<strong>Object-oriented programming</strong> is a way of structuring your code by bundling related lines of code into individual objects. In python, we create objects by creating classes and then we define methods under each class. For example

stuff = list()

stuff.append('dumebi')

stuff.sort()

The first line of code above constructs an object of type <code>list</code>, the second line calls the <code>append()</code> method and the third line calls the <code>sort()</code> method. This is an example of what object-oriented programming is like.

Classes are used to create user-defined data structure other than the default ones that come with python for example class “list”. Classes define functions called method inside the class. It is worth noting that a class is a blueprint for how an object would be made. An analogy I like relating to this is that a class is a cookie cutter. You can use the cookie cutter to make cookies as it is just a template. It is when you call the class that the object is created. When you call the class, an <em>instance</em> of that class is created. A class is like an empty form and an instance is a filled form.

class Player:

    def \_\_init\_\_(self, letter):

        self.letter = letter

To create a class you use the <code>class</code> keyword followed by the name of the class. It is worth noting that by convention, class names are written in “CapitalizedWords”. If you want every instance of the Player class to have some attributes you define then in the <code>\_\_init\_\_</code> method. This is when the class is initialized. You can pass in different attributes into the <code>\_\_init\_\_()</code> but the first parameter is always a variable called self. Self points to the particular instance of the Player class. Any object created from this Player class would have the syntax <code>x = Player(letter)</code> because we initialized the Player class with a letter parameter. <code>self.letter = letter</code> means that whatever is passed in as letter parameter when calling the class is assigned to the letter attribute of the class instance. <code>x = Player(“X”)</code> in this instance, the X is assigned to the letter. Attributes created inside the <code>\_\_init\_\_()</code> are called instance attributes. The value of the attribute is specific to the instance of that class. Class attributes however have the same value for all class instances.

class Player:

    game = "Tic-Tac-Toe"

    def \_\_init\_\_(self, letter):

        self.letter = letter

In the example above, game is a class attribute. In the example below. We create an instance of our class. We can access the attributes both class attributes and instance attributes by specifying the attribute name after a dot (.)

a = Player("X")

print(a.letter)

print(a.game)

Inside our class, we can define methods. These are instance methods that can only be called from an instance of the class. Methods are functions win the class.

class Player:

    game = "Tic-Tac-Toe"

    def \_\_init\_\_(self, letter):

        self.letter = letter

    #Instance method

    def print\_game(self, position):

        print(f"welcome to a game of {self.game}, you are letter {self.letter} and are in position {position}")

a = Player("X")

a.print\_game(5)

Methods like .\_\_init\_\_() are called <em>dunder methods</em> because they begin and end with double underscores. They can be used to customize classes in Python. To get a full list of the methods that a class has, you can make use of the <code>dir(class\_instance)</code> keyword e.g. <code>print(dir(a))</code> would give you all the methods that a has including the dunder methods.

<h5>Class Inheritance</h5>

Inheritance is when one class takes all the attributes and the methods of another class. The new class formed is referred to as a child class while the class that the new class is based on is called the parent class. Child classes can override or extend the methods and attributes of their parent class.

class HumanPlayer(Player):

    pass

a = HumanPlayer("X")

print(a.game)

You can override the methods by defining a method inside the child class with the same name inside the parent class. In the example below, we override the need to input a position when calling the <code>print\_game()</code> method of the <code>HumanPlayer</code> class

class HumanPlayer(Player):

    def print\_game(self, position=4):

        print(f"welcome to a game of {self.game}, you are letter {self.letter} and are in position {position}")

a = HumanPlayer("X")

a.print\_game()

when inside a method of a child class, we can get access to the parent class methods by using super().

class HumanPlayer(Player):

    def print\_game(self, position=4):

        super().print\_game(position)

a = HumanPlayer("X")

a.print\_game()

In the code above, super() searches the parent class for a method called <code>print\_game()</code> and then calls it. In a child function, we can also add our own methods so an instance/object of the child class would have all the methods and attributes of both the parent and the child class.