PYTHON – DAY 3

Day 3 Agenda

- Recap of Day 1 and Day 2
- Working with Classes / Objects
- Database Access
- MultiThreading
- GUI Programming
- Integrating with other tools

Classes and Objects

- Python is an object oriented programming language. Unlike procedure oriented programming, where the main emphasis is on functions, object oriented programming stress on objects.
- Object is simply a collection of data (variables) and methods (functions) that act on those data.
- And, class is a blueprint for the object.
- We can think of class as a sketch (prototype) of a house.
- It contains all the details about the floors, doors, windows etc.
- Based on these descriptions we build the house. House is the object.
- As, many houses can be made from a description, we can create many objects from a class.
- An object is also called an instance of a class and the process of creating this object is called instantiation.

Classes and Objects

What is a class?

A class is a code template for creating objects.

Objects have member variables and have behaviour associated with them.

In python, a class is created by the keyword *class*.

An object is created using the constructor of the class.

This object will then be called the instance of the class. In Python we create instances in the following manner

```
Instance = class(arguments)
```

Some more Terminologies ahead…

Classes and Objects

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.

Classes and Objects

OOP Terminology

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.

So how do we create a class now…

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__. It is optional.

The class_suite consists of all the component statements defining class members, data attributes and functions.

Classes and Objects

Class 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
```

Explanation:

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. Self represents the instance of the class. By using the "self" keyword we can access the attributes and methods of the class in python.

Classes and Objects

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("Ron", 2000)

"This would create second object of Employee class"
emp2 = Employee("Rita", 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
```

Classes and Objects

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'
```

Classes and Objects

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.

Garbage Collection

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.

Destructor Explained…

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

```
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
```

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
```

Class Inheritance

Example:

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

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

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

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

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

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

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:

```
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
```

Base Overloading Methods

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

Method, Description & Sample Call

__init__ (self [,args...])
Constructor (with any optional arguments)
Sample Call : *obj = className(args)*

__del__(self)
Destructor, deletes an object
Sample Call : *del obj*

__repr__(self)
Evaluable string representation
Sample Call : *repr(obj)*

__str__(self)
Printable string representation
Sample Call : *str(obj)*

__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

```
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
```

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
```

Multithreading

Running several threads is similar to running several different programs concurrently, but with the following benefits −

- Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.
- Threads sometimes called light-weight processes and they do not require much memory overhead; they are cheaper than processes.
- A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context it is currently running.
- It can be pre-empted (interrupted)
- It can temporarily be put on hold (also known as sleeping) while other threads are running - this is called yielding.

Starting a New Thread

- To spawn another thread, you need to call following method available in thread module –
- thread.start_new_thread (function, args[, kwargs])

- This method call enables a fast and efficient way to create new threads in both Linux and Windows.

- The method call returns immediately and the child thread starts and calls function with the passed list of args. When function returns, the thread terminates.

- Here, args is a tuple of arguments; use an empty tuple to call function without passing any arguments. kwargs is an optional dictionary of keyword arguments.

The *Threading* Module

- The newer threading module included with Python 2.4 provides much more powerful, high-level support for threads than the thread module discussed in the previous section.
- The *threading* module exposes all the methods of the *thread* module and provides some additional methods −
- threading.activeCount() − Returns the number of thread objects that are active.
- threading.currentThread() − Returns the number of thread objects in the caller's thread control.
- threading.enumerate() − Returns a list of all thread objects that are currently active.

Thread Methods

The methods provided by the Thread class are as follows −

run() − The run() method is the entry point for a thread.

start() − The start() method starts a thread by calling the run method.

join([time]) − The join() waits for threads to terminate.

isAlive() − The isAlive() method checks whether a thread is still executing.

getName() − The getName() method returns the name of a thread.

setName() − The setName() method sets the name of a thread.

Creating Thread Using Threading Module

To implement a new thread using the threading module, you have to do the following –

Define a new subclass of the Thread class.

Override the __init__(self [,args]) method to add additional arguments.

Then, override the run(self [,args]) method to implement what the thread should do when started.

Once you have created the new Thread subclass, you can create an instance of it and then start a new thread by invoking the start(), which in turn calls run() method.

Synchronizing Threads

The threading module provided with Python includes a simple-to-implement locking mechanism that allows you to synchronize threads. A new lock is created by calling the Lock() method, which returns the new lock.

The acquire(blocking) method of the new lock object is used to force threads to run synchronously. The optional blocking parameter enables you to control whether the thread waits to acquire the lock.

If blocking is set to 0, the thread returns immediately with a 0 value if the lock cannot be acquired and with a 1 if the lock was acquired. If blocking is set to 1, the thread blocks and wait for the lock to be released.

The release() method of the new lock object is used to release the lock when it is no longer required.

Multithreaded Priority Queue

The Queue module allows you to create a new queue object that can hold a specific number of items. There are following methods to control the Queue –

get() – The get() removes and returns an item from the queue.

put() – The put adds item to a queue.

qsize() – The qsize() returns the number of items that are currently in the queue.

empty() – The empty( ) returns True if queue is empty; otherwise, False.

full() – the full() returns True if queue is full; otherwise, False.

Databases

The Python standard for database interfaces is the Python DB-API. Most Python database interfaces adhere to this standard.

Python Database API supports a wide range of database servers such as −

GadFly

mSQL

MySQL

PostgreSQL

Microsoft SQL Server 2000

Informix

Interbase

Oracle

Sybase

Databases

- The DB API provides a minimal standard for working with databases using Python structures and syntax wherever possible. This API includes the following –
- Importing the API module.
- Acquiring a connection with the database.
- Issuing SQL statements and stored procedures.
- Closing the connection

Databases

What is MySQLdb?

MySQLdb is an interface for connecting to a MySQL database server from Python. It implements the Python Database API v2.0 and is built on top of the MySQL C API.

How do I Install MySQLdb?

Before proceeding, you make sure you have MySQLdb installed on your machine. Just type the following in your Python script and execute it −

```
import MySQLdb
```

Creating Database Table

Once a database connection is established, we are ready to create tables or records into the database tables using execute method of the created cursor.

Example: import MySQLdb

```
# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )
# prepare a cursor object using cursor() method
cursor = db.cursor()
# Drop table if it already exist using execute() method.
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")
# Create table as per requirement
sql = """CREATE TABLE EMPLOYEE (
         FIRST_NAME  CHAR(20) NOT NULL,
         LAST_NAME  CHAR(20),
         AGE INT,
         SEX CHAR(1),
         INCOME FLOAT )"""
cursor.execute(sql)
# disconnect from server
db.close()
```

Database Operations

- INSERT Operation
- It is required when you want to create your records into a database table.

READ Operation

READ Operation on any database means to fetch some useful information from the database.

Once our database connection is established, you are ready to make a query into this database. You can use either fetchone() method to fetch single record or fetchall() method to fetech multiple values from a database table.

fetchone() − It fetches the next row of a query result set. A result set is an object that is returned when a cursor object is used to query a table.

fetchall() − It fetches all the rows in a result set. If some rows have already been extracted from the result set, then it retrieves the remaining rows from the result set.

rowcount − This is a read-only attribute and returns the number of rows that were affected by an execute() method.

Database Operations

Update Operation

UPDATE Operation on any database means to update one or more records, which are already available in the database.

DELETE Operation

DELETE operation is required when you want to delete some records from your database.

Performing Transactions

Transactions are a mechanism that ensures data consistency. Transactions have the following four properties –

Atomicity − Either a transaction completes or nothing happens at all.

Consistency − A transaction must start in a consistent state and leave the system in a consistent state.

Isolation − Intermediate results of a transaction are not visible outside the current transaction.

Durability − Once a transaction was committed, the effects are persistent, even after a system failure.

The Python DB API 2.0 provides two methods to either commit or rollback a transaction.

Database Operations

COMMIT Operation

Commit is the operation, which gives a green signal to database to finalize the changes, and after this operation, no change can be reverted.

Here is a simple example to call commit method.

```
db.commit()
```

ROLLBACK Operation

If you are not satisfied with one or more of the changes and you want to revert back those changes completely, then use rollback() method.

Here is a simple example to call rollback() method.

```
db.rollback()
```

Disconnecting Database

To disconnect Database connection, use close() method.

```
db.close()
```

If the connection to a database is closed by the user with the close() method, any outstanding transactions are rolled back by the DB. However, instead of depending on any of DB lower level implementation details, your application would be better off calling commit or rollback explicitly.

QUESTIONS

Thank You