Recapping Python from Year One

Dr Ian Cornelius

Hello

Hello (1)

Learning Outcomes

1. Understand the basic concepts of the Python programming language
2. Demonstrate knowledge learnt by creating simple Python scripts

Using Python

Using Python (1)

- Python 3.x is the language of choice for this module
  - the simple nature of Python makes it a great programming language for beginners
  - it is cross-platform and works across all major operating systems
- Requires an installation of Python
  - Windows: Download the Executable Here
  - Linux: `sudo apt install python3`
  - macOS: Download the Installer Here

Using Python (2)

macOS and Linux

- Typing `python3` into the terminal window will call the Python interpreter

```
$ python3 filename.py
```

Windows

- Typing `py` or `python3` into the command-line or PowerShell will call the Python interpreter
  - only a single command will work
  - depends on the method of installation

```
$ py filename.py
```

Using Python (3)

Recommended Integrated Development Environment

- Supported IDE: **JetBrains IntelliJ IDEA Community/Ultimate**
  - Apply for an Educational Licence
  - Download JetBrains IntelliJ IDEA
- Features:
  - Debugging
  - Code Refactoring and Profiling
  - Version Control Integration
  - Python, Java, Kotlin, PHP, etc.

Variables and Data Types

Variables and Data Types (1)

Variables

- Variables in Python are a reserved memory location to store values of a particular data type
- They consist of two parts: a *name* and a *value*
    - the names of a variable can be long or short

```
x = 0
module_number = "5032CEM"
roomNumber =  "ECG-01"
```

Naming Convention

- There are a few rules to follow when naming your variables:
    1. Names must start with a letter or an underscore
    2. Names can only contain alphanumeric characters and underscores (`A-z`, `0-9`, and `_`)
- If you need to supply a comment to explain a name, then it does not reveal its true intent
    - if this is the case, then you may want to reconsider renaming your variable

```
et = 0 # Elapsed time
# Could be:
elapsedTime = 0
elapsed_time = 0
```

- Note that variable names are case-sensitive:
    - i.e. `module`, `Module` and `MODULE` are all different variables (and memory locations)

Variables and Data Types (2)

Data Types

- Python has **six** built-in data types:
  1. None: a null object
  2. Boolean: `True` or `False`
  3. Numeric: integer, float, and complex
  4. Sequence: strings, lists, and tuples
  5. Maps: dictionaries
  6. Sets

Variables and Data Types (3)

Determining the Type of Variable

- If you are unsure about the type of variable, you can use the `type()` function
- This will return the class type of the object/variable

```
stringExample1 = "Hello 5062CEM"
type(stringExample1)
type("Hello 5062CEM")
```

```
type(stringExample1) -> <class ‘str’>
type(‘Hello 5062CEM’) -> <class ‘str’>
```

Variables and Data Types (4)

Type Casting

- You can specify a data type to a value by a process known as **type casting**
- As Python is an object-oriented language, classes are used to define its data types
- To cast a value as a particular type, you can use the built-in data types class constructors:
  - i.e. `float()`, `int()` and `str()`

```
castingInteger1 = int(5.6)
castingInteger2 = int("8")
castingInteger3 = int(True)
```

```
castingInteger1 -> 5, <class ‘float’> to <class ‘int’>
castingInteger2 -> 8, <class ‘str’> to <class ‘int’>
castingInteger3 -> 1, <class ‘bool’> to <class ‘int’>
```

Operators

Operators (1)

- An operator is a character that represents an action of some sort
- They are used for performing operations on variables and values (otherwise known as operands)
- Python has a collection of operators built-in:
  - Arithmetic: `addition`, `subtraction`, `division`, `floor division`, `multiplication`, `exponentiation` and `modulus`
  - Comparison: `same as`, `not equal`, `greater than`, `greater than or equal to`, `lower than`, and `lower than or equal to`
  - Logical: `and`, `or` and `not`
  - Identity: `is` and `is not`
  - Membership: `in` and `not in`

Conditional Statements

Conditional Statements (1)

- Comparison and logical operators are used with conditional statements to ensure certain conditions have been met
- Recap on the comparison and logical operators:

Comparison Operators

Operator	Explanation
`==`	The Same
`!=`	*Not* the Same
`>`	Greater Than
`>=`	Greater Than or Equal To
`<`	Less Than
`<=`	Less Than or Equal To !

Logical Operators

Operator	Explanation
`and`	Both comparisons evaluate to `True`
`or`	One comparison evaluates to `True`
`not`	Inverts the evaluated boolean

Conditional Statements (2)

- A basic decision statement which is done using a selection structure
- The decision will be described to the interpreter by a conditional statement
  - whereby a result can only be `True` or `False`
- Python allows the following:
  - `if` statements
  - `if ... else ...` statement
  - `if ... elif ... else` statement
  - Nested `if` statements

Conditional Statements (3)

`if` Statements

- Often referred to as a decision-making statement
- Used to control the flow of execution for statements and to test an expression
    - tests logically whether a condition is `True` or `False`

```
if variable == value:
    ...
```

`if else` Statements

- Known as an alternative execution, whereby there are two possibilities
    - the condition statement determines which of the two statements gets executed
- The `else` is used as the ultimate result for a test expression
    - this result is only met if all other statements are `False`

```
if variable == value:
    ...
else:
    ...
```

Conditional Statements (4)

`else if` Statements

- `elif` is a keyword in Python to replace the `else if` conditions from other languages
- The condition allows for two or more possibilities, known as a **chained conditional**

```
if variable > value:
    ...
elif variable < value:
    ...
else:
    ...
```

Nested `if` Statements

- `if` statements can be written inside each other, and is known as **nesting**

```
if variable == value:
    if variable == value:
        ...
    else:
        if variable == value:
            ...
        elif variable == value:
            ...
        else:
            ...
elif variable != value:
    ...
else:
    ...
```

Control Statements

Control Statements (1)

- Typically, statements in code will be executed sequentially
- There are some situations which require a block of code to be repeated
  - i.e. summing numbers, entering multiple data points, capturing user input
- Control statements, otherwise known as **loop statements** are required
- Python has two loop structures:
  - `while` - conditional loops
  - `for` - counter controlled loops

Control Statements (2)

Structure of a Loop

- Loop structures can be likened to a conditional statement
  - they run on a `True` or `False` set of values
  - the loop will continuously loop until the condition is `True`
  - the loop will terminate when the condition is `False`
- Loops can run for a desired length of time
  - or until a user-defined flag terminates it
- Loops are great for re-using code
  - limiting the number of statements that are required
  - re-uses the same conditional arguments for testing instead of hundreds

Control Statements (3)

`while` Loops

- `while` loops, are loops that will execute zero or more times before it is terminated

```python
while variable < value:
    ...
    variable += 1
```

- If you are doing an incremental loop, you need to manually increase the variable
    - hence the `variable += 1`

Control Statements (4)

`for` Loops

- A `for` loop is a loop designed to increment a counter for a given range of values
- They are best suited for problems that need to iterate a specific number of times
  - i.e. looping through a directory or set of files
- The structure of a `for` loop consists of the following:

1. Initialisation of a counter
2. Test the counter-variable:
   a. less than: `start < stop`
   b. greater than: `start > stop`
3. Update the counter-variable

```
for variable in range(x, y, step):
    ...
```

Control Statements (5)

Terminating Loops

- `break` statements can be used to stop the loop if a condition is evaluated to `True`

```python
whileExample1 = 0
while whileExample1 < 10:
    print(f"whileExample1 -> {whileExample1}", end="\n\n")
    if whileExample1 == 5:
        break
    whileExample1 += 1
```

```
whileExample1 -> 0
whileExample1 -> 1
whileExample1 -> 2
whileExample1 -> 3
whileExample1 -> 4
whileExample1 -> 5
```

Functions

Functions (1)

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

Creating a Function

- Functions in Python begin with the `def` keyword followed by a function **name** and a set of 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():
  ...
```

Functions (2)

Using a Function

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

```
def hello():
    print("Hello 5062CEM")
hello()
```

```
hello() -> Hello 5062CEM
```

Functions (3)

Returning Values 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

```python
def hello():
    return "Hello 5062CEM"
hello()
```

```
hello() -> Hello 5062CEM
```

Functions (4)

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

```python
def hello(name):
    return f"Hello {name} and welcome to 5062CEM."
hello("Ian")
hello("Terry")
hello("Daniel")
```

```
hello(“Ian Cornelius”) -> Hello Ian Cornelius and welcome to 5062CEM.
hello(“Terry Richards”) -> Hello Terry Richards and welcome to 5062CEM.
hello(“Daniel Goldsmith”) -> Hello Daniel Goldsmith and welcome to 5062CEM.
```

Functions (5)

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

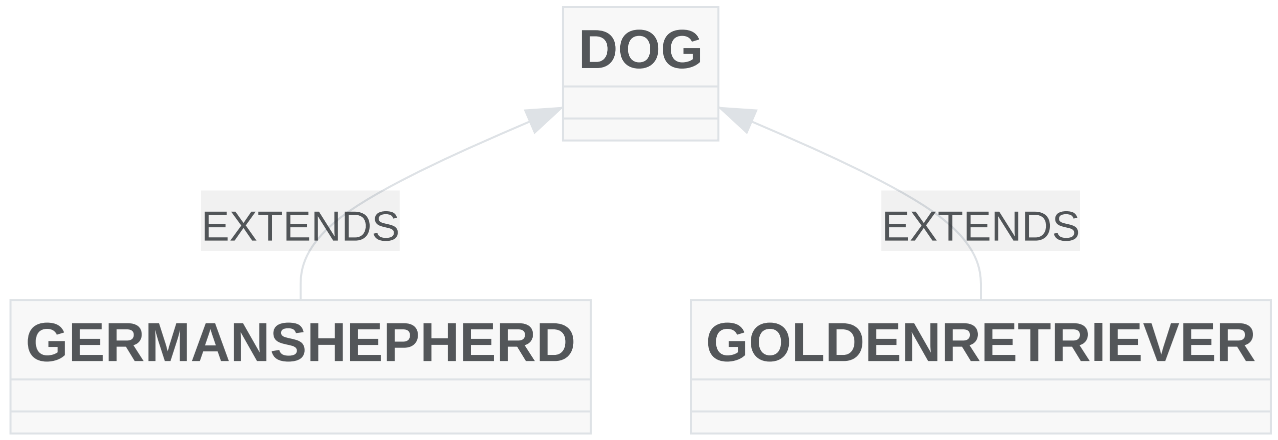
```python
def hello(name="Ian Cornelius", code="5062CEM"):
    return f"Hello {name} and welcome to {code}!"
hello()
hello("Terry Richards")
hello("Terry Richards", "5034CEM")
```

```
hello() -> Hello Ian Cornelius and welcome to 5062CEM!
hello(“Terry Richards”) -> Hello Terry Richards and welcome to 5062CEM!
hello(“Terry Richards”, “5034CEM”) -> Hello Terry Richards and welcome to 5034CEM!
```

Classes and Objects

Classes and Objects (1)

- Classes provide a structure for the objects
- They are used for defining:
  - a set of properties, represented by variables
  - the behaviour, which is represented by functions



Classes and Objects (2)

Creating a Class and Object

- Classes will be defined using the `class` keyword followed by the name you want to give it

```
class myClassName:
    ...
```

- Creating an object is achieved by creating a variable and calling our class name with a set of round brackets (`()`)
  - i.e., `objectExample1 = myClassName()`

Classes and Objects (3)

Using Class Constructors

- All classes consist of an in-built function which is used to execute code when it is being initiated
  - this is the function known as `__init__()`
- This initializer can be used to assign values to an object properties,
  - or other operations that are necessary to perform when an object is in the process of being created
- `__init__()` is called automatically each time the class has been used to create a new object

```python
class Lecturer:
    def __init__(self, _name, _age):
        self.name = _name
        self.age = _age
lecturer1 = Lecturer("Ian Cornelius", 34)
lecturer2 = Lecturer("Terry Richards", 1)
```

Classes and Objects (4)

Class Functions

- Classes can also consist of functions, and these will belong to the object that is created

```python
class Lecturer:
    def __init__(self, _name, _age):
        self.name = _name
        self.age = _age
    def hello(self):
        return f"Hello {self.name}, you are {self.age} years old."
lecturer1 = Lecturer("Ian Cornelius", 34)
lecturer2 = Lecturer("Terry Richards", 1)
```

```
lecturer1.hello() -> Hello Ian Cornelius, you are 34 years old.
lecturer2.hello() -> Hello Terry Richards, you are 1 years old.
```

Classes and Objects (5)

Public and Private

- Variables and functions can be set to private inside a class
    - this will restrict access to the variable *outside* the class
    - however, accessing the variable *inside* the class is permissive
- This is achieved by adding two underscores (`__`) to the beginning of the variable name

```python
class Lecturer:
    def __init__(self, _name: str, _age: int):
        self.name = _name
        self.__age = _age
    def change_age(self, new_age: int) -> None:
        self.__age = new_age
    def get_age(self) -> int:
        return self.__age
lecturer1 = Lecturer("Ian Cornelius", 34)
lecturer1.change_age(35)
```

```
[Before] lecturer1.get_age() -> 34
[After] lecturer1.get_age() -> 35
```

Goodbye

Goodbye (1)

Questions and Support

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