Py - Classes & Objects

Classes and Objects in Python

- In Python, Every piece of data you see or come into contact with is represented by an object
- An object is simply a collection of data (variables) and methods (functions) that act on those data.
- A class is a collection of objects defining the common attributes and behavior.

```
class className() //className is the class name
```

A class is a blueprint

We can think of a class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.

As many houses can be made from a house's blueprint, we can create many objects from a class. An object is also called an instance of a class and the process of creating this object is called instantiation.

Defining classes in python

- Like function definitions begin with the def keyword in Python, class definitions begin with a class keyword.
- A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions.

```
class MyNewClass:
    '''This is a docstring. I have created a new class'''
    pass
```

```
class Student():
 def      init (self, name, age, marks): //creating attributes function
   """ init method allows the class to initialize the attributes of a class"""
    self.name = name
    self.age = age
    self.marks = marks //Self is an instance of a class
```

Creating objects

- To create instances of a class, call the class using class name and pass in whatever arguments its `__init__` method accepts
- When creating instance of class, Python adds the `self` argument to the list for you. You don't include it when you call the methods

```
stud_1 = Student("Joe M", 22, 86)
stud_2 = Student("Janet K", 18, 96)

print(stud_1.__dict__) //Prints a dictionary
```

Special Class Attributes in Python

_dict	Dict variable of class name space
_doc	Document reference string of class
_name	Class Name
_module	Module Name consisting of class
_base	The tuple including all the superclasses

Class Scope

- Variable scope is the context in which it's visible to the program
- `Global variables` are available everywhere.
- `Member variables` are only available to members of a certain class
- `instance variables` are only available to a particular instance of a class

Garbage Collection

- Python will delete no longer needed objects - built-in types or class instances - automatically to free memory space
- You don’t notice when garbage collector destroys an orphaned instance and reclaims its space
- But, a class can implement the special `__del__()` method called a destructor
- You can use this method to clean up any non-memory resources used by an instance

Class Inheritance

- Inheritance enables us to define a class that takes all the functionality from a parent class and allows us to add more
- The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class.

```
class BaseClass:
  Body of base class
class DerivedClass(BaseClass):
  Body of derived class
```

```
class Polygon:
    def __init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]

    def inputSides(self):
        self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]

    def dispSides(self):
        for i in range(self.n):
            print("Side",i+1,"is",self.sides[i])
```

```
class Triangle(Polygon):
    def __init__(self):
        Polygon.__init__(self,3)

    def findArea(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)
```

Method Overriding

- Used when you want special or different functionality in your subclass

```
class Parent:
 def myMethod(self):
   print("Calling parent method")
class Child(Parent):
 def myMethod(self):
   print("Calling child method")

c = Child() # instance of Child
c.myMethod() # child calls overridden method
```

Operator Overloading

- Giving extended meaning beyond their predefined operational meaning e.g. we can overload the ‘+’ operator

```
class
 def  init  (self, a, b):
   self.a =
   self.b =
 def  str   (self):
   return  ("{:d}, {:d}".format(self.a,
 def  add   (self, other):
   return Vector(self.a + other.a, self.b +
   other.b)
v1 = Vector(2,
v2 = Vector(5,
print(v1 +
v2)
```

Method Overloading

- Not acceptable in Python
- Read about method overloading in java here to understand more.

More!

- Encapsulation
- Abstraction

See you at the next session!