Python has been an object-oriented language since it existed. Because of this, creating and using classes and objects are downright easy. This chapter helps you become an expert in using Python's object-oriented programming support.

If you do not have any previous experience with object-oriented (OO) programming, you may want to consult an introductory course on it or at least a tutorial of some sort so that you have a grasp of the basic concepts.

However, here is small introduction of Object-Oriented Programming (OOP) to bring you at speed:

Overview of OOP Terminology

- **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.
- **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.
- **Data member:** A class variable or instance variable that holds data associated with a class and its objects.
- **Function overloading:** The assignment of more than one behaviour to a particular function. The operation performed varies by the types of objects or arguments involved.
- **Instance variable:** A variable that is defined inside a method and belongs only to the current instance of a class.
- **Inheritance:** The transfer of the characteristics of a class to other classes that are derived from it.
- **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.
- **Instantiation:** The creation of an instance of a class.
- **Method:** A special kind of function that is defined in a class definition.
- **Object:** A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.
- **Operator overloading:** The assignment of more than one function to a particular operator.

Creation of Class:

A class is created with the keyword *class* and then writing the classname. The simplest form of class definition looks like this:

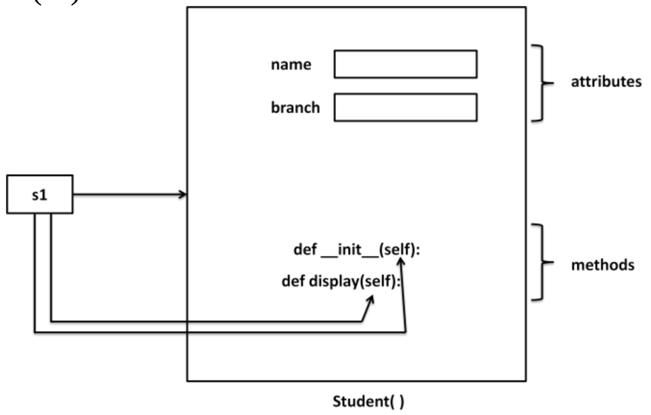
```
class ClassName:
        <statement-1>
        .
        .
        <statement-N>
```

Class definitions, like function definitions (def statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an if statement, or inside a function.)

Example:

```
class Student:
   def __init__(self):
      self.name="hari"
      self.branch="CSE"
   def display(self):
      print self.name
      print self.branch
```

- For example, If we 'Student' class, we can write code in the class that specifies the attributes and actions performed by any student.
- Observer that the keyword *class* is used to declare a class. After this, we should write the class name. So, 'Student' is our class name. Generally, a class name should start with a capital letter, hence 'S' is a capital in 'Student'.
- In the class, we have written the variables and methods. Since in python, we cannot declare variables, we have written the variables inside a special method, i.e. __init__(). This method is used to initialize the variables. Hence the name 'init'.
- The method name has two underscores before and after. This indicates that this method is internally defined and we cannot call this method explicitly.
- Observe the parameter 'self' written after the method name in the parentheses. 'self' is a variable that refers to current class instance.
- When we create an instance for the Student class, a separate memory block is allocated on the heap and that memory location is default stored in 'self'.
- The instance contains the variables 'name' and 'branch' which are called *instance variables*. To refer to instance variables, we can use the dot operator notation along with self as 'self.name' and 'self.branch'.
- The method display ( ) also takes the 'self' variable as parameter. This method displays the values of variables by referring them using 'self'.
- The methods that act on instances (or objects) of a class are called instance methods. Instance methods use 'self' as the first parameter that refers to the location of the instance in the memory.
- Writing a class like this is not sufficient. It should be used. To use a class, we should create an instance to the class. Instance creation represents allotting memory necessary to store the actual data of the variables, i.e., 'hari', 'CSE'.
- To create an instance, the following syntax is used:
  instancename = Classname( )
- So, to create an instance to the Student class, we can write as:
  s1 = Student ( )
- Here 's1' represents the instance name. When we create an instance like this, the following steps will take place internally:
  1. First of all, a block of memory is allocated on heap. How much memory is to be allocated is decided from the attributes and methods available in the Student class.
  2. After allocating the memory block, the special method by the name '__init__(self)' is called internally. This method stores the initial data into the variables. Since this method is useful to construct the instance, it is called 'constructor'.
  3. Finally, the allocated memory location address of the instance is returned into 's1' variable. To see this memory location in decimal number format, we can use id( ) function as id(s1).



Self variable:

'self' is a default variable that contains the memory address of the instance of the current class. When an instance to the class is created, the instance name cotains the memory locatin of the instance. This memory location is internally passed to 'self'.

For example, we create an instance to student class as:

s1 = Student( )

Here, 's1' contains the memory address of the instance. This memory address is internally and by default passed to 'self' variable. Since 'self' knows the memory address of the instance, it can refer to all the members of the instance.

We use 'self' in two eays:

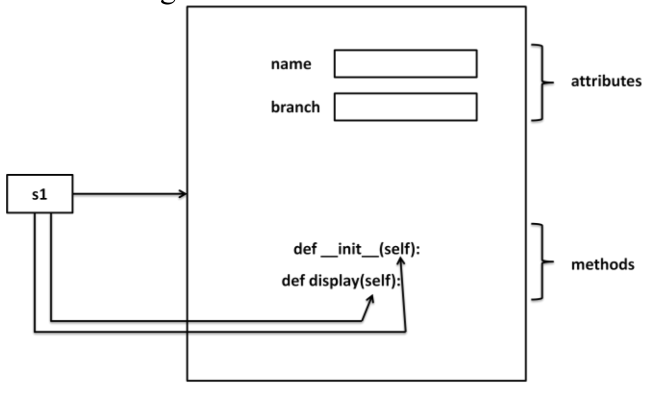
- The self variable is used as first parameter in the constructor as:

  def __init__(self):

  In this case, 'self' can be used to refer to the instance variables inside the constructor.
- 'self' can be used as first parameter in the instance methods as:

  def display(self):

  Here, display( ) is instance method as it acts on the instance variables. If this method wants to act on the instance variables, it should know the memory location of the instance variables. That memory location is by default available to the display( ) method through 'self'.



Student( )

Constructor:

A constructor is a special method that is used to initialize the instance variables of a class. In the constructor, we create the instance variables and initialize them with some starting values. The first parameter of the constructor will be 'self' variable that contains the memory address of the instance.

```
def __init__( self ):
        self.name = "hari"
        self.branch = "CSE"
```

Here, the constructor has only one parameter, i.e. 'self' using 'self.name' and 'self.branch', we can access the instance variables of the class. A constructor is called at the time of creating an instance. So, the above constructor will be called when we create an instance as:

s1 = Student()

Let's take another example, we can write a constructor with some parameters in addition to 'self' as:

```
def __init__( self , n = ' ' , b = ' ' ):
        self.name = n
        self.branch = b
```

Here, the formal arguments are ‘n’ and ‘b’ whose default values are given as ‘’ (None) and ‘’ (None). Hence, if we do not pass any values to constructor at the time of creating an instance, the default values of those formal arguments are stored into name and branch variables. For example,

s1 = Student( )

Since we are not passing any values to the instance, None and None are stored into name and branch. Suppose, we can create an instance as:

s1 = Student(“mothi”, “CSE”)

In this case, we are passing two actual arguments: “mothi” and “CSE” to the Student instance.

Example:

```
class Student:
    def __init__(self,n='',b=''):
        self.name=n
        self.branch=b
    def display(self):
        print "Hi",self.name
        print "Branch", self.branch
s1=Student()
s1.display()
print “------------------------------”
s2=Student("mothi","CSE")
s2.display()
print “------------------------------”
```

Output:

```
Hi
Branch
------------------------------
Hi mothi
Branch CSE
------------------------------
```

Types of Variables:

The variables which are written inside a class are of 2 types:

a) Instance Variables
b) Class Variables or Static Variables

a) Instance Variables

Instance variables are the variables whose separate copy is created in every instance. For example, if ‘x’ is an instance variable and if we create 3 instances, there will be 3 copies of ‘x’ in these 3 instances. When we modify the copy of ‘x’ in any instance, it will not modify the other two copies.

Example: A Python Program to understand instance variables.

```
class Sample:
    def __init__(self):
        self.x = 10
    def modify(self):
        self.x = self.x + 1
s1=Sample()
s2=Sample()
```

```
print "x in s1=",s1.x
print "x in s2=",s2.x
print "----------------"
s1.modify()
print "x in s1=",s1.x
print "x in s2=",s2.x
print "----------------"
```

Output:

```
x in s1= 10
x in s2= 10
----------------
x in s1= 11
x in s2= 10
----------------
```

Instance variables are defined and initialized using a constructor with 'self' parameter. Also, to access instance variables, we need instance methods with 'self' as first parameter. It is possible that the instance methods may have other parameters in addition to the 'self' parameter. To access the instance variables, we can use self.variable as shown in program. It is also possible to access the instance variables from outside the class, as: instancename.variable, e.g. s1.x

b) Class Variables or Static Variables

Class variables are the variables whose single copy is available to all the instances of the class. If we modify the copy of class variable in an instance, it will modify all the copies in the other instances. For example, if 'x' is a class variable and if we create 3 instances, the same copy of 'x' is passed to these 3 instances. When we modify the copy of 'x' in any instance using a class method, the modified copy is sent to the other two instances.

Example: A Python program to understand class variables or static variables.

```
class Sample:
    x=10
    @classmethod
    def modify(cls):
        cls.x = cls.x + 1
s1=Sample()
s2=Sample()
print "x in s1=",s1.x
print "x in s2=",s2.x
print "----------------"
s1.modify()
print "x in s1=",s1.x
print "x in s2=",s2.x
print "----------------"
```

Output:

```
x in s1= 10
x in s2= 10
----------------
x in s1= 11
x in s2= 11
----------------
```

Namespaces:

A *namespace* represents a memory block where names are mapped to objects. Suppose we write: n = 10

Here, ‘n’ is the name given to the integer object 10. Please recollect that numbers, strings, lists etc. Are all considered as objects in python. The name ‘n’ is linked to 10 in the namespace.

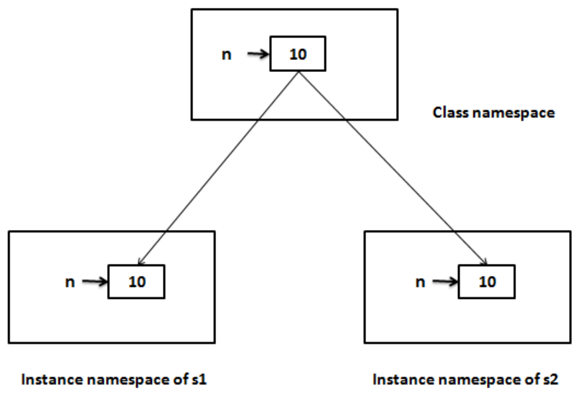
a) Class Namespace:

A class maintains its own namespace, called ‘class namespace’. In the class namespace, the names are mapped to class variables. In the following code, ‘n’ is a class variable in the student class. So, in the class namespace, the name ‘n’ is mapped or linked to 10 as shown in figure. We can access it in the class namespace, using classname.variable, as: Student.n which gives 10.

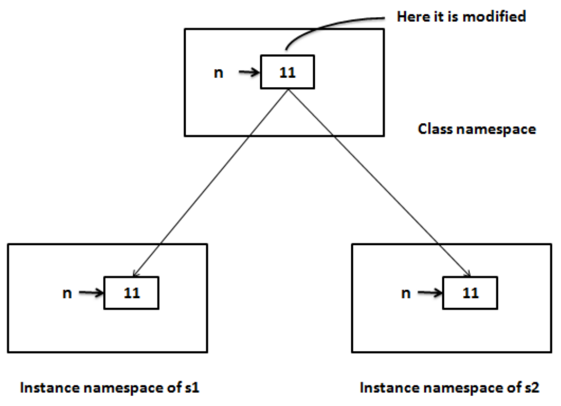
Example:

```
class Student:
        n = 10
print Student.n          # displays 10
Student.n += 1
print Student.n          # displays 11
s1 = Student( )
print s1.n               # displays 11
s2 = Student( )
print s2.n               # displays 11
```

Before modifying the class variable ‘n’



After modifying the class variable ‘n’



We know that a single copy of class variable is shared by all the instances. So, if the class variable is modified in the class namespace, since same copy of the variable is modified, the modified copy is available to all the instances.

b) Instance namespace:

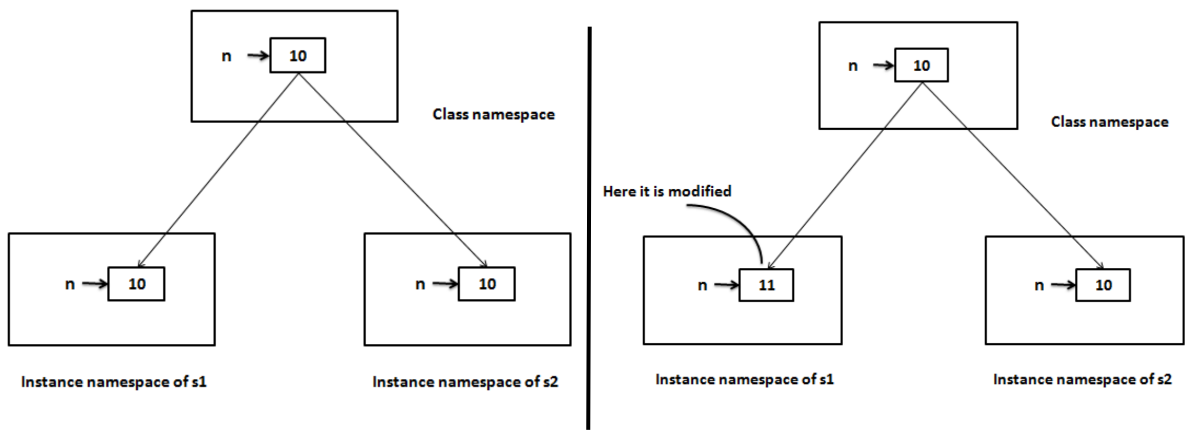
Every instance will have its own name space, called ‘instance namespace’. In the instance namespace, the names are mapped to instance variables. Every instance will have its own namespace, if the class variable is modified in one instance namespace, it will not affect the variables in the other instance namespaces. To access the class variable at the instance level, we have to create instance first and then refer to the variable as instancename.variable.

Example:

```
class Student:
        n = 10
s1 = Student( )
print s1.n               # displays 10
s1.n += 1
print s1.n               # displays 11
```

```
s2 = Student( )
print s2.n                # displays 11
```

Before modifying the class variable ‘n’   After modifying the class variable ‘n’



Types of methods:

We can classify the methods in the following 3 types:

a) Instance methods
   - Accessor methods
   - Mutator methods

b) Class methods

c) Static methods

a) Instance Methods:

Instance methods are the methods which act upon the instance variables of the class.instance methods are bound to instances and hence called as: instancename.method(). Since instance variables are available in the instance, instance methods need to know the memory address of instance. This is provided through ‘self’ variable by default as first parameter for the instance method. While calling the instance methods, we need not pass any value to the ‘self’ variable.

Example:

```
class Student:
    def __init__(self,n=",b="):
        self.name=n
        self.branch=b
    def display(self):
        print "Hi",self.name
        print "Branch", self.branch
s1=Student()
s1.display()
print “-----------------------------”
s2=Student("mothi","CSE")
s2.display()
print “-----------------------------”
```

- Instance methods are of two types: accessor methods and mutator methods.
- Accessor methods simply access of read data of the variables. They do not modify the data in the variables. Accessor methods are generally written in the form of getXXXX( ) and hence they are also called *getter* methods.
- Mutator methods are the methods which not only read the data but also modify them. They are written in the form of setXXXX( ) and hence they are also called *setter* methods.

Example:

```
class Student:
    def setName(self,n):
        self.name = n
    def setBranch(self,b):
        self.branch = b
    def getName(self):
        return self.name
    def getBranch(self):
        return self.branch
s=Student()
name=input("Enter Name: ")
branch=input("Enter Branch: ")
s.setName(name)
s.setBranch(branch)
print s.getName()
print s.getBranch()
```

b) Class methods:

These methods act on class level. Class methods are the methods which act on the class variables or static variables. These methods are written using *@classmethod* decorator above them. By default, the first parameter for class methods is 'cls' which refers to the class itself.

For example, 'cls.var' is the format to the class variable. These methods are generally called using classname.method( ). The processing which is commonly needed by all the instances of class is handled by the class methods.

Example:

```
class Bird:
    wings = 2

    @classmethod
    def fly(cls,name):
        print name,"flies with",cls.wings,"wings"

Bird.fly("parrot")  #display "parrot flies with 2 wings"
Bird.fly("sparrow") #display "sparow flies with 2 wings"
```

c) Static methods:

We need static methods when the processing is at the class level but we need not involve the class or instances. Static methods are used when some processing is related to the class but does not need the class or its instances to perform any work.

For example, setting environmental variables, counting the number of instances of the class or changing an attribute in another class, etc. are the tasks related to a class.

Such tasks are handled by static methods. Static methods are written with decorator @staticmethod above them. Static methods are called in the form of classname.method ( ).

Example:

```
class MyClass:
    n = 0
    def __init__(self):
        MyClass.n = Myclass.n + 1
    def noObjects():
        print "No. of instances created: ", MyClass.n
m1=MyClass()
m2=MyClass()
m3=MyClass()
MyClass.noObjects()
```

Inheritance:

- Software development is a team effort. Several programmers will work as a team to develop software.
- When a programmer develops a class, he will use its features by creating an instance to it. When another programmer wants to create another class which is similar to the class already created, then he need not create the class from the scratch. He can simply use the features of the existing class in creating his own class.
- Deriving new class from the super class is called *inheritance*.
- 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.

Syntax:

```
class Subclass(BaseClass):
        <class body>
```

- When an object is to SubClass is created, it contains a copy of BaseClass within it. This means there is a relation between the BaseClass and SubClass objects.
- We do not create BaseClass object,but still a copy of it is available to SubClass object.
- By using inheritance, a programmer can develop classes very easilt. Hence programmer's productivity is increased. Productivity is a term that refers to the code developed by the programmer in a given span of time.
- If the programmer used inheritance, he will be able to develop more code in less time.
- In inheritance, we always create only the sub class object. Generally, we do not create super class object. The reason is clear. Since all the members of the super class are available to sub class, when we crate an object, we can access the members of both the super and sub classes.

The super( ) method:

- super( ) is a built-in method which is useful to call the super class constructor or methods from the sub class.
- Any constructor written in the super class is not available to the sub class if the sub class has a constructor.
- Then how can we initialize the super class instance variables and use them in the sub class? This is done by calling super class constructor using super( ) method from inside the sub class constructor.
- super( ) is a built-in method which contains the history of super class methods.
- Hence, we can use super( ) to refer to super class constructor and methods from a aub class. So, super( ) can be used as:

```
super().init()  # call super class constructor
super().init(arguments) # call super class constructor and pass arguments
super().method() # call super class method
```

Example: Write a python program to call the super class constructor in the sub class using super( ).

```
class Father:
    def __init__(self, p = 0):
        self.property = p
    def display(self):
        print "Father Property",self.property
class Son(Father):
    def __init__(self,p1 = 0, p = 0):
        super().__init__(p1)
        self.property1 = p
    def display(self):
        print "Son Property",self.property+self.property1
s=Son(200000,500000)
s.display()
```

Output:

```
Son Property 700000
```

Example: Write a python program to access base class constructor and method in the sub class using super( ).

```
class Square:
    def __init__(self, x = 0):
        self.x = x
    def area(self):
        print "Area of square", self.x * self.x
class Rectangle(Square):
    def __init__(self, x = 0, y = 0):
        super().__init__(x)
        self.y = y
    def area(self):
        super().area()
        print "Area of Rectangle", self.x * self.y
r = Rectangle(5,16)
r.area()
```

Output:

```
Area of square 25
Area of Rectangle 80
```

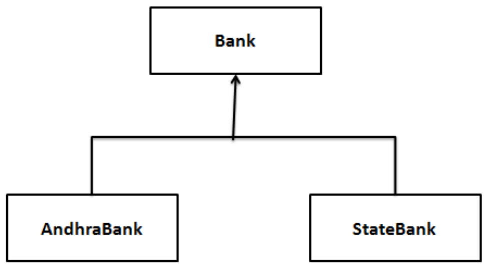
Types of Inheritance:

There are mainly 2 types of inheritance.

a) Single inheritance
b) Multiple inheritance

a) Single inheritance

Deriving one or more sub classes from a single base class is called 'single inheritance'. In single inheritance, we always have only one base class, but there can be n number of sub classes derived from it. For example, 'Bank' is a single base clas from where we derive 'AndhraBank' and 'StateBank' as sub classes. This is called single inheritance.



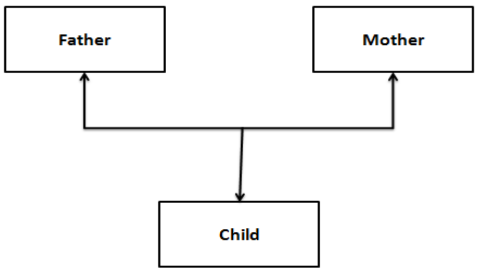
Example:

```
class Bank:
    cash = 100
    @classmethod
    def balance(cls):
        print cls.cash
class AndhraBank(Bank):
    cash = 500
    @classmethod
    def balance(cls):
        print "AndhraBank",cls.cash + Bank.cash
class StateBank(Bank):
    cash = 300
    @classmethod
    def balance(cls):
        print "StateBank",cls.cash + Bank.cash
a=AndhraBank()
a.balance()                 # displays AndhraBank 600
s=StateBank()
s.balance()                 #displays StateBank 400
```

b) Multiple inheritance

Deriving sub classes from multiple (or more than one) base classes is called 'multiple inheritance'. All the members of super classes are by default available to sub classes and the sub classes in turn can have their own members.

The best example for multiple inheritance is that parents are producing the children and the children inheriting the qualities of the parents.



Example:

```
class Father:
    def height(self):
        print "Height is 5.8 incehs"
class Mother:
    def color(self):
        print "Color is brown"
class Child(Father, Mother):
    pass
c=Child()
c.height() # displays Height is 5.8 incehs
c.color()  # displays Color is brown
```

Problem in Multiple inheritance:

- ➢ If the sub class has a constructor, it overrides the super class constructor and hence the super class constructor is not available to the sub class.
- ➢ But writing constructor is very common to initialize the instance variables.
- ➢ In multiple inheritance, let's assume that a sub class 'C' is derived from two super classes 'A' and 'B' having their own constructors. Even the sub class 'C' also has its constructor.

Example-1:

```
class A(object):
    def __init__(self):
        print "Class A"                    Output:
class B(object):                                          Class A
    def __init__(self):                                   Class C
        print "Class B"
class C(A,B,object):
    def __init__(self):
        super().__init__()
        print "Class C"
c1= C()
```

Example-2:

```
class A(object):
    def __init__(self):
        super().__init__()
        print "Class A"                    Output:
class B(object):                                          Class B
    def __init__(self):                                   Class A
        super().__init__()                                Class C
        print "Class B"
class C(A,B,object):
    def __init__(self):
        super().__init__()
        print "Class C"
c1= C()
```

Method Overriding:

When there is a method in the super class, writing the same method in the sub class so that it replaces the super class method is called 'method overriding'. The programmer overrides the super class methods when he does not want to use them in sub class.

Example:

```
import math
class square:
    def area(slef, r):
        print "Square area=",r * r
class Circle(Square):
    def area(self, r):
        print "Circle area=", math.pi * r * r
c=Circle()
c.area(15) # displays Circle area= 706.85834
```

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 are 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:/Python27/JustCounter.py", line 9, in <module>
    print counter.__secretCount
AttributeError: JustCounter instance 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
```

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

```
1
2
2
```

Errors and Exceptions:

As human beings, we commit several errors. A software developer is also a human being and hence prone to commit errors wither in the design of the software or in writing the code. The errors in the software are called 'bugs' and the process of removing them are called 'debugging'. In general, we can classify errors in a program into one of these three types:

a) Compile-time errors
b) Runtime errors
c) Logical errors

a) Compile-time errors

These are syntactical errors found in the code, due to which a program fails to compile. For example, forgetting a colon in the statements like if, while, for, def, etc. will result in compile-time error. Such errors are detected by python compiler and the line number along with error description is displayed by the python compiler.

Example: A Python program to understand the compile-time error.

```
a = 1
if a == 1
    print "hello"
```

Output:

```
        File ex.py, line 3
          If a == 1
                  ^
    SyntaxError: invalid syntax
```

b) Runtime errors

When PVM cannot execute the byte code, it flags runtime error. For example, insufficient memory to store something or inability of PVM to execute some statement come under runtime errors. Runtime errors are not detected by the python compiler. They are detected by the PVM, Only at runtime.

Example: A Python program to understand the compile-time error.

```
print "hai"+25
```

Output:

```
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print "hai"+25
TypeError: cannot concatenate 'str' and 'int' objects
```

c) Logical errors

These errors depict flaws in the logic of the program. The programmer might be using a wrong formula of the design of the program itself is wrong. Logical errors are not detected either by Python compiler of PVM. The programme is solely responsible for them. In the following program, the programmer wants to calculate incremented salary of an employee, but he gets wrong output, since he uses wrong formula.

Example: A Python program to increment the salary of an employee by 15%.

```
def increment(sal):
      sal = sal * 15/100
      return sal
sal = increment(5000)
print "Salary after Increment is", sal
```

Output:

```
Salary after Increment is 750
```

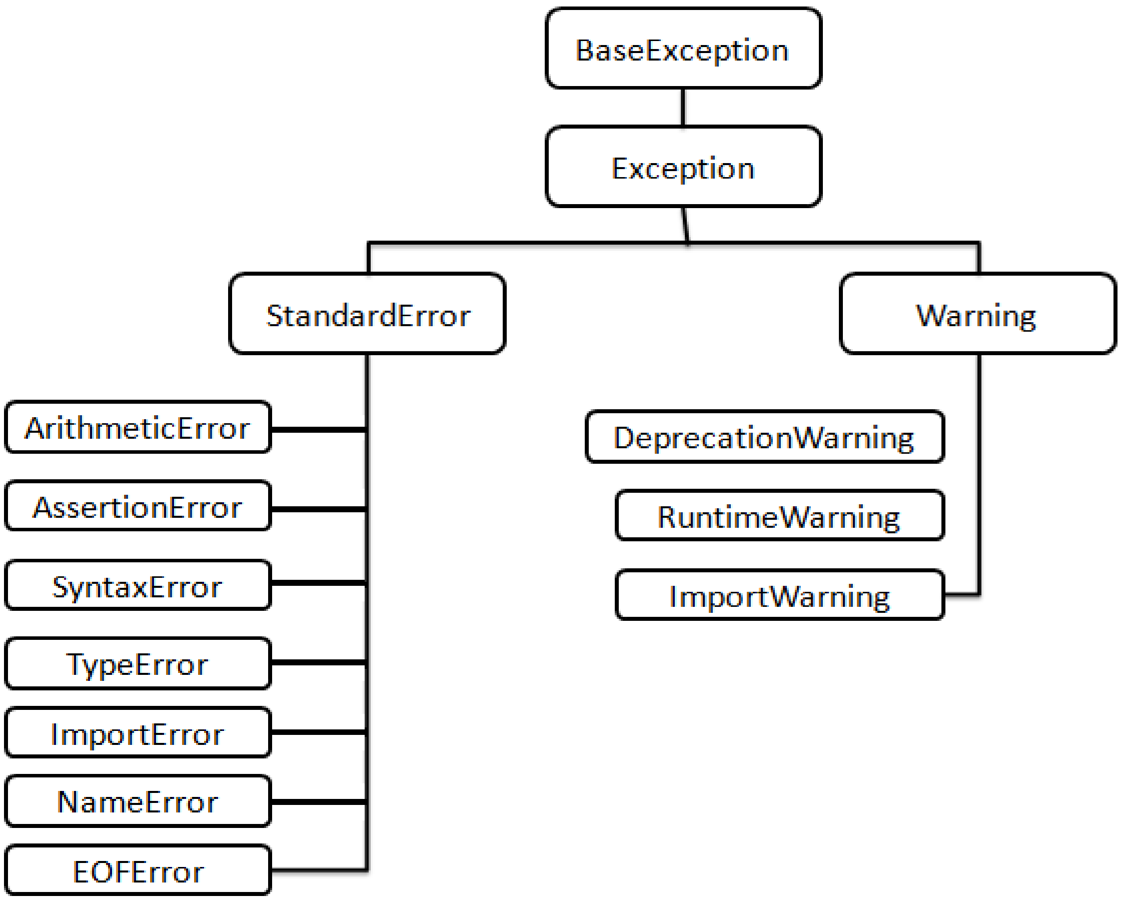
From the above program the formula for salary is wrong, because only the increment but it is not adding it to the original salary. So, the correct formula would be:

```
sal = sal + sal * 15/100
```

- ✓ Compile time errors and runtime errors can be eliminated by the programmer by modifying the program source code.
- ✓ In case of runtime errors, when the programmer knows which type of error occurs, he has to handle them using exception handling mechanism.

Exceptions:

- ➢ An exception is a runtime error which can be handled by the programmer.
- ➢ That means if the programmer can guess an error in the program and he can do something to eliminate the harm caused by that error, then it is called an 'exception'.
- ➢ If the programmer cannot do anything in case of an error, then it is called an 'error' and not an exception.
- ➢ All exceptions are represented as classes in python. The exceptions which are already available in python are called 'built-in' exceptions. The base class for all built-in exceptions is 'BaseException' class.
- ➢ From BaseException class, the sub class 'Exception' is derived. From Exception class, the sub classes 'StandardError' and 'Warning' are derived.
- ➢ All errors (or exceptions) are defined as sub classes of StandardError. An error should be compulsory handled otherwise the program will not execute.
- ➢ Similarly, all warnings are derived as sub classes from 'Warning' class. A warning represents a caution and even though it is not handled, the program will execute. So, warnings can be neglected but errors cannot neglect.
- ➢ Just like the exceptions which are already available in python language, a programmer can also create his own exceptions, called 'user-defined' exceptions.
- ➢ When the programmer wants to create his own exception class, he should derive his class from Exception class and not from 'BaseException' class.



Exception Handling:

- The purpose of handling errors is to make the program *robust*. The word 'robust' means 'strong'. A robust program does not terminate in the middle.
- Also, when there is an error in the program, it will display an appropriate message to the user and continue execution.
- Designing such programs is needed in any software development.
- For that purpose, the programmer should handle the errors. When the errors can be handled, they are called exceptions.

To handle exceptions, the programmer should perform the following four steps:

Step 1: The programmer should observe the statements in his program where there may be a possibility of exceptions. Such statements should be written inside a 'try' block. A try block looks like as follows:

```
try:
    statements
```

The greatness of try block is that even if some exception arises inside it, the program will not be terminated. When PVM understands that there is an exception, it jumps into an 'except' block.

Step 2: The programmer should write the 'except' block where he should display the exception details to the user. This helps the user to understand that there is some error in the program. The programmer should also display a message regarding what can be done to avoid this error. Except block looks like as follows:

```
except exceptionname:
        statements
```

The statements written inside an except block are called 'handlers' since they handle the situation when the exception occurs.

Step 3: If no exception is raised, the statements inside the 'else' block is executed. Else block looks like as follows:

```
else:
   statements
```

Step 4: Lastly, the programmer should perform clean up actions like closing the files and terminating any other processes which are running. The programmer should write this code in the finally block. Finally block looks like as follows:

```
finally:
  statements
```

The speciality of finally block is that the statements inside the finally block are executed irrespective of whether there is an exception or not. This ensures that all the opened files are properly closed and all the running processes are properly terminated. So, the data in the files will not be corrupted and the user is at the safe-side.

Here, the complete exception handling syntax will be in the following format:

```
try:
     statements
except Exception1:
     statements
except Exception2:
     statements
else:
     statements
finally:
     statements
```

The following points are followed in exception handling:

- ✓ A single try block can be followed by several except blocks.
- ✓ Multiple except blocks can be used to handle multiple exceptions.
- ✓ We cannot write except blocks without a try block.
- ✓ We can write a try block without any except blocks.
- ✓ Else block and finally blocks are not compulsory.
- ✓ When there is no exception, else block is executed after try block.
- ✓ Finally block is always executed.

Example: A python program to handle IOError produced by open() function.

```
import sys
try:
   f = open('myfile.txt','r')
   s = f.readline()
   print s
   f.close()
except IOError as e:
   print "I/O error", e.strerror
except:
   print "Unexpected error:"
```

Output:

```
I/O error No such file or directory
```

In the if the file is not found, then IOError is raised. Then 'except' block will display a message: 'I/O error'. if the file is found, then all the lines of the file are read using readline() method.

List of Standard Exceptions

Exception Name	Description
Exception	Base class for all exceptions
StopIteration	Raised when the next() method of an iterator does not point to any object.
SystemExit	Raised by the sys.exit() function.
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisionError	Raised when division or modulo by zero takes place for all numeric types.
AssertionError	Raised in case of failure of the Assert statement.
AttributeError	Raised in case of failure of attribute reference or assignment.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
LookupError	Base class for all lookup errors.
IndexError	Raised when an index is not found in a sequence.
KeyError	Raised when the specified key is not found in the dictionary.
NameError	Raised when an identifier is not found in the local or global namespace.
UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it.
EnvironmentError	Base class for all exceptions that occur outside the Python environment.
IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
OSError	Raised for operating system-related errors.
SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
RuntimeError	Raised when a generated error does not fall into any category.
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

The Except Block:

The ‘except’ block is useful to catch an exception that is raised in the try block. When there is an exception in the try block, then only the except block is executed. it is written in various formats.

1. To catch the exception which is raised in the try block, we can write except block with the Exceptionclass name as:

   except Exceptionclass:
2. We can catch the exception as an object that contains some description about the exception.

   except Exceptionclass as obj:
3. To catch multiple exceptions, we can write multiple catch blocks. The other way is to use a single except block and write all the exceptions as a tuple inside parantheses as:

   except (Exceptionclass1, Exceptionclass2, ...):
4. To catch any type of exception where we are not bothered about which type of exception it is, we can write except block without mentioning any Exceptionclass name as:

   except:

Example:

```
try:
    f = open('myfile.txt','w')
    a=input("Enter a value ")
    b=input("Enter a value ")
    c=a/float(b)
    s = f.write(str(c))
    print “Result is stored”
except ZeroDivisionError:
    print "Division is not possible"
except:
    print "Unexpected error:"
finally:
    f.close()
```

Output:

```
Enter a value 1
Enter a value 5
Result is stored
```

Raising an Exception

You can raise exceptions in several ways by using the raise statement. The general syntax for the **raise** statement is as follows.

raise [Exception [, args [, traceback]]]

Here, *Exception* is the type of exception (For example, NameError) and *argument* is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None.

For Example, If you need to determine whether an exception was raised but don’t intend to handle it, a simpler form of the raise statement allows you to re-raise the exception:

```
try:
   raise NameError('HiThere')
except NameError:
   print 'An exception flew by!'
   raise
```

User-Defined Exceptions:

- Like the built-in exceptions of python, the programmer can also create his own exceptions which are called 'User-defined exceptions' or 'Custom exceptions'. We know Python offers many exceptions which will raise in different contexts.
- But, there may be some situations where none of the exceptions in Python are useful for the programmer. In that case, the programme has to create his/her own exception and raise it.
- For example, let's take a bank where customers have accounts. Each account is characterized should by customer name and balance amount.
- The rule of the bank is that every customer should keep minimum Rs. 2000.00 as balance amount in his account.
- The programmer now is given a task to check the accounts to know every customer is maintaining minimum balance of Rs. 2000.00 or not.
- If the balance amount is below Rs. 2000.00, then the programmer wants to raise an exception saying 'Balance amount is less in the account of so and so person.' This will be helpful to the bank authorities to find out the customer.
- So, the programmer wants an exception that is raised when the balance amount in an account is less than Rs. 2000.00. Since there is no such exception available in python, the programme has to create his/her own exception.
- For this purpose, he/she has to follow these steps:

1. Since all exceptions are classes, the programme is supposed to create his own exception as a class. Also, he should make his class as a sub class to the in-built 'Exception' class.

```
class MyException(Exception):
def __init__(self, arg):
        self.msg = arg
```

Here, MyException class is the sub class for 'Exception' class. This class has a constructor where a variable 'msg' is defined. This 'msg' receives a message passed from outside through 'arg'.

2. The programmer can write his code; maybe it represents a group of statements or a function. When the programmer suspects the possibility of exception, he should raise his own exception using 'raise' statement as:

```
raise MyException('message')
```

Here, raise statement is raising MyException class object that contains the given 'message'.

3. The programmer can insert the code inside a 'try' block and catch the exception using 'except' block as:

```
try:
   code
except MyException as me:
    print me
```

Here, the object 'me' contains the message given in the raise statement. All these steps are shown in below program.

Example:

```
class MyException(Exception):
    def __init__(self, arg):
        self.msg = arg
def check(dict):
    for k,v in dict.items():
        print "Name=",k,"Balance=",v
        if v<2000.00:
            raise MyException("Balance amount is less in the account of "+k)

bank={"ravi":5000.00,"ramu":8500.00,"raju":1990.00}
try:
    check(bank)
except MyException as me:
    print me.msg
```

Output:

```
Name= ramu Balance= 8500.0
Name= ravi Balance= 5000.0
Name= raju Balance= 1990.0
Balance amount is less in the account of  raju
```