CLASSES AND OBJECTS:

OVERVIEW OF OOP (OBJECT ORIENTED PROGRAMMING)

- Python is an **object-oriented programming language**, which means that it provides features that support **object-oriented programming** (**OOP**).
- Object-oriented programming has its roots in the 1960s, but it wasn't until the mid 1990s that it became the main **programming paradigm** used in the creation of new software.
- It was developed as a way to handle the rapidly increasing size and complexity of software systems, and to make it easier to modify these large and complex systems over time. Up to this point we have been writing programs using a **procedural programming** paradigm.
- In procedural programming the focus is on writing functions or *procedures* which operate on data. In object-oriented programming the focus is on the creation of **objects** which contain both data and functionality together.

OVERVIEW OF OOP (OBJECT ORIENTED PROGRAMMING):

- a.**Class**: A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- b. **Class variable**: A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- c. **Data member**: A class variable or instance variable that holds data associated with a class and its objects.
- d. **Function overloading**: The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
- e. **Instance variable**: A variable that is defined inside a method and belongs only to the current instance of a class.

- f**. Inheritance**: The transfer of the characteristics of a class to other classes that are derived from it.
- g. **Instance**: An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- h. **Instantiation**: The creation of an instance of a class.
- i. **Method** : A special kind of function that is defined in a class definition.
- j. **Object**: A unique instance of a data structure that is defined by its class. An object comprises both data members (class variables and instance variables) and methods.
- k. **Operator overloading**: The assignment of more than one function to a particular operator.

CLASS DEFINITION

- *The class statement creates a new class definition. The name of the class immediately follows the keyword class followed by a colon as follows-*

```
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)
```

- The variable empCount is a class variable whose value is shared among all the instances of a in this class. This can be accessed as Employee.empCount from inside the class or outside the class.
- The first method 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.
- 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 OBJECTS

- To create instances of a class, you call the class using class name and pass inwhatever 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**
- You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows-
- emp1.displayEmployee() emp2.displayEmployee()
- print ("Total Employee %d" % Employee.empCount)

INSTANCES AS ARGUMENTS

- Instance variables are always prefixed with the reserved word self. They are typically introduced and initialized in a constructor method named init .
- In the following example, the variables self.name and self.grades are instance variables, whereas the variable NUM_GRADES is a class variable:

```
class Student:

    NUM_GRADES = 5

    def init_(self, name):
        self.name = name
        self.grades = []
        for i in range(Student.NUM_GRADES):
            self.grades.append(0)
```

- *Here “self” is a instance and “name” is a argument*
- The PVM automatically calls the constructor method when the programmer requests a new instance of the class, as follows:
- s = Student('Mary')
- *The constructor method always expects at least one argument*, self. When the method is called, the object being instantiated is passed here and thus is bound to self throughout the code. Other arguments may be given to supply initial values for the object’s data**.**

INSTANCES AS RETURN VALUES

```
class Numbers:
    MULTIPLIER=None
    def_init_(self,x,y):
        self.x=x
        self.y=y
    def add(self):
        return (self.x+self.y)
print("Enter two numbers for addition")
x=int(input())
y=int(input())
n=Numbers(x,y)
print("addition is : ",n.add())
```

Here return (self.x+self.y) are the instances as return values
Where self is a instance and .x and .y are variable associated with the instance

BUILT-IN CLASS ATTRIBUTES

- Every Python class keeps the following built-in attributes and they can be accessed
- using dot operator like any other attribute –
- • **dict :** Dictionary containing the class's namespace.
- • **doc :** Class documentation string or none, if undefined.
- • **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.
- For the above class let us try to access all these attributes-

```
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)

emp1 = Employee("Zara", 2000)
emp2 = Employee("Manni", 5000)
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      )
```

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

```
Employee. doc : Common base class for all employees
Employee.  name  : Employee
Employee. module : __main
Employee.  bases  : (<class 'object'>,)
Employee. dict : {'__module__': '__main__', '__doc__': 'Common base class for all
employees', 'empCount': 2, '__init__': <function Employee.__init at 0x00F14270>,
'displayCount': <function Employee.displayCount at 0x0304C0C0>,
'displayEmployee': <function Employee.displayEmployee at 0x032DFE88>,
'__ dict__ ': <attribute '__ dict__ ' of 'Employee' objects>, '__ weakref__ ': <attribute
'__ weakref__ ' of 'Employee' objects>}
```

INHERITANCE

- Instead of starting from a scratch, you can create a class by deriving it from a pre- existing class by listing the parent class in parentheses after the new class name.
- The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined In the child class. A child class can also override data members and methods from the parent.

```
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)

emp1 = Employee("Zara", 2000)
emp2 = Employee("Manni", 5000)
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      )
```

Syntax

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name –

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
        'Optional class documentation string'
        class_suite
```

Example

```
class Parent:    # define parent class
   parentAttr = 100
   def   init (self):
      print ("Calling parent constructor")
   def parentMethod(self):
      print ('Calling parent method')

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print ("Parent attribute :", Parent.parentAttr)

class Child(Parent): # define child class
   def   __init__(self):
      print ("Calling child constructor")

   def childMethod(self):
      print ('Calling child method')

c = Child()      # instance of child
c.childMethod()          # child calls its method
c.parentMethod()# calls parent's method
c.setAttr(200) # again call parent's method
c.getAttr()      # again call parent's method
```

- In a similar way, you can drive a class from multiple parent classes as follows-
- You can use issubclass() or isinstance() functions to check a relationship of two classes and instances.
- The issubclass(sub, sup) boolean function returns True, if the given subclass sub is indeed a subclass of the superclass sup.
- The isinstance(obj, Class) boolean function returns True, if obj is an instance of class Class or is an instance of a subclass of Class.

```
class A:        # define your class A
.....
class B:        # define your class B
.....
class C(A, B): # subclass of A and B
.....
```

METHOD OVERRIDING

- You can always override your parent class methods. One reason for overriding parent's methods is that you may want special or different functionality in your subclass.

Example

```
class Parent:    # define parent class
   def myMethod(self):
      print ('Calling parent method')


class Child(Parent): # define child class
   def myMethod(self):
      print ('Calling child method')

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

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

```
Calling child method
```

DATA ENCAPSULATION

- Simplifying the script by identifying the repeated code and placing it in a function. This is called 'encapsulation'.
- Encapsulation is the process of wrapping a piece of code in a function, allowing you to take advantage of all the things functions are good for.
- Generalization means taking something specific, such as printing the multiples of 2, and making it more general, such as printing the multiples of any integer. This function encapsulates the previous loop and generalizes it to print multiples of n:

```
def print_multiples(n): i = 1
    while i <= 6:
    print n*i, "\t",
    i += 1
print
```

- With the argument 4, the output is:
- By now you can probably guess how to print a multiplication table—by calling print multiples repeatedly with different arguments. In fact, we can use another loop:
- By now you can probably guess how to print a multiplication table—by calling print multiples repeatedly with different arguments. In fact, we can use another loop:

```
3       6       9       12      15      18

4       8       12      16      20      24
```

DATA HIDING

- An object's attributes may or may not be visible outside the class definition. You need to name attributes with a double underscore prefix, and those attributes then will not be directly visible to outsiders.

Example

```
class JustCounter:
    __secretCount = 0
    def count(self):
        self._secretCount += 1
        print (self.__secretCount)
counter = JustCounter()
counter.count()
counter.count()
print (counter.    secretCount)
```

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

```
1
2
Traceback (most recent call last):
  File "C:/Users/USER/AppData/Local/Programs/Python/Python36-32/try2.py", line 9, in
<module>
    print (counter.__secretCount)
AttributeError: 'JustCounter' object has no attribute '__secretCount'
```

Python protects those members by internally changing the name to include the class name. You can access such attributes as *object._className_attrName*. If you would replace your last line as following, then it works for you-

```
.....................
print (counter._JustCounter__secretCount)
```

```
1
2
2
```

MODULES: IMPORTING MODULE

- A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.
- Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.
- **Example**
- The Python code for a module named a name normally resides in a file namedaname.py. Here is an example of a simple module, support.py

```
def print_func( par ):
print "Hello : ", par
return
```

CREATING AND EXPLORING MODULES

- **The import Statement**
- You can use any Python source file as a module by executing an import statement in some other Python source file. The import has the following syntax-
- When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module hello.py, you need to put the following command at the top of the script
- Import module1[, module2[,... moduleN]
- When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module hello.py, you need to put the following command at the top of the script-
- Import module support import support
- # Now you can call defined function that module as follows support.print_func("Zara")

MATH MODULE

- This module is always available. It provides access to the mathematical functions defined by the C standard.
- These functions cannot be used with complex numbers; use the functions of the same name from the **cmath** module if you require support for complex numbers. The distinction between functions which support complex numbers and those which don't is made since most users do not want to learn quite as much mathematics as required to understand complex numbers.
- Receiving an exception instead of a complex result allows earlier detection of the unexpected complex number used as a parameter, so that the programmer can determine how and why it was generated in the first place.
- The following functions are provided by this module. Except when explicitly noted otherwise, all return values are floats.
- ***(a) Number-theoretic and representation functions***
- math.**ceil**(*x*)
- Return the ceiling of *x* as a float, the smallest integer value greater than or equal to *x*.
- math.**copysign**(*x*, *y*)
- Return *x* with the sign of *y*. On a platform that supports signed zeros, copysign(1.0, - 0.0) returns *-1.0*.
- *New in version 2.6.*
- math.**fabs**(*x*)
- Return the absolute value of *x*.
- math.**factorial**(*x*)
- Return *x* factorial. Raises **ValueError** if *x* is not integral or is negative.
- *New in version 2.6.*
- math.**floor**(*x*) Return the floor of *x* as a float, the largest integer value less than or equal to *x*.

- math.**fmod**(*x*, *y*)
- Return fmod(x, y), as defined by the platform C library. Note that the Python expression x % y may not return the same result. The intent of the C standard is
- that fmod(x, y) be exactly (mathematically; to infinite precision) equal to x - n*y for some integer *n* such that the result has the same sign as *x* and magnitude less than abs(y).
- Python’s x % y returns a result with the sign of *y* instead, and may not be exactly computable for float arguments. For example, fmod(-1e-100, 1e100) is -1e-100, but the result of Python’s - 1e-100 % 1e100 is 1e100-1e-100, which cannot be represented exactly as a float, and rounds to the surprising 1e100. For this reason, function **fmod()** is generally preferred when working with floats, while Python’s x % y is preferred when working with integers.
- math.**frexp**(*x*)

RANDOM MODULE

- This module implements pseudo-random number generators for various distributions. For integers, there is uniform selection from a range. For sequences, there is uniform selection of a random element, a function to generate a random permutation of a list in-

TIME MODULE

SN	Function with Description
1	**time.altzone** The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Use this if the daylight is nonzero.
2	**time.asctime([tupletime])** Accepts a time-tuple and returns a readable 24-character string such as 'Tue Dec 11 18:07:14 2008'.
3	**time.clock( )** Returns the current CPU time as a floating-point number of seconds. To measure computational costs of different approaches, the value of time.clock is more useful than that of time.time().
4	**time.ctime([secs])** Like asctime(localtime(secs)) and without arguments is like asctime( )
5	**time.gmtime([secs])** Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the UTC time. Note : t.tm_isdst is always 0

6	**time.localtime([secs])** Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the local time (t.tm_isdst is 0 or 1, depending on whether DST applies to instant secs by local rules).
7	**time.mktime(tupletime)** Accepts an instant expressed as a time-tuple in local time and returns a floating-point value with the instant expressed in seconds since the epoch.
8	**time.sleep(secs)** Suspends the calling thread for secs seconds.
9	**time.strftime(fmt[,tupletime])** Accepts an instant expressed as a time-tuple in local time and returns a string representing the instant as specified by string fmt.
10	**time.strptime(str,fmt='%a %b %d %H:%M:%S %Y')** Parses str according to format string fmt and returns the instant in time-tuple format.
11	**time.time( )** Returns the current time instant, a floating-point number of seconds since the epoch.
12	**time.tzset()** Resets the time conversion rules used by the library routines. The environment variable TZ specifies how this is done.