Discussion Group Leader should coordinate the task and report by December 4th 2 pm

Group 1 and 2

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
```

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.

<u>Exercise 26:</u> 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.

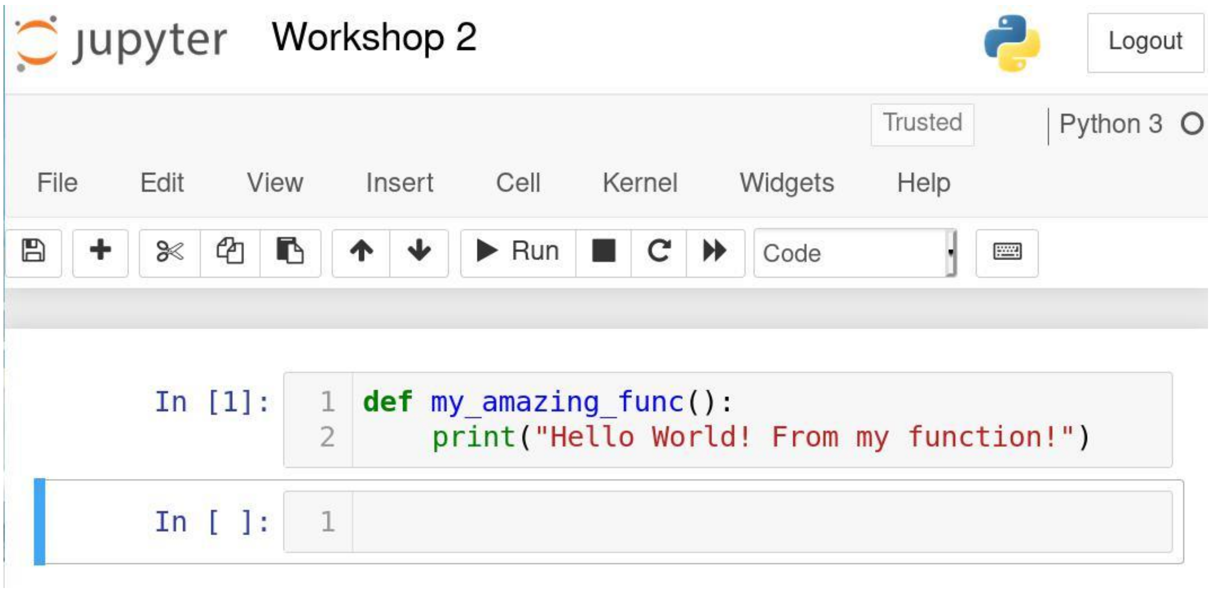
<u>Exercise 27:</u> Using the for k, v … code example from the lecture slides, iterate over your newly modified dictionary, showing the new entries.

Group 3 and 4

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

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

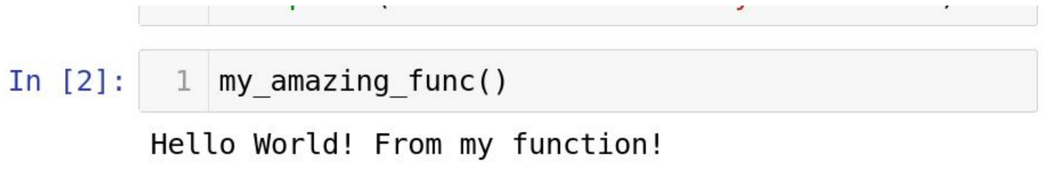
Once you have defined the above function in a cell, run the cell using **Shift+Enter**.
You should see something like the following



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

In [ ]:  1
```

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.

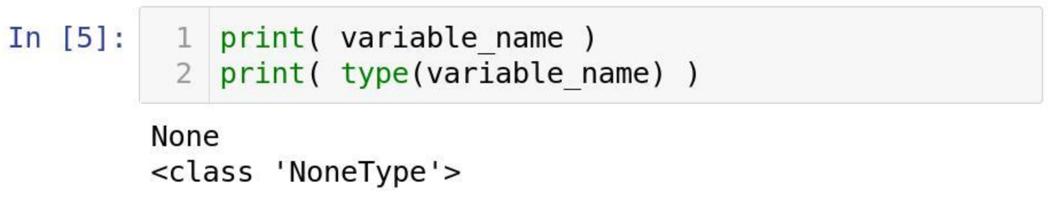
Exercise 2: In the lectures we covered the 'return' of a function. By using the **return** keyword. First, let's prove that by default a function without this keyword will return **None**.

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]:  print( variable_name )
         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.

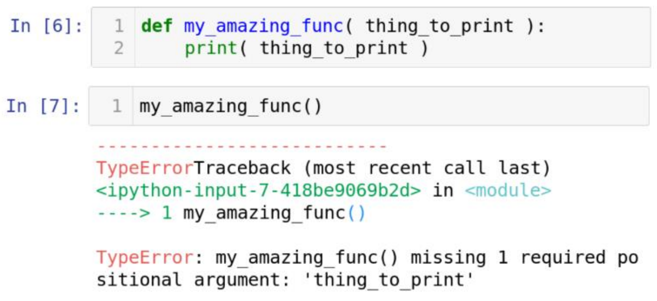
Exercise 3: At the moment our function can only ever do one thing. It can only print the exact phrase we wrote for it.

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 )
```

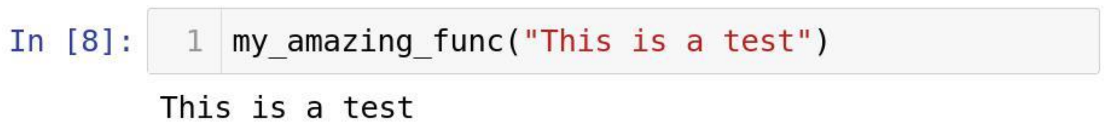
Note: Make sure you execute this cell once you have made the changes (**Shift+Enter**). You will now notice a new number next to the cell. Before this was **In [1]** and now it should be higher, such as **In [5]**. To Python, we have just re-executed our function code, as though we typed it all out again.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



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

In [7]:  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]:  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:

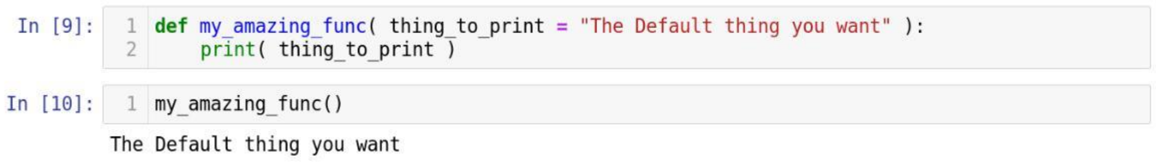
```
def my_amazing_func( thing_to_print ):
        print( thing_to_print )
```

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

In [10]: 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.

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.

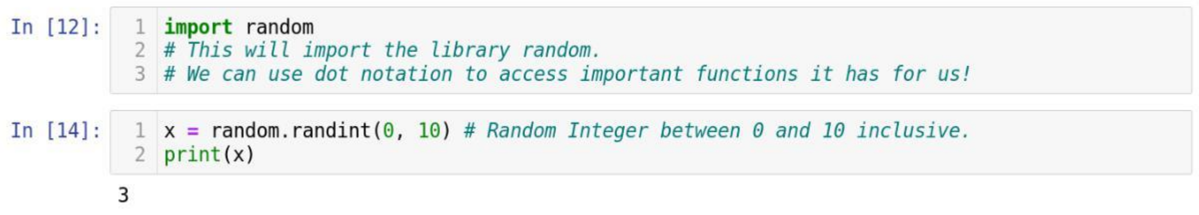
We only need to import this once! Typically we put all our import statements at the top of our Python file or Python Notebook.
E.g The first cell of our notebook will usually contain all import statements. As these need importing prior to using them.

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:

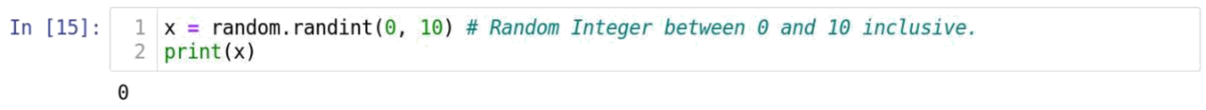
```
x = random.randint(0, 10) # Random Integer between 0 and 10 inclusive.
```

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



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

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

         3
```

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



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

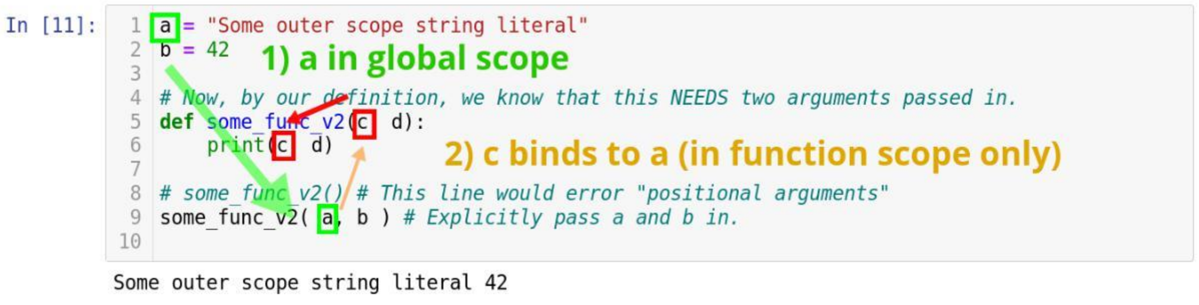
Ideally, our functions should be **self-contained**, and shouldn't rely on bleed through from parental scope. This reliance from the parental scope can lead to unintentional side-effects.

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.

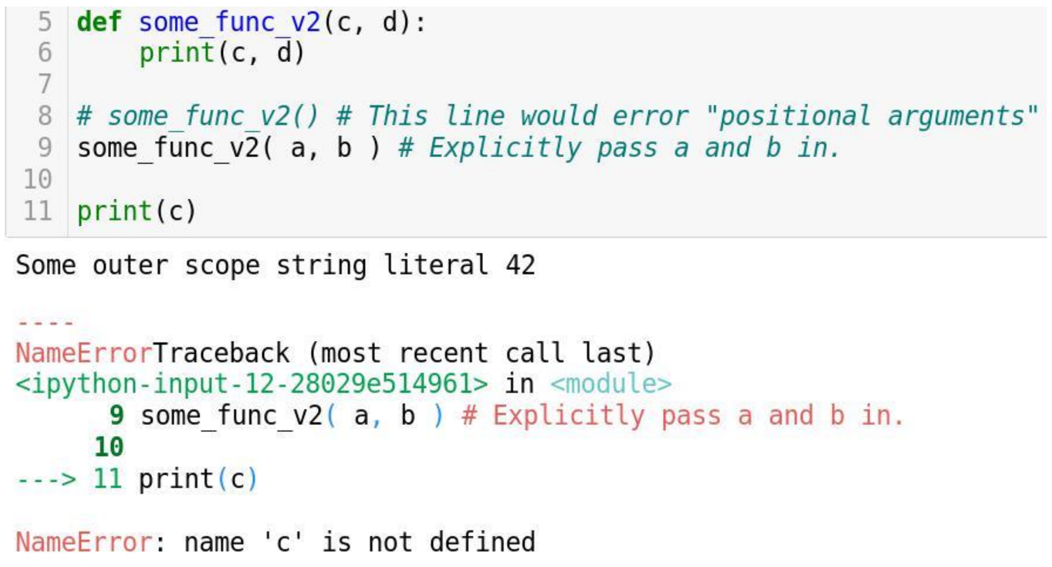
```
a = "Some outer scope string literal"
b = 42

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

2. 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.

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
```

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

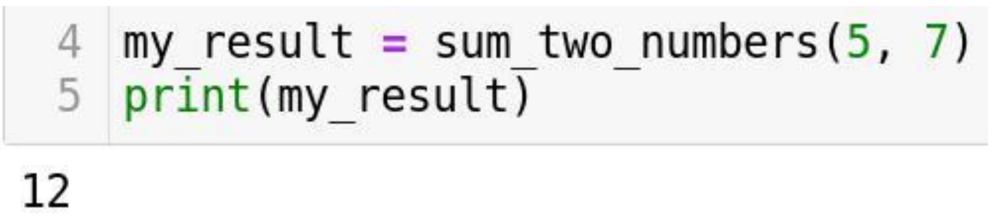
```
def always_4():
    return 4
```

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).



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

```
12
```

<ins>Exercise 10:</ins> 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:

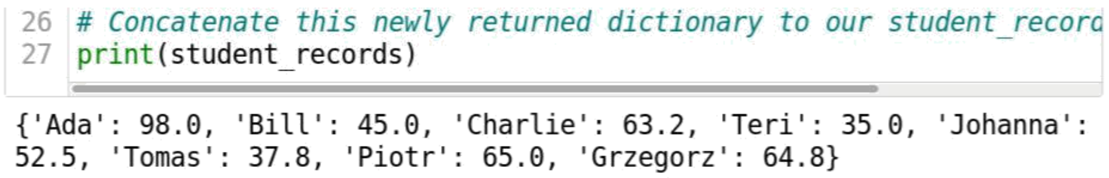
```
A=[5,2,9,-1,3,12]
indx_of_issue = find_negative_one( A )
print(indx_of_issue) # Should give me 3 (4th element in A).
```

<ins>Exercise 11:</ins> 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”.

<ins>Exercise 12:</ins> Taking the procedure you used for Ex 26 from Workshop 1:

4. Create a function which accepts a list of names, and a list of grades.
5. 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)
   E.g it might look like this { “Neva”: 72.2, “Kelley”: 64.9, “Emerson”: 32.0 }
   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.
4. The function should return this newly created dictionary.
5. 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]
```



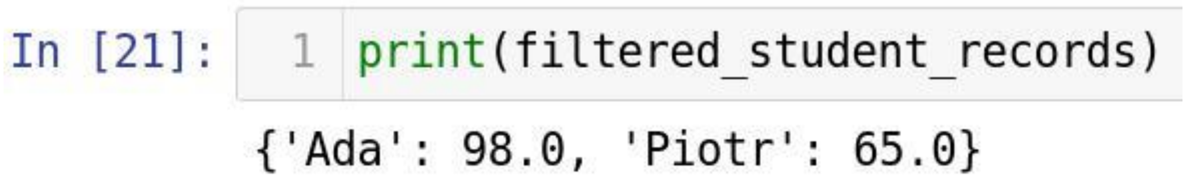
```
# Concatenate this newly returned dictionary to our student_recor
print(student_records)
```

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

```
# 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.

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]:  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) 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'}
```

<u>Exercise 15:</u> 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']
```

<u>Exercise 16</u>: 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.

<u>Exercise 17:</u> 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]
```

<u>Exercise 18:</u> 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?

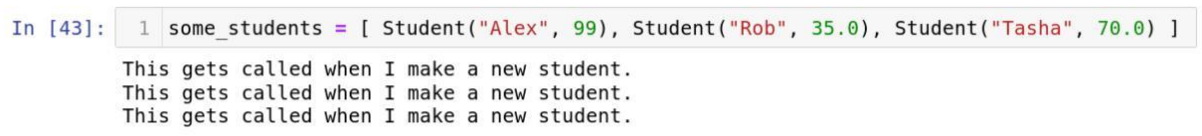
Exercise 21: Modify your instantiation of our Student object, to include a name and grade of your choice.
E.g We can give the variable alex, an actual name and grade (data attributes).

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

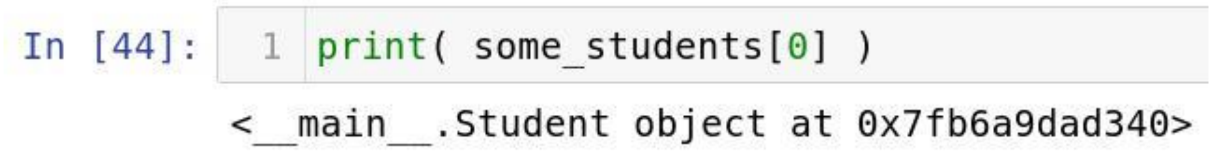
```
This gets called when I make a new student.
This gets called when I make a new student.
This gets called when I make a new student.
```

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]: 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.

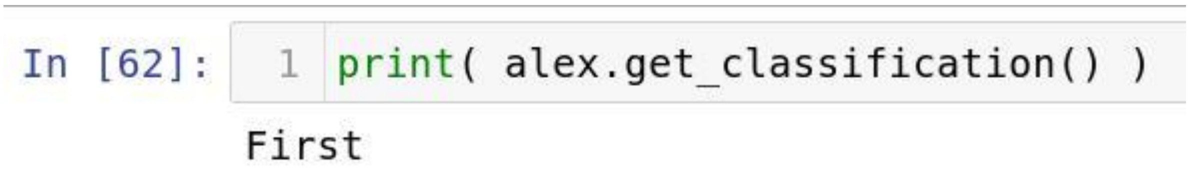
Just like how we got the Cats to speak based on the data attributes they hold, we can get this function to change behaviour based on the data its object holds. I.e the names and grades of each individual student object.

```
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.



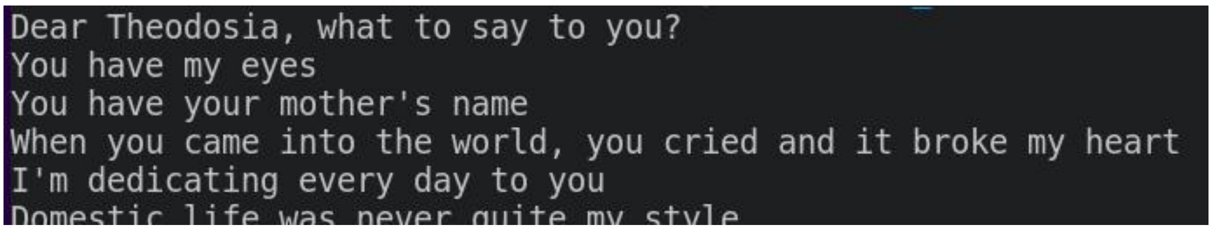
```
In [62]: print( alex.get_classification() )
First
```

Group 5 and 6

Reading Files

Exercise 1
Create a text file using notepad with song lyrics. “Dear Theodosia” from Hamilton (google this) and name this file **ex1_data.txt**.

Follow this example:



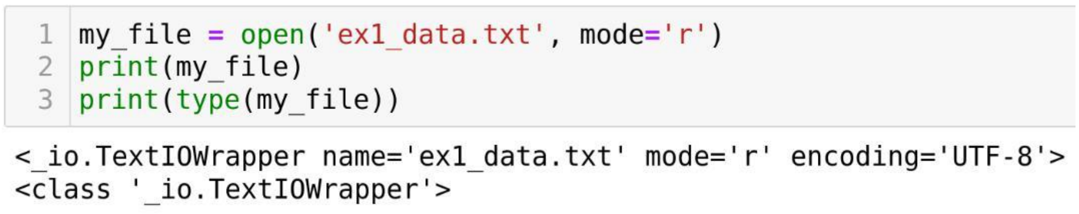
```
Dear Theodosia, what to say to you?
You have my eyes
You have your mother's name
When you came into the world, you cried and it broke my heart
I'm dedicating every day to you
Domestic life was never quite my style
```

Exercise 2
We can now use our notebook to read this data in via Python. We can use open() to open any file path we provide. Remember, open() returns a file object itself. Assign this to a sensibly named variable. For now, we will specify the file path as the local file path, and we will explicitly pass the argument mode='r' even though this is the default value.

Once we have this assigned to a variable, we can print the type of the variable (it’s an object), to see what it is.

```
my_file = open('ex1_data.txt', mode='r')
print( my_file )
print( type(my_file) )
```

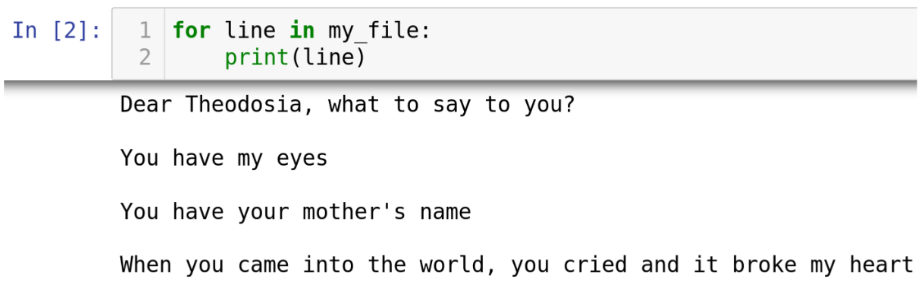
This should look something similar to the following:



```
my_file = open('ex1_data.txt', mode='r')
print(my_file)
print(type(my_file))
```

```
<_io.TextIOWrapper name='ex1_data.txt' mode='r' encoding='UTF-8'>
<class '_io.TextIOWrapper'>
```

Exercise 3
Let’s read some data out from this file object. We can use a for loop which iterates directly over lines in the file itself.
For each line in the file, print the line.

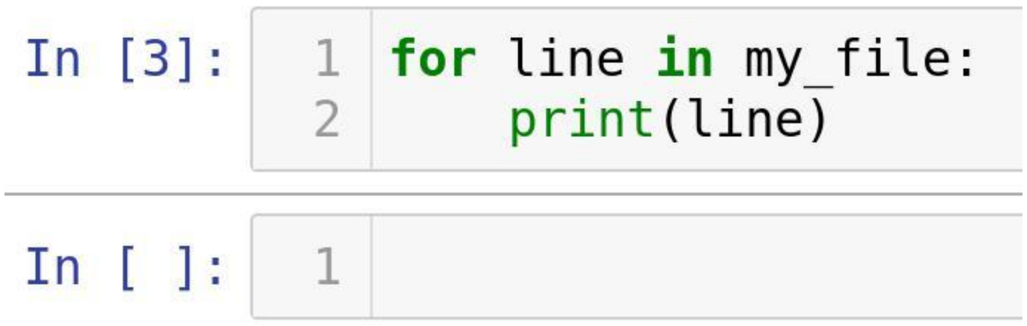


```
In [2]:  for line in my_file:
             print(line)
```

```
Dear Theodosia, what to say to you?

You have my eyes

You have your mother's name

When you came into the world, you cried and it broke my heart
```

Note: If things don't seem to print, read the beginning of Ex 4 and it might make sense.

Exercise 4
Execute the for loop again.
We have a problem with the previous exercise. If we try and execute the for loop again, just printing the line, we won't get anything out!



```
In [3]:  for line in my_file:
             print(line)

In [ ]:
```

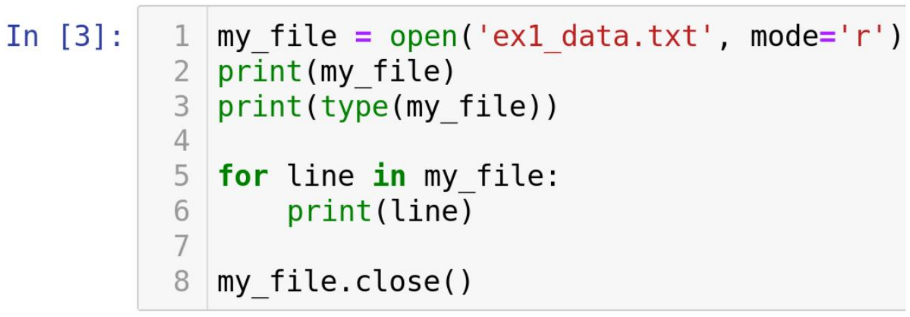
This is because we've already hit the end of the file. Python is doing some 'tracking' in the background which keeps track of where in the file we are. Once we iterate over the file once, we're at the end of it (in Ex3). When we try to iterate again, no new lines have been added to our file. Thus it will not print anything.

This isn't ideal. So everytime we want to iterate through the file from the beginning, we need to reopen the file!

Combine the Notebook cell which opens the file, with the iterating code from Exercise 3. From our File I/O lectures, we know that it's important to close a file once we've finished with it. Add that to the end of the cell.

To Merge cells: Select a Cell, go to Edit -> Merge Cell (Above/Below).

Once this is done we should have something which looks like the following:



```
In [3]:  my_file = open('ex1_data.txt', mode='r')
         print(my_file)
         print(type(my_file))

         for line in my_file:
             print(line)

         my_file.close()
```

This is good, as we know all of that code will execute in-order, at once. If they are in separate notebook cells we can technically execute them in any order! Now this cell will open, read, and close the file.

Exercise 5
Now that we have everything together, and it's not going to behave strangely when printing lines, we can begin to change our simple print expression to something more complex.

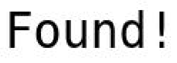
We can search for strings within other strings by executing the following boolean expression:

```
<string> in <other string>
```

E.g: In my example, I want to check if the line we're checking contains the word Theodosia. Note: "theodosia" != "Theodosia" (Capital letters are different to lowercase in Python programming!).

```
if "Theodosia" in line:
        print("Found!")
```

If I execute this cell again, it should open the file, iterate through it, and print "Found" for the line in which this comes up.

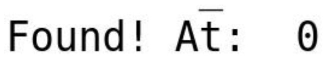


```
Found!
```

Depending on what is in your text document, search for some string which is present.

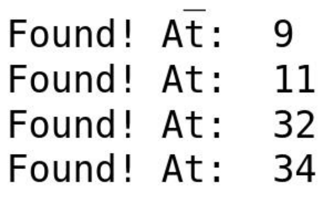
Exercise 6
At the moment, Ex 5 isn't very useful for us. I want to know the line number where this lyric comes up. We can use enumerate, covered in previous workshops, to provide line numbers alongside the elements themselves.

# Modify your file object for loop, make it use enumerate.
# Replace your print statement, to now print the line numbers which your string appears.



```
Found! At:  0
```

In my example, Theodosia only appears once, on line 0 (the first line).
If I wanted to find a more common word. I'm going to search for "We".



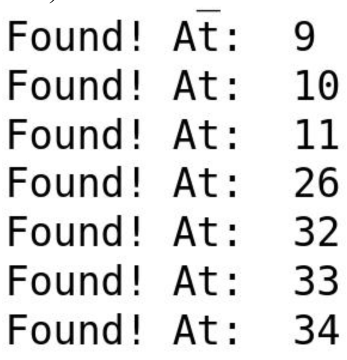
```
Found! At:  9
Found! At:  11
Found! At:  32
Found! At:  34
```

Note: I can call the method lower() on a string to convert it all to lowercase. This might make checking easier. I can check for "We" and "we" with a single check now.

Example:

```
if "we" in line.lower():
        print("Found! At: ", line_no)
```

I've now found some more entries which I missed before! (I was only checking for "We" prior and missed all the cases of 'we').



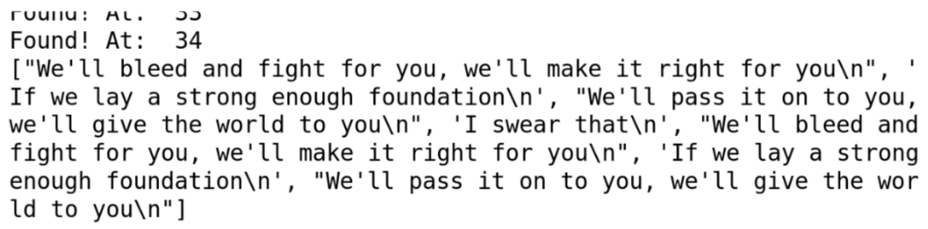
```
Found! At:  9
Found! At:  10
Found! At:  11
Found! At:  26
Found! At:  32
Found! At:  33
Found! At:  34
```

Exercise 7

Instead of just printing the lines where we've found "we" (lowercase and uppercase variants), let's append these lines to a fresh List. This is similar to what we did with numbers in Workshop 2.

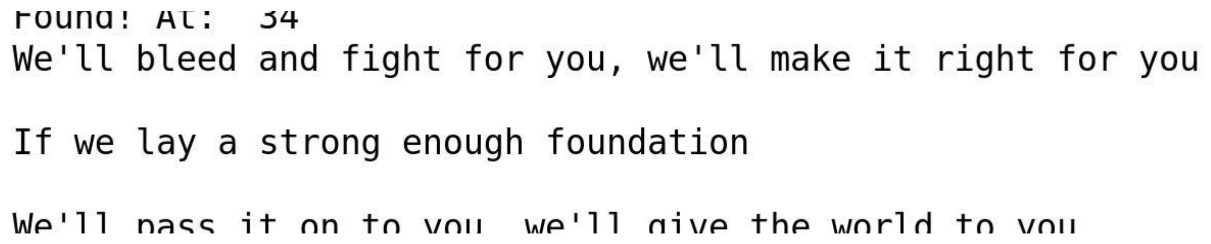
1. Create an empty list, giving it a suitable variable name. Make sure this is done before your for loop! Otherwise you'll constantly make empty ones.
2. If your chosen word is in the line, append the line to this List.
3. After the file close line, print the List (both ways, see below).

We can see that it is indeed a List of strings (Print the List object):



```
Found! At:  33
Found! At:  34
["We'll bleed and fight for you, we'll make it right for you\n", '
If we lay a strong enough foundation\n', "We'll pass it on to you,
we'll give the world to you\n", 'I swear that\n', "We'll bleed and
fight for you, we'll make it right for you\n", 'If we lay a strong
enough foundation\n', "We'll pass it on to you, we'll give the wor
ld to you\n"]
```

Or a nicer way, we can iterate through this List we made, and print each individual item (Iterate over the list and print items):

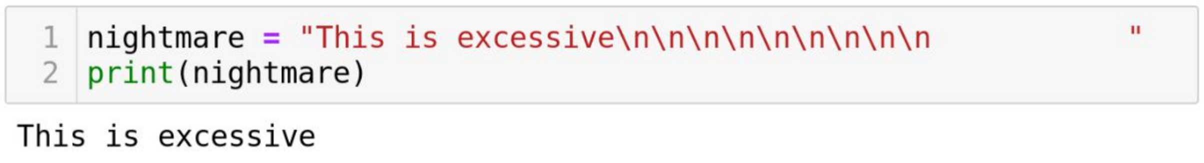


```
Found! At:  34
We'll bleed and fight for you, we'll make it right for you

If we lay a strong enough foundation

We'll pass it on to you, we'll give the world to you
```

Exercise 7: Notice how we seem to be getting an extra space between prints. This is due to that "\n" which appears at the end - or rather, all of the \n.

We can remove the trailing newlines and spaces by calling rstrip() on our strings. To prove this we can check the following:

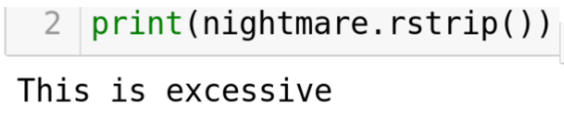
```
nightmare = "This is excessive\n\n\n\n\n\n\n\n\n                         "
print(nightmare)
```



```
nightmare = "This is excessive\n\n\n\n\n\n\n\n\n          "
print(nightmare)
```

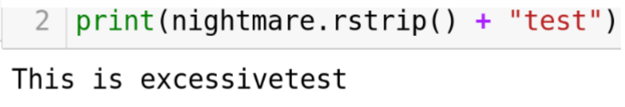
```
This is excessive
```

Yes, that is a lot of blank space…
If we print nightmare.rstrip() it will return a string which is rid of those special characters and all the spaces! The original variable is **NOT** modified.



```
print(nightmare.rstrip())
```

```
This is excessive
```



```
print(nightmare.rstrip() + "test")
```

```
This is excessivetest
```

It is immediately obvious that the newlines are gone (left), however, spaces are trickier. How can I check for trailing whitespace characters? We can do some string concatenation if we really wanted to prove the spaces are also gone (right)

Note: The existence of rstrip does indeed suggest the existence of a normal strip function, as well as an lstrip.

Writing Files

Exercise 8
In a new cell, we need to open a file to write. Let's write to the same file we've been working with. This time we will specify mode='w'. We know we need to close the file, so let's write
that in at the bottom before we forget.

```
ex8_file = open('ex1_data_copy.txt', mode='w')
print( ex8_file )
print( type(ex8_file) )

# Do stuff

ex8_file.close() # Might as well write this in before we forget.
```

As before, we should see it's a valid file, and the type is correct.

Even though we've not told Python to write anything, you may notice that a file has been created! (Check your file explorer) It's just empty.

Exercise 9
We can call ex8_file.write() passing it a string to write to the file. The behaviour of this depends on the file mode. In our case, we're on write, which will **erase** any existing content with what we put, if the file doesn't exist it will create it.

Write some content to this file by modifying the #Do Stuff section of our code.

After this open up the file and check what you intended to write. (Open in Notepad).

```
We're taking the hobbits to Isengard!
```

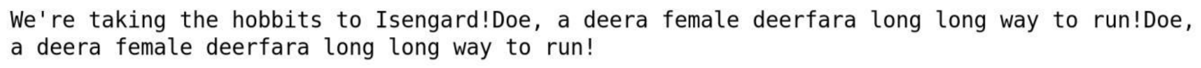
Exercise 10
Some time later we realised that we missed off some information. Let's append that to our file now.

1. Open the file in **append** mode
2. Create a List, and put some strings you want to write in it. (Note: These can also be numbers converted to strings - If we try numbers, we'll get an error).

   ```
   some_jibberish = ["Doe, a deer", "a female deer", "far", "a long long way to run!" ]
   ```

6. Iterate through this list, and for each element write this to the file.
7. Don't forget to close the file!

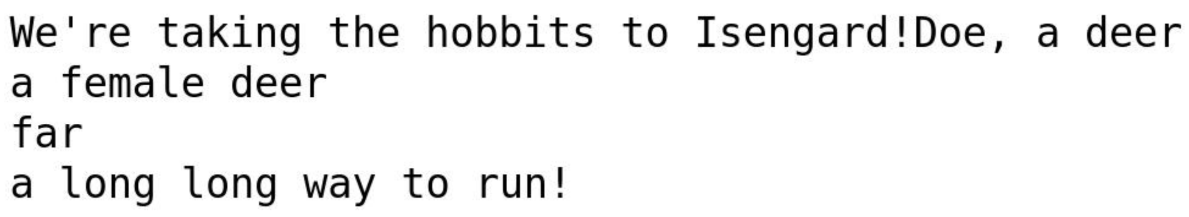
Open your file once finished, and check if the original content is present, with your additions added to the bottom in sequence order.



```
We're taking the hobbits to Isengard!Doe, a deera female deerfara long long way to run!Doe,
a deera female deerfara long long way to run!
```

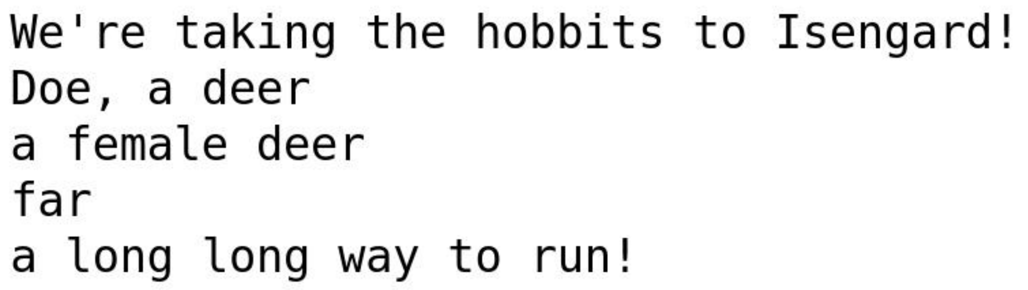
# **Whoops!** They're all on the same line in the file! Remember to add that special \n to each string you want to add! We can either change each string element literal. Or do some string concatenation in the write line itself!

E.g Instead of .write(s) we can do .write(s + "\n"). Saves us a lot of typing!



```
We're taking the hobbits to Isengard!Doe, a deer
a female deer
far
a long long way to run!
```

# Ahh! Almost. We need to make sure we write a newline at the end of the Exercise 9 bit, or before the first line written for this exercise. Try fixing this so you get the following output (swings and roundabouts on this one):



```
We're taking the hobbits to Isengard!
Doe, a deer
a female deer
far
a long long way to run!
```

Exercise 11

In the lectures we introduced Context Managers. Let's use those now.

Recall:

```
f = open(..)
4. Do stuff
f.close()
```

Becomes:

```
with open(..) as f:
    # Do Stuff.
```

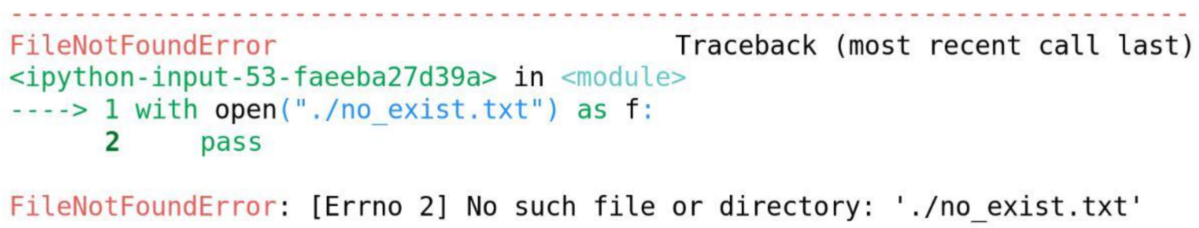
Duplicate your solutions to the previous answers involving file opening, replacing the clunky open, and close mechanisms with some Context Managers.

Exercise 12

Try the following in a new cell.

```
with open("./no_exist.txt") as f:
        pass
```

You should get the following output:

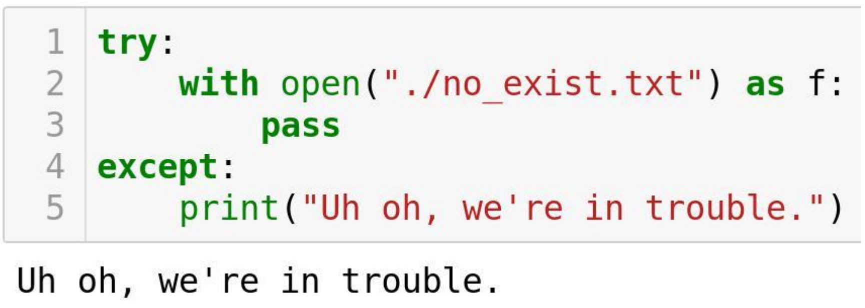


```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-53-faeeba27d39a> in <module>
----> 1 with open("./no_exist.txt") as f:
      2     pass

FileNotFoundError: [Errno 2] No such file or directory: './no_exist.txt'
```

This week we also introduced try and except for helping with Error Handling which follows the following format:

```
try:
        1. Do stuff here.
except:
        2. Any Issues, do this block.
```

3. Put the context manager, in its entirety, within the try block.
4. Add a nice print message into the except block which alerts you that something happened.



```
try:
    with open("./no_exist.txt") as f:
        pass
except:
    print("Uh oh, we're in trouble.")
```

```
Uh oh, we're in trouble.
```

Exercise 13
We can improve this, currently we have no idea what the error is. We only know that the print we put in the except block executed - so there is an error.

Modify the except line to catch the general exception Exception. Using the as keyword, give it a useful name. Then use this in your print statement.

```
Uh oh, we're in trouble:  [Errno 2] No such file or directory: './no_exist.txt'
```

Exercise 14

As we have already encountered this error, we can add a more specific except clause above the general one. Make this catch the FileNotFoundError exception, and get it to print something different.

```
except FileNotFoundError as not_found:
    print( "Didn't find the file, see message below" )
    print( not_found )
```

```
Didn't find the file, see message below
[Errno 2] No such file or directory: './no_exist.txt'
```

This try, except structure will prevent the whole of python from erroring out. Any code after this structure (try.... except...) will continue to be executed as normal. If this code is unrelated then it will execute just fine. Be cautious of putting code outside this try structure, if it relates to what's inside.

For example, if we open our file and it errors, but we try/except it and carry on afterwards to then attempt writing to the file... we're going to get another error as our file object will likely not be correct.

The behaviour of errors being swallowed by these except clauses is known as Error Hiding

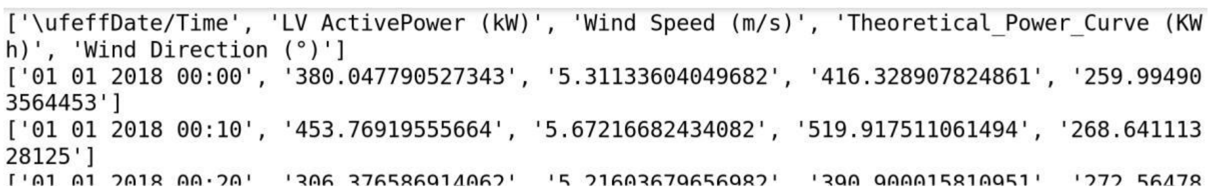
CSV

Exercise 15
Alongside this workshop you will find a wind_data.csv file. Download this to the same directory as your python notebook. Just like the text files from earlier.

Exercise 16
Import csv, and read the wind_data.csv file using a context manager. The file will be in 'read' mode.

Create a csv_reader, passing it the file we've just opened.

Print each line of this reader to see what data we're dealing with.



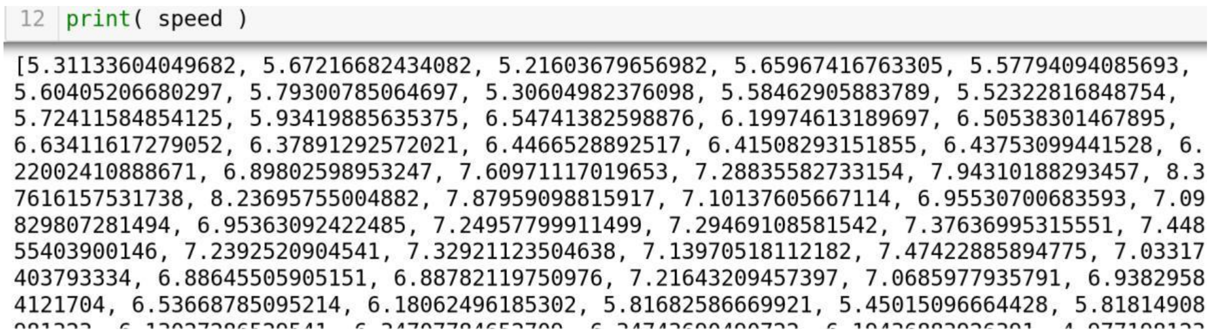
```
['\ufeffDate/Time', 'LV ActivePower (kW)', 'Wind Speed (m/s)', 'Theoretical_Power_Curve (KW
h)', 'Wind Direction (°)']
['01 01 2018 00:00', '380.047790527343', '5.31133604049682', '416.328907824861', '259.99490
3564453']
['01 01 2018 00:10', '453.76919555664', '5.67216682434082', '519.917511061494', '268.641113
28125']
['01 01 2018 00:20', '306.376586914062', '5.21603679656982', '390.900015810951', '272.56478
```

Exercise 17
Extract the first line (as this contains the headers) and store this as a variable. Take note of the type of this line, compared to our lines from the raw file reading earlier in the workshop. What do you notice?

Exercise 18

1. Which index would we need to take to extract the "Wind Speed" attribute?
2. For each line, obtain the Wind Speed (index the correct index).
3. This will be a string! Convert/cast it to a number (which type of number would be appropriate? An int? or a float? Decimal precision?)
4. Append it to a list of wind speeds.

Note: Be careful not to include the headers in this! Just data values we want. Numbers to do analyses on.



```
print( speed )
```

```
[5.31133604049682, 5.67216682434082, 5.21603679656982, 5.65967416763305, 5.57794094085693,
5.60405206680297, 5.79300785064697, 5.30604982376098, 5.58462905883789, 5.52322816848754,
5.72411584854125, 5.93419885635375, 6.54741382598876, 6.19974613189697, 6.50538301467895,
6.63411617279052, 6.37891292572021, 6.4466528892517, 6.41508293151855, 6.43753099441528, 6.
22002410888671, 6.89802598953247, 7.60971117019653, 7.28835582733154, 7.94310188293457, 8.3
7616157531738, 8.23695755004882, 7.87959098815917, 7.10137605667114, 6.95530700683593, 7.09
829807281494, 6.95363092422485, 7.24957799911499, 7.29469108581542, 7.37636995315551, 7.448
55403900146, 7.2392520904541, 7.32921123504638, 7.13970518112182, 7.47422885894775, 7.03317
403793334, 6.88645505905151, 6.88782119750976, 7.21643209457397, 7.0685977935791, 6.9382958
4121704, 6.53668785095214, 6.18062496185302, 5.81682586669921, 5.45015096664428, 5.81814908
```

Exercise 19

1. How many wind speed records/entries do we have? How might we find this out from the List we've just created.
2. What is the average Wind Speed recorded in these data? Hint: You may find the sum function useful here.
3. What is the minimum Wind Speed? (Hint: max(some_list))
4. What is the maximum Wind Speed? (Hint: min(some_list))

Exercise 20
Using a new Context Manager, this time set for **write** mode. Create a CSV Writer, and write out your Wind Speed List you created in Ex18.

View this file in a notepad to verify.

Small Introduction to Numpy

Exercise 21

Numpy is used for numerical computing, and allows us to convert from Lists to Numpy Arrays. These types have some useful functionality defined on them!

Have a look at the following code. Numpy arrays behave very similarly to Python standard Lists; however, they have more defined functions available to them for computational purposes. Previously you had to create an expression to get the summation and then the length and calculate the mean. Numpy has a function to do this. Remember: Functions are useful for utilities and routines we may often require. Therefore, numpy bundled a lot of these as behaviours on their own Data Type numpy ndarray.

For scientific computing, Numpy is written with several optimisations in mind. This makes it significantly faster for calculating on large sets of data. Ideal for crunching large amounts of data.

```
import numpy as np # Just an alias, I type np instead of numpy
x = np.array( speed )
print( type(speed) )
print( type(x) )

print(x[0] == speed[0]) # Standard Indexing. So far equivalent in how we use them!

print(x[-1] == speed[-1]) # Reverse Indexing.

#  useful statistics print("Mean:",
x.mean() ) print("Max:", x.max()
) print("Min:", x.min() )


#  More complex statistical measures of spread/dispersion print(x.std()) #
Standard Deviation
print(x.var()) # Variance = std ** 2 try:

        print(mean(speed)) # Try either of these.
        #print(speed.mean())
except Exception as e:
        print(e)
```

JSON Read/Write

Exercise 22

Recall Davey McDuck from this week's lectures. I have decided to add a new data attribute for our ducks, the number of people they follow on twitter. I have chosen to call this: 'following'.

```
duck_1 = {
        "first_name": "Davey",
        "last_name": "McDuck",
        "location": "Undercover",
        "insane": True,
        "followers": 12865,
        "following": 120,
        "weapons": ["wit", "steely stare", "devilish good looks"],
        "remorse": None
}
```

Let's build up our duck collection. We can represent this as a List of Dictionaries. Where each Dictionary follows the same pattern for outlining a Duck. First let's define some ducks. Feel free to add your own!

```
duck_2 = {
        "first_name": "Jim",
        "last_name": "Bob",
        "location": "Out at sea",
        "insane": False,
        "followers": 123,
        "following": 5000,
        "weapons": ["squeak"],
        "remorse": None
}

duck_3 = {
        "first_name": "Celest",
        "last_name": "",
        "location": "Throne Room",
        "insane" : True,
        "followers": 40189,
        "following": 1, # Her other account
        "weapons": ["politics", "dance moves", "chess grandmaster",
"immortality"]
}
```

We shall put these in a List called duck_collection

```
duck_collection = [ duck_1, duck_2, duck_3 ]
```

Go through the duck_collection, and make sure each element is in-place and all your ducks are accounted for.

Exercise 23

1. `import json`
2. Using a context manager, open a json file for writing
3. Using json.dump, write your duck_collection to the file you've opened.

   Remember that the arguments are almost backwards from what we're used to!

Open the file and we can copy its contents, which are rather unreadable at the moment, and use a pretty printing website to make it a bit easier on the eyes.

Paste your file contents into https://jsonformatter.org/json-pretty-print and see the output. Are all your ducks there? Do they all have their attributes?

Exercise 24
In a new cell. Using json.load, load your saved json file back in, assigning the output to a new list variable (other than duck_collection). We hope that the ducks loaded in are the same ducks we saved earlier. Therefore, they should be identical to your duck_collection. We can check for this by checking the equivalence between both lists.

Exercise 25
Write some code which, for each duck, will calculate the difference between the number of people following them and the number of followers of their twitter account. (Positive if more people follow them, than they follow).

1. Print these
2. Append them to an empty list called trendy_ducks

E.g for Davey, this would be his "followers" minus his "following" value. In this case 12865 - 120 = 12745. Note: This should be done programmatically! I might give you many more ducks. I want these resultant numbers for ALL ducks.

Hint: How do we index dictionaries? some_dict["key"] should give us the value. Our List we just got from json.load has a dictionary at every index of our List (Nested structures here).

```
12745
-4877
40188
Trendy Ducks:  [12745, -4877, 40188]
```

Exercise 26

Numpy has some useful functionality. If I wanted to find the trendiest duck (the one with the most 'net' followers (followers - following), I could use max or min. But this gives me the value back out, not necessarily which duck this relates to! I wanted an index so I could track the duck down.

Convert the trendy_ducks list to a numpy array

```
arr_trendy_ducks = np.array(trendy_ducks)
```

We can now call argmax() on this numpy ndarray object (or argmin() too) to show which duck has the most (or least).

```
print( arr_trendy_ducks.argmax() )
```

If I assign a variable to that function call, I now have the index of the trendiest duck. I can use this to go back to my original duck_collection List, which houses each duck dictionary, and pull things like their name.

Print the first name of the trendiest duck, programmatically, and print out their 'net' following count (the thing you calculated!).

No manual entry here. This code should work directly off of the List collection, so that I can add more ducks and your code would work exactly the same, and maybe a new Duck is crowned champion.

Hint: duck_collection[0] would get our first duck. We can replace 0 with any variable, so long as it returns an integer. duck_collection[0] would return a whole dictionary related to that duck. We can then reference the keys within that dictionary! duck_collection[0]["some_key"]

Hint 2: You may need to cast the net followers! Numpy likes to use it's own primitive data types for numbers. You will see :)

```
Congratulations Celest. You are the trendiest duck!
You have a net following of 40188 followers!
```

Exercise 27

Your boss has given you the task of creating a separate JSON file. In this file, he only wants ducks who have a net follower count > 0.
You must filter out the ducks who follow more accounts than who follow them, and save these ducks back out to a JSON file just like your input was.

Reusing your net follower calculations find indices of all the ducks you need.
Store them in a separate data structure (create this, and add the correct ducks using the indices found)
Open a new file in write mode to put this JSON data into. (Use a context manager).
Use json.dump to convert your separate data structure into JSON and store it in the file.

This exercise will involve a lot of juggling of variables, and data structures, and logic. Values and Indices will need to be managed appropriately. Try doing this before moving onto Ex 28, where we'll look at a nice feature of Numpy which might be more helpful.

Exercise 28

Numpy provides some functionality outside of just its objects. (https://numpy.org/doc/stable/reference/generated/numpy.where.html)

```
np.where( expression )
```

This returns a ndarray of indices where the expression holds True. For example, we can provide a whole array of values, and an expression which can be used to filter it. Numpy will then not only find where the condition is True or False, but then convert those into indices and provide them.

E.g
We can find all the indices of ducks, where their "net follower" count is even.

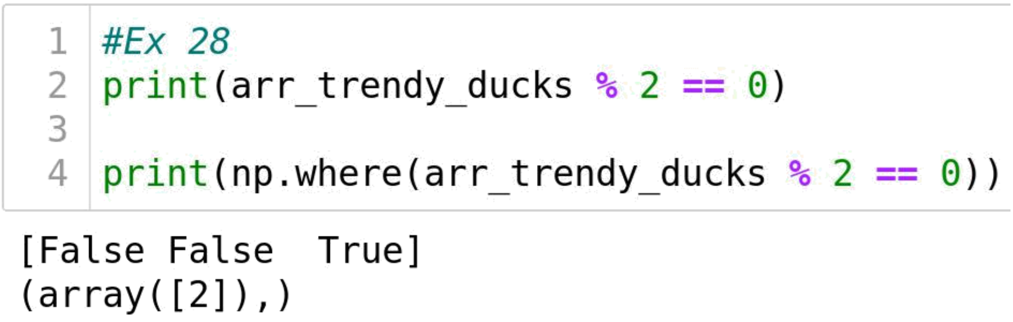
```
arr_trendy_ducks % 2 == 0
```

Normally we would pass an integer/number to this expression. However, with numpy arrays it can be applied to every value. If we print the output of this expression, we should get a List of True/False, where the condition holds!

We can pass this True/False List into np.where() and it will convert those into the indices for us.

Putting it together:

```
print(np.where(arr_trendy_ducks % 2 == 0))
```

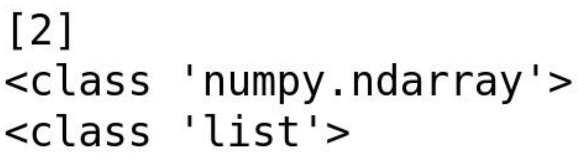


```python
#Ex 28
print(arr_trendy_ducks % 2 == 0)

print(np.where(arr_trendy_ducks % 2 == 0))
```

```
[False False  True]
(array([2]),)
```

This has returned a tuple. Remember, a tuple is defined as (item, item, item). The trick here is that it's a single element, you can see by the trailing comma. To get the actual array/List which contains the indices you'll need to call [0] on the np.where result.
Note: If we wanted to convert a numpy array back into a List, we can cast it just like we did with numbers weeks ago.

```python
actual_indices = np.where(arr_trendy_ducks % 2 == 0)[0]
print(actual_indices)
print(type(actual_indices))
print(type(list(actual_indices)))
```



```
[2]
<class 'numpy.ndarray'>
<class 'list'>
```

Technical Explanation: In our toy example here, we're using a single dimension (List of numbers). But np.where is very powerful and can actually be run over N-dimensional data. Therefore each entry in the tuple is a dimension. As we only have 1, we get a single one back.

As we can see, only one of our ducks has an even count. This is the duck at index 2. Surprise, surprise, this is Celest again.

Exercise 29
Copy your answer to Ex 27, and now use the np.where from Ex 28 to make your solution tidier, cleaner, more readable. In my example I did a boolean expression for even numbers, you need to find those > 0.

Hint: The array you get back, contains elements which are the index to look-up. A list comprehension is a perfect choice here. It sounds complicated, but it's just a List of indices you want to go through, and use them to reference the right ducks. Making a new list out of them.

```
[{'first_name': 'Celest',
  'last_name': '',
  'location': 'Throne Room',
  'insane': True,
  'followers': 40189,
  'following': 1,
  'weapons': ['politics', 'dance moves', 'chess grandmaster', 'immortality']}]
```

Group 7 Part A

Exercise 1: Research individually, or as part of a group, the trends in data of a company in your field or a company of interest for you.
In particular ensure you consider and answer the following:

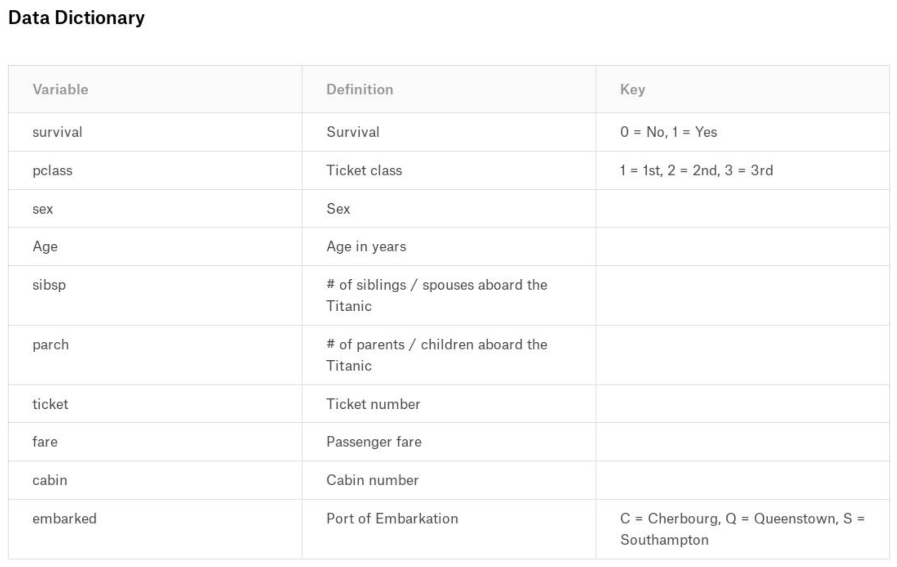
- # How much data is generated by this company (and the time frame)?
- # What data is used for day-to-day operations, is any generated data used for analysis?
- # How is this data used?
- # Where do they get the data from?
- # How do they get this data?
- # Who gave them permission for these data?

Exercise 2: Find an example of a company utilising data mining. How do they extract value from the data within these use-cases to drive their business? What technologies do they use for these, where applicable?
When answering these questions for the following use-cases, you should aim to find any relevant literature such as journals, conference proceedings, or technical whitepapers which may have been produced by, or about, the chosen company.

1. Log analytics
2. Commerce
3. Recommendation
4. Fault Detection / Prediction
5. Fraud Detection

Can you think of any other use-cases not listed above, for your respective programme of study?

Exercise 3: Read in the data from the titanic dataset CSV file found on the internet (I.e Context managers, File I/O).
Using Python, do some basic printing of the data, looking for attributes and the raw data stored in the CSV.
You may need to consult the Data Dictionary below for the titanic dataset.



Data Dictionary

Variable	Definition	Key
survival	Survival	0 = No, 1 = Yes
pclass	Ticket class	1 = 1st, 2 = 2nd, 3 = 3rd
sex	Sex	
Age	Age in years	
sibsp	# of siblings / spouses aboard the Titanic	
parch	# of parents / children aboard the Titanic	
ticket	Ticket number	
fare	Passenger fare	
cabin	Cabin number	
embarked	Port of Embarkation	C = Cherbourg, Q = Queenstown, S = Southampton

<u>Exercise 4:</u> Using the Titanic Data and Python (Numpy may be helpful here):

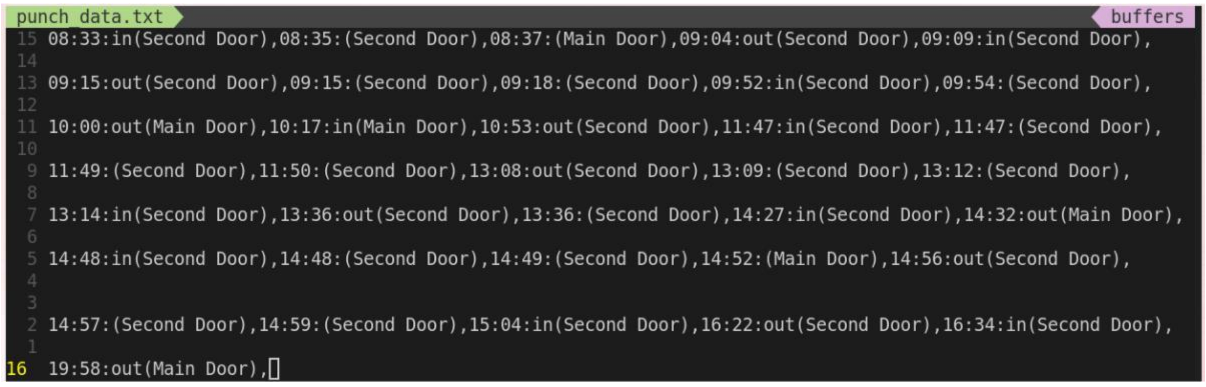
1. Calculate the <u>min</u> and <u>max</u> fare in the dataset.
2. What Data Type is the **fare** attribute? (Both Python data type, and Data Science)
3. If we were writing a better Data Dictionary, how would we represent this data attribute as a range descriptor? Remember the difference between [10, 1000] and (10, 1000).
4. Given the values within the dataset and what fare represents, what would be a good degree of precision to use for this attribute? Defend your position.
5. Separate out the data into the three pclass (1, 2, and 3). How does the <u>min</u> and <u>max</u> fare price change for each of these groups?
6. Consider the **name** attribute, what is it composed of, and how you might express this better?

    Example: How do we break up names in our society? What about titles (Mr, Mrs, Ms, Dr, Prof, …).

<u>Exercise 5:</u> Look at publicly available datasets for data mining / data analysis which have data dictionaries. Find 2 different data dictionaries and compare these against each other and the titanic dataset above.
You should note the level of description, the accuracy, as well as useful information for multiple roles within the organisation: analyst, designers, developers.

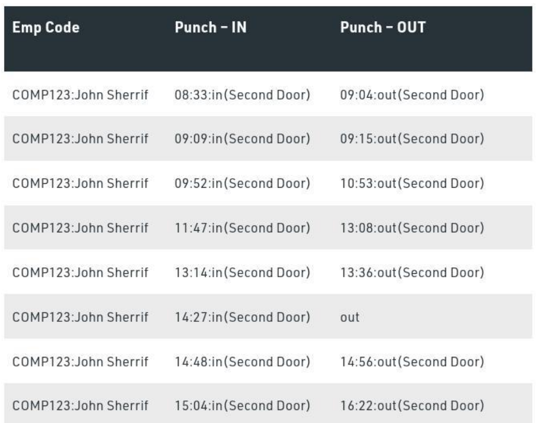
<u>Group 7 Part B</u>

<u>Exercise 5:</u> You are given the following data (**punch_data.txt** - See Canvas). This represents the security checkpoints logged by a security guard over the weekend.



```
punch_data.txt                                                                                    buffers
08:33:in(Second Door),08:35:(Second Door),08:37:(Main Door),09:04:out(Second Door),09:09:in(Second Door),

09:15:out(Second Door),09:15:(Second Door),09:18:(Second Door),09:52:in(Second Door),09:54:(Second Door),

10:00:out(Main Door),10:17:in(Main Door),10:53:out(Second Door),11:47:in(Second Door),11:47:(Second Door),

11:49:(Second Door),11:50:(Second Door),13:08:out(Second Door),13:09:(Second Door),13:12:(Second Door),

13:14:in(Second Door),13:36:out(Second Door),13:36:(Second Door),14:27:in(Second Door),14:32:out(Main Door),

14:48:in(Second Door),14:48:(Second Door),14:49:(Second Door),14:52:(Main Door),14:56:out(Second Door),


14:57:(Second Door),14:59:(Second Door),15:04:in(Second Door),16:22:out(Second Door),16:34:in(Second Door),

19:58:out(Main Door),
```

This data is not very useful for analysis. We desire it in the following format, whereby punch in and out events are paired:



Emp Code	Punch – IN	Punch – OUT
COMP123:John Sherrif	08:33:in(Second Door)	09:04:out(Second Door)
COMP123:John Sherrif	09:09:in(Second Door)	09:15:out(Second Door)
COMP123:John Sherrif	09:52:in(Second Door)	10:53:out(Second Door)
COMP123:John Sherrif	11:47:in(Second Door)	13:08:out(Second Door)
COMP123:John Sherrif	13:14:in(Second Door)	13:36:out(Second Door)
COMP123:John Sherrif	14:27:in(Second Door)	out
COMP123:John Sherrif	14:48:in(Second Door)	14:56:out(Second Door)
COMP123:John Sherrif	15:04:in(Second Door)	16:22:out(Second Door)

In **raw Python** (not using any external libraries; E.g do not use numpy!):

1. Read in the text data
2. Remove erroneous newlines
3. Split each content line based on their comma (,) - You can use string split for this (https://docs.python.org/3/library/stdtypes.html?highlight=split#str.split)
4. Extract all the values where the punch in and punch out events occur
5. Put these extracted values into a new Python data structure, E.g Dictionary

Exercise 6: You should notice that John Sherrif forgot to punch out twice during his rounds. For those specific entries in your Python Data Structure, add a new binary field / attribute called 'missed' and set this to True. For all others, this attribute should not be defined, or set.

Exercise 7: Currently the data is subject-focused around John Sherrif and his tap-in duties. However, I want to focus on a specific door, the Second Door. This is because I want to analyse how long guards are taking to ensure that this door is secure before moving on.

From your Exercise 6 output, filter all entries for Second Door.

CMP 411 Exercise 1 for all

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.

CMP 411 Exercise 2 for all

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]