SESSION 1.1

Let’s do *stuff*

WELCOME BACK!

- I hope you had a good break.
- Annnnnd I hope you're looking forward to the recap quiz that you're about to do.
- It's in the usual place, and as always it's not for marks, but it's a good way to check your understanding.

JUST REMEMBER

- `strs` are collections of alphanumeric characters enclosed in quotes (either single, double or triple)
- `ints` are whole numbers
- `floats` are decimal numbers
- `lists` are collections of items enclosed in square brackets (they can contain any datatype, including other lists)
- `dicts` are collections of `key : value` pairs enclosed in curly braces (they can contain any datatype, including lists and dicts)

RECAP QUIZ

- Up on brightspace, on the content page for this week.

TODAY'S PLAN

Now we've talked about how python handles different types of data, we're going to start by looking at the (basic) things we can do with that data.

In this session we're going to look at - Operations with strings (string addition and multiplication) - Indexing and slicing strings and lists - Including why we start counting at 0 - Functions (What they are, how to define them, and how to use them)

And we'll see how we might be able to put those things together to do some interesting stuff.

OPERATIONS WITH STRINGS

- When we are working with numbers, we are used to the idea that we can perform several type of operations on those numbers.
- By operation, I mean some process where we take in information (input) to spit out new information (output).
- For example, we can use addition to get the sum of two or more numbers: `5 + 2`; `3 + 4 + 5`.
- Or we could use multiplication to get the product of these numbers: `5 * 2`; `3 * 4 * 5`.

OPERATIONS WITH STRINGS

- When we work with text, however, we don't often think in terms of operations or of mechanical processes.
- For example, imagine you asked someone how to spell the word "Weird" and they said: "Okay so you start with a 'W', and then you add an 'e' to it to get 'We', and then you add 'i' to get 'Wei', and then you add 'r' to 'Wei' to get 'Weir', and then finally you add 'd' to 'Weir' to get Weird….. :)"
- You'd think they were weird (if not an outright moral degenerate), and you'd be right.
- Normally, when we use language, we use more intuitive processes (like sounding out how a word is spelt).

OPERATIONS WITH STRINGS

- But, in the magical world of Python, we can and often will perform operations on text (AKA strings). Two examples of string operations are “String Addition” and “String Multiplication”

STRING 'ADDITION' (STRING CONCANTENATION)

- In python we can add two or more strings together:

```
1 print('hand' + 'bag') #prints 'handbag'
2 print('bull' + 'dog') #prints 'bulldog'
3 print('counter' + 'clock' + 'wise') #prints 'counterclockwise'
```

```
handbag
bulldog
counterclockwise
```

STRING 'ADDITION' (STRING CONCANTENATION)

- This operation, adding two or more strings together, is called `String Concantenation`.
- Concantenation being the latin word for "gluing crap together" (no need to fact check this).

```
1 print('hand' + 'bag') #prints 'handbag'
```

```
handbag
```

- Concantenation (or string addition) takes whatever is inside one string and sticks it directly on to the end of the previous string with no space in between.
- This is in contrast to how addition works with numbers, where adding two or more numbers results in a separate number.

STRING 'ADDITION' (STRING CONCANTENATION)

- You can also perform string addition with string variables
- And if you want to include a space between your two variables, there are two main approaches.

```
## Approach 1. We can use a space either at the end of the first variable, or a space at the beginni

space_first_name = "Grampton " ## put a space between the end of the name and the quotation mark
no_space_last_name = "St.Rumpterfraple"
print(space_first_name + no_space_last_name)
```

```
Grampton St.Rumpterfraple
```

STRING 'ADDITION' (STRING CONCANTENATION)

- We just put a space at the end of the first variable, and then add the two variables together.
- This approach is fine if what you are printing is a once off.
- If using variables, however, spaces will remain a part of our variable until we specifically change it.

```
## Approach 2. A better approach is to `add` a string inbetween our two variables:
my_first_name = "Jean Luc"
my_last_name = "Picard"
print(my_first_name + " " + my_last_name)
# The " " string adds a space between our two variables.
```

```
Jean Luc Picard
```

STRING 'ADDITION' (STRING CONCANTENATION)

- An important thing to note about string addition, is that it works exclusively with strings. You cannot add a string to an int or a string to a float.
- If you try to do this, Python will throw an error message.
- But you can do something like this:

```
1 print("I am " + str(25) + " years old") #prints "I am 25 years old"
```

```
I am 25 years old
```

STRING 'ADDITION' (STRING CONCANTENATION)

- Play around a little with string concatenation in your own jupyter notebook.
- Also try making a couple of new examples of string variables and add them to int or floats with the `str()` function.
- Then we'll move onto string multiplication.

STRING MULTIPLICATION (STRING REPLICATION)

- You can also perform “string multiplication”, an operation known as `string replication`.

```
print("a" * 3)
```

```
aaa
```

```
print("do"*2) #like the bird
```

```
dodo
```

```
print("Are we there yet? " * 5) #Like your annoying little brother or sister.
# NOTE: I am using a space here at the end as I am just printing the string once off, I am not using
```

```
Are we there yet? Are we there yet? Are we there yet? Are we there yet? Are we there yet?
```

STRING MULTIPLICATION (STRING REPLICATION)

- You can see in each case, that when we 'multiply' a string, we replicate it, and then add the replication onto to the end of our original string.
- You will also notice that, in contrast to string addition, you can combine strings and integers.
- Integers are used to specify the number of times you want to replicate the string. However, string replication does not work with floats.

STRING MULTIPLICATION (STRING REPLICATION)

- Copy the cell below into your own jupyter notebook and run it.

```
lyrics = "Around the world, around the world\n" ## Do not worry about the \n for now, this is just a

print(lyrics * 8.0)
```

- Notice the “TypeError” message in the output.
- Have a read through it, and once you are ready, change the float 8.0 into an interval, and re-run the code.

STRING MULTIPLICATION (STRING REPLICATION)

- There you have just printed out the first verse of Daft Punk's 1997 hit single "Around the World"!
- You can change the 8 to 80 to print out the radio version of the song,
- or you change it to 144 to print out the entire album version of the song.
- But now lets move on to the actual best way to work with strings in python: `f-strings`!

STRING-CEPTION: PUTTING STRINGS INSIDE OTHER STRINGS

- A common operation that you will use in Python is `formatting` strings.
- Technically we could achieve this by using string concatenation

```
1 my_first_name = "Harry"
2 my_last_name = "Potter"
3 subject = "Wizardry"
4 university = "Hogwarts"
5
6 print("Hi there, my name is " + my_first_name + " " + my_last_name + ". I studied " + subject + " at
```

```
Hi there, my name is Harry Potter. I studied Wizardry at the University of Hogwarts.
```

STRING-CEPTION: PUTTING STRINGS INSIDE OTHER STRINGS

But in order to print out this sentence in a correct manner, we need to perform string concantenation (using the '+' operator) multiple times in order to:

- I. To `add` the string `"Hi there, my name is "` to the `my_first_name` variable
- ii. To `add` the `my_first_name` variable to the string `" "`
- iii. To `add` the string `" "` to the `my_last_name` variable
- iv. To `add` the `my_last_name` variable to the string `". I studied "`
- V. To `add` the string `". I studied "` to the `subject` variable
- vi. To `add` the `subject` variable to the string `" at the University of "`
- vii. To `add` the string `" at the University of "` to the `university` variable
- viii. Finally, to `add` the `university` variable to the string `"."` .

STRING-CEPTION: ` F-STRINGS

- This is a **collosal pain in the ass**.
- We are always likely to make small mistakes (maybe we forget to leave a space between variables, or forget to `add` a full-stop)
- Instead of using the string concantenation, use an `f-string operator`.
- `f-strings` are known as formatted strings (get it?), and they enable us to plug in our variables without the need to excessively worry about formatting.

```
1 print(f"Hi there, my name is {my_first_name} {my_last_name}. \n I studied {subject} at the Universit
2 # We know Hogwarts isn't a university you massive nerd.
```

```
Hi there, my name is Harry Potter.
 I studied Wizardry at the University of Hogwarts.
```

STRING-CEPTION: ` F-STRINGS

- We place a **f** at the start of our string,
- we use a curly bracket {} to tell Python we are inserting a variable
- and then we type the variable name inside the curly bracket.
- So much easier!

STRING-CEPTION: `F-STRINGS

- In your own jupyter notebook, play around with `f-strings`
- You've made lots of variables so far, but you can always make more and play around!

Next, we're going to look at how we can start to pick apart strings and lists using `indexing` and `slicing`.

INDEXING AND SLICING STRINGS

- When we work with strings, we can access chunks in the string using `indexing` and `slicing`.
- `Indexing` is the process of selecting a single character from a string.
- `Slicing` is the process of selecting multiple characters from a string.
- Let's start with `indexing` (this is where you learn why yesterday was Day 0!!)

INDEXING STRINGS

- The `index` of a particular object is it's position in a collection of objects.
- That might seem like jargon (and in fairness it kind of is) but let's use a string as an example to clear that up.

INDEXING STRINGS

```
1  ```{python}
2  #| echo: true
3  #| eval: true
4  m = 'hello world'#assigning 'hello world' to the variable m
5  ```
```

- You already know that a string is an object made of alphanumeric characters enclosed in either single or double quotes
- We often think of a string as an object unto itself, but, we can also think as a string as a collection of individual characters, each character (including spaces) can also be thought of as an object (essentially, python can treat a `string` as a `list` of alphanumeric objects enclosed in quotes instead of square brackets).

INDEXING STRINGS

- The variable `m` that we assigned just above is a string which is 11 characters long.
- Each of those characters has a position within the string and this position is identified with a number.
- You might think that in `m` the position of the first character ('h') would be 1, but python is what's called a '0-indexed language', this means that we start counting positions from 0 instead of 1, this is demonstrated in the table below.

index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
char	h	e	l	l	o		w	o	r	l	d

WHY ZERO-BASED INDEXING?

The concept of zero-based indexing has its roots in low-level programming and memory management:

- **Memory Addresses**: Memory addresses start at 0. The first element of an array is stored at the starting memory address, which is address 0.
- **Efficiency**: Zero-based indexing aligns with how memory is addressed in hardware, making the computation of the memory address of an element straightforward.
- **Historical Context**: Early programming languages like C adopted this convention to stay close to the hardware, and this tradition carried over into modern languages like Python.
- It is kind of annoying? Yes.
- Will you make mistakes because of it? Yes.
- Will you get used to it? Yes.

INDEXING STRINGS

- Indexing is quite powerful because it allows us to do things like search *within* an object, or pick an object apart.
- Let's start with just picking out the first character ('h') in `m`, it's really easy.

```
1 m = 'hello world' #assining 'hello world' to the variable m
2
3 print(m[0])# Note the square brackets (like a list) and the int 0
```

```
h
```

- The first line in the cell above you already know, we've just taken a `string` variable ('hello world') and given it the label (assigned it to the `variable`) - `m`.
- The second line is the magic.

INDEXING STRINGS

```
1 m = 'hello world' #assining 'hello world' to the variable m
2 #The next line prints just the first character of the string m
3 print(m[0])# Note the square brackets (like a list) and the int 0
```

```
h
```

- `m` represents our string so python knows that we are working on that particular object.
- Immediately after `m` we open square brackets `[` (like we used when we were making a list)
- then we put in the `index position` (remember the table above) and then we closed the square brackets `]`.
- This is the basic `syntax` for finding an object in a collection of objects.

NEW WORD - `syntax`

INDEXING LISTS

- So we can do the same thing with `lists`, and a very similar thing with `dicts` and full datasets (`DataFrames`, more on this in future sessions).
- lets take a list

```
beatles = ['John', 'Paul', 'George', 'Ringo']
```

- This list is indexed exactly the same way as `m`

index	0	1	2	3
item	'John'	'Paul'	'George'	'Ringo'

INDEXING LISTS

- So we can use exactly the same `syntax` to find out what is at a particular `index` in the list.

```
beatles = ['John', 'Paul', 'George', 'Ringo'] #creating the list, and assigning it the variable beat

print(beatles[0])# getting the first object from the list beatles, it's 'John'
```

```
John
```

- See? Simple.

INDEXING LISTS - NEGATIVE INDEXING

- We can also use negative numbers to index from the end of the list.
- This is useful if you don't know how long the list is, but you know you want the last item.
- As in the context where a list is updated regularly, and you just want to know the last item.

```
1 print(beatles[-1])# getting the last object from the list beatles, it's 'Ringo'
2 print(beatles[-2])# getting the second last object from the list beatles, it's 'George'
```

```
Ringo
George
```

INDEXING - ASSIGNMENT

```
1 j = m[-1]# Taking the last character from the string m, and assigning it to the variable j
2 k = beatles[2]# takine the second object from the list beatles, and assigning it to the variable k
3 print(j)
4 print(k)
```

```
d
George
```

- We did a couple of things there, and one is kind of sneaky (sorry/not sorry).
- At the simplest level, we used our `indexing syntax` to extract a character from a string and *store it as a new variable*
- then we used the same `syntax` to save 1 of the elements from our list sperately.

It's worth noting that we haven't actually changed the list or string.

When we say we've 'extracted' a part of them we don't mean that we have literally just pulled them out, we just mean we've selected those bits of information and given us a way to access them easily, but the original items are still intact.

INDEXING - ASSIGNMENT

```
print(m) #prints 'hello world'
print(f"The last letted of `m` is {j}") #prints 'd'
print(beatles) #prints ['John', 'Paul', 'George', 'Ringo']
print(k) #prints 'George' because we indexed the third object in the list beatles in an earlier cell
k = 'Derrick' #changing the variable k to 'Derrick'
print(f"New variable k: {k}") #prints 'New variable k: Derrick'
print(beatles) #prints ['John', 'Paul', 'George', 'Ringo'] because we only changed the variable k, n
```

```
hello world
The last letted of `m` is d
['John', 'Paul', 'George', 'Ringo']
George
New variable k: Derrick
['John', 'Paul', 'George', 'Ringo']
```

SLICING

- `Slicing` is the process of selecting multiple characters from a string.
- We can use slicing to select a range of characters from a string, or items from a list.
- The syntax for slicing is similar to the syntax for indexing, but we use a colon `:` to indicate a `range` of positions.

```
print(m[0:5])#prints 'hello'
print(beatles[1:3])#prints ['Paul', 'George']
```

```
hello
['Paul', 'George']
```

SLICING

```
1 print(m[0:5])#prints 'hello'
2 print(beatles[1:3])#prints ['Paul', 'George']
```

```
hello
['Paul', 'George']
```

- Slicing is a little more complex than indexing, but it's still pretty simple.
- We specify the `start` and `end` of the slice, and python will return everything between those two points.
- but, the `end` point is not included in the slice.
- In the first line we are slicing the string `m`, starting at 0 and ending at 5.
- So we start a 0 and take everything **up to but not including** position 5 (we stop with position 4).

SLICING

```
1 print(m[0:5])#prints 'hello'
2 print(beatles[1:3])#prints ['Paul', 'George']
```

```
hello
['Paul', 'George']
```

- In the second line we are slicing the list `beatles`, starting at 1 and ending at position 2.
- So we start at 1 and take everything **up to but not including** position 3 (we stop with position 2).
- Imagine you had a scale with 50 items in it, and you wanted just select the items between 10 and 20.
- Slicing would allow you to do that.

```
my_scale[10:21] #would give you items 10 to 20
```

INDEXING AND SLICING - YOUR TURN

- In your own jupyter notebook, create a string and a list, and then use indexing to extract the first and last items from each.
- Maybe try using `f-strings` when making your string? (you could make an `int` variable and `add` it to your string using an `f-string`).
- Then try using negative indexing to extract the second last item from each.

INDEXING RECAP AND MOVING FORWARD

- This was just an introduction to indexing, and we'll play around with it more when we start working with full datasets.
- If you want to learn more about it, you can check out https://pythonguides.com/indexing-and-slicing-in-python/.
- It will give you more examples and also introduce you to something called `slicing`.
- For now, we're going to move on to…

FUNCTIONS

- Before we get into what functions are, we need to apologize, for you see, we tricked you.
- We tricked you into using functions without telling you… and we're deeply sorry (cough)…

FUNCTIONS

- `print('Hello world')`
- That line of `code` tells the computer to write out the `string` 'Hello world'.
- Getting the computer to write something out actually requires a lot of code, but thanks to `functions` we can 'compress' all that code, all its functionality, into a simple-to-type statement *followed by a pair of brackets*.
- Let's look at another function:

FUNCTIONS

- `type('Hello world')`
- This line tells python to find the datatype of the object between the brackets.
- Again, there's a lot of code 'hidden' behind `type()` but we don't need to worry about that now.
- Right now we just want to pay attention to the `syntax` for both of those functions:

FUNCTIONS

- `function_name(thing to run the function on, also called an argument)`
- We start by typing the name of the function (print, or type in the examples we've used so far), then we open brackets, then we enter the object we want the function to work on, then we close the brackets.

```
1 a = 'This is a string'#
2 b = 10#
3 c = ['Sarah', 'Navid', 'Robert', 'Noura']#
4 print(f"c is a {type(c)}, but a is a {type(a)}")#
```

```
c is a <class 'list'>, but a is a <class 'str'>
```

FUNCTIONS

- The `len()` function is another function that should be pretty easy to understand.

```
l = len(c[1]) # We're using a function to assign a new variable
print(f"{c[1]}'s name is {l} characters long")
```

```
Navid's name is 5 characters long
```

FUNCTIONS - EXPLANATION

- In the above cells, we `assigned` a few `variables` and `called` a few `functions` on them.
- In some lines we actually `called` a `function` *within* another `function`,
- and in one, we used a `function` to *create* a `variable`, all with really small amounts of typing on our part.
- This should demonstrate that functions are really useful. Let's look at how `functions` are made (the `syntax` of `defining a function`).

DEFINING OUR OWN FUNCTIONS

- Let's imagine that we want to have the computer take any number and multiply it by 42 (get it?) and then print out the result.
- We know that this is a simple example, but stick with us here.

```
# let's just use 3 for this example
print(3*42)
```

```
126
```

- We could just use the code above, but what if we wanted to do this a lot?
- It would be really inefficient to keep typing out `print(n*42)` every time we wanted to multiply a number by 42.

DEFINING OUR OWN FUNCTIONS

- However, if we want to have the ability to do this often, we might want to make a function that does this for us…

```
1 def my_func(i): # def is short for define, i is a placeholder variable, note the colon after the bra
2     print(i*42) # note the indentation, indentation is really important, we indent using 'tab' (or 4
3
4 my_func(3) # we're calling the function we just defined, and passing the number 3 to it
```

```
126
```

- We've just defined a function called `my_func` that takes a single argument (a number) and multiplies it by 42 and prints out the result.
- Then we `called` the function and passed the number 3 to it.
- Take a sec and really look at the `syntax` of defining a function, it's really important.

WHAT DID WE JUST DO?

- When making a function we start on a newline with the term `def` which is short for 'define' (in technical terms we're '`defining`' a function).
- Then we type the name for our function (in this example we just called it `my_func`, using underscore instead of spaces).
- Then we open brackets, put in a placeholder variable (`i` in this case, but it could be `x` or `banana`), then close the brackets.
- Then we finish that line with a colon (:).
- Then we go to a newline.
- Then we `indent` the line with either a 'tab' or 4 spaces (don't use 4 spaces, it'll work but just… don't be that guy).
- Then we can start putting in the functionality.

INDENTATION IN PYTHON

- Indentation is a whole thing in Python, it has to do with something called `scope` and we'll cover it more in more detail in later sessions.
- For now just know that in simple functions, every new line under the `def` line (after the colon) starts with an indentation (press the `tab` key first).
- Because we're working in VSCode, and we've told VSCode that we're working in Python, it will *usually* put the indentation in for you.
- But there will be times when you need to manage the indentation yourself.

QUICK EXERCISE

- In your jupyter notebook, make a new cell and copy the code below.
- We've defined a function with incorrect indentation.
- Run the cell first, read the error, then go back and fix the indentation (and for the love of God, use tab not 4-spaces).

```
1 ```{python}
2 def new_function(): # we're not putting a placeholder variable in this cause we don't need it
3 print('This') # We're splitting the functionality up into multiple lines
4 print('Function')
5 print('Works')
6 ```
```

QUICK EXERCISE

- You'll notice that before we fix the cell up above we get an error (called a `traceback` in Python language, can you think of why?).
- Once we fix the indentation we get no output from the cell, this is because we just `defined` the `function`, we never `called` it.
- Calling a function is easy:

```
1 my_func(12) # calling the 'my_func()' function to multiply 12 by 42
```

```
504
```

- Make a new cell in your jupyter notebook and call the function `new_function()` that you just fixed.

EXERCISE

- OK, now for the final exercise of this session is to `define` a `function` that uses a little bit of everything we've covered so far.
- This function should take a string as an `argument` (the bit between the brackets) and then, using `f-strings` and `indexing` prints out something like:

```
This is my string: (string); and it's last letter is (last letter)
```

- All the `syntax` you need is in this notebook, just take your time and think about the 'order' that you want things to happen in, don't forget to comment your code and don't forget to call the function once it's working.

WRAP UP

- We've covered quite a bit, but none of it is particularly difficult.
- There's lots of chunks of `syntax` to remember, but you'll get used to it.
- Before we take a brake, take a stab at the recap quiz on Brightspace.
- Work together, ask me questions, and don't be afraid to make mistakes.