LAB Manual

PART A

(PART A: TO BE REFFERED BY STUDENTS)

Experiment No.02

A.1 Aim:

Creating functions, classes and objects using python. Demonstrate exception handling and inheritance.

A.2 Prerequisite:

1. C,JAVA Language

A.3 Outcome:

After successful completion of this experiment students will be able to To demonstrate basic concepts in python using OOP methodology.

A.4 Theory& Procedure:

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 behavior 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.

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 follows −

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

· 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.

Example

Following is the example of a simple Python class −

```
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 instances of a 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 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

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
```

Now, putting all the concepts together −

```
#!/usr/bin/python



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


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

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.
```

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

- The getattr(obj, name[, default]) − to access the attribute of object.
- The hasattr(obj,name) − to check if an attribute exists or not.
- The setattr(obj,name,value) − to set an attribute. If attribute does not exist, then it would be created.
- The delattr(obj, name) − to delete an attribute.

```
hasattr(emp1, 'age') # Returns true if 'age' attribute exists
getattr(emp1, 'age') # Returns value of 'age' attribute
setattr(emp1, 'age', 8) # Set attribute 'age' at 8
delattr(empl, 'age') # Delete attribute 'age'
```

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 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 −

```
#!/usr/bin/python


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


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__: ()
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>}
```

Destroying Objects (Garbage Collection)

Python deletes unneeded objects (built-in types or class instances) automatically to free the memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed Garbage Collection.

Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of aliases that point to it changes.

An object's reference count increases when it is assigned a new name or placed in a container (list, tuple, or dictionary). The object's reference count decreases when it's deleted with del, its reference is reassigned, or its reference goes out of scope. When an object's reference count reaches zero, Python collects it automatically.

```
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>
```

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.

Example

This __del__() destructor prints the class name of an instance that is about to be destroyed −

```
#!/usr/bin/python


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
print id(pt1), id(pt2), id(pt3) # prints the ids of the obejcts
del pt1
del pt2
del pt3
```

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

```
3083401324 3083401324 3083401324
Point destroyed
```

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

Class Inheritance

Instead of starting from scratch, you can create a class by deriving it from a preexisting 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.

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

```
#!/usr/bin/python

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
```

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

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

Similar way, you can drive a class from multiple parent classes as follows −

```
class A: # define your class A
.....

class B: # define your class B
.....

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

You can use issubclass() or isinstance() functions to check a relationships 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

Overriding Methods

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

Example

```
#!/usr/bin/python


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
```

Base Overloading Methods

Following table lists some generic functionality that you can override in your own classes −

Sr.No.	Method, Description & Sample Call
1	__init__ (self [,args...]) Constructor (with any optional arguments) Sample Call : obj = className(args)
2	__del__(self) Destructor, deletes an object Sample Call : del obj
3	__repr__(self) Evaluable string representation Sample Call : repr(obj)
4	__str__(self) Printable string representation Sample Call : str(obj)
5	__cmp__ (self, x) Object comparison Sample Call : cmp(obj, x)

Overloading Operators

Suppose you have created a Vector class to represent two-dimensional vectors, what happens when you use the plus operator to add them? Most likely Python will yell at you.

You could, however, define the __add__ method in your class to perform vector addition and then the plus operator would behave as per expectation −

Example

```
#!/usr/bin/python


class Vector:
 def __init__(self, a, b):
 self.a = a
 self.b = b


 def __str__(self):
 return 'Vector (%d, %d)' % (self.a, self.b)


 def __add__(self,other):
 return Vector(self.a + other.a, self.b + other.b)


v1 = Vector(2,10)
```

```
v2 = Vector(5,-2)
print v1 + v2
```

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

```
Vector(7,8)
```

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

```
#!/usr/bin/python


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 "test.py", line 12, 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
```

What is Exception?
An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.
When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.
Handling an exception
If you have some suspicious code that may raise an exception, you can defend your program by placing the suspicious code in a try: block. After the try: block, include an except: statement, followed by a block of code which handles the problem as elegantly as possible. Syntax
Here is simple syntax of try....except...else blocks −

```
try:
 You do your operations here;
 ......................
except ExceptionI:
 If there is ExceptionI, then execute this block.
except ExceptionII:
 If there is ExceptionII, then execute this block.
 ......................
else:
 If there is no exception then execute this block.
```

Here are few important points about the above-mentioned syntax −
A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
You can also provide a generic except clause, which handles any exception. After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
The else-block is a good place for code that does not need the try: block's protection. Example
This example opens a file, writes content in the, file and comes out gracefully because there is no problem at all −

```
#!/usr/bin/python

try:
 fh = open("testfile", "w")
 fh.write("This is my test file for exception handling!!")
except IOError:
 print "Error: can\'t find file or read data"
else:
 print "Written content in the file successfully"
 fh.close()
```
This produces the following result −
```
Written content in the file successfully
```
Example

This example tries to open a file where you do not have write permission, so it raises an exception −
```
#!/usr/bin/python

try:
 fh = open("testfile", "r")
 fh.write("This is my test file for exception handling!!")
except IOError:
 print "Error: can\'t find file or read data"
else:
 print "Written content in the file successfully"
```
This produces the following result −
```
Error: can't find file or read data
```
The except Clause with No Exceptions

You can also use the except statement with no exceptions defined as follows − try:
```
 You do your operations here;
 ......................
except:
 If there is any exception, then execute this block.
 ......................
else:
 If there is no exception then execute this block.
```
This kind of a try-except statement catches all the exceptions that occur. Using this kind of try except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

The except Clause with Multiple Exceptions

You can also use the same except statement to handle multiple exceptions as follows − try:
```
 You do your operations here;
 ......................
except(Exception1[, Exception2[,...ExceptionN]]]):
 If there is any exception from the given exception list,
 then execute this block.
 ......................
```

```
else:
 If there is no exception then execute this block.
```
The try-finally Clause

You can use a finally: block along with a try: block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this −
```
try:
 You do your operations here;
 ......................
 Due to any exception, this may be skipped.
finally:
 This would always be executed.
 ......................
```
You cannot use else clause as well along with a finally clause.

Example
```
#!/usr/bin/python

try:
 fh = open("testfile", "w")
 fh.write("This is my test file for exception handling!!")
finally:
 print "Error: can\'t find file or read data"
```
If you do not have permission to open the file in writing mode, then this will produce the following result −
```
Error: can't find file or read data
```
Same example can be written more cleanly as follows −
```
#!/usr/bin/python

try:
 fh = open("testfile", "w")
 try:
 fh.write("This is my test file for exception handling!!")
 finally:
 print "Going to close the file"
 fh.close()
except IOError:
 print "Error: can\'t find file or read data"
```
When an exception is thrown in the try block, the execution immediately passes to the finally block. After all the statements in the finally block are executed, the exception is raised again and is handled in the except statements if present in the next higher layer of the try except statement.

Argument of an Exception

An exception can have an argument, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows −
```
try:
 You do your operations here;
 ......................
except ExceptionType, Argument:
 You can print value of Argument here...
```
If you write the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.

This variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.

Example

Following is an example for a single exception −
```
#!/usr/bin/python

# Define a function here.
def temp_convert(var):
 try:
 return int(var)
 except ValueError, Argument:
 print "The argument does not contain numbers\n", Argument

# Call above function here.
temp_convert("xyz");
```
This produces the following result −
```
The argument does not contain numbers
invalid literal for int() with base 10: 'xyz'
```
Raising an Exceptions

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

Syntax
```
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.

The final argument, traceback, is also optional (and rarely used in practice), and if present, is the traceback object used for the exception.

Example

An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows −
```
def functionName( level ):
 if level < 1:
 raise "Invalid level!", level
 # The code below to this would not be executed
 # if we raise the exception
```
Note: In order to catch an exception, an "except" clause must refer to the same exception thrown either class object or simple string. For example, to capture above exception, we must write the except clause as follows −

```
try:
 Business Logic here...
except "Invalid level!":
 Exception handling here...
else:
 Rest of the code here...
```

User-Defined Exceptions

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example related to RuntimeError. Here, a class is created that is subclassed from RuntimeError. This is useful when you need to display more specific information when an exception is caught.

In the try block, the user-defined exception is raised and caught in the except block. The variable e is used to create an instance of the class Networkerror.

```
class Networkerror(RuntimeError):
 def __init__(self, arg):
 self.args = arg
```

So once you defined above class, you can raise the exception as follows −

```
try:
 raise Networkerror("Bad hostname")
except Networkerror,e:
 print e.args
```

PART B

(PART B : TO BE COMPLETED BY STUDENTS)

(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded on the Blackboard or emailed to the concerned lab in charge faculties at the end of the practical in case the there is no Black board access available)

Roll No.	Name:
Class :SE	Batch :

Date of Experiment:	Date of Submission
Grade :	

Document created by the student:

B.1 Observations and learning:

(Students are expected to understand the selected topic.

Students have to include the program/programs and output/outputs executed in the lab session. Have to list out the components & functionality. Prepare a flow of the algorithm defined in the aim. List the performance metrics that is used (if any))

B.2 Conclusion:

(Students must write the conclusion as per the attainment of the aim)

B.3 Question of Curiosity

(To be answered by student based on the practical performed and learning/observations)

Q.1] Discuss Inheritance with example?

Q.2] Discuss Exception handling. Also List out different types of Exceptions?

****