# Python  Programming

## Day 9: Classes and Objects



# Classes and Objects

| | |
|---|---|
| Instance | Methods |
| Property | OOP |



# Color and symbol meaning

**Hint**

**Preferred**

**Student's activity**

**Practice code**

| | |
|---|---|
| | **Keyword** |
| | **In-built functions** |
| | **Strings** |
| | **Output** |



# Object-Oriented Programming

Python is an object-oriented programming (OOP) language, which means that it provides features that support OOP.

- ✓ Encapsulation
- ✓ Inheritance
- ✓ Abstraction
- ✓ Polymorphism



# Object-Oriented Programming

In procedural programming the focus is on writing functions or procedures which operate on data.

This has been our approach since inception.

In Object-oriented programming the focus is on the creation of objects which contain both data and functionality together.



# OOP Terminology

**Instantiate**

To create an instance of a class.

**Instance**

An object that belongs to a class.

**Object**

A compound data type that is often used to model a thing or concept in the real world.

**Attribute**

One of the named data items that makes up an instance. This include both class variable and method.



# Classes and Objects

A class is an abstract definition of an object

- It defines the structure and behaviour of each object in the class.
- It serves as a template for creating objects
- A particular object of a class is an instance.



# Creating Classes

The ***class*** statement creates a new class definition. The name of the class immediately follows the keyword class followed by a colon (:) as shown below.

```
class ClassName:
    'Optional class documentation string'
    class_suite (block of codes)
```



# Creating Classes

The class has a documentation string, which can be accessed via ClassName.__doc__.

The class_suite consists of all the component statements defining class members, data attributes and functions.

```
class ClassName:
    'Optional class documentation string'
    class_suite
```



# Creating Classes – Class Constructor

**_init_()** is a special method, which is called **class constructor** or initialization method that Python calls when you create a new instance of this class. It is usually the first method within a class.

Difference between a function and a method is that a method is a function within a class



# Creating Classes – Self Parameter

You declare other class methods like normal functions with the exception that the first argument to each method is ***self***.

Python adds the self argument to the list for you; you do not need to include it when you call the methods.



# Creating Classes

*Sample Code*

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
      print("Total Employee %d" % Employee.empCount)

   def displayEmployee(self):
      print("Name : ", self.name, ", Salary: ", self.salary)
```

class variable whose value is shared among all instances of this class



# Creating Instance Objects

To create instances of a class, you call the class using class name and pass in whatever arguments its __init__ method accepts

```
"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
```



# Accessing Attributes

```
emp1.displayEmployee()
emp2.displayEmployee()
print ("Total Employee %d" %)
Employee.empCount
```

When the above code is executed, it produces the following result

```
Name :  Zara ,Salary:  2000
Name :  Manni ,Salary:  5000
Total Employee 2
```



# Modifying Object Instance

You can add, remove, or modify attributes of classes and objects at any time

```
emp1.age = 7  # Add an 'age' attribute.
emp1.age = 8  # Modify 'age' attribute.
del emp1.age  # Delete 'age' attribute.
```



# Modifying Object Instance

Instead of using the normal statements to access attributes, you can use the following functions

The **hasattr** (obj,name):
It checks if an attribute exists or not.

```
hasattr(emp1, 'age')    # Returns true if 'age' attribute exists
```



# Modifying Object Instance

The **getattr** (obj, name[, default]):
This is used to access the attribute
of object.

```
getattr(emp1, 'age')    # Returns value of 'age' attribute
```



# Modifying Object Instance

The **setattr** (obj, name,value):
This is used to set an attribute.
If attribute does not exist, then
it would be created.

```
setattr(emp1, 'age', 8) # Set attribute 'age' at 8
```



# Modifying Object Instance

The **delattr**(obj, name):
This is used to delete an attribute.

```
delattr(empl, 'age')    # Delete attribute 'age'
```



# Class Activity 1

Write a Python class to convert an integer to a roman numeral.



# Built-In Class Attributes

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute

- **__dict__**: Dictionary contains the class' namespace.
- **__doc__**: Class documentation string or none, if undefined.



# Built-In Class Attributes

- **__name__**: Class name.
- **__module__**: Module name in which the class is defined. This attribute is "__main__" in interactive mode.
- **__bases__**: A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list



# Built-In Class Attributes

Let us try to access all these attributes for the Employee class

```
print "Employee.__doc__:" Employee.__doc__
print "Employee.__name__:" Employee.__name__
print "Employee.__module__:" Employee.__module__
print "Employee.__bases__:" Employee.__bases__
print "Employee.__dict__:" Employee.__dict__
```



# Built-In Class Attributes

Expected Output

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: ()
Employee.__dict__: {'__module__': '__main__', 'displayCount':
<function displayCount at 0xb7c84994>, 'empCount': 2,
'displayEmployee': <function displayEmployee at
0xb7c8441c>,
'__doc__': 'Common base class for all employees',
'__init__': <function __init__ at 0xb7c846bc>}
```



# Class Activity 2

Write a Python class to convert a roman numeral to an integer.



# Destroying Objects (Garbage Collection)

Python deletes unneeded objects (built-in types or class instances) automatically to free the memory space. This process is referred to as **Garbage Collection**.

Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero.



# Destroying Objects (Garbage Collection)

An object's reference count changes as the number of aliases that point to it changes.

```
a = 40      # Create object <40>
b = a       # Increase ref. count  of <40>
c = [b]     # Increase ref. count  of <40>

del a       # Decrease ref. count  of <40>
b = 100     # Decrease ref. count  of <40>
c[0] = -1   # Decrease ref. count  of <40>
```



# Destroying Objects (Garbage Collection)

You normally will not notice when the garbage collector destroys an orphaned instance and reclaims its space.

But a class can implement the special method ***__del__( )***, called a destructor, that is invoked when the instance is about to be destroyed. This method might be used to clean up any non memory resources used by an instance.



# Destroying Objects (Garbage Collection)

Below is the sample implementation of the **_del_( )** special method in the point class. It prints the class name of an instance that is about to be destroyed.

Note: Ideally, you should define your classes in separate file, then you should import them in your main program file using import statement.



# Destroying Objects (Garbage Collection)

```
class Point:
  def __init( self, x=0, y=0):
    self.x = x
    self.y = y
  def __del__(self):
    class_name = self.__class__.__name__
    print (class_name, "destroyed")

pt1 = Point()
pt2 = pt1
pt3 = pt1
# prints the ids of the objects
print (id(pt1), id(pt2), id(pt3))
del pt1
del pt2
del pt3
```

## Output

```
3083401324
3083401324
3083401324
Point destroyed
```



# Class - Properties

A property is a special kind of attribute that computes its value when accessed.

The **@property decorator** makes it possible for the method that follows to be accessed as a simple attribute, without the extra () that you would normally have to add to call the method.



# Class - Properties

In this example, Circle instances have an instance variable c.radius that is stored. c.area and c.perimeter are simply computed from that value.

```
import math
class Circle(object):
    def __init__(self, radius):
        self.radius = radius
        #Some additional properties of circles
    @property
    def area(self):
        return math.pi*self.radius**2
    @property
    def perimeter(self):
        return 2*math.pi*self.radius
```

# Class - Properties

A circle object c is created. Execute the code to get the corresponding output.

```
c = Circle(4.0)
print('radius = ', c.radius)
print('area = ', c.area)
print('perimeter = ', c.perimeter)
```



# Class - Properties

```
c = Circle(4.0)
print('radius = ', c.radius)
print('area = ', c.area)
print('perimeter = ', c.perimeter)
c.area = 3
```

```
radius =  4.0
Traceback (most recent call last):
  File
"C:/Users/OluFemi/PycharmProjects/test/circle
.py", line 17, in <module>
    c.area = 3
AttributeError: can't set attribute
area =  50.26548245743669
perimeter =  25.132741228718345
```

The new line c.area=3 is added. The output shows c.area is not a variable but a property compared to c.radius.



# Class - Properties

Using properties in this way is related to something known as the **Uniform Access Principle**.

Essentially, if you're defining a class, it is always a good idea to make the programming interface to it as uniform as possible.



# Class - Properties

Without properties, certain attributes of an object would be accessed as a simple attribute such as c.radius whereas other attributes would be accessed as methods such as c.area().

**Keeping track of when to add the extra ( ) adds unnecessary confusion. *A property can fix this*.**



# Class Activity 3

Write a Python class named Triangle constructed by base and height and a method which will compute the area of a triangle.



# Next Lecture ...

## Python Programming

Day 10: Classes and Objects (2)