University of Sunderland

Technology Management for Organisations

Workshop 1

Jupyter Notebook and PIP environment

Aims of the workshop

The aim of this session is to introduce you to the Python Interpreter and Jupyter Notebook environment, which we've briefly seen this week. This workshop will serve as an introduction to Python, putting into practice all that we've covered this week.
It is important to take time to understand these basic concepts, and explore what they can do.

Please see 'Useful Information' below on how to lookup certain Python functionality. The concept behind this workshop is about discovery, and experimentation surrounding topics covered so far.

Feel free to discuss the work with peers, or with any member of the teaching staff.

Useful Information

Throughout this workshop you may find the following useful.

Python Documentation

https://docs.python.org/3.8/

This allows you to lookup core language features of Python 3.8 as well as tangential information about the Python Language. We mentioned this at the end of the introductory lecture.

Note: Depending on the Python version you have installed, the documentation may not be appropriate. Ensure you're on the correct version. E.g Python 3.8 in the case here. You can change this at the top of the page.

Jupyter Notebook Basics

Jupyter itself offers some basic documentation for people new to the editor. These can be found on https://nbviewer.org/github/ipython/ipython-in-depth/blob/master/examples/Notebook/Notebook%20Basics.ipynb

Jupyter can also use markdown cells for text input to describe things. If you wish to annotate each cell as to which Exercise it belongs to, you may find https://nbviewer.org/github/ipython/ipython-in-depth/blob/master/examples/Notebook/Working%20With%20Markdown%20Cells.ipynb useful.

Reminder

We encourage you to discuss the contents of the workshop with the delivery team, and any findings you gather from the session.

Workshops are not isolated, if you have questions from previous weeks, or lecture content, please come and talk to us.

The contents of this workshop are not intended to be 100% complete within the session; as such it's expected that some of this work be completed outside of the session. Exercises herein represent an example of what to do; feel free to expand upon this.

Running A Jupyter/IPython Notebook

A jupyter notebook is a server which runs on the local machine. It will use whichever directory it is invoked within as the root folder. It is then able to see all sub-folders within that. This is currently installed on the lab machines; however, if you are wishing to install this at home, you may need to follow pip instructions for installing jupyter.

Jupyter Notebook can be launched from the command line by invoking:

```
jupyter notebook
```

This will start a server, typically on http://localhost:8888/tree, and should automatically open a new tab in the browser.

Jupyter notebook server uses a token, generated at launch, to authenticate access. By default, the notebook server is accessible to anybody with network access to your machine over the port, with the correct token.

If you are asked for a password or token, you can always invoke:

```
jupyter notebook list
```

This will provide you a list of currently active Notebook servers (you can run multiple, on different ports, for different folders), along with their token, all within a single URL.

PIP and packages

PIP is python's packaging tool, it enables the installation of libraries. If you tried to import a library (e.g numpy) when starting your notebook, you may have noticed an error. This is because, by default, python doesn't include pandas as part of the default libraries. Anything not included by Python needs to be installed before it can be utilised.

Python has a rich community backing, with several libraries available over at PyPI which users can install. Users can even make their own libraries and host them online for others to use.

If you require to install python packages on-campus, the image we have is not an administrator. Therefore installing packages to the system level is not permitted. However, pip provides a --user flag for installing packages to the user-profile instead.

Most packages can be installed by using:

```
pip install packageName
```

and for the user-flag:

```
pip install --user packageName
```

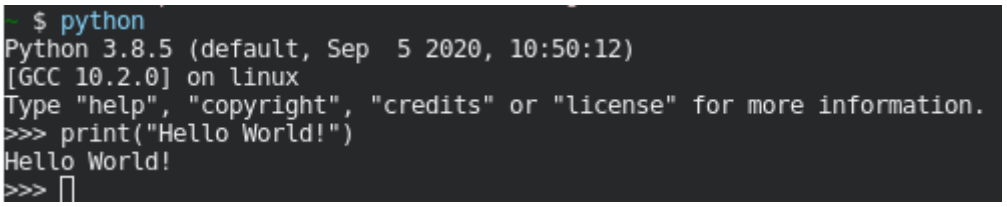
Remember, you can always use `pip help install` for specific help information on installing.

Using the Python Interpreter

Remember, you can launch the interpreter by **opening a command prompt** and typing ***python***

Note: Here I'm using a Python 3.8.5 installation (seen at the top of the screenshot), you may have a more recent version of Python. As long as the version is greater than 3.7 everything in these workshops should be okay.

Exercise 1: When prompted, type **print("Hello World!")** this should enter **Hello World!** On the line immediately afterwards. This is the output of the expression.



Congratulations! You've just run your first Python command.

Exercise 2: We can use **type()** to check the type of any value. We simply provide the value to type, between the parentheses.

E.g **type(932)**

Try this in the interpreter. What do you get out?

Hint: We don't need print in this case with the interpreter, because it is outputting it for us. If we executed this code in a **.py** file, this would not be the case. Any expressions which have a value will be spit back to us.

Using your knowledge of the other basic types we introduced, check some examples of these values with **type()**.

Exercise 3: We covered casting a value from one type to another. An example of this might be treating the integer 932 as a float.

- Write an expression which casts 932 to the type string (*shortened to str*)
- Verify the type output from the step above. *Hint:* We don't need to use any variables yet. Remember, **type()** can take any expression which provides a value.

Exercise 4: For the following expressions, what are the types of the two inputs values (three in the case of the last example), and what is the type of the result of executing the expression. Try to answer these first without writing code, then write some Python to verify if you are correct.

1. 3 + 4
2. 3.0 + 4
3. 37 % 7 ** 2
4. 'bob' + 'cat'
5. "bob" / "cat"
6. "banana" + "na" * 20

Do any of these combinations surprise you?
Try other operators which you know, and some other data types.

Exercise 5: You are given the following mathematical expression

12 + 28 / 7 * 2

Using your understanding of PEMDAS:

1. What would be the result of such an expression? Which numbers are calculated together first?
2. What would be the overall type of the result?
3. How might we use parentheses to make this expression less ambiguous? Remember, the expression must still return the same value!

Exercise 6: Create some variables (name of your choosing) to store the values 42.0 and 97.

1. Calculate addition, subtraction, divide, and multiply operators on these. (E.g a * b)
2. Using the expressions from 6.1, create a new variable which is assigned that expression. (E.g ***product = a * b***). Do this for all your variants.
3. Print your new variables. (E.g ***print(product)***)

Exercise 7: Pick your favourite expression from Ex 4. Take any variable from Ex 6 which you made (E.g **product**), and **re-bind** this to your favourite Ex 4 expression (E.g = 37 % 7 ** 2). Print the new value of your Ex6 variable (E.g print(product) again); observe the change in value from the end of Ex6.

Example:

```
# My Ex 4 expression for subtraction
my_subtract_expression = 42.0 - 97
print(my_subtract_expression)

# Re-bind my_subtract_expression variable to my favourite expression from Ex6
my_subtract_expression = 37 % 7 ** 2

# Print the new value
```

```
print(my_subtract_expression)
```

Using the Jupyter Notebook

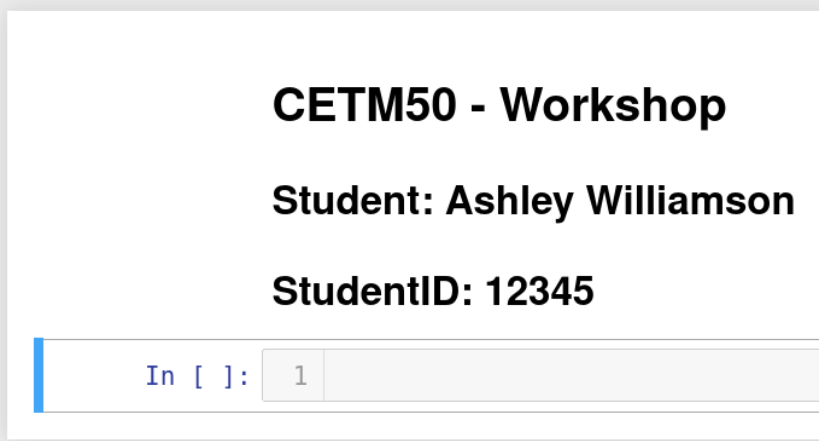
Exercise 8: Launch a Jupyter Notebook, and explore the interface and controls. Look at the useful information section above under Jupyter Notebook Basics.

Try executing the python exercises above (Ex 1 - 7) in Jupyter Cells, and noting their output. It is recommended that you make a fresh notebook for each workshop we run. This way you can reference back to previous works. Each exercise should be able to be placed in its own cell.

Exercise 9: Add a Markdown cell as the first cell at the top of the book. Enter the following, filling in your student ID in place of <StudentID>, likewise for name:

```
# CETM50 - Workshop 1
## Student Name: Ashley Williamson
## Student ID: <StudentID>
```

Use Shift+Enter, to execute this markdown cell and force it to display.



Exercise 10: We can use comparison operators on strings, just as numbers. However, these have some different behaviours. Comparing two strings “ada”, and “bill” will check the first letter of each and will use alphabetical ordering to determine which one is ‘first’ (ie. Lower than the latter).

1. In a fresh cell, try executing “ada” < “bill”

If the first letter is the same in both, it will then check the second letter to make the outcome. If the second letter is also the same, it will keep progressing until out of letters.

1. Compare “ada” with “adb”

If the two words have different lengths, but otherwise are identical as we go along, python will favour the shorter of the two strings.

1. E.g “ada” < “adalovelace” -> True

Exercise 11: Using ***input()*** ask the user to input a name. Let’s check if the name is equal to your name using an equality test!

E.g

```
their_name = input(“What is your name?: “)
if their_name == “ashley”:
      print(“I’m called that too”)
```

Exercise 12: Remember that we can perform logical operations on entire boolean expressions themselves. Modify the boolean expression in Ex11 to also accept if they enter your last name too!

E.g

```
their_name = input(“What is your name?: “)
if their_name == “ashley” or their_name == “williamson”:
      print(“I’m called that too”)
```

Test out providing other names, your first name, and your last name to see what the program outputs. Remember you can re-run the same cell that you have selected.
Notice how each side of ‘or’ is a boolean expression which evaluates to True/False in itself.

Exercise 13: Given the following input table, complete the truth table for the following expression:

(a and b) or c

a	b	c	a and b	(a and b) or c
True	True	True		
True	False	True		
False	True	True		
False	False	True		
True	True	False		
True	False	False		
False	True	False		

False	False	False		

Exercise 14: You are then told that the expression is incorrect, and should be:

(a and b) or not c

How does this change the overall boolean expression. Add a column to your table, and fill in the results of this new expression.

Exercise 15: We can use the expression **x % 2 == 0** to check if a variable is even. Write an **if statement** to check if a given variable is even. It should print out "The variable is even!" if it passes this condition. Remember, modulo provides a remainder. Even numbers can fit 2 into themselves perfectly with nothing left over, hence why we check if the remainder is 0.

Exercise 16: How could Ex15 be modified to also print "The variable is odd!" when the variable is odd? Can a variable be anything other than even or odd?

Exercise 17: Produce some code which first checks if a variable is divisible by 2, then checks if it is divisible by 3, otherwise prints that it is "not divisible by 2 or 3".
Would we use two separate if statements, or a single if, elif statement?
To help you decide, consider what should be output if we tested the number 6 through this system.

Exercise 18: You are given the following list of numbers:

```
A = [ 5, 2, 9, -1, 3, 12]
```

Using the counter method from lectures slides, create a while loop which will go over each item in this list.

1. Print each item
2. Check if the item is -1, if so, immediately break out of the loop.
3. Calculate the square of the element, and print it.

Exercise 19: Replace the while loop counter method from Ex18 with for loop and range(n)

Exercise 20: Repeat Ex19, but using the direct iteration version. E.g No indexing.

Exercise 21: Using the list, A, from Ex18, write a loop which will sum all of the numbers up, and print the resultant sum.

Exercise 22: The mean number can be calculated by taking the summation and dividing by how many elements there are. Using the output of Ex21 (the summation), create a new variable for the mean number of the list. (E.g [1, 3, 5] # Sum is 9. Mean is 9 / 3 which is 3.)

Exercise 23: You are asked to find the minimum, and maximum of a list of numbers. You are provided with code to determine the minimum.

```
my_items = [ -5, 3, 72, 1, 9, 24, -3]

minimum_so_far = None
for elem in my_items:
      if minimum_so_far == None or elem < minimum_so_far:
            minimum_so_far = elem

print("Minimum Value: ", minimum)
```

Modify the above program to also calculate the maximum so far, outputting it in a similar way.
In this example we used None as an initial value as it's not a number. If the value is None, then we know we haven't checked a single number yet, so our first value is always going to be our highest and lowest in that case!

Exercise 24: Execute the following:

```
A = 5
B = A
B = 10
print(A) =>  5
print(B) => 10
```

When we rebind B to be 10, A is left completely unmodified. This behaviour is unique to basic data primitives such as the basic types we have looked at. However, let's see what happens if we do something similar to a List.

```
A = [ 1, 6, 2, 7 ]
B = A
B.remove( 6 )
print( A ) => [ 1, 2, 7 ]
```

This behaviour will hold for Lists, Dicts, and most 'objects' in Python. **B is a reference to A.** We only have **one** copy of our list floating around, but now we have multiple variables capable of pointing to it.

If we wanted to make a copy of the List, such that we can modify them independent of each other we have to [:]. E.g

```
A = [ 1, 6, 2, 7 ]
B = A[:]
B.remove( 6 )
print( A ) => [ 1, 6, 2, 7 ]
print( B ) => [ 1, 2, 7]
```

This is known as 'slice' notation. Effectively [:] says to take the entire list, and it makes a copy of this.

Exercise 25: Using B (your **copy** of A), append some of your favourite numbers to the list.

Exercise 26: You are given the following to complete:

```
results = []
bought_cost = [ 10.0, 12.55, 17.99 ] # Price you pay to the
manufacturer for an item
sale_price = [ 12.0, 11.50, 20.0 ] # Price you sell it for

for i in range( len(item_cost) ):
      <Calculate the difference between sale_price and bought_cost>
      <Add these to the empty list, results>

print(results)

<Calculate and print the total profit/loss overall>
```

Note: Take care when calculating whether we made a loss, or a profit on our sale!
Adv Hint: There is such a function called zip, which can help iterate over two lists at once, should you wish to look this up. It is left out of here as to not overwhelm.

Exercise 26: You are given the following dictionary:

```
student_records = {
      “Ada”: 98.0,
      “Bill”: 45.0,
      “Charlie”: 63.2
}
```

You are then given two separate lists, one with names, and another with grades. You are to go through these lists, and insert the students correctly into the dictionary.

```
student_names    = [“Neva”,       “Kelley”,  “Emerson”]
student_grades   = [72.2,         64.9,      32.0 ]
```

1. Ensure both lists are the same length: How might we check for this?
2. Using a for loop, iterating in range sufficient for all the elements
    a. Each iteration here represents an index. E.g Index-0 is Neva and her grade.
    b. Insert these into the dictionary, student_records. Remember `dictionary[“theKey”] = value`. “theKey” can be replaced with any expression which evaluates as a string. It could be a variable, or another expression, or a string literal.

Exercise 27: Using the `for k, v …` code example from the lecture slides, iterate over your newly modified dictionary, showing the new entries.

The Extended Exercises are optional, and are offered as an advanced supplement for those who have completed the existing work and wish to expand on their knowledge and challenge themselves further.

Extended Exercise 1: Bubble sort is a naive sorting algorithm for taking an unordered list, and sorting them into ascending order. This is achieved by stepping through the entire list, comparing the current element to the next in the sequence. If the current element is greater than the following one, they are swapped in-place. The loop then continues until all elements are exhausted.
The effect of this is that the largest number will bubble up through the list to the top. Once we've reached the top, we start the comparison again for a second pass, third pass, fourth pass, etc.
This is repeated until the list is sorted. I.e no swaps have occurred that pass.

L = [9, 2, 12, 7]

Is 9 > 2 -> Yes. Swap them in-place. Move along by one.

L = [2, 9, 12, 7]
Is 9 > 12 -> No. Do nothing. Move along by one.

L = [2, 9, 12, 7]
Is 12 > 7 -> Yes. Swap them in-place. Move along by one.

L = [2, 9, 7 , 12]
We have run out of elements, start iterating from the beginning.

L = [2, 9, 7, 12]
L = [2, 9, 7, 12] -> SWAP
L = [2, 7, 9, 12]

Another pass
L = [2, 7, 9, 12]
L = [2, 7, 9, 12]
L = [2, 7, 9, 12]
No swaps have been made this pass -> terminate, the list should now be sorted.

Your task is to implement the above Bubble sort algorithm for a given list, L.