Python Constructors

A constructor is a special type of method (function) which is used to initialize the instance members of the class. Constructor can be parameterized and non-parameterized as well. Constructor definition executes when we create object of the class.

Creating a Constructor

- A constructor is a class function that begins with double underscore (_). The name of the constructor is always the same __init__().
- When we create a class without a constructor, Python automatically creates a default constructor that doesn't do anything.
- In Python, Constructors can be parameterized and non-parameterized as well. The parameterized constructors are used to set custom value for instance variables that can be used further in the application.

Python Non Parameterized Constructor Example

```
class Student:
   # Constructor - non parameterized
   def __init__(self):
      print("This is non parametrized constructor")
   def show(self,name):
      print("Hello",name)
student1 = Student()
student1.show("srikanth")
```

output:

```
====================== RESTART: E:/sri/non_perm
This is non parametrized constructor
Hello srikanth
```

Python Parameterized Constructor Example

```
class Student:
   # Constructor - parameterized
   def __init__(self, name):
      print("This is parametrized constructor")
      self.name = name
   def show(self):
      print("Hello",self.name)
student1 = Student("srikanth")
student1.show()
```

output:

```
========== RESTART: E:/sri/c
This is parametrized constructor
Hello srikanth
```

Inheritance:

Inheritance is a feature of Object Oriented Programming. It is used to specify that one class will get most or all of its features from its parent class. It is a very powerful feature which facilitates users to create a new class with a few or more modification to an existing class. The new class is called child class or derived class and the main class from which it inherits the properties is called base class or parent class.

The child class or derived class inherits the features from the parent class, adding new features to it. It facilitates **re-usability of code**.

Python Inheritance Terminologies

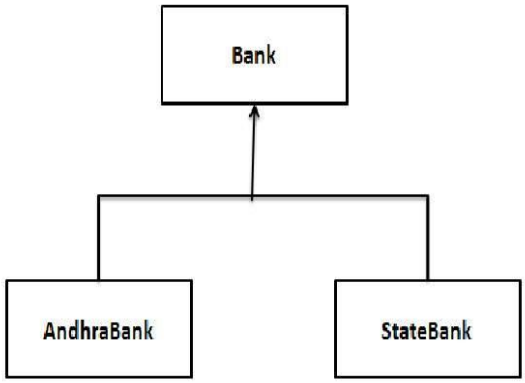
1. **Superclass:** The class from which attributes and methods will be inherited.
2. **Subclass:** The class which inherits the members from superclass.
3. **Method Overloading:** Redefining the definitions of methods in subclass which was already defined in superclass.

There are mainly 2 types of inheritance.

a) Single inheritance
b) Multiple inheritance

b 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.



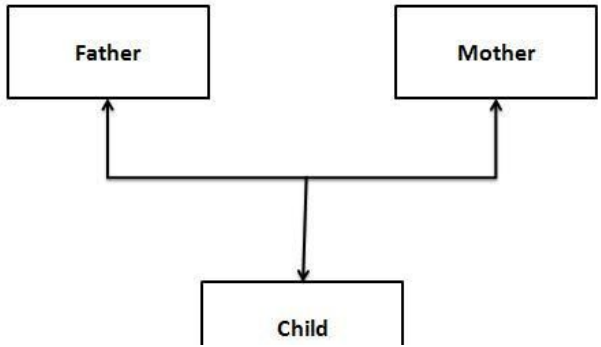
```
class Bank:
    cash = 100
    @classmethod
    def balance(cls):
        print (cls.cash)
class AndhraBank(Bank):
    cash = 500
    @classmethod
    def balance(cls):
        print ("AndhraBank",cls.cash + Bank.cash)
class StateBank(Bank):
    cash = 300
    @classmethod
    def balance(cls):
        print ("StateBank",cls.cash + Bank.cash)

a=AndhraBank()
a.balance()
s=StateBank()
s.balance()
```

Output:

```
= RESTART: C:/Users/acer/AppData/Local/Prog
AndhraBank 600
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.



```
class Father:
    def height(self):
        print ("Height is 5.8 incehs")
class Mother:
    def color(self):
        print ("Color is white")

class Child(Father, Mother):
    pass
c=Child()
c.height() # displays Height is 5.8 incehs
c.color()  # displays Color is white
```

Output:

```
========== RESTART: E:/sri/multip
Height is 5.8 incehs
Color is white
```

Overriding Methods:

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".

Redefining the definitions of methods in subclass which was already defined in superclass.

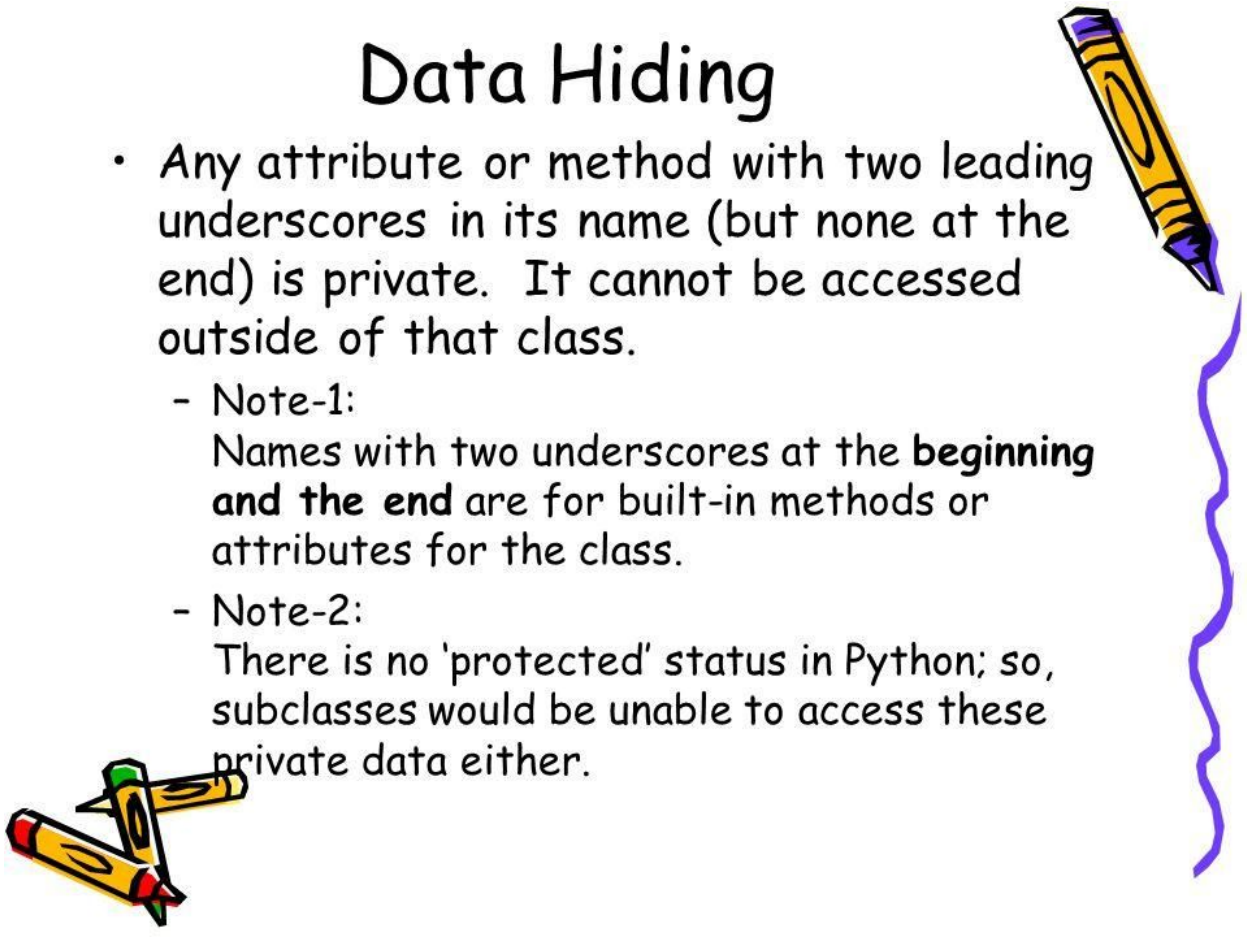
```
import math
class Square:
    def area(slef, r):
        print ("Square area=",r * r)
class Circle(Square):
    def area(self, r):
        print ("Circle area=", math.pi * r * r)
c=Circle()
c.area(15)
```

Output:

```
= RESTART: C:/Users/acer/AppData/Local/Programs/
Circle area= 706.8583470577034
```

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)

j = JustCounter()
j.count()
j.count()
j.count()
j.count()
print (j.__secretCount)
#print (j._JustCounter__secretCount)
```

Output:

```
======================== RESTART: E:/sri/datahiding.py ================
1
2
3
4
Traceback (most recent call last):
  File "E:/sri/datahiding.py", line 12, in <module>
    print (j.__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 (j._JustCounter__secretCount)

output:

1
2
3
4
4

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

    returnsal

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:

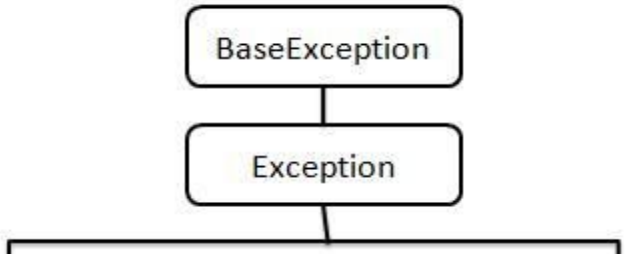
sal = sal + sal * 15/100

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 apossibility 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 shouldwrite the „except‟ block where he should display theexception 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:

exceptexceptionname:

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 blocklooks like as follows:

else:

statements

Step 4: Lastly, the programmer should perform clean up actions like closing the files andterminating 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:

```
= RESTART: C:/Users/acer/AppData/Local/Programs/Pytho
I/O error No such file or directory
>>>
```

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.

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.

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:

exceptExceptionclass:

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

exceptExceptionclass 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:

```
try:
    num1, num2 = eval(input("Enter two numbers, separated by a comma : "))
    result = num1 / num2
    print("Result is", result)

except ZeroDivisionError:
    print("Division by zero is error !!")

except SyntaxError:
    print("Comma is missing. Enter numbers separated by comma like this 1, 2")

except:
    print("Wrong input")

else:
    print("No exceptions")

finally:
    print("mic cse")
```

Output:

```
= RESTART: C:/Users/acer/AppData/Local/Programs/Python/Python36-32/class7.py =
Enter two numbers, separated by a comma : 0,0
Division by zero is error !!
mic cse
>>>
= RESTART: C:/Users/acer/AppData/Local/Programs/Python/Python36-32/class7.py =
Enter two numbers, separated by a comma : 1
Wrong input
mic cse
>>>
= RESTART: C:/Users/acer/AppData/Local/Programs/Python/Python36-32/class7.py =
Enter two numbers, separated by a comma : 1 2
Comma is missing. Enter numbers separated by comma like this 1, 2
mic cse
>>>
= RESTART: C:/Users/acer/AppData/Local/Programs/Python/Python36-32/class7.py =
Enter two numbers, separated by a comma : 100,10
Result is 10.0
No exceptions
mic cse
```

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:

Output:

```
= RESTART: C:/Users/acer/AppData/Local/Programs/Pytho
An exception raise!
Traceback (most recent call last):
  File "C:/Users/acer/AppData/Local/Programs/Python/P
    raise NameError('HiThere')
NameError: HiThere
```

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.

classMyException(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:

raiseMyException('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

exceptMyException as me:
    print me
```

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

```
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={"srikanth":5000.00,"krishna":8500.00,"mic":1990.00}
try:

    check(bank)
except MyException as me:

    print (me.msg)
```

Output:

```
= RESTART: C:/Users/acer/AppData/Local/Programs/Python
Name= srikanth Balance= 5000.0
Name= krishna Balance= 8500.0
Name= mic Balance= 1990.0
Balance amount is less in the account of mic
```