# PYTHON
# Day 3

# Day 3 - Agenda

- OO Concepts
- Standard Library

# OOP concepts

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

# 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.

# Sample Class

- **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)
```

# Creating object and accessing members

- **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
```

# Creating object and accessing members

- 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.
```

# Creating object and accessing members

- 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)
```

# Creating object and accessing members

- 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

```
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’
```

# Class Inheritance

```
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
.....
```

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

# 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+v3
```

# Data Hiding

- **Data Hiding:**
- An object's attributes may or may not be visible outside the class definition.
- For these cases, you can name attributes with a double underscore prefix, and those attributes will not be directly visible to outsiders.(Private variables)
- **Example:**

```
class JustCounter:
        __secretCount = 0
        def count(self):
                self.__secretCount += 1
                print self.__secretCount

counter = JustCounter()
counter.count()
counter.count()
print counter.__secretCount
```

**OUTPUT**

```
1
2
Traceback (most recent call last): File "test.py", line 12, in <module> print
    counter.__secretCount AttributeError: JustCounter instance has no attribute '__secretCount'
```

# Data Hiding

- 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 would work for you:

- ..... print counter._JustCounter__secretCount

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

1
2
2

# 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.

# Standard Python Library

# Python Library

- **Python is packaged with a large library of standard modules**
  - String processing
  - Operating system interfaces
  - Networking
  - Threads
  - GUI
  - Database
  - Language services
  - Security.
- **And there are many third party modules**
  - XML
  - Numeric Processing
  - Plotting/Graphics
  - etc.
- **All of these are accessed using ’import’**
  - import string
  - ...
  - a = string.split(x)

# Object Serialization

- **Motivation**
- Sometimes you need to save an object to disk and restore it later.
- Or maybe you need to ship it across the network.

- **Problem**
- Manual implementation requires a lot of work.
- Must come up with some kind of encoding scheme.
- Must write code to marshal objects to and from the encoding.

- **Fortunately...**
- Python provides several modules to do all of this for you

# The pickle and cPickle Module

**The pickle and cPickle modules serialize objects to and from files**

- To serialize, you 'pickle' an object
  - import pickle
  - file=open("object.txt","w")
  - p = pickle.Pickler(file) # file is an open file object
  - p.dump(obj) # Dump object
- To unserialize, you 'unpickle' an object
  - p = pickle.Unpickler(file) # file is an open file
  - obj = p.load() # Load object

**Notes**

- Most built-in types can be pickled except for files, sockets, execution frames, etc...
- The data-encoding is Python-specific.
- Any file-like object that provides write(),read(), and readline() methods can be used as a file.
- Recursive objects are correctly handled.
- cPickle is like pickle, but written in C and is substantially faster.
- pickle can be subclassed, cPickle can not.

# The marshal Module

**The marshal module can also be used for serialization**

- To serialize
  - import marshal
  - marshal.dump(obj,file) # Write obj to file
- To unserialize
  - obj = marshal.load(file)

**Notes**

- marshal is similiar to pickle, but is intended only for simple objects
- Can’t handle recursion or class instances.
- On the plus side, it’s pretty fast if you just want to save simple objects to a file.
- Data is stored in a binary architecture independent format

# OS Interface

```
>>> import os
>>> os.getcwd() # Return the current working directory
'C:\\Python31'
>>> os.chdir('/server/accesslogs') # Change current working directory
>>> os.system('mkdir today') # Run the command mkdir in the system shell
0
```

- Be sure to use the import os style instead of from os import *. This will keep os.open() from shadowing the built-in open() function which operates much differently.

- The built-in dir() and help() functions are useful as interactive aids for working with large modules like os:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

# OS Interface

For daily file and directory management tasks, the shutil module provides a higher level interface that is easier to use:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
>>> shutil.move('/build/executables', 'installdir')
```

# Command Line Arguments

- Command line arguments are stored in the sys module's argv attribute as a list.

- For instance the following output results from running
  python demo.py one two three
  at the command line:

  **>>> import sys**
  **>>> print(sys.argv)**
  ['demo.py', 'one', 'two', 'three']

# math module

- The math module gives access to the underlying C library functions for floating point math:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

- The random module provides tools for making random selections:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10) # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random() # random float
0.17970987693706186
>>> random.randrange(6) # random integer chosen from range(6)
4
```

# Internet Access

- There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are urllib.request for retrieving data from URLs and smtplib for sending mail:

```
>>> from urllib.request import urlopen
>>> for line in urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
... line = line.decode('utf-8') # Decoding the binary data to text.
... if 'EST' in line or 'EDT' in line: # look for Eastern Time
... print(line)
<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Have a good day.
... """)
>>> server.quit()
```

# Sending Email

- Simple Mail Transfer Protocol (SMTP) is a protocol, which handles sending e-mail and routing e-mail between mail servers.

- Python provides **smtplib** module, which defines an SMTP client session object that can be used to send mail to any Internet machine with an SMTP or ESMTP listener daemon.

- import smtplib
- smtpObj = smtplib.SMTP( [host [, port [, local_hostname]]] )

- **host:** This is the host running your SMTP server. You can specify IP address of the host or a domain name. This is optional argument.
- **port:** If you are providing *host* argument, then you need to specify a port, where SMTP server is listening. Usually this port would be 25.
- **local_hostname**: If your SMTP server is running on your local machine, then you can specify just *localhost* as of this option.

- An SMTP object has an instance method called **sendmail**, which will typically be used to do the work of mailing a message. It takes three parameters:
- The *sender* - A string with the address of the sender.
- The *receivers* - A list of strings, one for each recipient.
- The *message* - A message as a string formatted as specified in the various RFCs.

# Sending Email

```
>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Have a good day.
... """)
>>> server.quit()
```

# Data Compression

- Common data archiving and compression formats are directly supported by modules including: zlib, gzip, bz2, zipfile and tarfile.

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

# Performance Measurement

- To know the relative performance of different approaches to the same problem, Python provides a measurement tool that answers those questions immediately.
- For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments.
- The timeit module quickly demonstrates a modest performance advantage:
  **>>> from timeit import Timer**
  **>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()**
  0.57535828626024577
  **>>> Timer('a,b = b,a', 'a=1; b=2').timeit()**
  0.54962537085770791
- In contrast to timeit's fine level of granularity, the profile and pstats modules provide tools for identifying time critical sections in larger blocks of code.

# Quality Control

- The doctest module provides a tool for scanning a module and validating tests embedded in a program's docstrings.

- Test construction is as simple as cutting-and-pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the **doctest** module to make sure the code remains true to the documentation:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.
    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod() # automatically validate the embedded tests
```

# Quality Control

- The unittest module is allows a more comprehensive set of tests to be maintained in a separate file:

```
import unittest
class TestStatisticalFunctions(unittest.TestCase):
    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        self.assertRaises(ZeroDivisionError, average, [])
        self.assertRaises(TypeError, average, 20, 30, 70)
        unittest.main() # Calling from the command line invokes all tests
```

# Multi-threading

- Threading is a technique for decoupling tasks which are not sequentially dependent.
- Threads can be used to improve the responsiveness of applications that accept user input while other tasks run in the background.
- A related use case is running I/O in parallel with computations in another thread.

# Multi-threading

- The following code shows how the high level threading module can run tasks in background while the main program continues to run:

```
import threading, zipfile
class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
            super().__init__(self)
            self.infile = infile
            self.outfile = outfile
    def run(self):
            f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
            f.write(self.infile)
            f.close()
            print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')
background.join() # Wait for the background task to finish
print('Main program waited until background was done.')
```

# Database access

- 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:
  - GadFly
  - mSQL
  - MySQL
  - PostgreSQL
  - Microsoft SQL Server 2000
  - Informix
  - Interbase
  - Oracle
  - Sybase

- 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

# Database access

- **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.

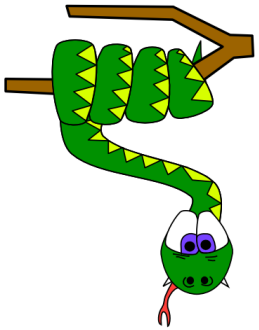
```
import MySQLdb
# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

# execute SQL query using execute() method.
cursor.execute("SELECT VERSION()")

# Fetch a single row using fetchone() method.
data = cursor.fetchone()

print "Database version : %s " % data # disconnect from server
db.close()
```




# Packages

# Packages

- Collection of modules in directory
- Must have __init__.py file
- May contain subpackages
- Import syntax:
  - from P.Q.M import foo; print foo()
  - from P.Q import M; print M.foo()
  - import P.Q.M; print P.Q.M.foo()
  - import P.Q.M as M; print M.foo()  # new

# Summary

- The following topics are covered so far
  - Classes and Objects
  - Standard Library

# Thank you