PYTHON
Part 2

Session 2 - Agenda

- Functions
- Modules
- File Management
- Exception Handling
- OO Concepts

Functions

What is a function?

- A function is a group of statements which are reusable and is used to perform a task.

- Function blocks begin with the keyword def followed by the function name and parentheses (( ))

- Any input parameters or arguments should be placed within these parentheses.

- The first statement of a function can be an optional statement - the documentation string of the function or docstring.

- The code block within every function starts with a colon (:) and is indented.

- Functions may return a value to the caller, using the 'return‘ keyword.

Syntax:

```
def functionname( parameters ):
   “””function_docstring“”””
   function_suite
   return [expression]
```

Example

```
def printme( str ):
   '''This prints a passed string into this function'''
   print(str)
   return
```

Calling a Function

- Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.
- Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt.

```
# Function definition is here
def printme( str ):
    print(str)
    return

# Now you can call printme function
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

Keyword arguments

```
# function definition

 def add ( x, y):
       result=x+y
       return result

# function invocation using keywords

print(add(x=10,y=20))
print(add(y=10, x = 10))
```

Default arguments

```
  # function definition

def divide ( a, b = 1):
        result=a/b
        return result

# function invocation

print(divide(10,5))
print(divide(10))
```

Pass by reference vs value

- All parameters are passed by reference. It means if you change the parameter within a function, the change also reflects back in the calling function.

```
# Function definition
def changeme( mylist ):
    mylist.append([1,2,3,4])
    print ("Values inside the function: ", mylist)
    return

# Now you can call changeme function
mylist = [10,20,30]
changeme( mylist )
print("Values outside the function: ", mylist)
```

Variable-length arguments

- User may need to process a function for more arguments than specified while defining the function. They are called variable-length arguments

- An asterisk (*) is placed before the variable name that holds the values of all non-keyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call.

```
# function definition

def  print_student_marks( name,  *student_marks):
    print(name)
    for marks in student_marks:
            print(marks)

# function invocation

print_student_marks(‘xyz’, 90,80,70)
print_student_marks(‘abc’, 90, 88)
```

The *Anonymous* Functions

- You can use the *lambda* keyword to create small anonymous functions. These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword.
- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression.
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to *inline* statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

The *Anonymous* Functions

- Syntax:
- lambda [arg1 [,arg2,.....argn]]:expression
- Example:

```
# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2



# Now you can call sum as a function
print("Value of total : ", sum( 10, 20 ))
print("Value of total : ", sum( 20, 20 ))
```

Scope of Variables

- A namespace is a dictionary of variable names and their corresponding objects. Scope refers to a namespace.

a. Local variables - Variables defined in a function are local and cannot be accessed outside it
b. Global variables - Variables defined outside all functions are global

- Therefore same names can be used for local and global variables. Local variable shadows global variable.

```
a = 99                    # Global variable
def func():
    a= 88                 # Local variable hides global variable
    print(a)              # Prints 88
func()
print(a)                  # Prints 99
```

Global vs. Local variables:

- To access a global variable inside a function, use the statement global VarName.

```
total = 0                    # This is global variable.
def sum( arg1, arg2 ):
        global total          # Without this, the next expression will lead to
                              # UnboundLocalError as we may try to access the value
                              # of a local variable before setting it.

        total = arg1 + arg2
        print("Inside the function local total : ", total)

# Now you can call sum function
sum(10, 20)
print("Outside the function global total : ", total )
```

Modules

Modules

- A module is a Python object with arbitrarily named attributes that user can bind and reference.

- A module allows to logically organize the Python code. Grouping related code into a module makes the code easier to understand and use. Simply, a module is a file consisting of Python code. A module can define functions, classes, and variables. A module can also include runnable code

Example:

- The Python code for a module named *aname* normally resides in a file named *aname.py*. Here's an example of a simple module, support.py

```
def print_func( par ):
   print("Hello : ", par)
   return
```

Creating a module

```
# calculator.py

def add(x,y):
    return x+y

def multiply(x,y):
    return x*y
```

Importing a module

- User can use any Python source file as a module by executing an import statement in some other Python source file.

```
import module1[, module2[,... moduleN]
```

Example

```
# sample.py
import calculator
print(calculator.add(10,10))
```

- A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur.

The *from...import* Statement

- Python's *from* statement lets you import specific attributes from a module into the current namespace.

```
from modname import name1[, name2[, ... nameN]]
```

- For example, to import the function fibonacci from the module fib, use the following statement:

```
from fib import fibonacci
```

- This statement does not import the entire module fib into the current namespace; it just introduces the item fibonacci from the module fib into the global symbol table of the importing module.

Locating Modules

- When you import a module, the Python interpreter searches for the module in the following sequences:
  - The current directory.
  - If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
  - If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.

Locating Modules

- The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH.
- Here is a typical PYTHONPATH from a Windows system:
    - set PYTHONPATH=c:\python20\lib;
- And here is a typical PYTHONPATH from a UNIX system:
    - set PYTHONPATH=/usr/local/lib/python

The dir( ) Function:

- The dir() built-in function returns a sorted list of strings containing the names defined by a module.

- The list contains the names of all the modules, variables and functions that are defined in a module.

```
# Import built-in module math
import math

content = dir(math)

print(content)
```

File Management

Files

- Until now, you have been reading and writing to the standard input and output. Now, we will see how to play with actual data files.
- Python provides basic functions and methods necessary to manipulate files by default.
- You can do your most of the file manipulation using a **file** object.

The *open* Function:

- Before you can read or write a file, you have to open it using Python's built-in *open()* function. This function creates a **file** object, which would be utilized to call other support methods associated with it.

```
file object = open(file_name [, access_mode][, buffering])
```

- **file_name:** Name of the file that user wants to access.

- **access_mode:** Mode in which the file has to be opened, such as, read, write, append, and so on. This is an optional parameter and the default file access mode is read (r)

- **buffering:** If the buffering value is set to 0, no buffering will take place. If the buffering value is 1, line buffering will be performed while accessing a file. If the buffering value is specified as an integer greater than 1, then buffering action will be performed with the indicated buffer size. If negative, the buffer size is the system default.

File modes

Modes	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
rb	Reading in binary format.
r+	Opens a file for both reading and writing.
rb+	Opens a file for both reading and writing in binary format.
w	Opens a file for writing only. Overwrites the file if it exists. If the file does not exist, creates a new file for writing.
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
ab	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
ab+	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

The *file* object attributes:

- Once a file is opened and you have one *file* object, you can get various information related to that file.
- Here is a list of all attributes related to file object:

Attribute	**Description**
file.closed	Returns true if file is closed, false otherwise.
file.mode	Returns access mode with which file was opened.
file.name	Returns name of the file.
file.softspace	Returns false if space explicitly required with print, true otherwise.

The *close()* Method:

- The close() method of a *file* object flushes any unwritten information and closes the file object, after which no more writing can be done.
- Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.

```
# Open a file
fo = open("foo.txt", "wb")
print("Name of the file: ", fo.name)

# Close opend file
fo.close()
```

The *write()* Method:

- The *write()* method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.

- The write() method does not add a newline character ('\n') to the end of the string
:

Syntax:

- fileObject.write(string)

```
# Open a file
fo = open("/tmp/foo.txt", "wb")
fo.write( "Python is a great language.\nYeah its great!!\n")

# Close opened file
fo.close()
```

The *read()* Method:

- The *read()* method reads a string from an open file. It is important to note that Python strings can have binary data and not just text.

- **Syntax:**

  fileObject.read([count])

- Here, passed parameter is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if *count* is missing, then it tries to read as much as possible, maybe until the end of file.

```
# Open a file
fo = open("/tmp/foo.txt", "r+")
str = fo.read(10)
print "Read String is : ", str
# Close opend file
fo.close()
```

Files: Input

inflobj = open(‘data’, ‘r’)	Open the file ‘data’ for input
S = inflobj.read()	Read whole file into one String
S = inflobj.read(N)	Reads N bytes (N >= 1)
L = inflobj.readlines()	Returns a list of line strings

Files: Output

outflobj = open(‘data’, ‘w’)	Open the file ‘data’ for writing
outflobj.write(S)	Writes the string S to file
outflobj.writelines(L)	Writes each of the strings in list L to file
outflobj.close()	Closes the file

Reading files

```
>>> f = open(“names.txt")
>>> f.readline()
'Yaqin\n'
```

File Output

```
input_file = open(“in.txt")
output_file = open(“out.txt", "w")
for line in input_file:
        output_file.write(line)
```

“w” = “write mode”
“a” = “append mode”
“wb” = “write in binary”
“r” = “read mode” (default)
“rb” = “read in binary”
“U” = “read files with Unix or Windows line endings”

Quick Way

```
>>> lst= [ x for x in open("text.txt","r").readlines() ]
>>> lst
['Chen Lin\n', 'clin@brandeis.edu\n', 'Volen 110\n', 'Office
   Hour: Thurs. 3-5\n', '\n', 'Yaqin Yang\n',
   'yaqin@brandeis.edu\n', 'Volen 110\n', 'Offiche Hour: Tues.
   3-5\n']
```

Renaming and Deleting Files:

- Python **os** module provides methods that help you perform file-processing operations, such as renaming and deleting files.
- To use this module you need to import it first and then you can call any related functions.
- **The rename() Method:**
- The *rename()* method takes two arguments, the current filename and the new filename.
- **Syntax:**

```
os.rename(current_file_name, new_file_name)
```

The *remove()* Method:

- You can use the *remove()* method to delete files by supplying the name of the file to be deleted as the argument.

- **Syntax:**

```
        os.remove(file_name)
```

- **Example:**
- Following is the example to delete an existing file *test2.txt*:

```
import os
# Delete file test2.txt
 os.remove("text2.txt")
```

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.
   ......................
```

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.
   ......................
```

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))
except ValueError:
        print("Could not convert data to an integer.")
except:
          print("Unexpected error:", sys.exc_info()[0])
```

Object Oriented Programming

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  # Class variable

   def __init__(self, name, salary):  #constructor
      self.name = name   #self.name – instance variable
      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:

```
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
```

Summary

- The following topics are covered so far
  - Functions
  - Modules
  - Local and Global variables
  - File Management
  - Exception Handling
  - OO Concepts

Thank you