COM4013
Introduction to Software Development

- Week 9
- Classes and Enums

- *Umar Arif – u.arif@leedstrinity.ac.uk*

Class: Recap

- Classes provide a means of bundling data and functionality together.
- Creating a new class creates a new type of **object**, allowing new ***instances*** of that type to be made.
- Each class instance can have **attributes/mVariables** attached to it for maintaining its **state**.
- Class instances can also have **methods** (defined by its class) for modifying its **state**.

Dot Operator

In Python the dot (.) operator is used to access a classes member variables and its methods.

```
class CSquare:
    . . . # Constructor and member variables above

    # Change member variable values with Setter method
    # This is good practice
    def SetLength(self, length:float):
        self.mLength = length
```

Before we went over Getters which are methods that we use to get a variable from a class. Above is an example of a Setter which is used to change the value of a member variable after it has been declared

```
square.SetLength(14) # We use the dot operator to access
the class method here
```

Function: Recap (example)

Why do we use classes instead of functions?

- Generally speaking before using objects, functions are the primary way of **organising** our code.
- They allow us to **reuse** the same code, and give it a name.
- Functions have **parameters**, and also their own **isolated scope**.
- The name and parameters of a function are often called the **definitions**.

```
def SayHello( name:str ): # The :str is a type hint
```

- The **definition above** tells you how to use the function.

The function definition above tells us that the function name is SayHello and that I must provide it with a string parameter from the main function.

Class: Recap (example)

Why do we use classes instead of functions?

- Sometimes in our code we find that there are functions and variables that all relate to one thing.
- Classes are a great way to group these many things together.
- Once a class has been created then we can use it as a template to create many objects from it.

- Let's look at this example:

    - We have a guy called Umar.
    - He is 27 years old.
    - His year of birth is 1996.
    - He has a BSc in Computer Science.
    - He graduated in 2023.
    - He went to the University of Central Lancashire.

- So let's write this in our main.

Class: Recap (example)

Umar, 27, born in 1996, holds a BSc in Computer Science from the University of Central Lancashire, graduated in 2023.

```
def main:
    name             = "Umar Arif"
    age              = 27
    yearOfbirth      = 1996
    degree           = "BSc in Computer Science"
    university       = "University of Central Lancashire"
    yearOfGraduation = 2023

    print(name + " " + age + " " + yearOfBirth...
```

Now let's say he has a brother called Ihtishaam, born in 1994, holds a BSc in Accounting and Finance from the University of Central Lancashire, graduated in 2021.

We would have to make more variables to store and then print this...

Class: Person

```
class CPerson:
    mName           = ""
    mAge            = 0
    mYearOfbirth    = 0
    mDegree         = ""
    mUniversity     = ""
    mYearOfGraduation = 0

    def __init__(self, name, age, yOB, degree, uni, yOG):
        self.mName = name
        ... # constructor used to set mVars in the class

    # All we want to do is print, so let's provide this...
    def __str__(self):
        return f"{self.mName}, {self.mAge} years old, born
        in {self.mYearOfbirth}, holds a {self.mDegree} from
        {self.mUniversity}, graduated in
        {self.mYearOfGraduation }"
```

Class: Constructor

```
class CPerson:
    ... Member variables above

    def __init__():
```

- The **constructor** is a special method, it is not called explicitly, but it is executed when a new object is created
- It can be used to assign or compute the initial values of some of a classes member variables/properties.
- We usually use getters and setter methods to change these values once a class has been initialised.

Class: Person in Main

```
def main():
    umar = CPerson("Umar", 27, 1996, "BSc in Computer
    Science", "University of Central Lancashire", 2023)
    ihty = CPerson("Ihty", 28, 1994, "BSc in Accounting and
    Finance", "University of Central Lancashire", 2021)

    print(umar)
    print(ihty)

    -- Output --
    Umar, 27 years old, born in 1996, holds a BSc on
    Computer Science from the University of Central
    Lancashire, graduated in 2023.

    Ihty, 28 years old, born in 1994, holds a BSc in
    Accounting and Finance from the University of
    Central Lancashire, graduated in 2021.
```

Class: Person

```
class CPerson:
    ... # Member variables above

    # Some of the class method definitions below
    def __init__(self, name, age, yOB, degree, uni, yOG):
    def GetAge(self):
    def SetDegree(self, degree):
    def GetYearsGraduated(self):
    def ApproximateDaysAlive(self):
    def IncrimentAge(self):
    def __str__(self):
```

- We've set up our constructor for initialising the class.
- Additionally, we've implemented getter and setter methods to access and modify member variables.
- There are also other handy functions available for optional use.
- Moreover, we've **overridden** the print function to neatly display all class information when needed.

Class: Person

```
class CPerson:
    ... # Member variables above

    # Some of the class method definitions below
    def GetYearsGraduated(self):
        return 2023 - self.mYearOfGraduation

    def IncrimentAge(self):
        self.mAge += 1
```

- We're using the self keyword to get access to the values inside of the member variables.
- The first method uses an ***expression*** to return the number of years a person has been a graduate for.
- The second method ***increments*** the mAge member variable once – maybe it's the persons birthday... ¯_(ツ)_/¯.

- Are there any magic numbers in the code?

Class: Person

Let's access this in main:

```
def main():
    # Same as before
    umar = CPerson("Umar", 27,
    1996, "Computer Science", "UCLan", 2023)
    ihty = CPerson("Ihty", 28, 1994, "Accounting
    and Finance", "UCLan", 2021)

    # Umar has had a birthday – let's increment his age
    umar.IncrimentAge()
    print( umar.mAge ) # What will the output be?

    # Let's add the two brothers ages together
    siblingsAgeMerged = umar.mAge + ihty.mAge
    print( siblingsAgeMerged ) # What will the output be?
```

Enumerated Types

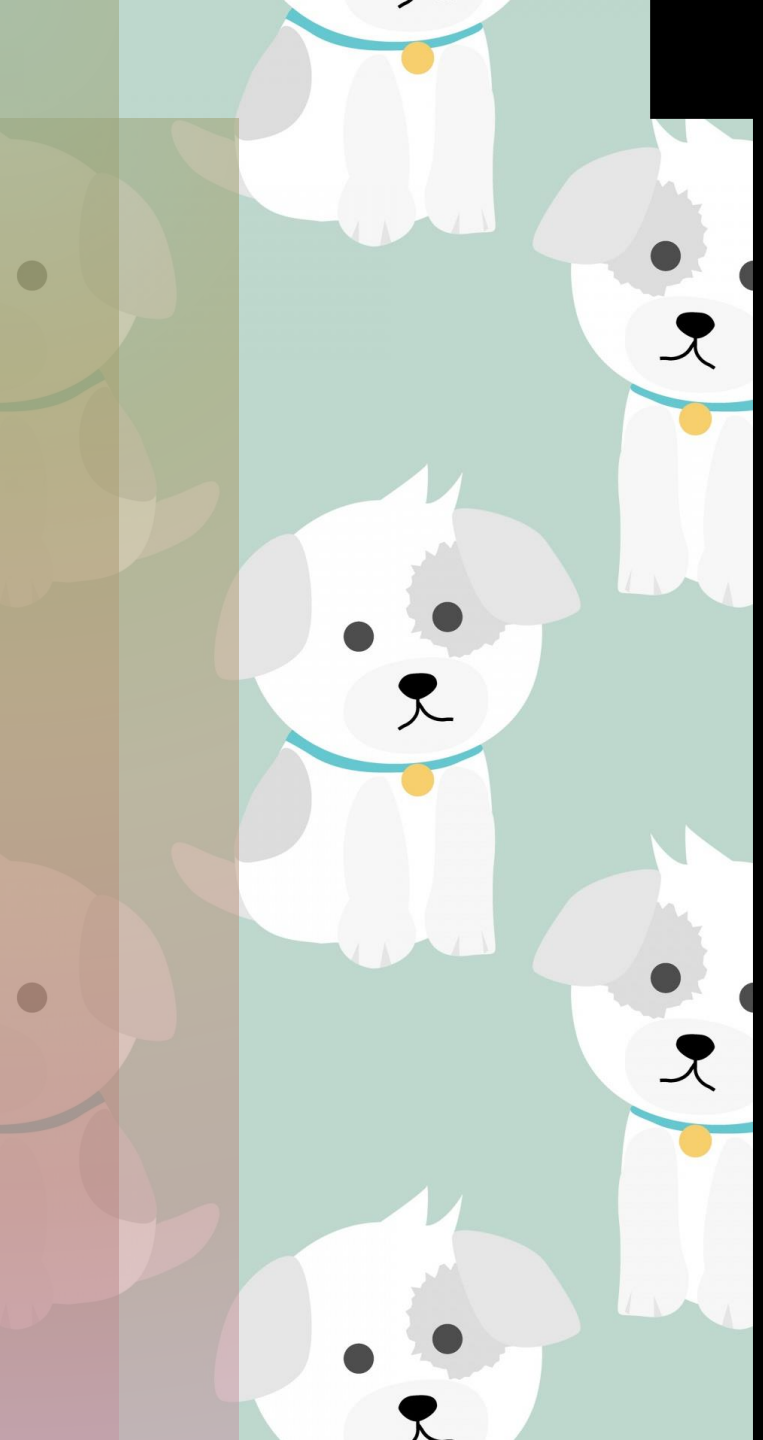
- In Python, an enumeration (enum) is a symbolic name for a set of values.
- Enums are created using the enum module.

```
from enum import Enum
```

- Enums offer a means to assign descriptive names to constant values, adding clarity and meaning beyond simple integers.
- Consider a scenario like building a Susceptible, Infected, and Recovered (SIR) model in Python, where using 0, 1, and 2 as placeholders is feasible but lacks intuitive understanding.
- By employing enums, we enhance code readability and avoid the challenge of recalling the significance of each numeric representation.

Enumerated Type: How

```
from enum import Enum       # Call this library
from random import uniform # Get a value between 0 and 1

# Define an enumeration class – Note the E in the name
class EHealthState(Enum):
    SUSCEPTIBLE = 0
    INFECTED    = 1
    RECOVERED   = 2

def main():
    personName   = "Umar"
    currentState = EHealthState.SUSCEPTIBLE

    randomValue  = uniform(0, 1)

    if ( randomValue > 0.75 ):
        currentState = EHealthState.INFECTED

    print(f"Current State: {current_state}")
```



Enumerated Breakdown

```
[1]from enum import Enum       # Call this library
[2]from random import uniform # Random value between 0 - 1

# Define an enumeration class – Note the E in the name
[3]class [4]EHealthState([5]Enum):
    [6]SUSCEPTIBLE = 0
    INFECTED       = 1
```

1. Must call Enum from enum library to access this functionality.
2. Must call uniform from random to access the uniform function.
3. Enums, in Python, are essentially classes, but in this context, you don't need to explicitly write a constructor or other methods.
4. When setting a variable to one of the enum states, use the name of the enum class.
5. This class inherits its functionality from the Enum class; therefore, it needs to be passed as a parameter during class definition.
6. We must explicitly define the states of the enum along with their individual values.

Encapsulation

- One important use of OOP is helping separation of concerns: by keeping some members (properties or methods) private, we can ensure that other objects only depend on a selected set of public members.
- We can then change the private members without (hopefully) breaking any external code.
- The public members of a class are the API: this is what we normally find in the documentation of a library.
- Python does not really have private members, but they are defined by convention using _underscore

```
class CPerson:
    mName   = ""
    mAge    = 0
    __mNino = 0 # National insurance number
```

In this example we probably wouldn't want individuals to be able to access their national insurance numbers, without going through some security checks.

Inheritance

- In some cases, it may be useful to have different classes that share some properties and methods.
- This may mean the respective real-world entities have something in common, they may be of the same kind.
- Inheritance allows us to re-use code and avoid duplication, while keeping the model understandable.
- We define a class to “inherit” the methods and properties of a parent class. This should only be used when the descendant is a **specialisation** of the parent.

- A square is a kind of rectangle – so it can inherit it's vars and methods.

```
class CSquare( CRectangle ):
    def __init__(self, x, y, size):
        super().__init__(x, y, size, size)
```

- When you create a subclass, you may want to reuse and **extend** the methods of the parent class. `super()` allows you to call a method from the parent class within the subclass.

Summary

We've covered quite a few topics today:

- Classes
- Methods
- Dot Operator
- Enumerated Type classes

Any questions?