PROGRAMMING: OBJECT-ORIENTED APPROACH 🧑‍💻

LOOPS

- Press `Space` to navigate through the slides
- Use `Shift+Space` to go back
- Save as **PDF**:
  - Open **Chrome** then **click here**
  - Press **Ctrl+P/Cmd+P** to print
    - Destination: **Save as PDF**
    - Layout: **Landscape**
  - Press **Save** button

LOOPS

- Loops are statements used in python to repeat things *iteratively*.
- Iterating just means to do something one after another.
  - For example: counting from one to 10 requires you to count the current number and then move on the the next *iteration* until you reach 10.
- In practice this means that loops are used to execute code multiple times until a certain condition is met.
  - For example: you could *loop* through the numbers 1-10 and print them out one after another.
- All loops are composed of the *loop type* (while, for etc.) followed by the *terminating condition* (when the loop should end) and the *loop body* (what the loop should do on each *iteration*.)

WHILE LOOP

- A *while* loop is exactly as it sounds, it **loops** will continue **until** the condition is **False**.
- The basic syntax for a while loop looks like this:

```
while condition:
    # Do stuff at this indentation level while condition is True
# Stuff at this indentation will run after the loop finishes
```

WHILE LOOP

- For example: if you wanted to count from 0-10 and print every number on each loop the code would look like this:

```
x = 0 # Setup a variable to use for the conditional

while x <= 10: # Continue looping until x is greater than 10
    print(x) # Print the current itterations value of x
    x += 1 # Inrement x by 1 (add 1 to the current value of x)

print("loop Finished") # This will execute after the loop since it's at a lower in
```

FOR LOOP

- For loops are a bit more complicated than while loops.
- They loop based on *iterables*, which are things in python that you can *iterate over*.
- Collections such as lists, and tuples (and even strings) are examples of *iterables*.
- The *terminating condition* during for loops is when you hit the end of the iterable (i.e. the last element in a list).
- The basic syntax of a for loop looks like this:

```
for variable in iterable:
    # Code at this indentation executes during the iteration---

# Code at this indentation does stuff after the loop
```

FOR LOOP

- Where variable is an arbitrarily named variable used to hold the current value of the iteration, and iterable is the item to iterate over.
- Let's take the example of a shopping list and say you wanted to loop through the values in the list and print them off one by one, you could use a for loop to do this:

```
shopping_list = ["Eggs", "Ham", "Milk"]

for item in shopping_list: # Iterate through the shopping list
    print(item) # Print the item at the current iteration
```

This would print:

```
>> Eggs
>> Ham
>> Milk
```

BREAK

- The break statement is used to end a loop immediately.
- It can (but does not have to) be used with if/elif/else statements to force a loop to stop iterating if a condition is met.
- Let's take the example that we want to create a for loop that print's out each letter of a string, but if the string contains a hyphen (-) then we want the loop to end.
- The code would look like this:

```
greeting = "Hello-World" # Setting up a string to iterate through

for character in greeting: # Iterate over the string one letter at a time
    if "-" in character: # if the current character is a hyphen
        print("Hyphen detected, ending loop!")
        break # End the loop
    else:
        print(character) # Print the current character
```

CONTINUE

- Continue statements are used to jump to the next iteration.
- So for example let's say you wanted to print the even numbers from 0-10 using a while loop, you could do it like this:

```
x = 0 # Initialize a variable to use in the condition

while x < 10:
    if ((x % 2) == 0): # If the number is even
        print(x)

    else: # If the number is odd
        continue # Go to next iteration
```

LOOP TRICKS AND TECHNIQUES

- Here are some additionally useful things you can do with loops, as well as things to avoid:
  - Infinite loops
  - Loop nesting
  - Using loops for validation
  - For loops over a set range

INFINITE LOOPS

- When you are making a *while* loop, inevitably you will create a loop that never ends.
- Some languages stop you from doing this, python is not one of them.
- If you create a loop that wont stop, you can force the program to stop running by pressing `Ctrl+C` on Windows or `Cmd+C` on Mac.

LOOP NESTING

- You can execute loops inside of loops, this is useful in many cases.
- A good real world example is that some people will put lists inside lists, or dictionaries inside dictionaries.
- Here is an example of going through a list of shopping lists:

```
shopping_lists = [["Eggs", "Milk", "Ham"],
                  ["Vinegar", "Mustard", "Ketchup"],
                  ["Burgers", "Lettuce", "Mayo"]]

for current_list in shopping_lists: # Steps through the list of lists
    for item in current_list: # Steps through each list
        print(item) # Prints the item in the current shopping list
```

This prints:

```
Eggs
Milk
Ham
Vinegar
Mustard
Ketchup
Burgers
Lettuce
Mayo
```

USING LOOPS FOR VALIDATION

- You can use a loop to validate input from a user.
- For example let's say you needed the user to input a number between 1-10, you could force them to do this in a loop like this:

```
while True: # This is an infinite loop
  number = int(input("Please type a number between 1 and 10: ")) # Take user inpu

  if number < 1: # Number is too small
    print("Number provided is less than 1")

  elif number > 10: # Number is too large
    print("Number provided is greater than 10")

  else: # If the input is in a valid range
    print("Number provided is between 1 and 10")
    break # End the loop
```

USING LOOPS FOR VALIDATION

- There is another way to do this syntactically:

```
valid_input = False # Used to mark if input is valid

while not valid_input: # Loop when valid input is False
  number = int(input("Please type a number between 1 and 10: ")) # Take user input

  if number < 1: # Number is too small
    print("Number provided is less than 1")

  elif number > 10: # Number is too large
    print("Number provided is greater than 10")

  else: # If the input is in a valid range
    valid_input = True # End the loop
```

FOR LOOPS OVER A SET RANGE

- It is possible to create for loops that will loop for a specific range of numbers (i.e. 2-8).
- Let's take the example of looping from 5-10:

```
for number in range(5,11):
    print(number)
```

- The range() function takes two arguments:
  - The first is the number to start at, and the second is the number to end on.
- Keep in mind – the end number needs to be 1 more than the value you want to end on (most people use this for lists so this ensures that you don't go past the range of the list).
- You can also opt to just specify an end number and the loop will start by default at 0. For example 0-10 would be:

```
for number in range(11):
    print(number)
```

EXERCISE TIME

- Check out the exercises_loops.py for some simple exercises to try out.

PROGRAMMING: OBJECT-ORIENTED APPROACH 🧑‍💻

FUNCTIONS

- Press `Space` to navigate through the slides
- Use `Shift+Space` to go back
- Save as **PDF**:
  - Open **Chrome** then **click here**
  - Press **Ctrl+P/Cmd+P** to print
    - Destination: **Save as PDF**
    - Layout: **Landscape**
  - Press **Save** button

FUNCTIONS

- We have already used a ton of functions; print(), *list*.append(), input(), int(), etc.
- This lecture will focus on learning how to write your own functions.
- Now there are a ton of use cases for functions, but typically there are 3 reasons to create a function vs just writing the code inline:
  - Reuse
  - Organisation
  - Modules/APIs

FUNCTIONS

1. Reuse

    - If your code can be generalized and reused in many places in your software, then usually it's better as a function.
    - If you're creating an e-commerce shop having a function to buy something is better than writing every possible purchase that could be made inline.

2. Organisation

    - Even if code is being used once it's sometimes easier to read if you are just calling functions instead of inline code.

FUNCTIONS

- Let's say for example you are creating a game, if you create functions with good names it's often easy to tell what's happening without having to go through many lines of code:

```
# Gameloop
while turns < 100:              # Game goes until 100 score
    player_move(player_one) # Player one's move
    player_move(player_two) # Player two's move
    turns += 1

if player_one.score > player_two.score:
    print("Player Two Wins!")
elif player_one.score < player_two.score:
    print("Player Two Wins!")
```

- Even without knowing what the functions are **exactly** doing you can get an idea of what's happening here.
- There's some game that goes on until the *turns variable* is 100 and player's take turn making moves, then after 100 *turns* whoever has higher score wins

FUNCTIONS

3. Modules/APIs

- Sometimes your code won't always be directly seen by people.
- Python uses a lot of modules (which can also be called, libraries, API's, or packages) to get a lot of functionality.
- For example: earlier in one of the challenges we *imported* the function random.randint() to generate a random number.
- If the creator of the module wrote the code inline then there would be no way to get that simple functionality out of the module (there will be more detail about this in the next module).

BASIC SYNTAX

- The basic syntax to define a function is using the keyword *def* followed by the function name (what you want to call it)...
    - then the arguments (what data the function needs to run), then the *docstring* (description of the function), then the function body (the code the function needs to run)
    - followed by an optional *return* statement (data the function should give back).
- Here is what it should look like if you have a function called *sum* that takes two integers, adds them, and returns the result:

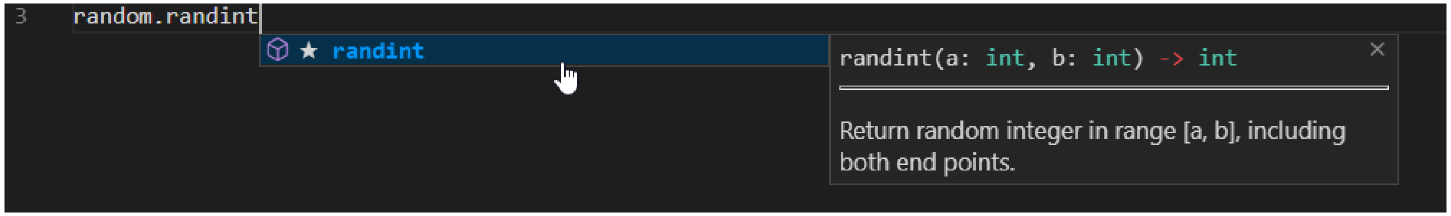
```
def sum(num_1, num_2):
    """
    Takes two variables (int's or floats), and
    adds them together, then returns the result
    """
    result = num_1 + num_2
    return result
```

COMMENTING/DOCSTRINGS

- Remember earlier in the course when I said that commenting is important?
- Well trust me it's really important for functions.
- Properly commenting functions can save you (and others who use your code) a ton of time.
- Specifically functions have special types of comments called *docstrings*, these *docstrings* are ways to tell people how to use your function properly.

COMMENTING/DOCSTRINGS

- Most text editors/IDE's use docstrings to help people write code with functions.
- For example: here is what the *docstring* for the random.randint() function looks like in visual studio code:



```
random.randint
```

```
randint(a: int, b: int) -> int
```

Return random integer in range [a, b], including both end points.

- As you can see it tells you what arguments you need (a and b), what types they should be (both ints), tells you what you will get back/returned (an int), and gives a short description of what the function does.
- Now this is sort of a best case scenario, but at least you should include a short description of what the function does, and what it returns even if you don't bother with anything else.

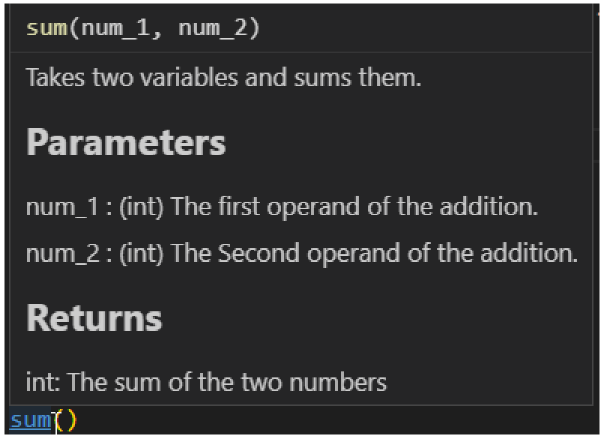
COMMENTING/DOCSTRINGS

- To create *docstrings* you simply need to add a multiline comment as the first line of a function definition.
- There's a ton of debate in python as to what is the best way to write *docstrings*, but the following convention is useful – numpy style.
- A basic docstring for the sum() example above would look like this:

```
def sum(num_1, num_2):
    """
    Takes two variables and sums them.

    Parameters
    ----------
    num_1 : (int)
        The first operand of the additio

    num_2 : (int)
        The Second operand of the additi

    Returns
    -------
    int:
        The sum of the two numbers
    """
    result = num_1 + num_2
    return result
```

COMMENTING/DOCSTRINGS

- Which then gives you something like this in visual studio code when you try to call the function:



```
sum(num_1, num_2)
```

Takes two variables and sums them.

Parameters

num_1 : (int) The first operand of the addition.

num_2 : (int) The Second operand of the addition.

Returns

int: The sum of the two numbers

```
sum()
```

VARIABLE SCOPE

- *Variable Scope* just refers to *where* you can access a variable from.
- In the case of functions you can only access variables that are in the functions from ***inside the function***.
- For example:

```
greeting = "Hello You!" # This would be called a global variable because it's not

def greet():
    """Prints a greeting"""
    greeting = "Good Bye!" # Now a 'function/local' greeting variable exists
    print(greeting) # This will use the 'function/local' variable

greet() # Prints: Good Bye!
print(greeting) # Prints: Hello You!
```

- As you can see the variable *greeting* that's outside the function has a lower 'priority' than the one inside the function, and the one inside the function has no effect on the one outside the function.
- As a rule of thumb you should always aim for 'function/local' variables since only the function has access to them.

ARGUMENTS/PARAMETERS

- Arguments (sometimes called parameters), are pieces of information you use to complete whatever your function needs to do.
- In the example I gave of the sum() function at the beginning of this module, there were two arguments: num_1 and num_2.
- Arguments are *passed* to functions and create a 'function/local' variable of the same name, to be used in any code running inside the function.
- They can be of any type, but there are a few different ways to setup arguments:

POSITIONAL ARGUMENTS

- These are the most common arguments, this is exactly what I used in sum() from earlier.
- They are arguments that the values are given based on *where* they were inputed.
- For example: let's take a function called difference which **subtracts** two numbers

```
def difference(num_1, num_2):
    """
    Takes two variables and subtracts the

    Parameters
    ----------
    num_1 : (int)
        The first operand of the subtract

    num_2 : (int)
        The Second operand of the subtrac

    Returns
    -------
    int:
        The difference of the two numbers
    """
    result = num_1 - num_2
    return result
```

POSITIONAL ARGUMENTS

Depending on which order I *pass* my arguments to the function changes which value is assigned to num_1 and num_2:

```
difference(4, 2) # Returns 2 because num_
difference(2, 4) # Returns -2 because nur
```

```
def difference(num_1, num_2):
    """
    Takes two variables and subtracts the

    Parameters
    ----------
    num_1 : (int)
        The first operand of the subtract

    num_2 : (int)
        The Second operand of the subtrac

    Returns
    -------
    int:
        The difference of the two numbers
    """
    result = num_1 - num_2
    return result
```

KEYWORD ARGUMENTS

- These are less common but incredibly useful ways of declaring variables that have 2 distinctions compared to positional arguments:

    1. They can be in any order
    2. They have a default value provided

- Let's take the example of the greet() function from earlier and give people the ability to customise the message:

```
def greet(name="John doe", greeting="Hel
    """Greets a person with the greeting

    Parameters
    ----------
    name: (str)
        The name to greet by.
    greeting: (str)
        The greeting to greet by.
    """
    print(name, greeting)
```

KEYWORD ARGUMENTS

- Now there are a few ways to call this function, because there are defaults for both variables already set we could just call it as it is:

```
greet() # Prints: Hello there: John Doe
```

- Or we can just change 1 of the variables:

```
greet(name = "Kieran Wood") # Prints: Hel
greet(greeting = "How it be: ") # Prints:
```

- Or both in any order:

```
greet(name = "Kieran Wood", greeting = "H
greet(greeting = "How it be: ", name = "K
```

```
def greet(name="John doe", greeting="Hel
    """Greets a person with the greeting

    Parameters
    ----------
    name: (str)
        The name to greet by.
    greeting: (str)
        The greeting to greet by.
    """
    print(name, greeting)
```

FUNCTION BODY

- As we have seen the function body is simply the statements that make up what the function needs to do.
- A function body for example could be a calculation, running the code necessary to print something to the screen, or even creating and returning information (such as in random.randint()).

RETURNS

- Return statements are optional, by default if one is not defined the function returns what's called a None (basically nothing, it just let's python know it's done running).
- You can return any type; an int, float, a collection (such as a string, list, dictionary), a logical or arithmetic operation, or even return a call to another function.

TYPE DECLARATIONS

- Optionally you can give what are called *type declarations*, these allow you to specify what *type* arguments and/or *returns* should be.
- This can be helpful to let people know what the function is expecting to receive and return.
- It can also help cut down on the amount you need to write for *docstrings*.
- Let's take the example of the sum() function from earlier:

```
def sum(num_1: int, num_2: int) -> int:
    """
    Takes two variables and sums them.

    Parameters
    ----------
    num_1 :
        The first operand of the additior

    num_2 :
        The Second operand of the additic

    Returns
    -------
    The sum of the two numbers
    """
    result = num_1 + num_2
    return result
```

TYPE DECLARATIONS

- As you can see all you need to do is add a `: type` after an argument, and an `-> return_type` after the arguments to specify argument types and return types respectively.
- One thing to keep in mind is that this **DOES NOT ENFORCE THE TYPES**, it merely gives people an idea of what they **should** do not what they **have to do**.

```
def sum(num_1: int, num_2: int) -> int:
    """
    Takes two variables and sums them.

    Parameters
    ----------
    num_1 :
        The first operand of the additior

    num_2 :
        The Second operand of the additic

    Returns
    -------
    The sum of the two numbers
    """
    result = num_1 + num_2
    return result
```

EXERCISE TIME

- Check out the exercises_functions.py for some simple exercises to try out.