Python lab, lecture 1

Welcome to the labs for MATH1058, Operational Research I and Mathematical Computing. We are going to introduce Python, which will accompany you throughout the module (and in the assessments!).

You will need to open `spyder`. On the bench PC, go to the Start menu, All Programs, Programming Languages, Anaconda3 (64 bit), and then choose `spyder`. You may try searching for it but there may be multiple versions installed: you need to ensure that the version you use has Python version 3, not version 2.

You may use your own laptop if you want. Python is free: follow the instructions on Blackboard for how to install on your own machine. This may take 15 minutes or so, so please *don't* do it right now (although you may want to download the installer on campus - it is not small).

First steps

Once you have opened `spyder`, we will first look at the *console* in the bottom right part of the screen. That allows us to type in Python commands and get an *immediate* response. Let's use it like a calculator:

```python
1 + 1
```

```
2
```

```python
1 / 2
```

```
0.5
```

```python
(1 + 2.3 * 4.5) / 6.7
```

```
1.6940298507462686
```

```python
3**2
```

```
9
```

```python
print("Hello", 1 + 2)
```

```
Hello 3
```

Things to note: the `**` command raises one number to the power of another. The `print` function displays something to the screen.

Using the console is fine, but has problems. We want to keep our work for future re-use. We want to systematically build up a lot of code. Neither of these is easy within the console. Instead, we can use the *editor* on the left half of the screen. This acts like a word-processor, allowing us to type in commands that we can then use or run later.

In the editor, remove any existing text and type in some of the commands you've previously used:

```python
1 + 1
1 / 2
print("Hello", 1 + 2)
```

Save the file in a sensible location (your filestore, or the Desktop) under the name `lab1.py`.

We then want to run the commands in the file. To do this, choose "Run" from the Run menu, *or* press the big green play button on the toolbar, *or* press `F5`.

The output you see should look like:

```python
1 + 1
1 / 2
print("Hello", 1 + 2)
```

```
Hello 3
```

This shows one key difference between files and the console: only output that is *explicitly* `print`ed appears on the screen.

Exercise

Break for a quick exploration. For example, explicitly compute $6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1$ in the console. Then do the same in the editor, printing it out with explanatory text.

Variables (and tab completion)

Variables

We want to be able to store results of calculations in ways we can re-use. For this we define variables:

```python
x = 3
print(x)
```

```
3
```

A variable is a *label* that's put on a thing somewhere in the computer's memory. Think of it like a post-it note with `x` written on it. The above lines locate `3` in the computer's memory and attach the label `x` to it. Then, when asked to `print` the value of `x`, it `print`s the value with that label on it.

In Python we can assign "things" of almost any type to avariable and Python doesn't care:

```python
x = 1.2
print(x)
x = "Hello"
print(x)
```

```
1.2
Hello
```

This is not true in all programming languages, so take care.

There are certain rules and conventions for variable names. To keep it short:

- always start with a letter;
- only use lower case Latin letters, or numbers, or underscores;
- in particular, never use spaces or hyphens (which can be interpreted as a new variable or a minus sign respectively).

Tab completion

To save time in the console you can access previous commands by pressing the "up" arrow cursor key. You can then edit the command you typed. To find a previous command that started with a certain letter, first type that letter and then press the "up" key.

To save time and reduce errors in both console and editor, particularly when using long variable names, you can use the TAB key to auto-complete variable names (and later function names etc). For example, define the variable

```
width = 2
```

Then type the first two (say) letters, `wi`, and press TAB. You will *either* see it automatically complete this to show `width`, or you will see a list of options starting `wi` from which you can choose.

Exercise

A rectangular box has width 2, height 3, and depth 2. Create a variable for each. Compute the volume of the box, assigning that to a variable. Print the result with explanatory text.

Functions and import

We won't get very far with just basic algebraic operations. We want to compute more complicated mathematical things. Let's try computing the $\sin$ of some number:

```
print(sin(1))
```

```
-----------------------------------------------------------------------
----

NameError                                 Traceback (most recent call
last)

<ipython-input-12-40b0bf07e290> in <module>
----> 1 print(sin(1))


NameError: name 'sin' is not defined
```

Python has given us an error because it doesn't know the $\sin$ function. The core part of Python, in fact, knows very little.

We can get access to lots of additional functions (and variables, and other things) using the `import` statement. We're going to `import` the "numerical Python" package `numpy` that we will use often for this course. We can then *use* the contents of this package - its variables and functions - in our own code. So try:

```
import numpy
print(numpy.sin(1))
print(numpy.pi)
```

```
0.8414709848078965
3.141592653589793
```

We see that, to use anything from within the `numpy` package, we type `numpy`, then the dot or full stop, and then the name of the function (like `sin`) or variable (like `pi`) that we want to use.

To call a function, like `print` or `numpy.sin`, we pass along any arguments (the thing to `print`, or the number to `sin`) within round brackets `()`. If there are multiple arguments we separate them with commas (as in the `print` examples).

We can use `import` on *our own* files: it works exactly the same as on other Python packages. For example, make sure that your file `lab1.py` has the line

```python
width = 2
```

somewhere within it. Then open a new file and save it as `lab1v2.py`. Make sure it's saved in the same directory as `lab1.py`. In the new file type

```python
import lab1

print("The width from lab1 is", lab1.width)
```

The output should be `The width from lab1 is 2`.

Tab completion again

When you `import` packages you can usefully use TAB completion to rapidly find out about its useful functions. For example, type `numpy.co` and press TAB to see its $\cos$ and $\cosh$ functions, amongst many others.

Exercise

Import the `math` library. Find, using the help or TAB completion, how it computes factorials, and use that to check your calculation of $6!$.

Defining your own functions

Using other people's functions is essential, but we will also want to define our own. Let us define a simple function to start:

```python
def my_add(a, b):
    """
    Add two numbers.
    """
    return a + b
```

The Python keyword to *define* a function is `def`. The thing that comes next is the name that we will assign to the function: in this case `my_add`. As the name is a label to something that lives within the computer's memory, this should obey the same rules as variables.

The arguments to the function are then a comma-separated list between round brackets `()`. This is in the same format as calling the function, but we are inventing the variable names to refer to the input within the function. So, however the user calls the function, the first argument that is passed in will be assigned the label `a` within the function itself.

Finally there is a colon `:` to end the line. That says that whatever follows is the content, or body, of the function: the lines that will be executed when the function is called. All lines within the function to be executed **must** then be indented by four spaces. `spyder` should start doing this automatically.

Note that in most programming languages (e.g. C, C++, Java) indendations *do not* carry any semantical meaning. Keep in mind that in Python *they do*!

The three quotes are *documentation* for the function: they have no effect. However, **from a perspective of code usability and maintainability, any undocumented function is broken**. We can see the documentation by using the `help` function:

```
help(my_add)
```

```
Help on function my_add in module __main__:

my_add(a, b)
    Add two numbers.
```

We then include all the commands with the function that we want to run each time the function is called. Once we have a result that we want to send back to the place that called the function, we `return` it: this sends back the appropriate value(s).

We can now call our function:

```
print(my_add(1, 1))
print(my_add(1.23, 4.56))
```

```
2
5.789999999999999
```

Exercise

Define a function that computes the volume of a rectangular box with input arguments `width`, `height`, and `depth`. Check that it works as in the exercise above.

Assessed lab exercises

For future labs and for weekly work there will be exercises set. To complete those exercises you have to write functions. These will be tested *automatically*: you will provide your functions to a software system which will automatically `import` your file and call the functions within in, checking that they `return` correct output for specified input. You will need to get used to writing functions in files and testing that they work as intended.

*When you are asked to write a function that computes something in the assignments, that function must **return** that something, rather than just print it on screen!*