how would the program know what function to use?

1.3.3 Importing parts of packages

Some packages can be quite large and importing lots of unused code into your programme has implications for speed and efficiency. Hence, if we know what module or function we are going to need we can import only the modules we need.

Restart the kernel (this means no code has been executed yet and the maths package we installed earlier is no longer imported, see 1.1.3) and try the following code snippet.

In [ ]:

```
from math import sin
print(sin(0.2))
```

There are two things to note. Firstly, we only imported the 'sin' function. Try changing sin to cos and you will find it gives an error (remember to restart the kernel and clear all outputs). Secondly, we don't need to include 'math' in front of the sin() to execute the function we require. The code certainly looks less cumbersome.

We have used the 'print' function again, this time to print out the value of a variable. The print function is especially important if we run multiline code from which we wish to output values at different places. We will see it used again shortly and the reasons why should become more apparent. In the meantime, remove the print() function from the code cell above and exceute the sin(0.2) only. Note the difference in the output. You should see that our Out [n]: prompt has returned.

It is important to realise that you can't import two functions from different packages with identical names in this way. For example, the following import statements in the same code would produce an error in python

from math import sin
from mymath import sin

1.3.4 Importing all the functions from a package.

It would be exceedingly tedious if we had to import each trigonometric function in this way. If we wish to use all the functions in the maths package without the math.func() syntax. The way to do this is shown below.

Restart the kernel and clear all outputs and try the following:

In [ ]:

```
from math import *
print(sin(0.2))
print(cos(0.2))
```

In this case we have imported all the functions in the math package (the meaning of the *) without naming them individually. This can be helpful for the standard math package or other commonly used packages but the method should be avoided for the majority of the packages you wish to use in your code as errors will be produced if identical function names appear in different packages. It also avoids bloating your code.

You might also note the use of the 'print' statement in the above example. If we don't do this the output will only be shown for the last line executed. You might like to check this by editing the cell.

1.3.5 How do I find the names of the functions included in the package?

If you are lucky you may find full documentation for all the functions in a package (its attributes). The python website, www.python.org (http://www.python.org) is a good point of reference but it can be difficult to find what you are looking for! For example, the details about the math package may be found at : https://docs.python.org/3/library/math.html (https://docs.python.org/3/library/math.html) . You may wish to try to get to this page by searching on the main web site. You will soon realise how extensive the python language is!

The following page has links to the most important python packages:

https://docs.python.org/3/library/index.html (https://docs.python.org/3/library/index.html)

It is also possible to get the attributes of any *object* in python using the dir() function. An example that generates this list for the maths package is given below. How many of the functions to recognise?

In [ ]:

```
import math
dir(math)
```

1.4 Summary of this Notebook

In this notebook you will have become familiar:

- with the concept of a Jupyter notebook,
- how to control the notebook by restarting the kernel,
- the methods for creating markdown and code cells an the differences between them,
- with the methods for entering and running simple code snippets for basic calculations,
- with the method for incorporating packages into your code to extend the core python functons,
- with the math package needed for doing caclulator like calculations in python

Notes.

You may like to incorporate your own notes or code fragments in cells below. Alternatively. you might like to create you own notebook of tips and hints.

In [ ]:

```
```

Type *Markdown* and LaTeX: α^2