Chapter 4

Object Oriented Programming

Programming methodologies can be broadly divided into two types 1) Functional Programming 2) Object Oriented Programming. In this chapter we will present an overview of object oriented programming as it will come in handy when using artificial intelligence and machine learning tools in `Python` .

4.1 What is Objected Oriented Programming?

Object oriented programming is a paradigm of programming where data and tools to manipulate it, are grouped together and this grouped structures is called an object (hence the name object oriented programming). Let's motivate the object oriented programming (OOP) with an example, say we want to design a system for an organization to manage it's employees. We have to store employee information and functions to handle employee pay raises, transfers etc.

The procedural approach to do this will be to write some functions which handle the pay raises and transfers, etc, and separately write data structures to hold employee data, when the functions are called be on the data structure we will call them by passing the data structure to them and transferring the output back to the data structure.

The OOP approach to this problem is much simpler: here, we define an object called the employee. This object will have all the data associated with the employee and the functions that control the data manipulation for the employee. If we need to call a function for an employee (object) we don't need to specifically pass any data around as the object has access to it's data.

4.2 Classes

On the surface the above idea of grouping data and methods in an object seem quite good but how do we actually do it in programming. Say for the above example if we have a couple thousand employees, how can we write the data and methods for each one of them in an object? If we have to do it for each employee individually it will be excessively prohibitive and virtually impossible for large number of objects.

In OOP similar objects are grouped together in classes. Classes can be thought of as templates to which object's belong. Objects can be generated by using the blueprint of a class. Below is the code for generating a new class in `Python` .

```
class Employee:
    pass
```

The above code generates a class called Employee. Right now the class is empty, we have used the keyword pass to prevent the error which is generated when you don't add any code under the class definition

Good Practice: Class names use CamelCase notation starting with a capital letter in Python to differentiate them from objects.

Note: Class definition in Python 2 is slightly different.

```
class Employee(object):
    pass
```

Notice that we have again used Python keyword pass here, which is used as a a place holder in Python , pass is where the code will go eventually.

Let's add some methods and attributes to the class Employee that we have defined above

```
class Employee:
    "a simple class"
    def __init__(self, fn, ln):
        self.first_name=fn
        self.last_name=ln

staff1234=Employee('Naeem', 'Khan')
print(staff1234.first_name, staff1234.last_name)
```

Those of you who are familiar with object oriented programming in other languages like C++ will see the similarities between constructors in C++ and the __init__ method in Python . I like to give the simplest definition that every time you create an object of a class the __init__ of that class is called and basically acts as an interface between the class and your code. Another important concept to understand is that of self (we can use a different name for this as well) variable of classes in Python . The variable self represents a particular instance object of the class inside the class definition. For example, let's say we create an instance object1 of a a particular class, then when the __init__ function is executed self would imply object1 and we can access the attributes and methods of object1 inside the class using the keyword self.

In the above code the 5th line of the code generates an instance of class Employee called staff1234. Naeem and Khan are passed as fn and ln to the __init__ function of the class employee and staff1234 is passed to self. The __init__ function assigns the value Naeem to the first_name attribute of staff1234 and assigns the value khan to the last _name attribute of staff1234.

Note: The . operator is used to access attributes and methods in Python . Try printing __doc__attribute for the above class and see what do you get.

Good Practice: Usually the instance object variable in methods inside the class is labelled self.

Note: If a class does not have an __doc__method instantiation the class creates an empty object.

Note: Instance object is passed as the first argument to the methods in a class.

```python
def full_name(first_name, last_name):
    return first_name+ ' '+last_name

class Employee:

    def __init__(self, fn, ln):
        self.first_name=fn
        self.last_name=ln

    def full_name(self):
        return self.first_name+ ' '+self.last_name

staff1234=Employee ('Naeem', 'Khan')
print(staff1234.first_name, staff1234.last_name)
print(staff1234.full_name())
print(full_name(staff1234.first_name,staff1234.last_name))
```

The real advantage of the object oriented programming is the ability to the different methods to connect to objects in a seamless manner. For example for the above example if we wanted a function to join the first and last name of each employee we would have to provide the first name and last name of each object to that function and that function would have then returned the result to us. But using the object oriented programming we can define a method inside the class which will have access to the attributes of the object and we don't need to explicitly provide the data to the function, rather we just call the method for the instance of the class and it returns the desired results.

```python
def full_name(first_name, last_name):
    return first_name+ ' '+last_name

class Employee:

    company='AI Course Work Module'

    def __init__(self, fn, ln):
        self.first_name=fn
        self.last_name=ln

    def full_name(self):
        return self.first_name+ ' '+self.last_name

staff1234=Employee ('Naeem', 'Khan')
print(staff1234.first_name, staff1234.last_name)
print(staff1234.full_name())
print(full_name(staff1234.first_name,staff1234.last_name))

print(staff1234.company)
```

In the above code we have introuduced a new attribute of the class Employee called company. Notice unlike the attriubutes like first_name and last_name the attribute company is not an attribute of each instance uniquely rather it is common among all attribute, such attributes are called class attributes.

Note: Check the __class__attribute of the names in your code and see what it returns.

Inheritance

Inheritance is the concept in object oriented programming where we derive a class from a different class. Let's work with the above example for Employee class, suppose we have different departments in an organization and we want to write separate class for each of these departments. Since these departments still consist of the employees of the organization, Do we need to re-define the methods we have defined in Employee class again for each department? The answer is no. The class for each department can inherit from the Employee class. Let's try to put this somewhat convoluted explanation in code and it would make things clearer.

```
#inheritance
class Employee:

    company='AI Course Work Module'

    def __init__(self, fn, ln):
        self.first_name=fn
        self.last_name=ln

    def full_name(self):
        return self.first_name+ ' '+self.last_name

class Dept1(Employee):
    department='Department 1'


staff1234=Dept1('Naeem','Khan')
print(staff1234.full_name())
```

In the above example the class Dept1 inherits the methods __init __and full_name from Employee. Notice that we have not defined these methods in the body of Dept1 but when we create an instance of class Dept1 it is using these methods from class Employee.
Now, let's run this code

```
#inheritance
class Employee:

    company='AI Course Work Module'

    def __init__(self, fn, ln):
        self.first_name=fn
        self.last_name=ln


    def full_name(self):
        return self.first_name+ ' '+self.last_name

class Dept1(Employee):
    dpeartment='Department 1'

    def __init__(self, fn, ln):
        self.first_name_dept=fn
        self.last_name_dept=ln


staf1234=Dept1('Naeem','Khan')
print(staf1234.full_name())
```

In the above code you will get an error when using the method full_name. The reason is that now our child class Dept1 has it's own __init __method which takes preference over the __init __of the parent class and hence attributes first _name and last _name are not defined when we call the full _name method, because the __init __of the parent class (Employee)is not executed.

But this seems to be a problem if this is the case then we will not be able to use the methods of the parent class. The answer to this problem is simple and we will explain it with an example below.

```
#inheritence
class Employee:

    company='AI Course Work Module'

    def __init__(self, fn, ln):
        self.first_name=fn
        self.last_name=ln


    def full_name(self):
        return self.first_name+ ' '+self.last_name

class Dept1(Employee):
    dpeartment='Department 1'

    def __init__(self, fn, ln):
        super().__init__(fn,ln)
        self.first_name_dept=fn
        self.last_name_dept=ln


staf1234=Dept1('Naeem','Khan')
print(staf1234.full_name())
```

The above code will work fine, despite the fact that we have method __init __in both parent and the child class(and we need to run both of these). Notice that the only change is the super().__init __command. The keyword super refers to the super class of the class and in the above code it will call the __init __method of the Employee class and fix the first _name and last _name so then we are able to use the methods of the Employee class. The keyword super can be used in OOP inheritance and extend a parent class.

Good Practice: It is better to keep the class attributes and instance attributes separate.

Note: The lookup for an attribute in an object follows the following rule:

- Arrtibute is looked for in the class first.
- if not found in above it is looked for in the parent-list from left to right in the class definition.
- In the parent class, it is looked recursively in the parents of parent class.
- if not found in above it is looked for in next parent and so on.

TO DO: isinstance issubclass

Exercises

Exercise 4.1: Write a child class with 2 parents and validate the above scheme of attribute search in instance of a class.

Exercise 4.2: Write a program (without using OOP) which takes in $(x, y) \in \mathbb{R}^{+}2$ a tuple of positive number representing the size of rectangular grid. It then takes an arbitrary number of parameters $(x, y), r$, where the (x, y) represents the center of a circle and r represents the radius. For each entry the program returns a "True" if the circle can be placed on the grid without it intersecting with any other circles and then places the circle on the grid. Else "False" if the circle can not be placed on the grid because it intersects with other previously placed circle.

Exercise 4.3: Re-implement the above with OOP.

Exercise 4.4: Re-implement the above in "n" dimensional space. where "n" is the input of the class constructor.

Exercise 4.5: Re-implement the above in 2 dimensional where we can now draw, circle, squares, triangles, and rectangles. The format for the shapes is:
circle: x, y, r where x, y is center of circle and r is it's radius
square: x, y, d where x, y is center of square and d is it's side length
square: x, y, l, w where x, y is center of square and l, w are it's length and width
triangle: $x_1, y_1, x_2, y_2, x_3, y_3$ where x_i, y_i represents the i^{th} corner of the triangle