Object-Oriented Python

Classes and Instance, Special Methods, Decorators

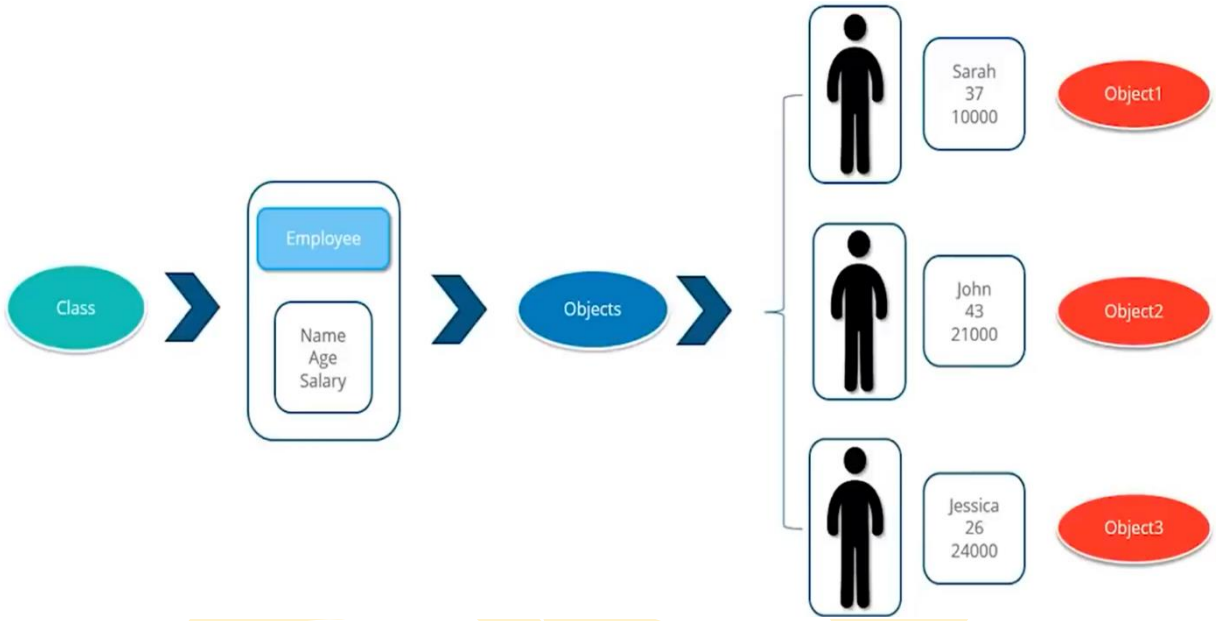
Gaurav Kr. suman

4/11/20

DSPMU_MCA/MIT

Class

A class is a user-defined blueprint from which specific objects are created. 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 attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state



Class Definition Syntax:

```
class ClassName:
    # Statement-1
    .
    .
    .
    # Statement-N
```

Example 1:

```
class student:
    roll=10
    name="Vinay"
    Marks=98
    def disp(self):
        print("Working")
obj=student()
print("Roll = ",obj.roll)
print("Name = ",obj.name)
print("Marks = ",obj.Marks)
obj.disp()
```

Instance Variables

Instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class. Instance variables are variables whose value is assigned inside a constructor or method with self whereas class variables are variables whose value is assigned in the class.

Special Method

The self

- Class methods must have an extra first parameter in method definition. We do not give a value for this parameter when we call the method, Python provides it.
- If we have a method which takes no arguments, then we still have to have one argument.
- This is similar to this pointer in C++ and this reference in Java.

When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.

__init__ method

The __init__ method is similar to constructors in C++ and Java. Constructors are used to initialize the object's state. Like methods, a constructor also contains collection of statements(i.e. instructions) that are executed at time of Object creation. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

Example (_init_):

```
 class boy:

# init method or constructor
    def __init__(self, name):
        self.name = name

    # Sample Method
    def say_hi(self):
        print('Hello, my name is', self.name)

obj = boy ('Gaurav')
obj.say_hi()
```

Output:

```
Hello, my name is Gaurav
```

Decorators in Python

Python has an interesting feature called decorators to add functionality to an existing code. This is also called metaprogramming as a part of the program tries to modify another part of the program at compile time. Decorators are very powerful and useful tool in Python since it allows programmers to modify the behavior of function or class. Decorators allow us to wrap another function in order to extend the behavior of wrapped function, without permanently modifying it.

Example

```
 def first(msg):
    print(msg)

first("Hello")

second = first
second("Hello")
```

Example: Decorating Functions with Parameters

```
def smart_divide(func):

   def inner(a,b):

      print("I am going to divide",a,"and",b)

      if b == 0:

         print("Whoops! cannot divide")

         return


       func(a,b)

   return inner


@smart_divide

def divide(a,b):

    return a/b
```