SESSION 1.2

Loops, Modules, and Methods (‘Oh My!’)

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

- We can add strings together, multiply strings by numbers, and use `f-strings` to include variables in strings.
- We can use the `index` to find a particular character in a string, and we can use `slicing` to get a range of characters from a string.
- We can do the same thing with lists!
- We can use `variables` to 'hook' onto bits of data and use them later, and make them easier to type.
- We can use `functions` like `print()` and `type()` to do things with data
- And we can `define` our own functions to do things with data in a more structured way with the `variables` we create.

RECAP QUIZ

- Up on brightspace, on the content page for today.

TODAY'S PLAN

We've talked about some simple operations on strings and lists, we've talked about variables and functions. In this session we're going to start diving a little deeper, but again, we're going to keep it simple and we promise that the topics are relevant to the work you'll be doing as researchers.

We're going to: - Go further into functions and talk about the idea of the `placeholder` and the `return` statement. - We're going to talk about `loops` and how they can help us do things over and over again. - We're going to talk about `modules` and how they can help us do things that are a bit more complicated, by using code that other people have written. - We're going to briefly touch on the idea of `methods` (but these will be covered more when we dive into actually working with dataframes later on).

FUNCTIONS: A RECAP

- The last thing we worked on was `functions`.
- We can think of them as a way of wrapping up a lot of code in a short and easy to type statement
- like the `print()` and `type()` `functions`.
- The `syntax` of using `functions` is pretty simple.
- But lets look at it again.

FUNCTIONS: A RECAP

```
1 ```{python}
2 # This is the syntax for using a function
3 # the function name is followed by a pair of brackets
4 # and the object you want to use the function on is inside the brackets
5 print('Hello world')
6 ```
```

- You already know that the `print` bit is the name of the `function` and that this is followed by a pair of brackets `()`.
- The brackets are what actually tell python to execute the `function`.
- The thing (or things) *between* the brackets is called an `argument`,
- so in the example above `'Hello world'` is the `argument` for the `print()` function. All of the premade `functions` we've looked at all take 0 or 1 argument.

FUNCTIONS: A RECAP

```
```{python}
print('Hello world')
type('Hello world')
```
```

- Each specific `function` will have a required number of `arguments`
- Don't worry about it too much because you can *always* look up a specific `function` when you need it, and the more you use a particular one, the better you'll remember it.
- A lot of the time you'll be using functions that other people have written (they are `packaged` into `modules` you can `import` into your own projects)
- But you also know how to `define` you're own `functions` and we're going to go a little deeper into that here.

DEFINING FUNCTIONS

- Last session we showed you the `syntax` to `define` your own `functions`.
- Remember we use the `def` keyword to tell python that we're going to `define` a `function`.
- We then give the `function` a name, and then a pair of brackets `()`.
- After the brackets we put a colon `:`
- Then we `indent` the code that we want to be part of the `function` by 1 tab (or 4 spaces, but seriously, just use 1 tab)
- We can then `call` the `function` by using its name followed by a pair of brackets `()`

```
1 # This is the example you fixed in the last session
2 def my_function_1(): # This function takes no arguments so we're not putting anything in the bracket
3     # all the lines below are the first one are indented by 1 tab (or 4 spaces, but sersiously, just
4     print('This')# We're splitting the functionality up into multiple lines
5     print('Function')
6     print('Works')
```

DEFINING FUNCTIONS - ARGUMENTS

- In this second example we want the `function` to be able to operate on a `variable`
- We can do this by putting a `placeholder` in the brackets of the `function definition`
- We can then use that `placeholder` in the `function code` to do things with the `variable` that we `call` the `function` with.
- We can use any `alphabetic` character as a `placeholder` but it's good practice to use something that makes sense in the context of the `function` you're writing.

```
1 def my_function_2(placeholder):# notice that this line is almost exactly the same except for the pla
2     # again, all the lines are indented
3     print(f'The variable is: {placeholder}')# using an f-string to print out the variable
4     print(f'it is a {type(placeholder)} object')
```

DEFINING FUNCTIONS - ARGUMENTS

- So now that we have defined these two functions in the cells above we can call them in subsequent cells without having to type out all that code again.

```
1 my_function_1()
2 my_function_2('Banana')
3 z = 24.5
4 my_function_2(z)
```

```
This
Function
Works
The variable is: Banana
it is a <class 'str'> object
The variable is: 24.5
it is a <class 'float'> object
```

DEFINING FUNCTIONS - ARGUMENTS

- We can also define a function that takes multiple arguments
- We just put a comma between the placeholders in the `function definition`
- The `arguments` in a function can be any type of object, and you can mix and match them as you like.
- They can be `variables` we want the function to operate on, or they can be `values` that control how the works in some way.
- They can get pretty complex, but when we're making them we can think about how to keep them as simple as *they need to be* to do the job we want them to do.

```
1 def my_function_3(a,b): #we are using two placeholders here
2     print(a*b)# this function takes numbers and multiplies them
3 my_function_3(2,3)
4 my_function_3(100, 2.24)
```

```
6
224.00000000000003
```

PLACEHOLDER VARIABLES

- One of the most important things to understand about functions (and Python in general) is the use of placeholders variables.
- Typically, when you write functions, you will have a specific purpose in mind.
- For example, last week we asked you to write a function that took your name, and then printed both your name and the last letter of your name within a f-string.
- Since we had already asked you to store your first name in a variable (`first_name`) the temptation is to write your function specifically to print out that * specific* variable.

```
1 first_name = "Eve"
2 def last_week_function_v1():
3     print(f"Hi there, my name is {first_name} and the last letter in my name is {first_name[-1]}.")
4 last_week_function_v1()
5 print(first_name)
```

```
Hi there, my name is Eve and the last letter in my name is e.
Eve
```

PLACEHOLDER VARIABLES

- But, that version will only work if:
  1. You have a variable called `first_name` that is a string
  2. You have stored your first name in that variable
  3. You don't ever want it to work with any other string
- So if you wanted the same functionality but with a different string, you would have to write a whole new function.
  - This is not very efficient, and it's not very `pythonic` (which is a fancy way of saying it's not the best way to do things in python).

PLACEHOLDER VARIABLES

- This all makes the `function` pretty inflexible (or just not `functional`)
- Instead, we can use a `placeholder` variable in the `function definition` to allow the `function` to take an `argument` when it is `called`.
- This is the `syntax` we used in the second example above.

```
1 def last_week_function_v2(x):# we are using a placeholder variable here
2     print(f"Hi there, my name is {x} and the last letter in my name is {x[-1]}.")
3 last_week_function_v2(first_name)
```

```
Hi there, my name is Eve and the last letter in my name is e.
```

PLACEHOLDER VARIABLES

- If the two functions above produce the exact same output, then why should we use a placeholder?
- The value of a placeholder is that it significantly increases the flexibility of our code.
- The way we have defined our last_week_function1 limits our code only to print whatever is stored into our first_name variable.
- The last_week_function2, however, can take any string as an argument and print out the last letter of that string.

PLACEHOLDER VARIABLES

- `last_week_function_v2(first_name)` enables us to be much more flexible.
- To use an analogy, imagine you bought a fitbit, and rather than the fitbit calculating your resting heart rate, or the resting heart of anyone who wore it, it instead calculated the heart rate of some shmuck in Minnesota named Larry. And while you *may* be deeply concerned about Larry's slightly high RHR, it would not be a particularly useful product, because it would not be adaptable to the person wearing it.
- We want our `functions` to be as adaptable as possible. Adaptable functions prevent us from writing more functions. And we have not told you yet, but as people who use code, we all worship the majestic creatures of the sloth (more on that and the necessary blood sacrifices in session 7).

PLACEHOLDER VARIABLES

- Again, there are lots of functions 'built in' to python, and you can make your own with the `syntax` you've used above, but we don't want to overwhelm you with the idea that you need to learn all the functions in python *right now*.
- In most cases you'll find functions to solve a particular problem in tutorials or on **stackoverflow** answers (more about this in later sessions) and the more you use a particular function, the better you'll get to know it and the less you need to look it up.

PLACEHOLDER VARIABLES

If you are interested in diving deeper we strongly recommend 2 books:

1. Python crash course 2nd edition by Mathews (the first edition focuses on an outdated version of python)

- This is a really good, project based, introduction to python, with chapters on web development, game development, and data science.

2. Pandas for everyone by Chen

- This is *the* book on using python for data science and it includes a good general introduction to python.

For now, lets look at another place that we can use `placeholder` variables, and that's in `loops`.

LOOPS

- Since we conned… I mean convinced you to start learning python as a tool for data management and analysis we have been mentioning the power of getting the computer to do things for you.
- the power of `automation`.
- This is just taking a repetitive task and getting the computer to carry out those tasks for you, and `loops` are core of that process.
- We're going to start with `for loops` because they are the most common type of loop you'll use in python.

FOR LOOPS

- Lets imagine that you want to print out every item in a list, we've talked about how you might do this with a separate `print()` statement for each item

```
1 beatles = ['John', 'Paul', 'George', 'Ringo']
2 print(beatles[0])
3 print(beatles[1])
4 print(beatles[2])
5 print(beatles[3])
```

```
John
Paul
George
Ringo
```

FOR LOOPS

However, there are a lot of things wrong with that approach:

1. It's not really automation, you're giving each instruction line by line
2. It's not efficient code, you're wasting time typing out essentially the same statement
3. It's ugly (trust us, it's ugly code) and it can make it harder to fix issues when they pop up

This is where the power of `for loops` comes in, take a look at the example below:

```
1 # name is a placeholder variable
2 for name in beatles: # starts with 'for' and ends with a :
3     print(name) # each new line is indented
```

```
John
Paul
George
Ringo
```

FOR LOOPS

It might be apparent that this `syntax` is really similar to the `syntax` we use for `defining` a `function`, with some specific differences.

1. we start with `for` instead of `def` (this tells python that we're going to use a `for loop`, it's another `keyword`)
2. we enter a placeholder variable
3. we type 'in'
4. we type the name of the collection we want to work on (if it has one)
5. we end with a colon
6. we start each new line with an indentation
7. we `call` what ever function or functionality we want passing the placeholder variable as the argument (the placeholder can be any alphabetic character)

FOR LOOPS

````
```{python}
for i in collection:
    do_a_thing_with(i)
```
````

FOR LOOPS

- This `syntax` works on any `iterable`, which is just another name for a collection of objects like a list or string.

```
1 for l in beatles[0]: #using the first item in 'beatles', which is a string
2     print(l)# printing each letter of the string
```

```
J
o
h
n
```

FOR LOOPS - YOUR TURN

- So as you can see, `for loops` are quite simple, in essence, but they are really powerful, and honestly, they are just handy.
- In your note book:
  - make a list, it can be any list you like, but keep it simple,
  - then write a `for loop` that prints out each object in that list.
- In a new cell:
  - make a list of number, floats or ints,
  - write a for loop that multiplies all of those number by 10 and `print` out the result.

FOR LOOPS - EXAMPLE

- Lets imagine you have a raw data set from qualtrics in an excel file, and the person (or *Absolute flippin' ejit*) didn't name the questions in qualtrics when they were programing the survey (*blargh*).
- All of the items in the raw data are just called 'Q1, Q2' etc, and you have to clean the data, giving a useful name to each item.

FOR LOOPS - EXAMPLE

One of the ways that we might do this (ignoring actually importing the data into python for now) is make an empty list, and then populate that list using a `for-loop`. Lets imagine you want to name the items of the BFI-44:

```
1 bfi = [] #making an empty list
2
3 for i in range(1,45):# begining the for loop, and using the range function
4     bfi.append(f'bfi_{i}')#using the 'append' method and an f-string to append a formatted string to
5
6 print(bfi[0:10])# printing out the first ten items of the bfi list to check if the formatting has wo
```

```
['bfi_1', 'bfi_2', 'bfi_3', 'bfi_4', 'bfi_5', 'bfi_6', 'bfi_7', 'bfi_8', 'bfi_9', 'bfi_10']
```

FOR LOOPS - EXAMPLE

- You now have a clean list that you can use to rename columns, no copying and pasting, no moving between excel and spss, just 3 lines of code and your object is ready.
- We'll show you how to actually do this on 'real' data in future sessions but for now we just want to give you a sense of how to use simple `for-loops` and the kind of thing they might be used for.
- Ok we're nearly done with `for loops` for now, but there's a couple of other things we want to show you.

NESTED FOR LOOPS

- `for loops` can be `nested`, meaning that multiple `for loops` can be run at the same time, if, for example, you want to work with two or more lists at the same time.
- In the example below we have two lists of numbers and we want each number in the first list (called `ns`) to be multiplied by each number in the second list (called `np`) and then want some output to be produced that updates the user on what's happening (using an `f-string`).
- Take a look at the example we've used below, pay close attention to the `syntax`; specifically the `indentation`.

NESTED FOR LOOPS

```
ns = [6,5,4,3]# making a list of numbers
np = [4,3,2,1]# making another list of numbers

for i, n in enumerate(ns):# enumerate() is a new function that allows us to access the index and the
    #note the first indentation
    for j in np:
        # note the second indentation
        k = n*j
        print(f'The number at position {i}(which is {n}) has been multiplied by {j}. The result is {
```

```
The number at position 0(which is 6) has been multiplied by 4. The result is 24
The number at position 0(which is 6) has been multiplied by 3. The result is 18
The number at position 0(which is 6) has been multiplied by 2. The result is 12
The number at position 0(which is 6) has been multiplied by 1. The result is 6
The number at position 1(which is 5) has been multiplied by 4. The result is 20
The number at position 1(which is 5) has been multiplied by 3. The result is 15
The number at position 1(which is 5) has been multiplied by 2. The result is 10
The number at position 1(which is 5) has been multiplied by 1. The result is 5
The number at position 2(which is 4) has been multiplied by 4. The result is 16
The number at position 2(which is 4) has been multiplied by 3. The result is 12
The number at position 2(which is 4) has been multiplied by 2. The result is 8
The number at position 2(which is 4) has been multiplied by 1. The result is 4
The number at position 3(which is 3) has been multiplied by 4. The result is 12
The number at position 3(which is 3) has been multiplied by 3. The result is 9
The number at position 3(which is 3) has been multiplied by 2. The result is 6
The number at position 3(which is 3) has been multiplied by 1. The result is 3
```

NESTED FOR LOOPS

- Let's talk through that cell because it's one of the more complex examples we've thrown at you so far.

1. **Create Lists**:
    - `ns = [6, 5, 4, 3]` # Making a list of numbers.
    - `np = [4, 3, 2, 1]` # Making another list of numbers.
2. **First For Loop**:
    - `for i, n in enumerate(ns):` # `enumerate()` allows access to both index and item from `ns`.
    - *Callout*: Handy function for getting index and item in one go.
3. **Second For Loop**:
    - `for j in np:` # Looping through items in `np`.
    - *Callout*: Nested loop to access items in the second list.

NESTED FOR LOOPS

4. **Calculations**:

- `k = n * j` # Multiply items from both lists.
- `print(f'The number at position {i} (which is {n}) has been multiplied by {j}. The result is {k}')` # Print the result.
- *Callout*: Each level of indentation shows the relationship between code blocks.

KEY POINTS

- **Indentation**:
  - Shows Python the relationship between code lines and code blocks.
  - Essential for nested structures.
- As a further example, below is a function using a for loop to create a list of SPSS-like column names.

NESTED FOR LOOPS

```
```{python}
#Making a function that we might use often
def spss_names(x,y):
    '''This function takes a string and a number and uses them to make a list of
    SPSS like names that we can use to rename columns of our dataset'''
    t = []#making a temporary list
    for i in range(1,y+1):# starting a for loop, which is indented within the function
        t.append(f'{x}_{str(i)}')# this is the functionality of the for-loop
    # note the indentation here, it's part of the function, not the for-loop so its indentation move
    return(t)# This passes our new list back out to us so that we can work with it, we'll explain th

bfi = spss_names('bfi', 44)# using the new function we've made to make assign this list to a variabl

print(bfi)# printing out that new list to make sure that its what we want
```
```

- The cell above might seem a little daunting,
- but if you just relax (*breath in*) and take it line-by-line (*breath out*), you'll see that there's almost nothing in it that you haven't seen before.

NESTED FOR LOOPS

There are only 2 new things in that example above.

1. `Docstrings`: strings encased in triple quotes (”””) allow you to write `strings` that are often used in `functions` to write explanations of what the function does, you can think of them as long #comments.
2. The ‘`return()`’ statement: This is something that you have to put into a `function` if you want it to give you back a `variable`.

- Again there are loads of online tutorials about this
- basically, this tells python that the `function` is finished and to pass something back to the user.
- In earlier lessons you’ve just used function to print things out, but with a `return()` statement you can use `functions` to actually make `objects` that you can work with.

FOR LOOPS - YOUR TURN

- In your notebook:
  - `define` a function that takes a list of numbers and returns a list of those numbers squared.
  - hint: remember use can use the `**` operator to square a number.
  - hint: you can use the `append()` method to add items to a list (remember we made an empty list in the example above)
- In a new cell:
  - `call` the function to make a new list of squared numbers.
  - `my_variable = my_function([1,2,3,4,5])`
  - then use a `for loop` to print out each item in the new list.

This one is the most complex task we've given you so far. You can definitely do it, just work with your partner.

MODULES

- One of the most useful things Python enables you to do is to use code that has already been written. This code can either have been written by someone else (and they have been generous enough to the whole community to share their work) or by a prior version of you.
- Python enables this by using `modules`.

MODULES

Modules are other Python files that we want to incorporate into our code.

- Today, we want to walk you through how you import and use a module in Python.
- Python has a number of modules that readily available through something called the standard library, which is just a collection of modules that are readily available and easy to import when you use python.
- The modules we are going to work with today are both useful and easy to implement, they are called
    1. `pathlib`,
    2. `datetime`,

PATHLIB

- We've already talked a lot about `paths` in this course.
- You know that they are just the location of a file on your computer.
- And you also know that, depending on the operating system you are using, the way that you write a path can be different.
- We can think of `paths` as `strings` that tell the computer where to find a file.
- But there is a problem (surprise, surprise)

THE \ PROBLEM

- The problem is that the `\` character is used in python to signify other special characters.
- For example, `\n` is used to create a new line in a string.

```
1 print('This is a string with a new line \n in it')
```

```
This is a string with a new line
 in it
```

- This means that if you want to use a `\` in a string, you have to use `\\` instead.
- This can get really annoying, really quickly, especially if you are working with a lot of paths, or collaberating with someone who uses a different operating system (windows uses `\` but mac and linux use `/` remember?).
- The `pathlib` module solves this problem by providing a way to work with paths that is operating system agnostic.

PATHLIB

- To get you started with pathlib (and any module really), we need to `import` it into our document.
- `import` is a keyword in python that tells the computer to bring in a module that we want to use.

```
1 import pathlib #This line imports the pathlib module
```

- `pathlib` contains a number of `classes` and `functions` that we can use to work with paths.
- a `class` is just another word for a `data type`, so a `string` is a class, and an `integer` is a class, and a `list` is a class.
- `pathlib` has a `class` called `Path` that we can use to work with paths.
- and a bunch of `methods` that we can use to do things with those paths.

PATHLIB

```
1  current_dir = pathlib.Path.cwd() #Line 3
2  print(current_dir) #Line 4
```

```
/home/kevin/my_projects/ULpsych_programing_club/presentations/Day 1
```

- We created a variable `current_dir`.
- We then assign `pathlib.Path.cwd()` to that variable.
- To understand what this `pathlib.Path.cwd()` doohickey is doing, we will break it down into three parts, going from left to right.
    1. `pathlib`: We specify that this code is working from the pathlib module first.
    2. `.Path`: We specify that from the pathlib module, we are working with the Path class.
    3. `.cwd()`: From the path class, we are using the cwd() method.

PATHLIB

- Generally, whenever we import a module, and then use something from a class in its module, we can use the following syntax.

```
module.class.method() #if we are using a Class and a method

module.function() #if we are using a function
```

- If we know ahead of time that we will only be working with one class, then we actually cut the amount of code we need to write.
- For example, assuming we are only using the `Path class`, we could write

```
1 from pathlib import Path # Line 5
2 current_dir1 = Path.cwd() #Line 6
3 print(current_dir1) #Line 7
```

```
/home/kevin/my_projects/ULpsych_programing_club/presentations/Day 1
```

PATHLIB

- In the second example, rather than asking our computer to pull all the information from `pathlib`.
- I specify that `from pathlib`, I want the `Path` class.
- This means that I don't need to write pathlib in my code again after importing it.
- We will be coming back to more to pathlib in coming weeks. For now, we just want to show it to you as an introduction to modules.

The other module we will look at is `datetime`

IMPORTING THE MODULE DATETIME

- Python has a module named datetime to work with dates and times.
- This is useful as it enables us to handle different measurements of time (years, hours, days, seconds, miliseconds), to track timezones, the current date, day, along with a host of other information.
- It also allows us to work with dates regardless of the national format (American, European, etc).
- As you get further into complex data analysis, you will find that you will need to work with dates and times (timeseries data, for example).

IMPORTING THE MODULE DATETIME

- Let’s import the datetime module and, this time, let’s give it a nickname.

```
import datetime as dt #saving the module datetime under the nickname: dt
today = dt.datetime.now() #From the datetime module, select the class datetime (confusing I know),
# and use the now() function
print(f'Today is: {today}') #printing out the current date and time using an f-string
```

```
Today is: 2024-06-25 17:04:30.415917
```

IMPORTING THE MODULE DATETIME

- This time we imported the datetime module and gave it the nickname `dt`.
- This is really common practice as it cuts down the amount of time you need to write out the module name.
- We could have imported `pathlib as pl` and then used `pl` instead of `pathlib` in our code.
- We then used the `datetime` class from the `datetime` module and the `now()` method to get the current date and time.

USING THE MODULE DATETIME

- There is also a `date` class in the `datetime` module that we can use to work with dates.
- This is useful if you only need to work with dates and not times.
- The `date` class takes three arguments: year, month, and day which are all integers.
- The `datetime` class takes the same arguments as the `date` class, but it also takes the hour, minute, second, and microsecond as integers. For when you reeeeeeeally don't wanna miss lunch.

```
1 yester_year = dt.date(1993, 11, 6) #This uses the American format: Year, Month, Day
2 print(yester_year)
3 #we can even go further and print the hour, minute, second, and microsecond
4 yester_year1 = dt.datetime(1993, 11, 6, 18, 32, 12, 342380)
5 print(yester_year1)
```

```
1993-11-06
1993-11-06 18:32:12.342380
```

USING THE MODULE DATETIME

- That's just a small taste of what the `datetime` module can do.
- Anytime you need to work with dates and times, you can use the `datetime` module to make your life easier.
- As with all these modules you can find a huge amount of info online, and the more you use them, the more you'll get to know them.
- But you don't need to be an 'expert' at them, you can look up how to do specific things as you need them.

We're going to get you to practice with modules in a few minutes but first we need to take a little dive into `methods` and `attributes`.

METHODS AND ATTRIBUTES - FUNCTIONS RECAP

- In previous sessions we've talked about `functions` and `variables`.
- A `function` is a 'wrapper' for a sclonch of code that does something with a piece of data.
- A `variable` is a 'hook' that we can use to store data and then use that data later on.
- `functions` take in `arguments` which can be variables, or direct values, and they can `return` a value to us.

```
1 def multiply_numbers(number,multiplier): # we're defining a function that takes two arguments
2     return number * multiplier# this function multiplies the two arguments together and returns the
3
4 print(multiply_numbers(2,3)) # we're calling the function and passing it two values *within* the pri
```

METHODS

- Essentially a `function` is a piece of code that does something with and often it works on general data types like `integers`, `strings`, `lists`, etc. (This is a simplification but it's a good way to think about it for now).
- A `method` is a piece of code that is attached to a specific `type` (or `class`) of object.
- So where the `print()` function works on any object, the `append()` method works on `lists` (remember you saw this in an earlier session).

```
1 my_list = [1,2,3,4] # making a list
2 my_list.append(5) # using the append method to add a new item to the list
3 for number in my_list:
4     print(number) # printing out the list to see the new item
```

```
1
2
3
4
5
```

METHODS

- `Methods` are called using the `.` operator, and they can take `arguments` just like `functions`.
- Where the `function syntax` is `function_name(argument)`, the `method syntax` is `object.method(argument)`.
- So when you're trying to tell the difference between a `function` and a `method`, just remember that a `method` is attached to a specific `type` of object, and you call it using the `.` operator.

` function(argument) # this is a function

object.method(argument) # this is a method `

ATTRIBUTES

- `Attributes` are like `methods` in that they are attached to a specific `type` of object.
- But where `methods` are pieces of code that do something, `attributes` are pieces of data that are attached to an object.
- So where a `method` is a verb, an `attribute` is a noun.
- You can think of `attributes` as the 'properties' of an object, and `methods` as the 'actions' that an object can take.

```
1 from pathlib import Path # importing the Path class from the pathlib module
2 current_dir = Path.cwd() # using the cwd() method to get the current working directory
3 print(current_dir.parent) # using the parent attribute to get the parent directory of the current wo
```

```
/home/kevin/my_projects/ULpsych_programing_club/presentations
```

ATTRIBUTES

- `Attributes` are called using the `.` operator, just like `methods`, but they don't take `arguments`.
- So where the `method syntax` is `object.method(argument)`, the `attribute syntax` is `object.attribute`.
- When you're trying to tell the difference between a `method` and an `attribute`, just remember that a `method`, like a `function` is followed by `()`, and an `attribute` is not.

```
...object.method(argument) # this is a method

object.attribute # this is an attribute
```

ANNNNND RELAX

- That was a lot of information, and we've thrown a lot of new concepts at you.
- And we're going to give you the chance to get it all under your fingers in a moment.
- But before we do that, go take 10 minutes to stretch, get a drink, and relax a little.
- When you come back there will be a quiz to help you consolidate what you've learned today
- and then you're going to actually do stuff with all this new information.

LET'S GET DOWN TO BUSINESS (TO DEFEAT...)

- First, go do the quiz on brightspace, use it to remind you of the key points from today's session.
- Then, you and your partner are going to work together to do some bits and pieces.

YOUR MISSION (YOU DON'T HAVE A CHOICE ABOUT ACCEPTING IT)

- Start by making a new notebook called `day_1_project.ipynb`
- Go to brightspace and download the `instructions.txt` file and then copy it into the folder where you have your new notebook.
- Open it (in VScode) and read the instructions.
- Work *with your partner* to complete the tasks in the instructions.
- If you get stuck - call me. for_loops