University of Sunderland

Technology Management for Organisations

Workshop 2

Functions, Scope, Default Values, List & Dict Comprehensions

Aims of the workshop

In Week 1 we looked at the fundamentals of computation, and introduced you to basic constructs used within Python. These consisted of basic data types, control flow structures (if, elseif, else), boolean expressions, and loops (for, and while).

This week we looked at building upon this by exploring functions, and how they can wrap common procedures that we might want to execute multiple times. We expanded upon the concept of “everything is an object” and looked at abstraction and decomposition and how we can use these when constructing and representing data in our programs to make code more reusable, readable, and maintainable.

Note - We will be utilising ePortfolios on Canvas from now-on to document progress and learning within Workshops. Please familiarise yourself with these in the appropriate helper section below.

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.

ePortfolios

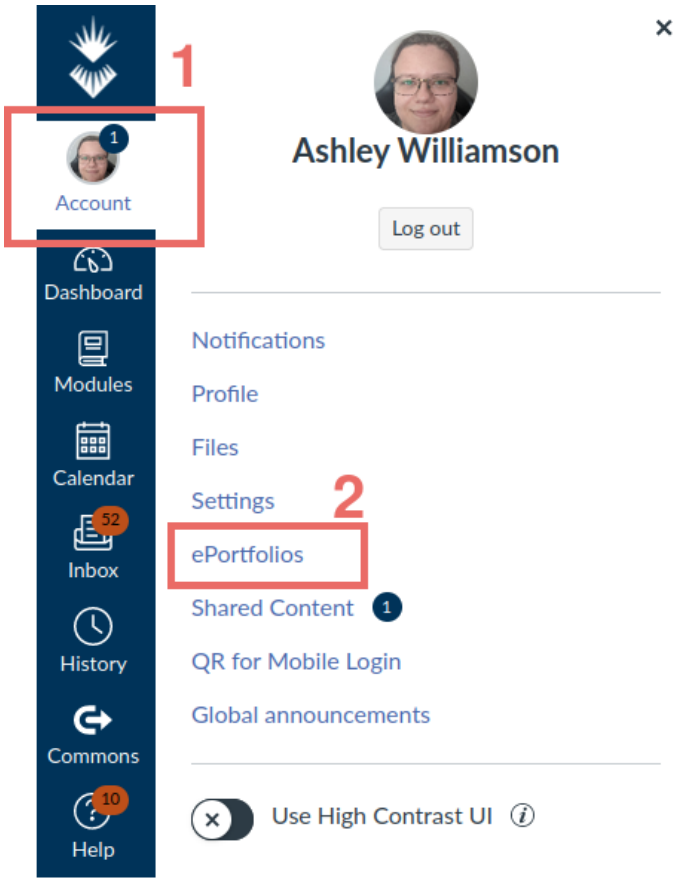
As part of the assignment for this module, evidencing your processes and learning will be a vital component. Therefore, now is a good time to introduce ePortfolios. These allow you to write and present work in a structured way which we can add to as we go through workshops.

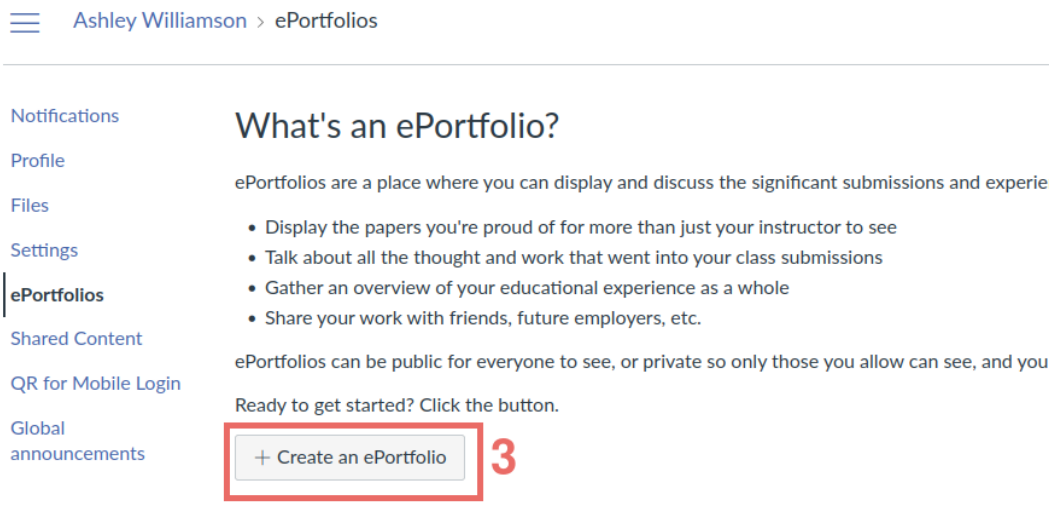
Each week you are expected to write a short page for that week's workshop. It should include reflective writing. What worked well? What didn't work so well? Any error messages that you struggled with? Did you fix them? What helped you in getting over that barrier?

You can think of this approach similar to a development diary, keeping track of your progress. You could even have entries in future weeks finally fixing something you've been struggling with, or finally understanding a concept that has eluded you.

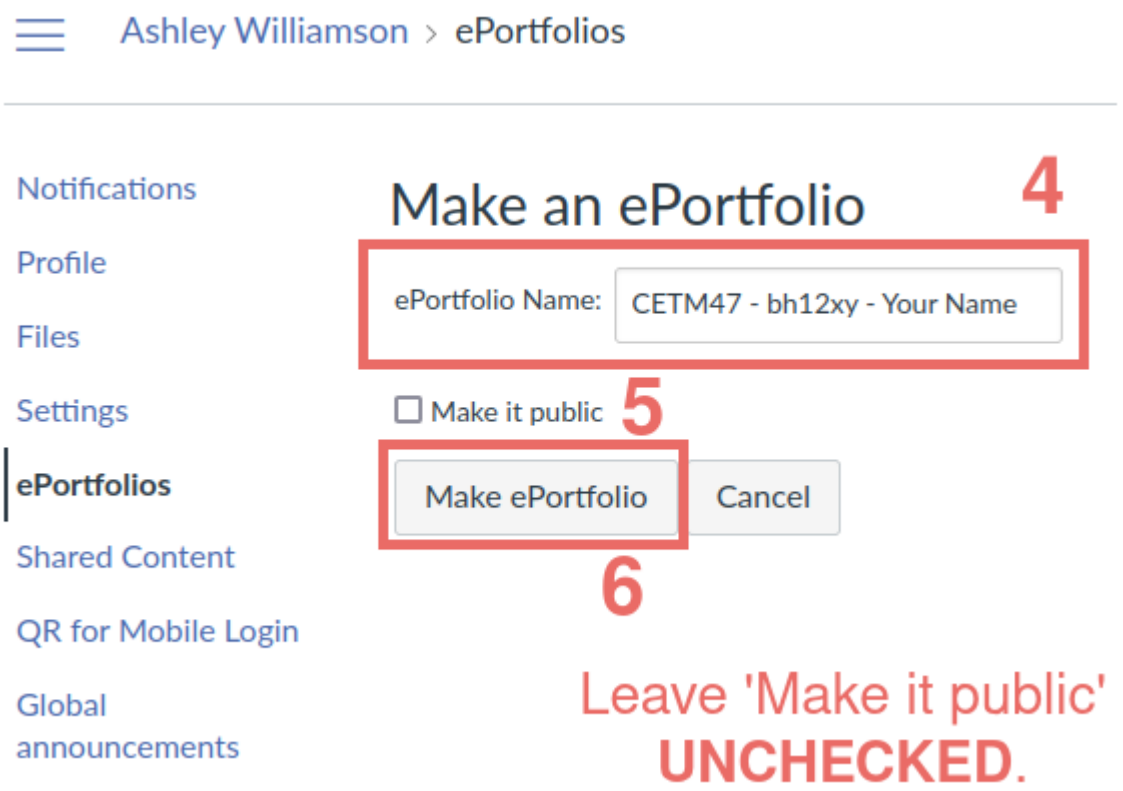
When writing your entries, **you should include screenshot evidencing to support what you are writing**. If you are talking about an error message, show it! If you had issues, and then solved them, show it broken and then working with the correct fix.

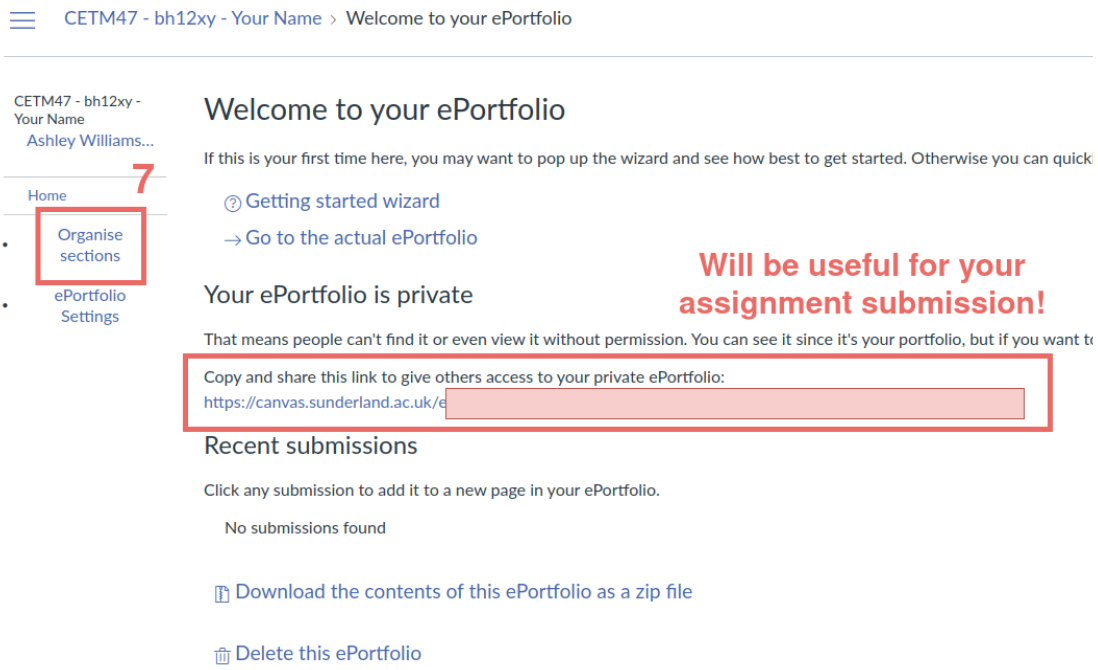
This type of reflective practice can help consolidate your learning, help you identify areas of improvement, and track your progress and approaches. Follow the steps below to get started.



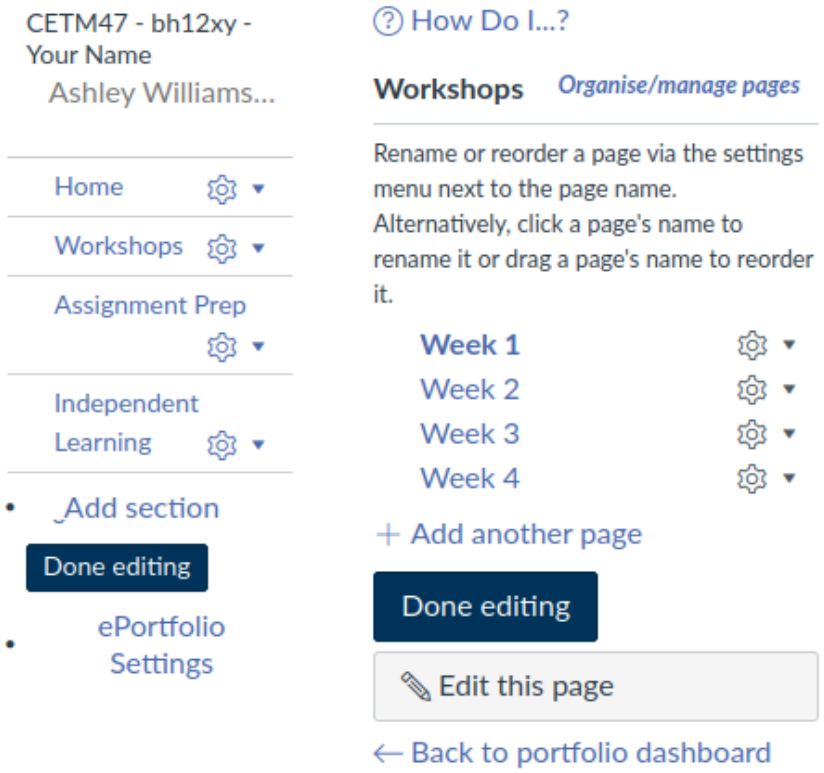


When creating your ePortfolio, you should give it a suitable name. Consider the below as a template. As you can make many ePortfolios, you should be descriptive about which module it is for, who you are (student number and name).





As a recommended structure you may wish to consider the below. Sections (left), and within the “Workshops” section, you could have a Page (right) for each week.



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.

Functions & Scope

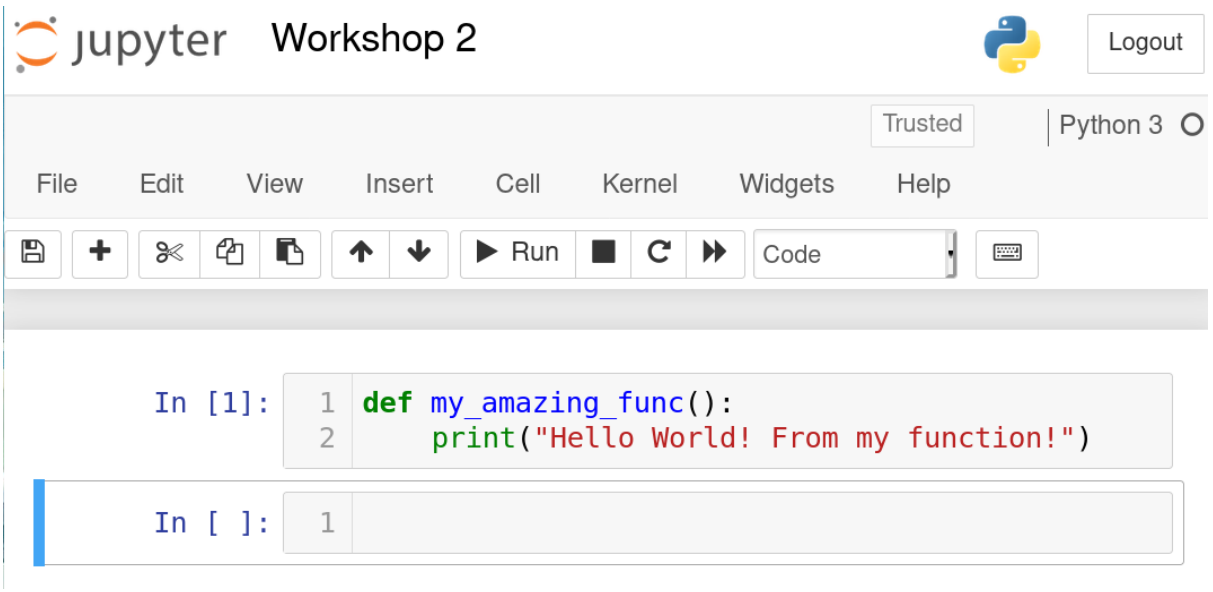
For this workshop, create a **new** notebook. Name this something memorable.
E.g “Workshop 2”

Remember, you can use Markdown cells to add some plain text description of the task. This might be useful to put the question above the cell.

Exercise 1: Let’s create a function to print something whenever we invoke it.

```
def my_amazing_func():
      print("Hello World! From my function!")
```

Once you have defined the above function in a cell, run the cell using **Shift+Enter**.
You should see something like the following



To view line numbers for each of your cells you can go to **View -> Toggle Line Numbers.**

We can now invoke (or call) our newly defined function in the following cell:

```
In [2]: 1 my_amazing_func()
Hello World! From my function!
```

And you should notice that the cell prints what we typed in the function definition.

Exercise 2: In the lectures we covered the ‘return’ of a function. By using the **return** keyword. First, let’s prove that by default a function without this keyword will return **None**.

In a separate cell (**Insert -> Insert Cell Below**), assign a variable to your function's output. We can do this like so:

```
variable_name = my_amazing_func()
```

Remember, you get to decide what to call your variable. Similarly, you get to define what you call your function. At the moment, these are toy examples so we're using names which might not mean much, but later on we should try to use variable and function names that make sense for what we are attempting to do.

After assigning a variable to your function's output: print the value, and type, of your newly made variable.

```
In [5]:  1 print( variable_name )
         2 print( type(variable_name) )

None
<class 'NoneType'>
```

From this, we can confirm that by default a function will return **None**, if we don't add a return statement in the function definition. Using our knowledge of types, we can check this and it does indeed return as **NoneType** which is one of the special primitive data types we covered last week.

Exercise 3: At the moment our function can only ever do one thing. It can only print the exact phrase we wrote for it.

We can modify our function to include **parameters**, this will allow us to pass values and variables to our function to use them in some way.

Let's modify our function to accept a string for printing. We can go back to our first cell in the notebook, and change the definition of our function. Previously, this function would print the string literal `"Hello World! From my function!"`. However, we now want it to directly print whatever we pass as an argument into this function.

```
def my_amazing_func( thing_to_print ):
    print( thing_to_print )
```

Note: Make sure you execute this cell once you have made the changes (**Shift+Enter**). You will now notice a new number next to the cell. Before this was **In [1]** and now it should be higher, such as **In [5]**. To Python, we have just re-executed our function code, as though we typed it all out again. If we were to run the remaining cells in the notebook, these should behave very differently now!

```
In [6]:
1 def my_amazing_func( thing_to_print ):
2     print( thing_to_print )
```

```
In [7]:
1 my_amazing_func()
```

```
---------------------------
TypeErrorTraceback (most recent call last)
<ipython-input-7-418be9069b2d> in <module>
----> 1 my_amazing_func()

TypeError: my_amazing_func() missing 1 required po
sitional argument: 'thing_to_print'
```

Uh-oh! Because we changed the definition of our function, the code we used before to invoke the function is now incorrect. Our function is expecting something to be given to it, and because we haven't done that it complains.

We can resolve this quite easily. We can provide it a string to use. Modify the erroneous cell to call `my_amazing_func("This is a test")` then run the cell again (**Shift+Enter**).

```
In [8]:
1 my_amazing_func("This is a test")
```

```
This is a test
```

Success!

Introducing, default values

Exercise 4:

Alternatively, we can tell our function that the parameter we defined is optional. That is, we can call `my_amazing_func()` and `my_amazing_func( "Beep" )`, and both would work without Python erroring.

Let's take a look at our function definition:

```
def my_amazing_func( thing_to_print ):
      print( thing_to_print )
```

We can do something like the following to tell Python that if thing_to_print is not provided, that it can take on a default value. This provides a default binding for our parameter within the function scope.

```
def my_amazing_func(thing_to_print = "The Default thing you want"):
      print( thing_to_print )
```

If we now invoke `my_amazing_func()`, missing out that argument of what to print. We should be able to observe the function using the default.

Implement these changes, making sure to redefine your original function, and re-execute the cells for the function definition, and your invocation cells. This should look like:

```
In [9]:
1 def my_amazing_func( thing_to_print = "The Default thing you want" ):
2     print( thing_to_print )

In [10]:
1 my_amazing_func()

The Default thing you want
```

Create a new cell beneath this, and try passing a string to your function. It should print whatever you pass it.

Introducing, libraries

We talked briefly about Python Modules, and why we might want to use libraries which abstract away complex functionality.
Python has some standard libraries we can import to help us with common tasks.

When dealing with more complex tasks, we typically want to use some behaviours/utilities which have already been defined and provided for us. This could be telling Python to wait 2 seconds before continuing, it might be telling Python to exit itself (if we've hit a stopping condition and want to exit our program without human input). We will be introducing more powerful libraries later on in the module (E.g Visualisations and Database Connectors).

Exercise 5:
An example of this is the **random** library. (https://docs.python.org/3.8/library/random.html).
Using this we can generate some random numbers.

To use this, we first need to tell Python where to get this functionality:

```
import random
# This will import the library random.
# We can use dot notation to access important functions it has for us!
```

Once this is imported, it is accessible via the name of the library itself, **random**. We can then access all the functionality via dot notation.

We only need to import this once! Typically we put all our import statements at the top of our Python file or Python Notebook.
E.g The first cell of our notebook will usually contain all import statements. As these need importing prior to using them.

Let's look at generating a random integer between some lower value and an upper value, inclusive. (https://docs.python.org/3.8/library/random.html#random.randint)

In a new cell put the following:

```
x = random.randint(0, 10) # Random Integer between 0 and 10 inclusive.
```

If we print the value of x, every time we evaluate this cell, I should get a new random integer.

```
In [12]:
1 import random
2 # This will import the library random.
3 # We can use dot notation to access important functions it has for us!

In [14]:
1 x = random.randint(0, 10) # Random Integer between 0 and 10 inclusive.
2 print(x)

3
```

Re-running the same cell (**Shift+Enter**):

```
In [15]:
1 x = random.randint(0, 10) # Random Integer between 0 and 10 inclusive.
2 print(x)
```

```
0
```

Exercise 6: Write a standard for loop to get a random number, and print it, the loop should execute **exactly** 20 times.

Scoping

Exercise 7: In this week's lecture we looked at scoping, and the concept of function scope We learned that a function has access to all the variables defined in the scope in which itself is defined.

Consider the following:

```
a = "Some outer scope string literal"
b = 42

def some_func():
      print(a, b)

some_func()
```

If we execute this, our function definition doesn't have any defined parameters and we do not provide it any arguments to execute on. The only place which it can possibly get a, and b, is from its parent scope - the global scope in this example.

This can be useful in some cases, but also dangerous. What if we moved our some_func definition to a completely different section of our code? Or change a, and b, in our parent scope?

Ideally, our functions should be **self-contained**, and shouldn't rely on bleed through from parental scope. This reliance from the parental scope can lead to unintentional side-effects.

Remember: some_func can access a, and b, getting their value. But it **cannot** change them from within the function! -> UnboundLocalError

The solution is to make the function require parameters to be passed in. If our function wants to print some variables, they should be supplied.

```
a = "Some outer scope string literal"
b = 42

# Now, by our definition, we know that this NEEDS two arguments passed
in.
def some_func_v2(a, b):
      print(a, b)

# some_func_v2() # This line would error "positional arguments"
some_func_v2( a, b ) # Explicitly pass a and b in.
```

Remember, that the function scope now makes a, and b binding it to whatever we pass in. The below code is equivalent:

In [11]:
```
a = "Some outer scope string literal"
b = 42
# Now, by our definition, we know that this NEEDS two arguments passed in.
def some_func_v2(c, d):
    print(c, d)

# some_func_v2() # This line would error "positional arguments"
some_func_v2( a, b ) # Explicitly pass a and b in.
```

1) a in global scope

2) c binds to a (in function scope only)

```
Some outer scope string literal 42
```

If we try to print(c) after our function call to some_func_v2(...), c should not exist. It was only ever defined within the scope of our function. We should expect an error.

```
def some_func_v2(c, d):
    print(c, d)

# some_func_v2() # This line would error "positional arguments"
some_func_v2( a, b ) # Explicitly pass a and b in.

print(c)
```

```
Some outer scope string literal 42

----
NameErrorTraceback (most recent call last)
<ipython-input-12-28029e514961> in <module>
      9 some_func_v2( a, b ) # Explicitly pass a and b in.
     10
---> 11 print(c)

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

More Useful Functions

Exercise 8: Let's write a function which can return something. In a new cell, define the following:

```
def always_4():
      return 4
```

Create a ***variable***, and ***assign*** it to the return of our function. (We will need to ***call*** our function to get anything back).

(Reference to: https://xkcd.com/221/)

Exercise 9: In our previous workshop we wrote some expressions for operations between two primitive types. E.g 5 + 7.

Write a function, with a suitable name of your choosing, which returns the additive sum of the two inputs. (Don't call the function **sum;** this is a protected keyword. Some of you may have experienced a problem with this in Workshop 1).

```
4 my_result = sum_two_numbers(5, 7)
5 print(my_result)
```

```
12
```

Exercise 10: Consider the following task. You are asked to write a function which accepts a list of any length as its input. This list will be of numbers which may contain -1 as a data element. The return of this function should be the index at which -1 was found.

For this example, you may assume that the input list will **always** have an entry which is -1 somewhere.

Example:

```
A = [ 5, 2, 9, -1, 3, 12 ]
indx_of_issue = find_negative_one( A )
print(indx_of_issue) # Should give me 3 (4th element in A).
```

Exercise 11: You find out later that the input list may sometimes **not** contain a negative one. What would be the output of your function in this case (as it is currently written)? And how could I check the `indx_of_issue` variable to determine whether I did find a -1 or not? **Modify your code** such that it prints the index if a -1 was found, otherwise it prints "There is no -1 here".

Exercise 12: Taking the procedure you used for Ex 26 from Workshop 1:

1. Create a function which accepts a list of names, and a list of grades.
2. The body of the function should initialise an empty dictionary, and combine the names and grades as the key and value (as in Ex 26)
    a. E.g it might look like this { "Neva": 72.2, "Kelley": 64.9, "Emerson": 32.0 }
    b. Remember: What if we had 100,000 entries! -> Write code to do the key: value mix for you, don't just write the dictionary literal itself.
3. The function should return this newly created dictionary.
4. Take the existing student_records dictionary and concatenate the return of calling your newly made function.

```
def get_combined_namegrades( names, grades ):
      # Initialise an empty dictionary.
      # Combine names with grades to make the dictionary
      # Keys are the names,
      # Values are the grades for the given student.
      # Return the dictionary

student_records = {
      "Ada": 98.0,
      "Bill": 45.0,
      "Charlie": 63.2
}

student_names = ["Teri", "Johanna", "Tomas", "Piotr", "Grzegorz"]
student_grades= [35.0, 52.5, 37.8, 65.0, 64.8]

# Invoke the get_combined_namegrades function
# passing the appropriate lists
# Use the return

# Concatenate this newly returned dictionary to our student_records.
# Print the updated student_records variables to show it worked.
print(student_records)
```

Note: Take care when copy-pasting from a Word Document. " and ' may be different, leading to syntax error.
Hint: We can **.update(...)** our dictionary with another.

```
26 # Concatenate this newly returned dictionary to our student_record
27 print(student_records)
```

```
{'Ada': 98.0, 'Bill': 45.0, 'Charlie': 63.2, 'Teri': 35.0, 'Johanna':
52.5, 'Tomas': 37.8, 'Piotr': 65.0, 'Grzegorz': 64.8}
```

Exercise 13: Taking our updated student_records, create a **dictionary comprehension** which will only include items with a grade >= 65. (I.e act as a filter). Assign this to a variable called **filtered_student_records.**
This should output the following once complete:

```
In [21]: 1 print(filtered_student_records)
```

```
{'Ada': 98.0, 'Piotr': 65.0}
```

Exercise 14: Create a function, which will take as input a grade as a float, and output a string which denotes the grade classification.
E.g

< 40.0 is a "Fail"
40.0 - 50.0 is a "Pass"
50.0 - 60.0 is "2:2"
60.0-70.0 is a "2:1"
> 70.0 is a "First"

```
def grade_to_classification( grade ):
    if grade < 40.0:
        return "Fail"
    elif ...
```

The above template is provided to help you get started. You will need to check values of the grade provided to determine what should be returned. A consideration once you've completed your if, elif, else conditionals is: "Have you covered every eventuality? Could I provide a grade which doesn't match one of your cases? Thus will return None."

Create a dictionary comprehension which will apply **grade_to_classification** for every value.
Hint: `{k: v ...` Would store the value, v, at the key k. We want to store the return of calling `grade_to_classification` on that grade (I.e `grade_to_classification(v)` in this case)

Hint 2: For determining if a value is within two ranges you will need to use the logical operators for boolean expressions from last week.

E.g `grade >= 40.0 and grade < 50.0`

If you print the result of this more complex dictionary comprehension, you should get the following:

```
{'Ada': 'First', 'Bill': 'Pass', 'Charlie': '2:1', 'Teri': 'Fail', 'J
ohanna': '2:2', 'Tomas': 'Fail', 'Piotr': '2:1', 'Grzegorz': '2:1'}
```

Exercise 15: You are given a list of just grades, and asked to calculate which grade gets which classification. Using your newly defined function for this, write a **List Comprehension** which will apply the function on every element.

```
more_grades = [0.0, 50.0, 49.9, 79.0, 101.0, 65.0, 54.2, 48.2, 78.9]
```

The output should be as follows:

```
['Fail', '2:2', 'Pass', 'First', 'First', '2:1', '2:2', 'Pass', 'First']
```

Exercise 16: The University administrators need a count of how many have failed. Modify your list comprehension to only get the classification for grades in the fail category (< 40) - This will output a smaller list. Secondly, using the new list returned from that list comprehension, obtain the length of this smaller list to give to the admins as a count.

Exercise 17: Suspicions have been raised that the IT system may have introduced an error into the grading system. The administrators want you to write a List Comprehension which will ***modify*** any values exceeding 100, to cap them at the maximum grade possible 100. **Returning the raw grades back (We're dealing with the marks, not the classifications).**

Hint: There were two types of conditional. One at the end for **filtering**, and one near the beginning for **modifying** values.

Output should look like this:

```
[0.0, 50.0, 49.9, 79.0, 100, 65.0, 54.2, 48.2, 78.9]
```

Classes

Exercise 18: Create the following empty class definition

```
class Student(object):
      pass # We need something on this line otherwise python complains
```

We can create a new Student Object from this; however, it doesn't do much currently.

```
alex = Student()
```

Exercise 19: Verify that alex is a Student object. Remember what happens if we **print** classes, or print their **type**?

Exercise 20: At the moment we can't really provide any student related data when making the student object. Let's add a constructor.
Define an empty constructor

```
class Student(object):
      def __init__(self):
            print("This gets called when I make a new student.")
```

If we execute the `alex = Student()` line again, we should now get something printed. This proves that `__init__` actually gets executed. Hint: Don't forget `self`

Exercise 21: What data do we currently hold for students? A **name** and a **grade**. Modify the constructor to accept two parameters, name, and grade.

Inside the constructor body, assign two attributes to self. (E.g `self.name = ...`) representing the two parameters you just put in the method definition.

What happens if we try to run `alex = Student()` now?

Exercise 21: Modify your instantiation of our Student object, to include a name and grade of your choice.
E.g We can give the variable alex, an actual name and grade (data attributes).

```
alex = Student("Alex", 99)
```

Remember, the variable name itself doesn't impact the data of our object!

I could say `x = Student("Alex", 99)` it's just a friendly/descriptive name to make the programmer's life easier.
Exercise 22: To prove this point, let's make a List directly (literally) with Student Objects. The only variable here is pointing to the List. Everything else is a raw data object (like 5, "bob", [], and now our newly defined Type Student).

```
some_students = [ Student("Alex", 99), Student("Rob", 35.0),
Student("Tasha", 70.0) ]
```

```
In [43]: 1 some_students = [ Student("Alex", 99), Student("Rob", 35.0), Student("Tasha", 70.0) ]
         This gets called when I make a new student.
         This gets called when I make a new student.
         This gets called when I make a new student.
```

Warning: Notice how I mixed integer grades and floats. This won't cause us a problem now, but in our constructor we may want to explicitly cast the incoming grade to be a float. We will look at error handling further in the module and what to do if something unexpected comes in instead.

Note: Feel free to remove the print line from the constructor, if it gets annoying.

I can index this list just like any list, and print the element:

```
print( some_students[0] )
```

```
In [44]: 1 print( some_students[0] )
         <__main__.Student object at 0x7fb6a9dad340>
```

Exercise 23: We already have written some code which will take a numeric grade, and converts it to a classification. Let's add a method to our Student class which can do this for us.

Just like how we got the Cats to speak based on the data attributes they hold, we can get this function to change behaviour based on the data its object holds. I.e the names and grades of each individual student object.

```
class Student(object):
      def __init__(self, name, grade):
          ...
      def get_classification(self):
            pass # Need this here for now.
```

When we made our function before, we had to pass it a grade to process on explicitly. However, with this method we already have access to a grade! With `self.grade`
The function definition needs `self` as this is required for methods of an object.

Replace `pass` with your grade classification logic that you wrote previously. Except instead of using `grade` use `self.grade`
Once this is complete, re-execute the class cell (updates our definition with python) (**shift+enter**). Re-execute all the other cells where we made objects (so they get this new method we've added) (**shift+enter**).

With a bit of luck we should now be able to get a student to directly give you their classification. Let's use our first student object that we made: alex.

```
In [62]:  1 print( alex.get_classification() )
          First
```

Note: I changed the name of the method in this case as we are wanting the classification of our student. Remember that we use classes for hiding away information we don't really care about. In this instance, do I need to know that the classification is derived from grade? No, I just want the classification of the student itself. E.g This could be a complex process of adding all of their module marks up, any late penalties, deductions, etc. As the user interfacing with this student, I just want their classification and the mechanism behind it can be unknown to me.

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.

Try to complete this without immediately searching Google. You have all the necessary knowledge and tools to solve this, it just needs applying. Remember, documentation is your friend, and you can read documentation on strings.

Extended Exercise 1: A palindrome is a string which, when reversed, is identical. Some example words which are palindromes are: “Taco cat”, “Stressed desserts”, “Anna”, “kayak”, “racecar”. They can be read forwards and backwards and remain the same.

Write a function which accepts a List of strings.
This function should return a List of True/False, where True means the entry at that index was a palindrome.

Example:
input_list = [“taco cat”, “bob”, “davey”]
output = palindromes(input_list)
-> [True, True, False]

Harder Challenge: The body of the function should be a singular List Comprehension