FUNCTIONS

DR IAN CORNELIUS

HELLO

- Learning Objectives
    1. Understand the purpose of functions in Python
    2. Under the difference between parameters and arguments
    3. Demonstrate the ability to use functions

INTRODUCING PYTHON FUNCTIONS

- Functions are a block of reusable code that are used to perform a single action
- They provide an aspect of modularity to your code and ensures a high-degree of code reuse

CREATING A FUNCTION

- Functions in Python begin with the `def` keyword followed by a function **name** and brackets (“`()`”)
- The code within the function then starts with after the colon (“`:`”) at the end of the brackets, and is indented once

```
def function_name():
    print("Hello 4061CEM")
```

USING A FUNCTION

- Functions can be called using their function name, followed by a set of brackets
  - this is often known as the **function caller**

```
def function_name():
    print("Hello 4061CEM")
```

```
function_name()
```

```
function_name() -> Hello 4061CEM
```

RETURNING A VALUE FROM A FUNCTION

- Functions can also return data from inside it using the `return` statement
- Useful if you have performed some operations inside a function and need to use the output

```
def my_sum():
    x = 2
    return 5 + x
```

```
my_sum()
```

```
my_sum() -> 7
```

- You may want to store the returned value from a function in a variable, or print it to the terminal

```
summed = my_sum()
print(my_sum())
```

EMPTY FUNCTIONS (1)

- The purpose of a function is to have some re-usable code, therefore they cannot be empty
  - if you insist on having a function with no code, then you can use the `pass` statement

```
def my_function():
    pass
```

EMPTY FUNCTIONS (2)

- Empty functions offers to purpose or use; unless it is a placeholder for future code
  - in this instance, you would want it to raise a warning, such as: `NotImplemented`

```
def my_function():
    return NotImplemented
```

PARAMETERS AND ARGUMENTS

- Data can be passed through to a function and these are known as either parameters or arguments
- Parameter and argument can be used for the same thing
  - simply it is data that is passed into a function
- But they do have a slightly different meaning:
  - **parameter** is the variable listed inside the brackets in the function definition
  - **argument** is the value that is sent to the function

PARAMETER AND ARGUMENT EXAMPLE

- Parameters/arguments are specified after the declaration of the function name and inside the brackets
  - you are able to add as many parameters/arguments as you want, separating them with a comma (,)

```
def hello_person(name):
    print("Hello " + name + " and welcome to 4061CEM")
```

```
hello_person("Ian")
hello_person("Terry")
hello_person("Daniel")
```

```
hello_person(“Ian”) -> Hello Ian and welcome to 4061CEM
hello_person(“Terry”) -> Hello Terry and welcome to 4061CEM
hello_person(“Daniel”) -> Hello Daniel and welcome to 4061CEM
```

PASSING ARGUMENTS TO FUNCTIONS (1)

- There are two ways of passing arguments to a function: pass by value or pass by reference

PASS BY VALUE

- The function creates a copy of the variable passed to it as an argument
  - the actual variable itself is not affected

```
x = 10
def change_int(x):
    x = 20
```

```
change_int(x)
```

```
Before Function Call: x = 10 [Address = 2313581363728]
Inside Function: x = 20 [Address = 2313581364048]
After Function Call: x = 10 [Address = 2313581363728]
```

PASSING ARGUMENTS TO FUNCTIONS (2)

PASS BY REFERENCE

- The actual variable is passed to the called function
  - changes made to the variable inside the function will affect the original value

```
x = [4, 0, 6, 1]
def change_value(_list):
    _list[1] = -9
```

```
change_value(x)
```

```
Before Function Call: x = [4, 0, 6, 1] [Address = 2313582751296]
Inside Function: _list = [4, -9, 6, 1] [Address = 2313582751296]
After Function Call: x = [4, -9, 6, 1] [Address = 2313582751296]
```

NUMBER OF ARGUMENTS

- When calling a function, it must be called with the correct number of arguments
  - if you have a function with three arguments then you have to call the function with three arguments

```
def hello_person(name, code):
    print(f"Hello {name} and welcome to {code}!")
```

```
hello_person("Ian", "4061CEM")
hello_person("Terry", "4059CEM")
```

```
hello_person(“Ian”, “4061CEM”) -> Hello Ian and welcome to 4061CEM!
hello_person(“Terry”, “4059CEM”) -> Hello Terry and welcome to 4059CEM!
```

DEFAULT PARAMETER VALUES

- A function can be called without an argument if a default value has been assigned to the parameter
- The default value will only be evaluated once and makes a difference when the default value is a mutable object
  - i.e. a list, dictionary or an instance of most classes

```
def hello_person(name="Ian", code="4061CEM"):
    print(f"Hello {name} and welcome to {code}!")
```

```
hello_person()
hello_person(name="Terry", code="4059CEM")
hello_person("Daniel")
```

```
hello_person() -> Hello Ian and welcome to 4061CEM!
hello_person(name="Terry", code="4059CEM") -> Hello Terry and welcome to 4059CEM!
hello_person("Daniel") -> Hello Daniel and welcome to 4061CEM!
```

KEYWORD ARGUMENTS

- Keyword arguments are related to the function calls
- When they are used in a function call, the caller identifies the arguments by its parameter name
  - the notation of using this method is: `parameter = value`
  - when used in a function call, the order of arguments do not matter

```
def hello_person(code, name):
        print("Hello " + name + " and welcome to " + code + "!")
```

```
hello_person(name="Ian", code="4061CEM")
hello_person(code="4059CEM", name="Terry")
```

```
hello_person(name="Ian", code="4061CEM") -> Hello Ian and welcome to 4061CEM!
hello_person(code="4059CEM", name="Terry") -> Hello Terry and welcome to 4059CEM!
```

ARBITRARY ARGUMENTS

- When you do not know the number of arguments that will be passed into a function, add an asterisk (*) before the parameter name
- The function will then receive a `tuple` of arguments and access the items accordingly
  - the tuple will remain empty if no arguments are passed through

```
def hello_person(*details):
    print("Hello " + details[0] + " and welcome to " + details[1] + "!")
```

```
hello_person("Ian", "4061CEM")
hello_person("Terry", "4059CEM")
```

```
hello_person("Ian", "4061CEM") -> Hello Ian and welcome to 4061CEM!
hello_person("Terry", "4059CEM") -> Hello Terry and welcome to 4059CEM!
```

ARBITRARY KEYWORD ARGUMENTS

- When you do not know the number of keyword arguments that will be passed into a function, add a double asterisk (**) before the parameter name
- The function will receive a `dictionary` of arguments and access the items accordingly

```
def hello_person(**details):
    print("Hello " + details['name'] + " and welcome to " + details['code'] + "!")
```

```
hello_person(name="Ian", code="4061CEM")
hello_person(code="4059CEM", name="Terry")
```

```
hello_person(name="Ian", code="4061CEM") -> Hello Ian and welcome to 4061CEM!
hello_person(code="4059CEM", name="Terry") -> Hello Terry and welcome to 4059CEM!
hello_person(name="Ian", code="4061CEM") -> Hello Ian and welcome to 4061CEM!
hello_person(code="4059CEM", name="Terry") -> Hello Terry and welcome to 4059CEM!
```

FUNCTION ANNOTATIONS

- Function annotations are optional metadata information about the various data types used by user-defined functions
  - these annotations are stored in the `__annotations__` attribute of a function
- Annotations for a parameter are defined with a colon (`:`) after the name of the parameter
- Annotations for a return are defined by a `->` followed by the data type
  - this is placed between the list of parameters and the colon denoting the end of the `def` statement

```
</> def my_sum(x: int) -> int:
        return 5 + x
```

```
</> print(my_sum.__annotations__)
```

```
▶ {'x': <class 'int'>, 'return': <class 'int'>}
```

GOODBYE

- Questions?
  - Post them in the **Community Page** on Aula
- Contact Details:
  - Dr Ian Cornelius, ab6459@coventry.ac.uk