Agenda

- Functions
- Modules
- File Management

Functions

What is a function?

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

Defining a Function

- 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. You can also define parameters inside 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.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

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 ):
   "This prints a passed string into this function"
   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 (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.

```
# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   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

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

An asterisk (*) is placed before the variable name that will hold the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example:

```
# 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 *return* Statement:

- Syntax:

```
# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2
   print "Inside the function : ", total
   return total;

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

Scope of Variables:

- All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.
- The scope of a variable determines the portion of the program where you can access a particular identifier.
- There are two basic scopes of variables in Python:
  - Global variables
  - Local variables

Global vs. Local variables:

- Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.
- This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions.
- When you call a function, the variables declared inside it are brought into scope.

Global vs. Local variables:

```
#!/usr/bin/python

total = 0; # This is global variable.
# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   global total
   total = arg1 + arg2; # Here total is local variable.
   print "Inside the function local total : ", total
   #return total;

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

Modules

Modules

- A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use.
- A module is a Python object with arbitrarily named attributes that you can bind and reference.
- 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

- You can use any Python source file as a module by executing an import statement in some other Python source file. The *import* has the following syntax:

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

#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.
- The *from...import* has the following syntax:
- 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 Handling

Opening and Closing 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.

- **Syntax:**
- file object = open(file_name [, access_mode][, buffering]) Here is paramters' detail:

- **file_name:** The file_name argument is a string value that contains the name of the file that you want to access.
- **access_mode:** The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is 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 you specify the buffering value 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(default behavior).

The *open* Function:

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	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Opens a file for both reading and writing. The file pointer will be at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer will be at the beginning of the file.
w	Opens a file for writing only. Overwrites the file if the file 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.

The *open* Function:

Modes	Description
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.

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 opend 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

File Positions:

- The *tell()* method tells you the current position within the file; in other words, the next read or write will occur at that many bytes from the beginning of the file.
- The *seek(offset[, from])* method changes the current file position. The *offset* argument indicates the number of bytes to be moved.
- The *from* argument specifies the reference position from where the bytes are to be moved.
- If *from* is set to 0, it means use the beginning of the file as the reference position and 1 means use the current position as the reference position and if it is set to 2 then the end of the file would be taken as the reference position.

File Positions:

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

# Check current position
position = fo.tell();
print "Current file position : ", position

# Reposition pointer at the beginning once again
position = fo.seek(0, 0);
str = fo.read(10);
print "Again read String is : ", str
# Close opend file
fo.close()
```

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*:

```
#!/usr/bin/python
import os
# Delete file test2.txt
 os.remove("text2.txt")
```

Summary

- The following topics are covered so far
  - Functions
  - Modules
  - File Management

Thank you