Agenda

- OOPS Concepts

Classes and Objects

OOP concepts

- Python has been an object-oriented language from day one.
- Because of this, creating and using classes and objects are downright easy.

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 aren't 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.

Overview of OOP Terminology

- **Function overloading:** The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects (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.

Overview of OOP Terminology

- **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.

Creating Classes:

```
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
```

Creating Classes:

- The variable *empCount* is a class variable whose value would be 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 don't 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" % emp1.displayCount())

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

- 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.
```

- Methods may call other methods by using method attributes of the self argument:

```
class Bag:

        def __init__(self):
                self.data = []
        def add(self, x):
                self.data.append(x)
        def addtwice(self, x):
                self.add(x)
                self.add(x)
```

- Instead of using the normal statements to access attributes, you can use 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'

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.

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

- **Syntax:**

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

Class Inheritance

- When the base class is defined in another module:

  class DerivedClassName(modname.BaseClassName):

- When the class object is constructed, the base class is remembered.
- This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class.
- This rule is applied recursively if the base class itself is derived from some other class.

Class Inheritance

- Instantiation of derived classes is as usual :

  DerivedClassName()

  creates a new instance of the class.

- Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.

Class Inheritance

- The base class constructor will not get called automatically whenever a child object is created.

- We have to call it explicitly using super().__init__()

```
class Child(Base):
    def __init__(self, value, something_else):
        super().__init__(value)
        self.something_else = something_else
```

Class Inheritance

- 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**.
- **Example** : issubclass(bool, int) is True since bool is a subclass of int. However, issubclass(float, int) is False since float is not a subclass of int.

- 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
- **Example** : : isinstance(obj, int) will be True only if obj.__class__ is int or some class derived from int.

Method overriding

- Derived classes may override methods of their base classes.
- An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name.
- There is a simple way to call the base class method directly: just call
  BaseClassName.methodname(self, arguments).

Overriding 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
```

Multiple Inheritance

- Python supports a form of multiple inheritance as well.

```
class DerivedClassName(Base1, Base2, Base3):
<statement-1>
.
.
.
<statement-N>
```

Example :

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

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

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

Overloading Operators

- Suppose you've created a Vector class to represent two-dimensional vectors
- You could 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):
     a=self.a+other.a
     b=self.b+other.b
     v=vector(a,b)
      return v
v1 = Vector(2,10)
v2 = Vector(5,-2)
Print v1+ v2
```

Exception Handling

Python Exceptions Handling

- **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 can't 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 would terminate and come out.

Python Exceptions Handling

- 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:**

```
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.
   ......................
except:
  default exception cluase.
else:
   If there is no exception then execute this block.
```

Python Exceptions Handling

- 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.

Python Exceptions Handling

```
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()
```

When you try to open a file where you do not have permission to write in the file, it raises an exception:

Python Exceptions Handling

- **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 e1:

except e2:

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.

Python Exceptions Handling

- **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.
```

Python Exceptions Handling

- **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.
   ......................
```

Note that you can provide except clause(s), or a finally clause, but not both. You can not use *else* clause as well along with a finally clause.

Python Exceptions Handling

```
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**

Python Exceptions Handling

```
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“
```

Python Exceptions Handling

```
import sys
try:
        f = open('myfile.txt')
        s = f.readline()
        i = int(s.strip())
except IOError as err:
        print("I/O error: {0}".format(err))
        print(a,b)
        print('a=%d b=%d'%(a,b))
        print('a = {0} b= {1}'.format(a,b)
except ValueError:
        print("Could not convert data to an integer.")
except:
        print("Unexpected error:", sys.exc_info()[0])
```

Exceptions Are Classes Too

- User-defined exceptions are identified by classes as well.

- There are two new valid (semantic) forms for the raise statement:
  - **raise** Class
  - **raise** Instance

- In the first form, Class must be an instance of type or of a class derived from it. The first form is a shorthand for:
    - **raise** Class()

- A class in an except clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around—an except clause listing a derived class is not compatible with a base class).

- For example, the following code will print B, C, D in that order:

```
class B(Exception):
    pass
class C(B):
    pass
class D(C):
    pass
```

Exceptions Are Classes Too

```
for c in [B, C, D]:
    try:
        raise(c)
    except D:
        print(“D")
    except C:
        print(“C")
    except B:
        print(“B")
```

- If the except clauses were reversed (with except B first), it would have printed B, B, B — the first matching except clause is triggered.

Summary

- The following topics are covered so far
  - Classes and Objects
  - Exception Handling

Thank you