Python Programming

Suryanarayana Chandrasekhar

Draft Version - 1

Preface

Draft Copy Version - 1

Table of Contents

1. **Introduction to Scripting and Python**
    a. Introduction to Scripting
    b. Timeline of scripting languages
    c. Introduction to Python
    d. History of Python
    e. Features of Python
    f. Benefits of Python
    g. Python Pre-requisites
    h. Environment Setup for Python
2. **Language Fundamentals**
    a. Language Fundamentals
    b. Basic Datatypes
    c. Variables and Program Execution
    d. Operators
    e. Data Structure Objects
3. **I / O Statements**
    a. Introduction
    b. Inputting statements
    c. Outputting statements
4. **Control Statements**
    a. Introduction
    b. Conditional Control Statements
    c. Loop Control Statements
5. **Functions**
    a. Introduction
    b. Built-In Functions
    c. User Defined Functions
    d. Anonymous Functions
6. **Modules**
    a. Introduction
    b. Built-In Modules
    c. User Defined Modules
    d. Setting up Module Search Path

7. **Packages**
    a. Introduction
    b. Built-In Packages
    c. User Defined Packages
    d. Setting up Package Search Path
8. **Exception Handling**
    a. Introduction
    b. Built-in Exceptions
    c. Exception Handling
    d. Default Exceptions
    e. Program closure statements
9. **Command Line Arguments**
    a. Introduction
    b. Command Line Arguments
    c. Type casting
10. **File Handling**
    a. Introduction
    b. File handling operations
    c. File handling modules

Draft Copy

Chapter 1 – Introduction to Scripting and Python

Introduction to Scripting

A scripting language or script language is a type of programming language that supports scripts, programs written for a special run-time environment that can interpret (rather than compile) and automate the execution of tasks that could alternatively be executed sequential line by line. These days, the line between a scripting language and a programming language is mostly blurred. As such, in business applications, the differences may be meaningless. Some of the traditional way to differentiate can be based on its execution styles; Where high level programming languages are mostly compiled and scripting languages are mostly interpreted by execution style.

Compiled Languages

Computers do not actually understand the code we written in high level language, we need to translate human-readable code to machine-readable code. Program source code need to be translated into bits' representation which can be executed by computers. This translation process is called compilation. Languages that require compilation are called compiled languages. Eg. C, C++ and Java are compiled languages.

Scripting Languages

Scripting languages are those that not require any pre-compilation process of source code and mostly execute the instructions on the execution. Eg. Shell Script, Python, Perl, PHP and Ruby are of the initial types scripting languages. Scripting languages don't

Draft Copy Version - 1

have compiler or a compilation process, these languages use interpreters to translate program source-code to machine executable code on the fly.

Typically, a scripting language can be characterized by the following properties:

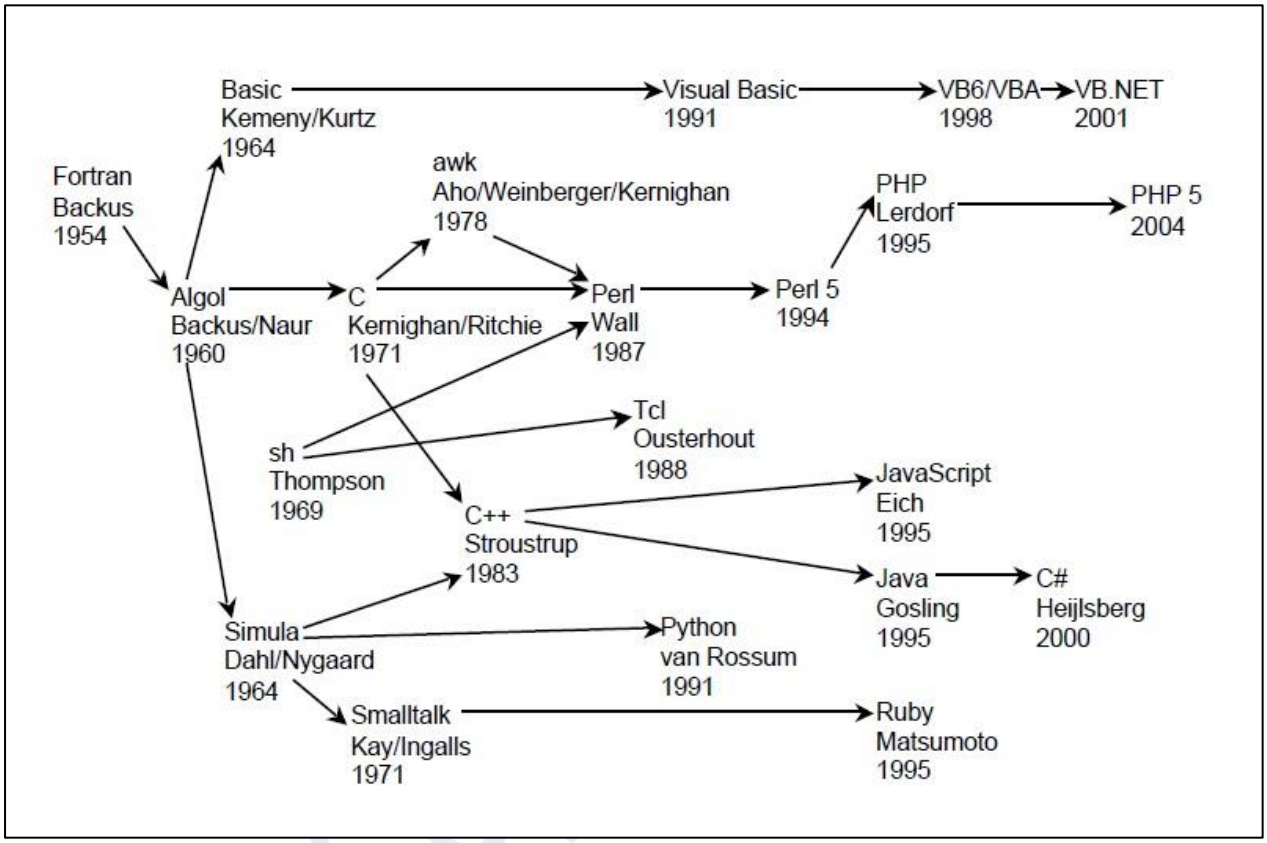
- **Ease of use:** Scripting languages are intended to be very fast to pick up and author programs in. This generally implies relatively simple syntax and semantics.
- **OS facilities:** Especially file system and related, built in with easy interfaces. Scripting is usually aimed at desktops, limiting the portability needs of the pre-built libraries.
- **Interpreted from source code** - to give the fastest turnaround from script to execution. On a desktop, the performance of even a slow interpreter is often non-problematic. In comparison, non-scripting languages intended for large programs are often precompiled in at least some sense for superior performance.
- **Relatively loose structure**: It would be difficult to use HLL as a scripting language due to the rules about which language exist, where it's possible to customize / define simple ways to achieve intended purpose.

Timeline of scripting language

- Early mainframe computers (in the 1950s) were non-interactive, instead using batch processing. IBM's **Job Control Language (JCL)** is the archetype of languages used to control batch processing.

Draft Copy Version - 1

- The first interactive shells were developed in the 1960s to enable remote operation of the first time-sharing systems, and these used **Shell Scripts**, which controlled running computer programs within a computer program, the shell.



- Michael Cowlishaw initially conceived the **Rexx language** in 1979 to simplify programming tasks on IBM's timesharing system. It became popular as a macro language for arbitrary application programs, and its usage spread to many other platforms, including PCs and Unix.
- Larry Wall in the late 1980s developed **PERL** as a way to bring together in one place the functions of many popular Unix text processing applications such as **sh**, **sed**, and **awk**, **PERL** quickly became a favorite tool of system administrators.

- John Ousterhout in the late 1980s developed **TCL** as an embeddable command language for interactive tools.
- **Python** is a dynamic object-oriented language, created by Guido van Rossum in the early 1990s to bridge the gap between shell and C programming. Its elegant, easy to learn syntax, high level data types, elaborate library, portability, and ease of extending and embedding in C/C+ + all contribute to its popularity.
- **JavaScript** was Created in the mid-1990s by Netscape Corporation to perform scripting functions in Web browsers, such as simple form validation. JavaScript has become the de-facto standard for client-side Web scripting, although it doesn't have much to do with Java
- Microsoft **VB Scripting** became popular in the late 1990s designed as a "lightweight" language with a fast interpreter for use in a wide variety of Microsoft environments

My sincere **apologies** to the authors and advocates of those many other important scripting languages, which may **not** have been mentioned here.

Translating source programs on the fly may affect the application's performance hence scripting languages may have some performance challenges. Typically, pre-compiled code comparatively runs faster than on the fly translation. But, good news is, there are mechanisms to speed up scripted languages, including techniques such as code caching and persistent interpreters, which is taken care sometimes internally by the language himself.

Draft Copy Version - 1

The vendor website refers Python as a programming language in the given definition "Python is a programming language that lets you work quickly and integrate systems more effectively." (http://www.python.org/), so obviously they consider it to be a programming language.

Introduction to Python

Python is a great language to learn. It is an ideal language for those, who are new to programming. After going through this self-paced documents, you should be confident to continue understand and doing learning with Python. You can create programs and applications using Python.

What is Python?

"**Python is a programming language that lets you work quickly and integrate systems more effectively**". – python.org –

"Python is a general purpose high-level, interpreted, interactive and high supportive of object oriented programming language".

Python was created by a Guido van Rossum in late 80's to be a highly readable language which uses english like representations and has fewer syntactical constructions than other programming languages. Python is an open source programming language, that was made to look good and easy to use. The language is named after his favorite television show Monty Python's Flying Circus.

Draft Copy Version -1

- **Python is Interpreted:** This means in python everything is processed at runtime by the interpreter and you do not need to compile your program before executing it.
- **Python is Interactive:** This means you can sit inside a python prompt and interact with the interpreter directly to write your code / implementations for understanding or get quick help.
- **Python is Object-Oriented:** This means; python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is Beginner's Language:** Python is a great language for the beginner programmers & supports the development of a wide range of applications, from simple text processing to web applications.

Python is a general-purpose, dynamic, object-oriented programming language. The design purpose of the Python language emphasizes programmer productivity and code readability. It was first officially released in 1991. Python was inspired by many other programming languages and has those features which might have been found in the languages like ABC, LISP, Haskell, Java and Perl programming languages. Python is a high level, general purpose, multiplatform, interpreted language. One of its most visible features is minimizes use semicolons and parentheses. Python uses indentation instead for better code representation.

History of Python

- Python was developed by **Guido van Rossum** in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

Draft Copy Version-1

- Python has been derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, Unix Shell and other scripting languages.
- Python is copyrighted under the GNU General Public License (GPL), Python source code is now available under the GNU General Public License (GPL).
- Though all Open sources are taken care by MIT, Python is still maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.

Python supports several programming styles. It does not force a programmer to a specific paradigm though has its own recommendations or style guide in PEP documentation. It supports object oriented and procedural programming. There is also a limited support for functional programming.

Features of Python

- **Easy-to-learn:** Python has relatively few keywords, simple structure, and a clearly defined syntax. This allows the user to understand the language in a relatively short period of time
- **Easy-to-read:** Python code is much more clearly defined and visible
- **Easy-to-maintain:** Python's success is that its source code is fairly easy-to-maintain
- **A broad standard library:** One of Python's greatest strengths is the bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and other OS

Draft Copy Version - 1

- **Interactive Mode:** Support for an interactive mode in which you can enter results from a terminal right to the language, allowing interactive testing and debugging of snippets of code
- **Portable:** Python can run on a wide variety of hardware platforms and OS which has the same interface on all platforms.
- **Extendable:** You can add built in and third party modules to the python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases:** Python provides interfaces to all major commercial databases.
- **GUI Programming:** Python supports GUI applications that can be created and ported to many system calls, libraries, and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable:** Python provides a better structure and support for large programs than any other scripting and highly scalable.

Python is much better suited as a "**glue**" language wherein, some of the key benefits are listed below which could be also a key highlight;

Benefits of Python

- Open Source
- Software Testing & Automation
- Desktop Development
- Integration
- Network Intensive Applications
- Web Developments

Draft Copy Version -1

- Prototyping
- Numeric and Scientific Applications
- Application Scripting

Python pre-requisites

Though there are no hard thumb rules to say as python pre-requisites, the following would be good choice for any user to have as pre-requisite which makes learning better, Pre-requisites can be looked with;

1. **Learning pre-requisite**
2. **Programming pre-requisite**

Learning pre-requisite will help user to have better conceptual understanding and quick implementations understanding in python for real time scenarios. The following are learning pre-requisites;

1. Any programming language (at-least C Programming from academics)
2. Understanding of Object Oriented Programming concepts
3. Understanding of Unix or any of Unix type environment
4. Shell scripting would be an added advantage

Programming pre-requisite will help to write and execute the python programs. The following are programming pre-requisite;

1. **Python Software** can be downloaded based on specific platform and user requirement from vendor site at https://www.python.org/downloads/

Draft Copy Version -1

2. **Text editor** is good to start writing python programs and every Operating System provides a default text editor. Eg. Notepad in Windows Environment / vi editor in Unix environment
3. **IDE** short for **Integrated Development Environment** is a tool having many features that makes programmer life easier. Python provides a default IDE called **IDLE**. There are many Opensource and other Commercial IDE's; of which **PyScripter** is one Opensource IDE available at the given URL code.google.com/p/pyscripter, and also a generic IDE called **Notepad++** which can be downloaded from www.notepad-plus-plus.org. Eclipse IDE has also support for Python programming environment using **PyDev** Plugin.

Disclaimer: **The mentioned suggestions in any part of documents are suggestive and subjective based on personal experience**. Hence every individual should try, test and experiment before use. "**Please take the necessary approvals before the software installation on official laptop**"

Environment setup for Python

Python software would be mostly available on Unix and Unix type of Operating System. If the software is not available, it can be downloaded and installed from the vendor site **python.org**.

Installation of Python on Windows: Visit https://www.python.org/downloads/ and download the latest version. At the time of preparing this document, it was **Python 2.7.11** with 2.x version and **Python 3.5.1** with python 3.x version. The installation steps would be as simple like any other Windows-based software.

Draft Copy Version - 1

Note: Ensure to enable the check option of add Python 3.x / 2.x to PATH during the installation for adding the environment path setting of python. One stop URL for all your Python needs http://python.org/

Step 1: Download the Python Software from the following link and Installation Guide can be found at;

a. https://docs.python.org/2/using/windows.html

b. https://docs.python.org/3/using/windows.html

c. http://docs.python-guide.org/en/latest/starting/install/win/

Software can be obtained from;

a. https://www.python.org/downloads/

b. Setup of IDE (**Optional**)

c. https://www.ics.uci.edu/~pattis/common/handouts/introtopythonineclipse/

Step 2: Identify the appropriate version based on your business requirement, recommended would be **Python 2.x** version for this assignment and demos demonstrated throughout this document;

Step 3: Execute the downloaded software, follow through click and go to install Python with default settings / parameters.

a. To change install location, click on Customize installation, then Next and enter C:\python35 as install location.

b. If not checked, check Add Python to environment variables. This does the same thing as Add Python 3.5 to PATH on the first install screen.

Step 4: Open Command Prompt and ensure python software is installed and accessible by issuing a command "c:\> python –V", If the command throws error as

"'python' is not recognized as an internal or external command, operable program or batch file.", now ensure the python software path is set in the environment variable. Open properties of My Computer → Advanced System Settings → Advanced Tab → Environment Variables → System Variable Section, edit **path** variable and **append** the installed python path. Close all windows by choosing OK button once setup. Once the path is applied re-open Command prompt to apply the new settings if already opened and check for correctness of the setup.

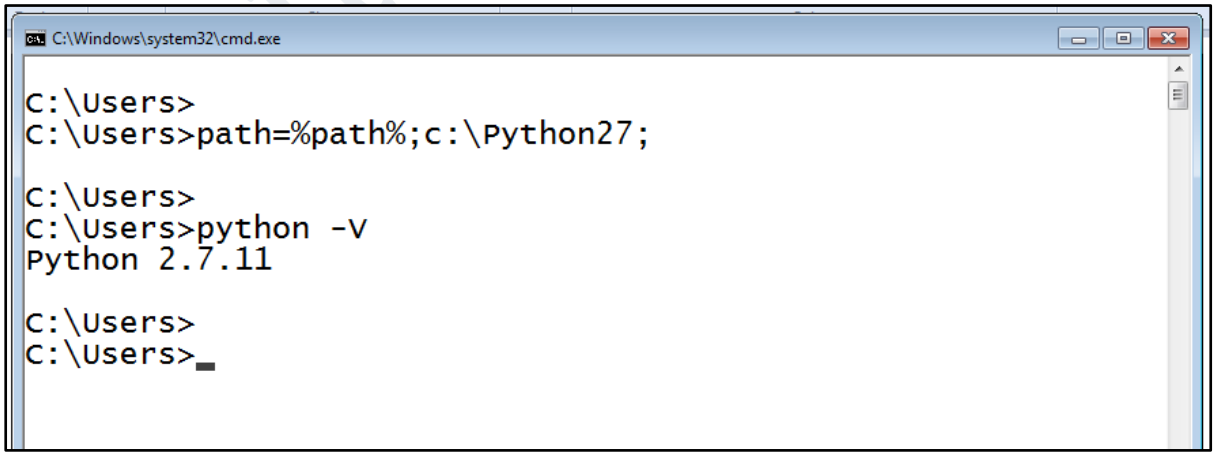
Step 5: Open a Notepad and good to go with Python Programming.

Command Line / DOS Prompt access

If you want to be able to use Python from the Windows command line i.e. the DOS prompt, then you need to set the PATH variable appropriately. Open a command prompt window and type the following at the command prompt.

path=%path%;c:\Python27;

Above command should be issued without any spaces in between the command. This would be just a temporary path setting at the command prompt valid for the current session and has to be issued every time the command prompt is launched.



For older versions of Windows, open the file C:\AUTOEXEC.BAT and add the line **PATH=%PATH%;C:\Python27** and restart the system. For Windows NT, use the AUTOEXEC.NT file.

Installation on Mac OS X

For Mac OS X users, use Homebrew: brew install python3. To verify, open the terminal by pressing [Command + Space] keys (to open Spotlight search), type Terminal and press [enter] key. Now, run python3 and ensure there are no errors.

Installation on GNU/Linux

Most likely on Unix and Unix type machines python would be readily available. This can be verified using which command. If not found then one can follow the step for installation for GNU/Linux users, use your distribution's package manager to install Python 3, e.g. on Debian & Ubuntu: sudo apt-get update && sudo apt-get install python3.

To verify, open the terminal by opening the Terminal application or by pressing Alt + F2 and entering gnome-terminal. If that doesn't work, please refer the documentation of your particular GNU/Linux distribution. Now, run python3 and ensure there are no errors.

You can see the version of Python on the screen by running:

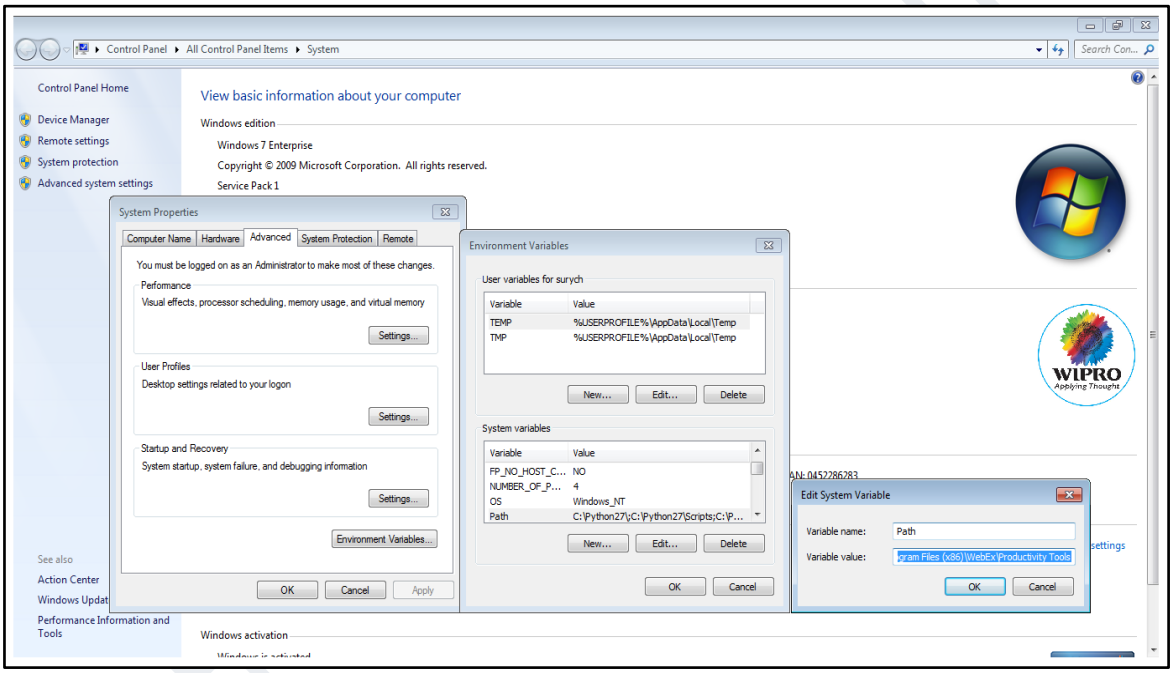
```
$ python3 -V
Python 3.5.1
```

Draft Copy Version - 1

NOTE: $ is the prompt of the shell. It will be different for you depending on the settings of the operating system on your computer, hence I will indicate the prompt by just the $ symbol.

CAUTION: Output may be different on your computer, depending on what version of Python software has been installed on your computer.

Screen shot of path setting



Try it out

1. List out various scripting languages and their abilities and write a comparison chart to define most essential components and its comparison to other scripting languages.

Draft Copy Version - 1

Chapter 2 - Python Language Fundamentals

Language Fundamentals

Python language contains many different components of the program statements which are broken down as **Python lexical structure**. Computer languages similar to human language, have their lexical structure. A source code of a Python program which contains instructions can be broken down into smallest representation called **tokens**. tokens are atomic code elements. In Python language, a token can be comments, variables, literals, operators, delimiters, and keywords. In Python, everything has a well-defined type, and these types can be handled completely at runtime in a dynamic and self-inspective fashion which is why python is known as dynamic typing programming language.

Comments: Comments are descriptive statements given by the programmer for any further better understanding of program. Comments can be categorized based on the type of information provided as;

1. **Script comments**
2. **Code Comments**

All comments in Python language are prefixed with # symbol. Script comments are those which are used to provide higher level of information about the script or the overview of the code. Script comment may further be divided as;

a. **Title of the script / code**

b. **Author**

Draft Copy Version - 1

c. **Date and Time**

d. **Version**

e. **Summary**

Code comments are those which are specific to a section / block / individual line of code to have better understanding of those. Everything that follows the # character is considered as comment and will be ignored by the python interpreter.

Variables: Variables are reserved memory locations which are used to store some intended type of value in it. In python variables are termed as **labels** which are references to the value, which are actually stored in the reserved locations. In python everything works on references only.

Literals: Literals are the values which are associated to the reserved memory locations that are stored in it. There are different types of literal classifications, which are termed as Data type.

Operators: Operators are special symbols that are intended to do some specific operations based on the values associated to it. Operators are categorized based on the type of intended functionality.

Indentation: Indentation is used to group a set of statements together as block in Python. Where in other programming languages we may use curly parentheses or some keywords as begin, end of blocks. Python uses white space as an indented block

Draft Copy Version - 1

and any un-indentation signifies the end of the current block. The Python style guide recommends using 4 white spaces as indentation level.

Delimiters: A delimiter is a sequence of one or more characters used to specify the boundary between to separate independent regions in expression or data representation. Delimiters are used in various representations of the Python language. They are used to build expressions, string literals, tuples, dictionaries, or lists.

Keywords: A keyword is a reserved word which are intended to do some specific operations. Keywords are used to perform a specific task within a computer program. A programmer should not use a pre-defined keyword for any other references or as an ordinary identifier.

and	elif	if	print
as	else	import	raise
assert	except	in	return
break	exec	is	try
class	finally	lambda	while
continue	for	not	with
def	from	or	yield
del	global	pass	

```
C:\Windows\system32\cmd.exe - python
>>>
>>> help()

Welcome to Python 2.7!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/2.7/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help> keywords

Here is a list of the Python keywords.  Enter any keyword to get more help.

and                 elif                if                  print
as                  else                import              raise
assert              except              in                  return
break               exec                is                  try
class               finally             lambda              while
continue            for                 not                 with
def                 from                or                  yield
del                 global              pass

help>
```

We can query for help() documentation within python interpreter, the above screen shot displays the list keywords available in 2.7 version of Python.

Basic Datatypes

Data types are the type specification given to the reserved memory location to store some compatible type of required values. In python for easy understanding datatypes can be categorized as;

a. **Primitive Type**

b. **Non Primitive Type**

Primitive types are those which are the simple types and store mostly unary values in it. **Non primitive** types are those which are extension of simple types to store multiple values. In Python any type of value will stored internally as an **object** type only.

Primitive types can be further broken down as;

1. **Number type**
2. **Non Number type**

Number data types can be further categorized as

a. **Integer**: This can store integer type of values which can be a signed number
b. **Floating**: This type can store floating number type of values
c. **Complex**: This type can store complex type representations numbers
d. **Long**: This can store long integers and they also can be represented in octal and hexadecimal forms

Non number data types could be

a. **Strings**: This can store set of characters or any raw text
b. **Boolean**: This can store True or False as Boolean values
c. **None**: This type of representation indicates non-existent, not known or empty values or can be treated similar to NULLs in other programming language

Non primitive types are Data structure type representation; There are many such objects with some underlying Data structure implementations, some are mentioned;

a. List
b. Tuple
c. Dictionary
d. Sets
e. Datetime

Draft Copy Version - 1

```
C:\Windows\system32\cmd.exe - python

C:\>python
Python 2.7.11 (v2.7.11:6d1b6a68f775, Dec  5 2015, 20:32:19) [MSC v.1500 32 bit (
Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 10
>>> b = 'Wipro Technologies'
>>> c = 123.45
>>> d = True
>>> e = 5+7j
>>> f = None
>>> type(a)
<type 'int'>
>>> type(b)
<type 'str'>
>>> type(c)
<type 'float'>
>>> type(d)
<type 'bool'>
>>> type(e)
<type 'complex'>
>>> type(f)
<type 'NoneType'>
>>>
```

In python the values have to be defined before it is accessed. The type specification is understood by the system based on the type of associated value.

Variables and Program Execution

Variable is a reserved memory location which is intended to store some type of value in it. There are two components of any variable;

1. **Literal**
2. **Identifier**

Literals are the values associated to any reserved location and **identifiers** are the associated names through which these stored values are accessed. Variable can be categorized by nature of its implementation as two types;

1. **Implicit Variable**
2. **Explicit Variable**

Draft Copy Version

Implicit variables are those which are defined and used within the program without any exclusive declarative statement. Mostly this type of variable is used within Scripting languages to provide a greater flexibility to storage of values. The type specification of value is declared / specified internally by the language himself based on the associated value in the reserved location. Eg. no1 = 10, here value **10** will be stored in some reserved location, which is accessed using the identifier **no1** and internally mapped to a number / integer data type as defined by language himself.

Explicit variables are those which are declared, defined and then used within the program having an exclusive declarative statement. With this style of defining, the type specification is fixed for the variable throughout its life and cannot change its type. Mostly this type of variables is used in high level programming languages. The type specification depends on the business requirement and decided by the programmer as available in the language specification. Eg. int a; a = 10 and then use a within any expression. Hence variable "**a**" will remain to store only of type integer throughout its existence.

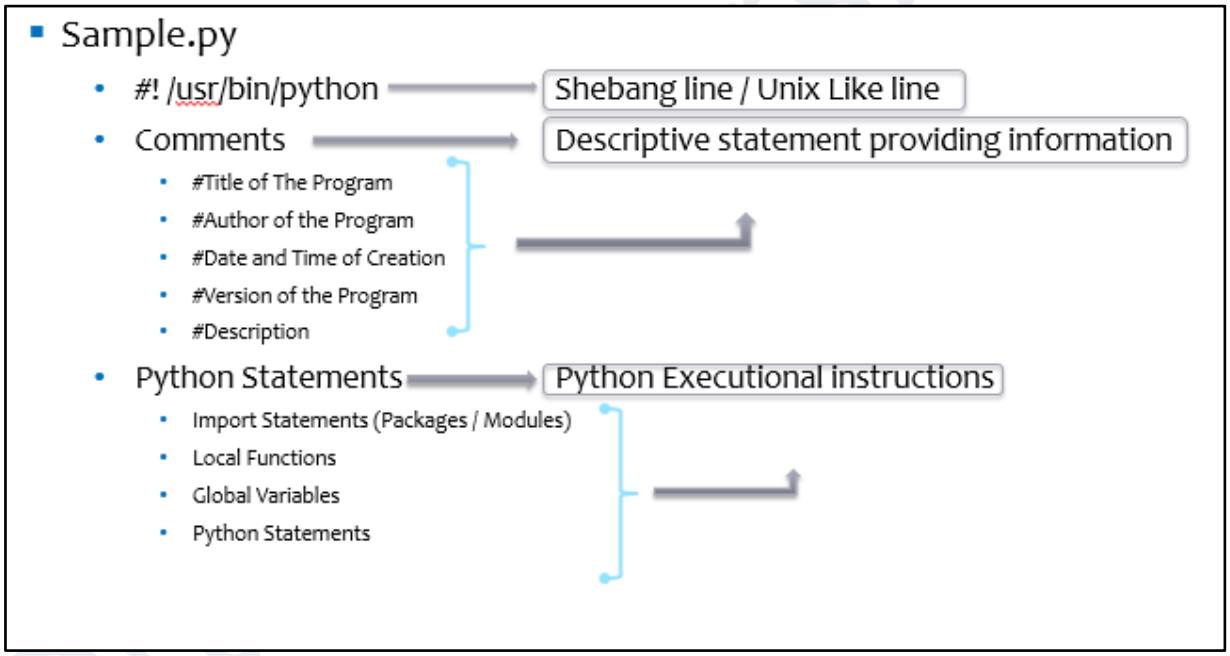
In python variables are referred as **labels** which are just reference or identifier to values stored. In python everything works on references only. Identifiers in python can be created using alphanumeric characters and also use of underscore _ character to separate multiple words. A variable name should not begin with a number and not recommended to have any keyword as names. Python language provides style guide for naming conventions and other language specifications found in the style guide of PEP documentation in python.org.

Draft Copy Version - 1

Some of the **valid identifiers** are Value, value, company_name etc., Some of the **invalid identifiers** are 12Val, exx$, first-name, some thing, etc., In python variables are case sensitive. This mean that upper case 'A' is not same as lower case 'a'.

Python Program Execution: Program execution can be done in different ways. The following are suggested way in Windows and Unix / Unix type environment. Python file should be created with any "**goodname**" having an extension "**.py**". The extension is required for the Python language himself.

Python Program Structure



Shebang line: Shebang line is Unix like line which is used only in Unix and Unix type of environment to provide the path of the executing utility. It should be very first line in the executing program and has a special symbol which is combination of **"#!"** prefixed to the path of python.

Comments: Comments are the descriptive statements which provides additional information of program here, which are further broken down as shown in screen shot.

Python Statements: These are the language statements to perform specific intended purposes. Python does not need any line delimiter unless, multiple statements are given in single line.

```
#!  /usr/bin/python
print "Welcome to Python Programming!"
print ("This is second line")
print ("This is third line")
```

Type the above program within any Text editor of your choice as available and save this program as "**sample.py".** The program is intended to display the messages which are always enclosed in either 'single quotes' or in "Double Quotes" it will be simply treated as string / raw text. To execute the above stored program, issue the command **python sample.py** If there are no challenges encountered you should be able to see the three messages printed on your standard output device.

"**print**" statement is used to display anything onto the standard output device. This topic is discussed more in detail "outputting styles".

The program shown below demonstrates various types of value defined and displayed using different styles of print statement. It supports "**C**" programming style of formatted print and pythonic style of appending. print statement provides a default newline and also supports various escape sequence operators.

Draft Copy Version - 1

```
#! /usr/bin/python
integerValue = 10
floatingValue = 123.456
stringValue = "Welcome to Wipro Technologies"
print "The value stored in integerValue variable is ", integerValue
print "The value stored in floatingValue variable is", floatingValue
print "The value stored in stringValue variable is ", stringValue
print "The value stored in integerValue variable is %d" % integerValue
print "The value in floatingValue variable is %f" % floatingValue
print "The value stored in stringValue variable is %s" % stringValue
```

There are various flexible ways of writing programs and using pythonic statements within the program.

```
#! /usr/bin/python
# Demonstrate of print styles using formatted and unformatted styles

intValue = 10
floatValue = 123.456
strValue = "Welcome to Wipro Technologies"

print 'The values are', intValue,',',floatValue,'and', strValue
print "The values are %d , %.3f and %s" % (intValue,floatValue,strValue)
```

Operators

Operators are special symbols which are intended to do some specific operations based on the values associated to it. Operators are categorized based on the type of intended operations. Operators are categorized based on the type of intended operations and associated with the value types are as follows;

a. Arithmetic Operators

b. Logical Operators

c. Relational Operators

d. Shift Operators

e. Membership Operators
f. Bitwise Operators

Arithmetic Operators are those which are intended to do specific mathematical operations like Addition (+), Subtraction (-), Multiplication (*), Division (/), Modulus (%) and so on.

Logical Operators are those which are used to perform logical binding of two or more conditions using “and”, “or” and “not” logical operators.

Relational Operators are those which are used to perform condition based execution which returns **True** or **False**.

Shift Operators are those which performs shifting of bits in the internal representations for computing new values.

Membership Operators are those which are used to find the membership associations within values or objects.

Bitwise Operators are those which are used to perform bitwise operations for internal computations.

Each of the operators are used to perform associated type operations;

Draft Copy Version - 1

```
#! /usr/bin/python
# Demonstration of Operators in Python
a=10
b=20
print "The given numbers are %d and %d" % (a,b)
res = a + b
print "The Sum is %d " % res
res = a - b
print "The Difference is %d" % res
res = a * b
print "The Product is %d" % res
res = a / b
print "The Division is %d" % res
res = a ** b
print "The Exponential is %d " % res
res = a % b
print "The Modulus is %d" % res
```

Data Structure Objects

Data structure is a mechanism to provide a necessary layout and organize data which could be stored easily and retrieved quickly based on the intended purpose. Python supports extensible structured types without any need to do a workaround like pointers references, or even worse things like explicit type-casts with structured objects though the type casting is available in Python which is purely for specific business requirements. There are many built in Data Structure elements present in python and we would be discussing with only three of the main types;

a. List

b. Tuple

c. Dictionary

Draft Copy Version - 1

List is an ordered set of mutable elements which are heterogeneous, dynamic and non-sequential by nature. Ordered here indicates the elements preserve their insertion order, heterogeneous indicate the elements can have any type of values stored, extensible indicates the memory can shrink or grown and dynamic indicates that every memory location is non-sequential or non-contiguous. A list object is represented with set of values enclosed within [ ] square parentheses.

Label = [val1, val2, val3, val4,...,valn] to access the values of the list, we use indexes of the respective positions.

```
#! /usr/bin/python
listValue=[10,'one',20,'two',30,'three',40,'four',50,'five']
print "The given values of the list is" , listValue
print "The first element of the list item is", listValue[0]
print "The fifth index of the list item is", listValue[5]
listValue [1] = "ONE"
print "The modified values of the list is " , listValue
print "The length of the given List is ", len(listValue)
```

List methods – Method is a member of class that defines a behavior or functionality required and can be applied to an object of the class where method belongs to.

append(): This method is used to insert value at the end of list object. append takes single / one group value as applicable. Eg. listobject.append(value)

count(): This method returns the number of times the value is found in the given list object. Eg. label = listobject.count(value)

extend(): This method is used to add multiple values of any iterable object arbitrarily at the end of the list. Eg. listobject.extend(otherlistobject)

```
#! /usr/bin/python
listValue=[10,'one',20,'two',30,'three',40,'four',50,'five']
listValue1 = ['a','b','c','d','e']
print "The given values of the list is" , listValue
print "The length of the given List is ", len(listValue)
listValue.append(60)
print "The modified values of the list is " , listValue
x = listValue.count(10)
listValue[1] = "One"
print "The value has occured %d times" % x
print "The length of the given List is ", len(listValue)
listValue.extend(listValue1)
print "The extended list of values of the list is " , listValue
print "The length of the given List is ", len(listValue)
del(listValue[5])
print "Value at 4,5,6 index :",listValue[4],listValue[5],listValue[6]
print "The modified list is " , listValue
print "The length of the given List is ", len(listValue)
```

index(): This method returns the index of first successive search of the given element if found in the list object. Eg. label = listobject.index(value)

insert(): This method is used to insert a value anywhere between the list object. It takes two parameters where the first one is offset (postion / index) value, and second is the actual value to be inserted. Eg. listobject.insert(index,value)

pop(): This method is used to return and remove the last element found in the list object if no parameter given, if the parameter given as an index, it will return the given index position value and also remove from the listobject. Eg. label = listobject.pop([index])

Draft Copy

reverse(): This method is going to flip the list, which means the last element becomes first and the first element becomes last as the order and so as other subsequent elements in the list object will be placed. Eg. listobject.reverse()

sort(): This method is going to sort the elements in the list in ascending order and apply the changes to the same object. Eg. listobject.sort()

```
#! /usr/bin/python
listValue=[ 10 , 'one', 20 , 'two' , 30 , 'three', 40 , 'four' , 50 , 'five' ]
print "The given values of the list is" , listValue
x = listValue.index(30)
print "The value given value 30 is present in %d index position" % x
y = listValue.pop()
print "The value popped out of the list is " , y
print "The new values of the list is" , listValue
listValue.remove(10)
print "The new values of the list is" , listValue
listValue.sort()
print "The sort values of the list is" , listValue
listValue.reverse()
print "The reversed of list is" , listValue
listValue.insert(0,10)
print "The new list of list is" , listValue
```

Range Function: Range is a function in python which is returns a list object containing range of values within the given limits and can be used to generate lists quickly. Range function takes three parameters. **range([startvalue], stopvalue, [stepvalue])**. If start value is not given, it takes default start value as zero. If step value is not given, it takes default step value as 1. stop value is mandatory which always specify the end of list, and this value will not be considered for the values in list. It always takes one prior to the last value.

```
>>> x = range(10)
>>>
>>> type(x)
<type 'list'>
>>>
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
>>> # Range with start value and stop value
...
>>> x = range(1,10)
>>>
>>> x
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
>>> # Range with start value, stop value and step value
...
>>> x = range(1,10,2)
>>>
>>> x
[1, 3, 5, 7, 9]
>>>
```

[ ] is referred as a range operator and is used to extract values from a list object given as [startvalue: stopvalue: stepvalue], here also startvalue and stepvalue is optional.

```
>>> #Build a list containing some values using range function
...
>>> x=range(10,110,10)
>>> x
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
>>>
>>> #Extract all values of list using range operator.
...
>>> x[::]
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
>>>
>>> #Extract all values upto 5th index of list using range operator.
...
>>> x[:5:]
[10, 20, 30, 40, 50]
>>>
>>> #Extract values from 3rd Index to 7th index of list using range operator.
...
>>> x[3:7:]
[40, 50, 60, 70]
>>>
>>> #Extract values from 3rd index to 7th index with step 2
...
>>> x[3:7:2]
[40, 60]
>>>
```

Tuple: Tuple is another Data structure object which is like list but having a read only property. Tuple is an ordered immutable object, which does not contain any methods. This is like read only list. Any time if user needs to create a set of items which is constant by nature, then tuple would be a choice to create. tuple is represented with () parentheses. The values once inserted to tuple cannot be modified in anyway. So tuples are read only list. Tuplelabel = (val1, val2, val3,....., valn) The value extract could be similar to list and can be accessed using the index positions.

```
#! /usr/bin/python
tupleValue=(10 , 'one', 20 , 'two' , 30 , 'three', 40 , 'four' , 50 , 'five')
print "The given values of the tuple is " , tupleValue
print "The first element of the tuple item is", tupleValue[0]
print "The fifth element of the tuple item is", tupleValue[4]
print "The range of values from 0 - 5 is", tupleValue[0:5]
#tupleValue[1]="ONE"
print "The length of the given tuple is ", len(tupleValue)
```

Dictionary: Dictionary is a type of Data structure object in python which is unordered and mutable object represented with a pair of representation as key and value. Dictionary is also known as hashes, associative arrays etc., in other programming languages. Dictionary is a mechanism to bind the index value (key) to associate element (value) which was a challenge to store earlier. Eg. If we would like to store Population of different countries / Headcount of various location of Wipro we can use Dictionaries. Keys are like indexes which are unique by nature and the associated element will be stored as value for the key. Dictionary can be represented with {} parentheses. The representation for dictionary should be always in pair of representation as key and value. Label = { key1 : value1 , key2 : value2, key3 : value3, ... keyn : valueN} The insertion order is not preserved in case of Dictionary. To access the

values of the dictionary, we use keys and the respective values of the given key positions are returned.

```
# /usr/bin/python

#Demonstration of Hash in Python
dictValue= {"India":10000000,"China":15000000,"Japan":1000000,"Usa":5000000,"Brazil":200000}
print "The Complete Hash Values present is ", dictValue
print "The population of India is ", dictValue["India"]
print "The key values of dictValues are ", dictValue.keys()
print "The values present in the hash variables are ", dictValue.values()
print "The type of dictValue is ", type(dictValue)
```

There are couple of methods in dict class which can be applied on dictionary object, of which some are; **keys ()**: This returns a list object which is going to contain the keys present in the dictionary object. **values ()**: This returns a list object which is going to contain the values present in the dictionary object.

```
# /usr/bin/python
#Demonstration of Hash in Python
dictValue={}
dictValue["India"]=10000000
dictValue["China"]=15000000
dictValue["Japan"]=1000000
dictValue["Usa"]=5000000
dictValue["Brazil"]=200000
print "The Complete Hash Values present is ", dictValue
x = len (dictValue)
print "The Length of the Dictionary is ",x
print "The type of Variable is ", type(dictValue)
dictValue1 = dictValue.copy()
print "The copied values are ", dictValue1
dictValue1["Pakistan"] = 100000
print "The Complete Copied Hash Values present is ", dictValue1
print "The Complete Original Has Values present is ", dictValue
del(dictValue1["Pakistan"])
print "The Complete Hash Values present is ", dictValue1
```

Try it out

1. Write a program to build a tuple object of integers having range of values from 10 to 100 with step of 10
2. Write a program to build a tuple object containing 10 baby names
3. Write a program to build a List object of integers having range of values of even numbers between 2 to 100
4. Write a program to build a List object of top 10 populated country names and sort in alphabetical order
5. Write a program to various methods on List object
6. Write a program to retrieve sorted 5 Baby names and 5 Country names from above program to build new list
7. Write a program to implement various techniques to concatenate Baby name and Country name into new object

Draft Copy Version - 1

Chapter 3 – I / O Statements

Introduction

Any programming or scripting languages does need to provide a mechanism to perform I/O operations to read and write data into program which is necessary for internal computation and display. Python also provide I/O statements to perform relevant operations using built-in functions.

Inputting Statements

Inputting is a mechanism to read the values from the user which is required for internal computation. There are two styles of inputting values to the program with regard to the execution mode of the program;

1. **Interactive execution**
2. **Non Interactive execution**

Interactive execution is one where, the program will mostly have human intervention during the execution of the program, say to read values for the program from the user. This style of reading values is not recommended as a good practice due under utilization of resources.

Non interactive execution is one where, the program will have mostly no human intervention during the execution of the program, say if the values are to be read, we would use the command line arguments. This style of reading values is highly recommended and a good practice which will improve the utilization of resources also.

Draft Copy Version - 1

Command line syntax

Command line syntax is a way or style used to execute the programs based on the OS interpretation. The commonly followed syntax for the command line is as follows;

<**command**> [-options] <**programname**> [<arguments>]

Command is the executing utility which is required to execute any given program. It may be a compiler or any interpreter or any tool to execute the program.

Options are the flags which are provided to customized the output or tweak the execution as required and supported by the command utility.

Programname is a file which contains the executable instructions given to the command for execution.

Arguments are the required values given in hand to the program before the execution which can be used internally for any further computation. Eg : python sample.py val1 val2

To read the values during the program execution as interactive mode we would use the function **input**() provided by the language. **Input**() function reads a value from the standard input device which is keyboard and retains the type of the value as is.

Example:

1. x = **input**("Enter any number")
2. x = **input**("Enter any string")
3. x = **input**("Enter any float value")

Draft Copy Version - 1

```
>>> x  = input('Enter any value : ')
Enter any value : 10
>>> type(x)
<type 'int'>
>>> x
10
>>>
>>> x  = input('Enter any value : ')
Enter any value : 123.45
>>> type(x)
<type 'float'>
>>> x
123.45
>>>
>>> x  = input('Enter any value : ')
Enter any value : "Wipro Technologies"
>>> type(x)
<type 'str'>
>>> x
'Wipro Technologies'
```

raw_input() function reads a value from the standard input device which is keyboard and always stores any given value type as string.

Example:

a. x = **raw_input**("Enter any value")

The **raw_input**() function will always store any given type of value as raw text and hence it is necessary to do type conversion to the appropriate type if required either by using **eval**() function or type specific type casting functions like **int**(), **float**(), **bool**() etc.

```
#! /usr/bin/python

#Demonstration of String

firstname=raw_input("Enter your First Name : ")
lastname=raw_input("Enter your Last Name : ")

print "Hello! Your first name is %s" % firstname
print "Hello! Your last name is %s" % lastname

fullname = firstname + lastname

print "Hello!  Your full name is %s" % fullname
```

It is recommended to use **eval**() which will do the string quoted value into appropriate type casting without much explicit specification. This is extensively used along with command line arguments to do the type conversion.

```
#! /usr/bin/python

#Demonstration of String

val1=raw_input("Enter First Value : ")
val2=raw_input("Enter Second Value : ")

res = val1  + val2
print "Hello!  Resultant is %s" % res

val1=eval(raw_input("Enter First Value : "))
val2=eval(raw_input("Enter Second Value : "))

res = val1  + val2
print "Hello!  Resultant is %s" % res
```

Outputting Statements

Python programming language provides numerous built-in functions and various mechanism that are readily available to perform I/O operations. Some of the functions like **input**(), **raw_input**() and **print**() are widely used for standard input and output operations respectively. print function is a statement which is used to print any given value to standard output device, which is monitor by default. There are two types of print statements which can be used;

1. **Formatted Print**
2. **Unformatted Print**

Formatted Print statement allows the user to format the output value and display as intended types. This is very similar to the way we format in other high level programming languages. Python provide some special format styles, but here we restrict to format only value type specification using format specifier of C programming style.

Unformatted Print statements are those where the formats of the value are decided by the language himself based on the value and we use "**,**" operator, which is called append operator to enclose the print members.

Print statement can be defined in three styles using both types.

a. **print** 'Any message here'
b. **print** variable_name
c. **print** "Message" , variable name

Draft Copy Version - 1

The unformatted style will always append the values at the end of previous value and the return type would be always string. It is much easier to use this way by inserting "," at the end of any type of each value to be appended to the previous one.

```
#! /usr/bin/python
# Demonstration of Operators in Python
a=10
b=20
print "The given numbers are :", a, b
res = a + b
print "The Sum of two numbers is :", res
res = a - b
print "The Difference of two numbers is :", res
res = a * b
print "The Product of two numbers is :", res
res = a / b
print "The Division of two numbers is :", res
res = a ** b
print "The Exponential of two numbers is :", res
res = a % b
print "The Modulus of the given number is :", res
```

Formatted style

Python has a support of C programming print style of formatting using various format specifiers and displaying message accordingly. The "**%**" operator is used as a format specifier and bind the type specification of the format specifier with the value associated "argument list", like %f %s and %d etc., which is also considered as old style formatting and It will throw error If the format specifier and value does not match.

a. **print** "%s" % "Welcome to python programming"

b. **print** "%d", x; where x is a label defined with some integer value x=10.

c. **print** "The value stored in x is %d" % x

There are other ways of formatting statements using new style formatting using .**format**() method of String class (str), which was introduced in Python 2.6 to have greater flexibility and formatting features. With new style formatting, it is possible to specify the placeholders at explicit positional index. This allows users a flexible way of re-arranging the order of display without changing the arguments you can check for more information about formats from the URL https://pyformat.info/.

```
#! /usr/bin/python
# Demonstration of Operators in Python
a=10
b=20
print "The given numbers are %d and %d" % (a,b)
res = a + b
print "The Sum is %d " % res
res = a - b
print "The Difference is %d" % res
res = a * b
print "The Product is %d" % res
res = a / b
print "The Division is %d" % res
res = a ** b
print "The Exponential is %d " % res
res = a % b
print "The Modulus is %d" % res
```

In the above program due to mismatch in format specifier and value type defined results may vary.

Try it out

1. Write a program to read integer values and perform basic mathematical operations
2. Write a program to read first name and last name and print person fullname
3. Write a program to read various types of values and print their format type
4. Write a program to read values and use format specifier to print different type of values
5. Write a program to perform addition operation on complex, octal and hex representation

Draft Copy Version - 1

Chapter 4 – Control Statements

Introduction

In programming and scripting languages many times, it would not be required to execute the program statements sequentially. It would be required to have a better controlled execution to perform different computations or actions depending on whether it satisfies the required conditions or not.

Control statements are used in such situations to which controls the flow of execution in the program based on the requirement. Statements can be either simple or compound statements. Simple statements are single line and self-executional expressions. Compound statements are set of simple statements.

Python compound statements have no explicit begin or end, and no curly braces to mark like traditional programming languages, where the block of code starts and stops. The only delimiter is Python to define a block is colon "**:**" and the indentation of the code itself. It becomes now essential to now understand to create a block or group of statements as one we would not use "**{ }**" curly parentheses, In Python to do the same rather we would use indentation and un-indentation to group of statements as a block.

Every compound statement which encloses the code blocks are defined by their indentation. By "**code block**", It means control statements, function statements, class statements, methods and so forth. Indenting starts a block and unindenting ends it. There are no explicit braces or keywords. This means that whitespace is significant, and must be consistent. Python recommends 4 white spaces as an indent to create

Draft Copy Version - 1

any compound block and anything un-indented statement would not fall into the group of block statements. Python cannot mix tabs and white spaces for indentation and it is good practice to avoid tabs all together, because the semantics of tabs are not well defined, and they can be interpreted differently on different types of systems and editors.

A Python IDE would be handy in such cases to make representations easy by providing self-indents for the compound statements and saves time.

There are two types of control statement within python;

a. **Conditional Control statement**
b. **Loop Control statements**

Conditional Control Statements

Conditional control statements are those wherein conditions are the mechanism to control the flow of execution and execute a block of statements as required. In Python we have only "**If**" conditional control statements as Python internally works only on references and hence "**Switch Case**" statements of other high level programming languages would not make any much difference than using "if ladder style".

There are four styles of if conditional control statements

a. **Bare if**
b. **If else**
c. **If nested**
d. **If ladder**

Draft Copy Version - 1

Bare If

Bare if is used to execute a block of statement only if the condition is true and ignore if the condition does not satisfy.

Syntax of Bare If:

If condition:

I executed condition evaluated true

I executed condition evaluated true

I executed condition evaluated true

I don't belong to if block

```
#! /usr/bin/python
#Demonstration of Bare if Conditional Statement
x=10
y=20

if x > y:
    print "X is greater than Y"
    print "This is next line in condition"

print "Thats all folks"
```

If else

If else is used to execute a block of statement relating to if the condition being true and else execute another block of statement if the condition does not satisfy.

Syntax of if else:

If condition:

I executed condition evaluated true

I executed condition evaluated true

else:

I executed condition evaluated false

I executed condition evaluated false

I don't belong to either of if block

```
#! /usr/bin/python
#Demonstration of Conditional Statement

x = 10
y = 20

if x > y:
    print "X is greater than Y"
    print "This is true part of the if condition"
else:
    print "Y is greater than X"
    print "This is false part of the if condition"

print "Thats all folks"
```

If nested

If nested is used when multiple levels of conditional statements are to be checked and executed. Simple we can term if nested as "if" conditional statement within another "if" conditional statement.

Syntax of If nested:

If condition1:

If condition2:

I executed true inner if of true outer

else:

I executed false inner of true outer

else:

If condition2:

I executed true inner if of false outer

else:

I executed false inner of false outer

```
#! /usr/bin/python
#Demonstration of Conditional Statement
x = 10
y = 20
z = 30
if x > y:
    if x > z:
        print "X is Biggest of All"
    else:
        print "Z is Biggest of All"
else:
    if y > z:
        print "Y is Biggest of All"
    else:
        print "Z is Biggest of All"
print "Thats all folks"
```

If ladder

If ladder is used to specify multiple conditional statements to be given with multiple optional values which could execute the relative option as matched with the given criteria and terminate.

Syntax of If ladder:

If condition1:

I belong to condition 1 and execute if satisfy

I also belong to condition 1 and execute if satisfy

elif condition2:

I belong to condition 2 and execute if satisfy

Draft Copy - 1

I also belong to condition 2 and execute if satisfy

elif condition3:

I belong to condition 3 and execute if satisfy

I also belong to condition 3 and execute if satisfy

else:

I execute if none of the above conditions are satisfied.

I execute if none of the above conditions are satisfied.

I don't belong to if ladder.

```
#! /usr/bin/python
#Demonstration of Conditional Statement
a='U'
if a=='a':
    print "The given character is vowel = a"
elif a=='e':
    print "The given character is vowel = e"
elif a=='i':
    print "The given character is vowel = i"
elif a=='o':
    print "The given character is vowel = o"
elif a=='u':
    print "The given character is vowel = u"
else:
    print "The given character not found"
print "Thats all folks"
```

Applying logical operator with conditional statements

Many times it would be required to bind two or more conditions to meet specific requirement. Python provides logical operators to bind such two or more conditions using **and, or, not** logical operators.

and logical operators is used when both the condition satisfies, then true part of the conditional statement gets executed.

```
#! /usr/bin/python
#Demonstration of Conditional Statement
x = 100
y = 20
if x > y and x == 10:
    print "True part of if is executed"
else:
    print "False part of if is executed"
print "Thats all folks"
```

or logical operator is used when either of the two given conditions satisfies, then the true part of the conditional statement gets executed.

```
#! /usr/bin/python
#Demonstration of Conditional Statement
x = 10
y = 20
if x > y or x  == 10:
    print "True part of the Conditional statement executed"
else:
    print "False part of the Conditional statement executed"
print "Thats all folks"
```

not operator is used when the flip side of true or false has to be executed, which means if true is "**not**'ed" becomes false and vice versa.

Draft Version - 1

```
#! /usr/bin/python
#Demonstration of Conditional Statement
x = 100
y = 20
if x > y and not x == 1:
    print "The evaluation of the expression is true"
else:
    print "The evaluation of the expression is false"
print "Thats all folks"
```

Loop Control Statements

Loop control statements are iterative statements which are executed for known number of times. Whenever there is a requirement to execute a bunch of statements repeatedly for finite number of times, looping statements come in handy. Looping statements are iterative statements which are executed for known number of times. There are two types of looping statements in python;

1. **while** loop statement
2. **for** loop statement

Either of types of loops has three components for construction;

a. Initial Value / Start value
b. Condition / Final Value
c. Step Value / Increment / Decrement

while loop is a type of iterative statement which is a generic loop used when the loop pattern is known, loop has homogenous type of values and loop is sequential by nature. Most of the generic looping scenarios we would use typical a **while** loop. The above three components of the loop are to be explicitly defined in **while** loop.

Draft Copy Version - 1

Syntax of **while** loop:

Initialization

while condition:

looping statements here

step value

```
#! /usr/bin/python
#Demonstration of Loops

x=1

while x<=10:
    print x,
    x+=1

print "Thats all folks!"
```

for loop is a specific type of iterative statement which is used on domain of values. Domain is a group or set or collection of values. This type of loop is used when the loop has no pattern, loop has heterogeneous values, loop has no sequence. Most of the object iterations we would use for loop. **for** loops are faster and less overheads compared to **while** loop. The above three components of loop are implicit in **for** loop.

for loop is characterized by the use of an implicitly referenced label wherein each iteration step a label is referencing to a value in given sequence or collection. This kind of for loop is mostly used in most of other programming and scripting.

```
#! /usr/bin/python
#Demonstration of Loops


for x in 10,5,-2,40,100,1,25,"Wipro",1.234:
    print "Value is : " , x," and type is ",  type(x)

print "Thats all Folks!"
```

In Python there is no straight forward syntax for "**do while**" and "**switch case**" but programmatically we can achieve this. I would demonstrate "**do while**" and defer "**switch case**" for certain reasons.

```
#! /usr/bin/python
a=10;
print "Executing While Loop"
while a!=10 :
    print "I am inside WHILE LOOP"
    print "This is a simple while loop"
    a+=1

print "Executing Do While Style using While Loop"

while True :
    print "I am inside the DO WHILE LOOP"
    print "This is simple do while style construct"
    if a==10:
        break
    a+=1
print "Out of all Loops"
print "Thats all FOLKS"
```

Try it out

1. Write a program to read integer and display number is positive or negative
2. Write a program to read age and sex and display the person is eligible to marry
3. Write a program to read an integer and display the number is prime or not
4. Write a program to read an integer and display the number the number of times
5. Write a program to read an integer and display Fibonacci series till the given number
6. Write a program to read number and display the number of digits present in the given number
7. Write a program to read a number until zero is given and display the count of numbers, biggest, smallest, sum of all, sum of even, sum of odd, number of positive and negative numbers

Draft Copy Version - 1

Chapter 5 – Functions

Introduction

Functions are block of code which are used to define a specific functionality that can be used multiple times anywhere in the program without re-writing the code again and again. This enables reusability. Which means write once and execute many times.

In Python, basically functions are namespaces memory reserved to store the executional statements and its data members which can be repeatedly used. Namespaces are logical grouping of the names which are uniquely identified and mapped with objects and can be used within a program block. Scope defines the availability of a member within a memory boundary. It is used for preventing name ambiguity or overlapping of names, so that we may have two memory representation with similar names present in different namespaces. We can leave it to the system to have better organized way of storing objects within memory.

Functions would help to break program into smaller units to provide a modular representation. As program becomes complex and larger, functions make it more organized and manageable representations. Furthermore, it avoids repetition and makes the code reusable.

Python provides three types of function implementations;

1. **Built-in function**
2. **User defined function**
3. **Anonymous function**

Draft Copy Version - 1

Built-in Functions

These are provided by the python language himself with a predefined functionality. These functional blocks are always available throughout the program code. Python has many predefined functionalities as built-in function which are given by Python himself to suite day to day activities and can be used as per business requirements. These built-in functions are always available to every program without explicitly including it within program. The following table provides the list of built-in functions provided by Python himself;

Built-in Functions				
abs()	divmod()	input()	open()	staticmethod()
all()	enumerate()	int()	ord()	str()
any()	eval()	isinstance()	pow()	sum()
basestring()	execfile()	issubclass()	print()	super()
bin()	file()	iter()	property()	tuple()
bool()	filter()	len()	range()	type()
bytearray()	float()	list()	raw_input()	unichr()
callable()	format()	locals()	reduce()	unicode()
chr()	frozenset()	long()	reload()	vars()
classmethod()	getattr()	map()	repr()	xrange()
cmp()	globals()	max()	reversed()	zip()
compile()	hasattr()	memoryview()	round()	__import__()

complex()	hash()	min()	set()	
delattr()	help()	next()	setattr()	
dict()	hex()	object()	slice()	
dir()	id()	oct()	sorted()	

Python's built-in functions are grouped into a special module called **__builtins__** named internally as **__builtin__**. It also contains other members like exception classes and other built-in data members. Python imports all the "**built-in**" functions into the namespace when the interpreter is loaded so that programmer can use them directly anywhere in the program.

Some of the built-in functionalities:

```
>>> abs(-10)
10
>>> type(10)
<type 'int'>
>>> cmp(10,20)
-1
>>> len('Wipro Technologies')
18
>>> type(eval('10'))
<type 'int'>
>>> max(10,20)
20
>>> min(10,20)
10
>>> range(1,11,2)
[1, 3, 5, 7, 9]
>>> pow(2,3)
8
>>> help(abs)
```

Draft Version-1

User defined Functions

These are defined by the programmer based on the business requirement and will be available locally across within the same program. User defined functions are those which are created by the programmer based on the business requirements. There are two of User defined components of functions.

a. **Function Body**
b. **Function Call**

Function Body: Function body defines the actual code which needs to be executed at any specific requirement. In Python the body of function has to be defined first in the program before the function is getting called. Function body can be defined using a keyword "**def**" prefixed in front of given function name.

Function call: Function call will transfer the control of current execution into the respective namespace / function body and executes the code defined within the function body. Once the function completes the execution the control would return back to the calling environment.

Scope defines the availability of any member within any defined memory block or boundary of code. There are three type of scopes which can be defined in Python as;

1. Local Scope
2. Global Scope
3. External Scope

Draft Copy Version - 1

Local scope are those members; in which these members are only available within a specific defined block. If we define any members within a function, we can say its scope is localized within the defined function.

Global scope are those members; in which these members are available anywhere across the program. If we define any members within the **main** block of the program which is also the default block where the execution starts available in a built-in namespace "**__main__**", then we say the member is within a global scope.

External scope are those members; in which these members are defined in other external programs and are accessed within any other program. Members in other external files defined such as modules, packages, classes can be treated as external scope. These members can be used by importing / including the files within any source program.

Function Example:

```
def sample():
    print "The call of the function sample is executed"
    print "End of function"
print "Executing first line in Body"
sample()
print "Back to main Body"
```

Draft Copy Version - 1

The above example demonstrates the example of function. A function is defined with the name "**sample**" before the function call is made. When this function is called, Python looks for the given "Namespace" as in current case "**sample**" and if found will transfer the control to this block for execution and once the execution is completed within function it will transfer the control back to calling environment.

Multiple function definitions:

```
def functioncall1():
    print "The call of the functioncall1 is executed"
    functioncall2()
    print "End of function1"

def functioncall2():
    print "The call of the functioncall2 is executed"
    print "End of function2"

print "Executing first line in Body"
functioncall1()
print "Back to main Body"
```

Passing parameters to functions: It is possible to pass the required values from the calling environment to the function for internal computations within function. It can receive any number of arguments and any type of arguments. The labels within the

Draft Copy Version - 1

function definition will receive the passed values as arguments. The following guidelines are intended to be taken care during passing of values.

a. The number of values passed is the number of arguments defined to hold values
b. The order of the values passed is the order of the values received in arguments

```
def functioncall(a,b):
    print "The call of the functioncall is executed"
    print "The address of a and b in function ", id(a), id(b)
    print "The values passed to function %d and %d" % (a,b)

print "Executing first line in Body";
x = 10
y = 20
print "The address of x and y in main is ", id(x), id(y)
functioncall(x,y)
print "Back to main Body"
```

The above program demonstrates the passing of values and also the scopes of the labels. The labels a, b are having local scope that exist only within the function and will gets garbage collected once the control comes out of the function. Whereas labels x, y is having a global scope which can be accessed anywhere within this program.

Returning values from function back to the calling environment: As passed values to function it is possible to return values back to the calling environment and there is no

Draft Copy Version - 1

restriction with number of values to be returned or type of values returned. As mentioned guidelines for passing values, it is necessary to take care the return values similarly.

```
def functioncall(x,y):
    print "The call of the functioncall is executed"
    print "The values passed to function are %d and %d" % (x,y);
    x = 20; y = 10
    return x,y


x = 10; y = 20
print "Executing first line in Body"
x,y=functioncall(x,y)
print "Back to main Body"
print "The value of the Variables is %d and %d" % (x,y)
print type(x), type(y)
```

Default Values: Python supports the use default values within a function block to have a reserved value that can be used, if the required values are not passed within the function call. It is also beneficial to use default values to suite with varied parameter requirements for internal computations.

```
def functioncall(x=0, y=0):
    print "The call of the functioncall is executed"
```

Draft Copy Version - 1

```
    print "The values passed are %d and %d" % (x,y)
    return x+y,x-y

print "Executing first line in Body"
x,y=functioncall()
print "The value of the Variables is %d and %d" % (x,y)
x=100
x,y=functioncall(x)
print type(x), type(y)
print "The value of the Variables is %d and %d" % (x,y)

x=10; y= 20
x,y=functioncall(x,y)
print "The value of the Variables is %d and %d" % (x,y)
```

The above example demonstrates the example of default values with three cases having function call with varied parameter call. If the values are received from calling environment it will dereference default values and reference the received values. If values are not available, then it will use the reserved default values.

Recursive Functions: Recursion is the programmatic approach of calling a function within self with changing value sets. Recursive style help programmer to design simpler and code clarity. It would be beneficial to use recursive functions for recursive data structures (like trees) or recursive algorithms.

Draft Copy Version - 1

```
def factorial(x):
    if x == 1:
        return 1
    else:
        fact = x * factorial(x-1)
    return fact
x = 5
print 'Executing first line in main'
fact = factorial(x)
print 'Factorial of the given number is: ', fact
```

Anonymous Functions:

These are functional blocks without any name. These are used to reduce the overheads caused in traditional style functions and does code replacement like the macros in **C** programming. Anonymous functions are those which does not have any function name but they are used to reduce functional overheads which are frequently called having a single line execution.

If a there is a business functional requirement which has to be frequently called and having a single line of code execution, then using traditional function definition and calling will instill overheads and performance challenges. In Python, we can overcome such challenges of traditional functions styles with using code substitution mechanism as anonymous function, which is also a function that is defined without a name. While

Draft Copy Version - 1

normal functions are defined using the "**def**" keyword, whereas anonymous functions are defined using the "**lambda**" keyword. It uses the signature style to map the requirement and does the code substitution. Anonymous functions can be chained together with other functions to reduce the code complexity and execute functions on the fly.

Anonymous Function definition

Label = **lambda** signature: expression

```
res = lambda x, y: x + y
print "Executing the Main Block functions"
a = 10
b = 20
z = res(a,b)
print "The sum of two numbers %d, %d is %d" % (a,b,z)
a = 1.2
b = 2.3
z = res(a,b)
print "The sum of two numbers %f, %f is %f" % (a,b,z)
a = 'm'
b = 'n'
z= res(a,b)
print "The sum of two no %s, %s is %s" % (a,b,z),type(res)
```

Draft Copy Version - 1

The above program defines an anonymous function handle called res having a single line code substitutions for various instances and performs the operations accordingly.

Draft Copy Version - 1

Try it out

1. Write a program to define function with basic mathematical functionalities like addition, subtraction, multiplication and division
2. Write a program to define function to find power, ceil, log, sin, cos and tan of given numbers using built-in functions
3. Write a function to implement factorial of a given number using recursive functions
4. Write a function which takes list as parameter and maps list of words into a list of integers representing the lengths of the corresponding words
5. Write a Python function that checks whether a passed string is palindrome or not
6. Write a function that takes a list as parameter containing words and returns the length of the longest one

Draft Copy Version - 1

Chapter 6 – Modules

Introduction

Modules are mechanism in Python to group a set of functions having similar goal or functionality as a separate file which can be included within any other program for reusability and concurrent access of the defined functionalities present in the modules. Functions are local to program, whereas Modules can be extended to any other external program where the functionalities can be consumed.

Modules are also mechanism in Python to secure the source code by implicitly Python himself compiling the module program into byte code representation and this byte code can execute on any platform and is non-human readable which understood only by python interpreter himself. Module files should have mandatory an extension of **somename.py**, and therefore any Python program can act as a module file if it satisfies the rules.

Modules mostly contains two components;

1. **Function definitions**
2. **Data Members**

There are two types of Modules;

a. **Built-in Modules**

b. **User Defined Modules**

Draft Copy Version - 1

Built-in Modules

These are the modules provided by the Python language himself having a group of predefined functionality within it. These are **not** always available to the program code; it has to be included into the program using **import** statement. We can list out the available built-in modules using help('**modules'**), this will list out the version specific modules which are integrated along with the current version of software itself. Apart from these software integrated modules we can download a list of modules from the online python repository which can be found at pypi.python.org which contains the recommended modules that can be used on day to day activities which are in ready to be used form using pip tool.

There are two components of Modules.

1. **Module Body:** Module body defines set of functionalities to be included.
2. **Module Use**: Module use includes the defined code into any source program.

Modules can be included into the program by using **import** statement, and **import** can be applied in three styles.

a) **import** module1, module2, module3,..., module
b) **from** module **import** *
c) **from** module **import** specific1, specific2,....,specific n

Listing the members of the namespace using dir() function. dir() is a built in function which is used to return a list object containing the members of the current working namespace.

Draft Copy Version - 1

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>>
>>> import math, os, sys
>>>
>>> dir()
['__builtins__', '__doc__', '__name__', 'math', 'os', 'sys']
>>>
>>> dir(math)
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan', 'atan2
', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs', 'floor',
'fmod', 'frexp', 'hypot', 'ldexp', 'log', 'log10', 'modf', 'pi', '
pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
>>>
```

Above examples demonstrate the use of **dir()** function to display the members of the current namespace and also the members of a specific namespace. **math**, **os** and **sys** are built-in modules provided by Python himself with some associated functionalities grouped inside the respective modules. The above style create appropriate name spaces and include the functionalities in the respective namespaces and can be used math.sqrt(10) to use the functionality.

The above import style will create exclusive namespaces for individual members and will further have namespaces for every member individually present in the module as functionality and hence overcomes any name conflicts when used. To localize the availability of members, present in any module can be done differently as shown below;

Draft Copy Version

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>>
>>> from math import *
>>>
>>> dir()
['__builtins__', '__doc__', '__name__', 'acos', 'asin', 'atan', 'a
tan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs', 'floo
r', 'fmod', 'frexp', 'hypot', 'ldexp', 'log', 'log10', 'modf', 'pi
', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
>>>
>>> sqrt(10)
3.1622776601683795
```

The above example demonstrates a style of import which is normally used in situations when the functionalities defined within the modules are to be localized for the current program, this will not create any module namespace will rather create only the functionality namespaces. These are directly available to the program using the function name. The limitation with the above style, it will inevitably import every member present in the module and will be made available to the current namespace. But if the programmer wants only specific members of the module as required to be imported and overcome importing unnecessary members we can use the following style.

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>>
>>> from math import sin, cos, tan
>>>
>>> dir()
['__builtins__', '__doc__', '__name__', 'cos', 'sin', 'tan']
>>>
>>> sin(45)
0.85090352453411844
>>>
```

Resolving name ambiguities:

Many times there would be situations that encounters name collision of local functionalities in the program with the imported module functionalities and hence it becomes necessary to remove name collisions using aliasing mechanism without changing the original name of the functionality or the module itself.

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>>
>>> import os as OS
>>>
>>> from math import sqrt as SQRT, sin as SINE
>>>
>>> dir()
['OS', 'SINE', 'SQRT', '__builtins__', '__doc__', '__name__']
>>>
>>> SINE(45)
0.85090352453411844
>>>
>>> SQRT(10)
3.1622776601683795
```

User Defined Modules

These are types of module which is defined by the programmer himself based on the business requirement and can be made available within a program by including it using import statement. When complexity of application grows and if many functionalities are required to be defined a part of application and make these functionalities to be concurrently accessed by other files and also other users, it would be good to separate the function definitions and their implementations. Modules are organized way of defining relative functionalities and grouping them together in a single file. We would require user defined modules for three key purposes.

Draft Copy

1. **Modularity**
2. **Reusability**
3. **Security**

Modularity will improve the code representation by breaking down a large program into smaller components as modules which can be pluggable to any program wherever the defined functionality is required. This improves the **Reusability** of the functionality without re-writing the code again and again and can be accessed concurrently from the stored repository. Python creates a special file which is non-human and can be only interpreted by the Python interpreter adds **Security** to the Module file.

User defined modules refer to a Python file having only function definitions and no other standalone statements within it. **Module body** file contains only function definitions and its data members and this file can be used by other programs and consumed as **Module use** by importing it, instead of copying their source code into different programs.

To provide security to module files, Python creates a non-human readable code which is called Python byte code which secures your module file and this can be shared across users without revealing the code stored in central repository. Python byte code is created by Python himself when the **import** statement is used on any module file and this creates a .pyc file which called as Python Compiled file. This file can be pushed to a storage repository for **Concurrent access** and **Code reusability**.

Draft Copy Version -1

Steps to create user defined Modules:

- **Step 1**: Create any file with a good naming convention and must have **.py** as file extension, this is module body file.
- **Step 2:** Edit the created file and define all the required functionalities into the body file intended for your business requirement.
- **Step 3:** Create another file as a source program which will consume the defined functionalities in the module body file (This will be consuming file).
- **Step 4:** Define appropriate **import** statements to include the required module body file into the module use file as required.
- **Step 5:** Use fully qualified naming convention to access the defined module functions as modulename.functionname().
- **Step 6:** Execute the module source file, which in turn will compile the module body file and creates a new file which is the byte code file having samename.pyc called python compiled code.
- **Step 7:** Move this .pyc byte code file to the storage repository for concurrent access and code reusability.

Module Body File: Create a file called mathmod.py and insert the mentioned code in it, This acts as the module body file.

```
#create a file mathmod.py and add the following functionalities
def add(x,y) :
   return x+y


def sub(x,y) :
```

Draft Copy Version - 1

```
    return x-y


def mul(x,y) :
    return x * y


def div(x,y) :
    return x / y
```

Module Source file / use file: Create another file called Mathsrc.py file and insert the mentioned code to consume the above module file.

```
#create another file mathsrc.py module source to consume module file mathmod.py
import mathmod
z=mathmod.add(10,20)
print 'The sum of two numbers is ', z
z=mathmod.sub(10,20)
print 'The sub of two number is ', z
z=mathmod.mul(10,20)
print 'The prod of two number is',z
z=mathmod.div(20,10)
print 'The div of two number is', z
```

The above program will be able to access the module file called mathmod and consume the functionalities defined in this file. This also demonstrates use of fully qualified naming conventions. We can also make the module functionalities to be

Draft Copy Version - 1

localized by using a different style of import statement demonstrated in the program below.

```
#create new file mathsrclocal.py module source to consume module file mathmod.py
from mathmod import *
z = add(10,20)
print 'The sum of two numbers is ', z
z = sub(10,20)
print 'The sub of two number is ', z
z = mul(10,20)
print 'The prod of two number is',z
z = div(20,10)
print 'The div of two number is', z
```

The import style clearly makes the functions defined in the module file to be included as a local functions and hence module namespace is will not available. It would be similar to those functions as defined locally in the program. If there are any functions having similar names it would have name ambiguity.

Hence programmer should overcome these challenges by appropriative taking measures. The current style will be able to include only those modules which are available within the same directory or the default locations where python searches for the module files.

Draft Copy Version - 1

Setting up Module Search Path

Module search path is the locations where python interpreter would search for the importing module files. If the user defined module file exist anywhere outside this default search path, then we should include the exact location within the module search path. Earlier we had created both the Module body file and Module use file in the same directory which would not be the case in real time scenarios. Modules can be made available to access anywhere across the storage repository. If the source program and the Module program file does not exist in the same directory, then it is necessary to include the module search path in the following ways.

There are two ways to set the module search path.

1. **Platform level**
2. **Script level**

To define the module search path at the platform level, we create a Python environment variable called **PYTHONPATH** and include the module search path to look for the defined modules.

To define the module search path at the script level the python provides a built-in list object inside **sys** module called **path**. This list object will be used by Python to search the module path when import statement is defined with module use file. We have to import sys module and append the new search path into the **path** object. It is left to programmer's choice to decide defining style of module search path.

Create a file called mathmod.py and store this file inside a folder **modulecollection** present in d:\ (D drive) or any other drive as available. We assume that this folder will

Draft Copy Version - 1

store the application module files and has to be added into the module search path accessing the defined functionalities.

The folder created to store all module files should not contain any special character or use any python reserved name and also no spaces within the folder name.

```
#create a file mathmod.py and add the following functionalities
def add(x,y) :
    return x+y

def sub(x,y) :
    return x-y

def mul(x,y) :
    return x * y

def div(x,y) :
    return x / y
```

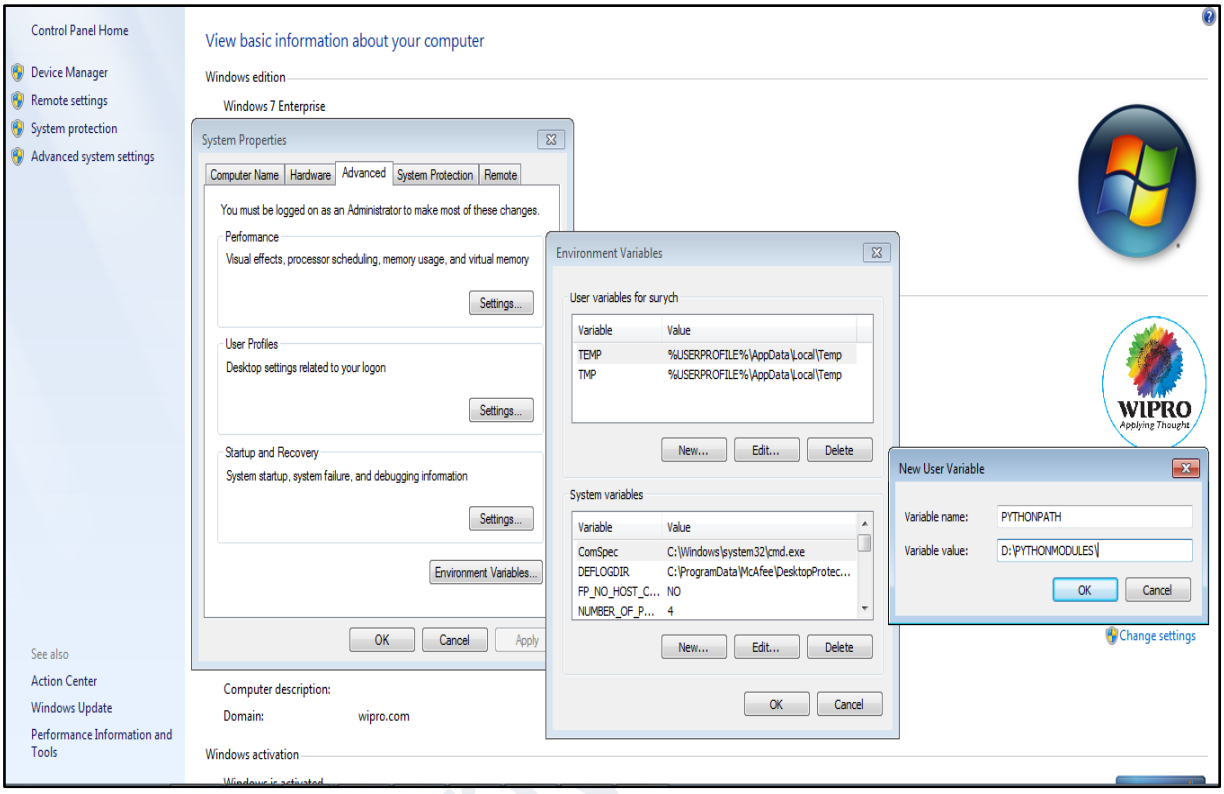
Create another file called mathsrc.py file anywhere and insert the mentioned code to consume the above module file. Observe the module search path included in the program which acts as a program start up statement. It is recommended to have double slashes when defining the path and give appropriate path where the module file exist.

Draft Copy Version - 1

```
#create new file mathsrc.py anywhere to consume module use file mathmod.py

from sys import path

path.append("D:// modulecollection//")

import mathmod

z=mathmod.add(10,20)

print 'The sum of two numbers is ', z

z=mathmod.sub(10,20)

print 'The sub of two number is ', z

z=mathmod.mul(10,20)

print 'The prod of two number is',z

z=mathmod.div(20,10)

print 'The div of two number is', z
```

```
$ pwd
/home/surya
$
$ ls -l ~/Surya2014Backup/Python/Modules/mathmod.py
-rwxrwx--- 1 surya surya 193 2016-06-13 11:42 /home/surya/Surya2014Backup/Pyt
hon/Modules/mathmod.py
$
$ python
Python 2.5.1 (r251:54863, Jul 10 2008, 17:24:48)
[GCC 4.1.2 20070925 (Red Hat 4.1.2-33)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import mathmod
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named mathmod
>>>
>>> import sys
>>> sys.path.append('/home/surya/Surya2014Backup/Python/Modules')
>>> import mathmod
>>> dir()
['__builtins__', '__doc__', '__name__', 'mathmod', 'sys']
>>> mathmod.add(10,20)
30
```

Platform definition: To define python path at the platform level, user needs to have admin access to the system and follow the screen shot to define the PYTHONPATH variable.



PYTHONPATH variable is recommended to be created within the **USER SECTION** of environment variable since the variable is only used by current user and becomes appropriate to individuals.

Draft Copy

Try it out

1. Write a two module program areamodule.py containing basic functionalities to find area of Triangle, Circle and Rectangle and define another module called Trigmodule with basic trigonometric functionalities like sin, cos and tangent of given angles using the built-in functionalities. Implement this module and check the functionalities using different styles of import statement.
2. Write a module program dictcopy.py containing function called mergedict(dict1, dict2) which computes the union of two dictionaries. It should return a new dictionary, with all the items in both its arguments (assumed to be dictionaries). If the same key appears in both arguments pick any from either. What happens of you pass lists instead of dictionaries? How could you generalize your function to handle this case too?

Draft Copy Version - 1

Chapter 7 – Packages

Introduction

When complexity of application grows and if many modules are required to be defined for the application, managing and representation becomes difficult. Packages are mechanism in Python to create hierarchical grouping of module files having similar goal or functionality or integrate module for application as folders which can be included in any program for reusability and concurrent access.

Package pack module files together providing a single interface and separates the design and code. Packages mostly contain module files and also a special file called package constructor having the name **__init__.py** file in every folder of the package level. This package constructor file contains appropriate import statements to include the module files as part of package. Packages can be either a flat package or hierarchical package.

There are two types of packages;

1. **Built in package**
2. **User defined package**

Built-in Packages

Built-in packages are those which are provided by the python himself containing many predefined modules within it, which are required for day to day business requirements. Python greatest strength is its large standard library, which is containing ready to use API's, offering a wide range of functionalities for day to day needs as packages and

Draft Copy Version - 1

there are around 80K+ and growing python recommended packages provided. We can explore PyPI (Python Package index) to explore additional more about it in vendor site.

PyPi packages are not part of the python built-in language, it has to be downloaded, installed and then used based on business requirements.

Some of the built in packages (**Python 3.x version**) which are used for specific purposes are;

1. **http**
2. **urllib**
3. **email**
4. **unittest**
5. **xml**

To understand the package or module contents we would require to query the file using **help** () function. This function displays the help documentation of the package / module / function to have further understanding of functionalities or the members available. This function can be applied only after doing the **import** of the package / module to get help on the respective component.

http is a package that collects several modules for working with the HyperText Transfer Protocol:

- http.client is a low-level HTTP protocol client; for high-level URL opening use urllib.request

Draft Copy Version - 1

- http.server contains basic HTTP server classes based on socketserver
- http.cookies has utilities for implementing state management with cookies
- http.cookiejar provides persistence of cookies

http is also a module that defines a number of HTTP status codes and associated messages through the http.HTTPStatus enum:

urllib is a package that collects several modules for working with URLs:

- urllib.request for opening and reading URLs
- urllib.error containing the exceptions raised by urllib.request
- urllib.parse for parsing URLs
- urllib.robotparser for parsing robots.txt files

email package is a library for managing email messages, including MIME and other RFC 2822-based message documents. It is specifically not designed to do any sending of email messages to SMTP (RFC 2821), NNTP, or other servers; those are functions of modules such as smtplib and nntplib. The email package attempts to be as RFC-compliant as possible, supporting in addition to RFC 2822, such MIME-related RFCs as RFC 2045, RFC 2046, RFC 2047, and RFC 2231.

unittest package is unit testing framework was originally inspired by JUnit and has a similar flavor as major unit testing frameworks in other languages. It supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework.

Draft Copy Version - 1

To achieve this, unittest supports some important concepts in an object-oriented way:

- **test fixture**: A test fixture represents the preparation needed to perform one or more tests, and any associate cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.
- **test case**: A test case is the individual unit of testing. It checks for a specific response to a particular set of inputs. unittest provides a base class, TestCase, which may be used to create new test cases.
- **test suite**: A test suite is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.
- **test runner**: A test runner is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

xml is Python's interfaces for processing XML are grouped in the xml package.

The XML handling submodules are:

- xml.etree.ElementTree: the ElementTree API, a simple and lightweight XML processor
- xml.dom: the DOM API definition
- xml.dom.minidom: a minimal DOM implementation
- xml.dom.pulldom: support for building partial DOM trees
- xml.sax: SAX2 base classes and convenience functions
- xml.parsers.expat: the Expat parser binding

Draft Copy Version - 1

Depending on various requirement and going through the documents of python package index one can include the appropriate package / module as available to achieve intended business requirements.

User defined Package

User defined packages are those which are created by the programmers based on the business requirements and is used to achieve some intended functionality by integrating application based module files. Packages can be included to the any source program with import statement similar to module include styles.

When we have got a large number of program components like modules and classes we would want them to be organized in a better way as packages in Python. User defined packages are those which are created by the programmers based on the business requirements to achieve some intended functionality and integrate application based module files. Packages can be included to the any source program with import statement similar to module include styles.

There are two components of Packages;

1. **Package Body**
2. **Package Use File**

Package body is the folder which will house all the module files and also the package constructor file.

Draft Copy Version - 1

Package use file is the source which consumes the package body file and implements the functionalities defined in the module files.

Steps to create Python Package;

- **Step 1**: Create any folder with a good naming convention, which will act as a package.
- **Step 2**: Get inside this created folder and start adding the module files which are to be part of package.
- **Step 3**: Create a special file __init__.py which is called package constructor necessary for package construction to define and integrate the module files.
- **Step 4**: Edit this package constructor file and define appropriate import statements inside this __init__.py file.
- **Step 5**: Repeat the step 2, step 3 and step 4, if there are any sub packages to be created under the main package to increase the modularity.
- **Step 6:** Come outside of root package folder and create a package source file which will consume the package members.
- **Step 7:** Use appropriate **import** statement defined in the package use file to access the package members using fully qualified naming conventions like packagename.modulename.specificfunctionalities.
- **Step 7**: After successful compilation only retain the .pyc files and move out .py files from the folder structure.

Draft Copy Version - 1

```
$ ls
mathsrc.py  samplepack
$
$
$ tree samplepack/
samplepack/
|-- __init__.py
|-- mathmod.py
`-- trigmod.py

0 directories, 3 files
$
```

The above screen shot demonstrates the contents of the directory, where samplepack is a folder which is created as a package and other module files will be inside this folder.

samplepack folder contains three files __init__.py which is a package constructor file and two other are the module files namely mathmod.py and trigmod.py. The contents of all the files are shown in the following screenshots.

Package constructor file __init__.py contents;

```
#create package constructor file __init__.py
#this file should be present for every directory

import mathmod
import trigmod
```

trigmod.py module file contents;

```
#Create a trigmod.py module file which
#defines basic trignometric functionalities

from math import sin, cos, tan

def sine(x):
    return sin(x)

def cosine(x):
    return cos(x)

def tangent(x):
    return tan(x)
```

Draft Copy Version - 1

mathmod.py module file contents;

```
#create a file mathmod.py and
#add the following functionalities

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

def sub(x,y) :
    return x-y

def mul(x,y) :
    return x * y

def div(x,y) :
    return x / y
```

Package source file

```
#create another file mathsrc.py module
#source to consume package file samplepack
import samplepack

print 'The addition of two numbers is', samplepack.mathmod.add(10,20)
print 'The subtraction of two number is', samplepack.mathmod.sub(10,20)
print 'The multiplication of two number is', samplepack.mathmod.mul(10,20)
print 'The division of two number is', samplepack.mathmod.div(20,10)

print 'The sine of given number is', samplepack.trigmod.sine(45)
print 'The cosine of given number is', samplepack.trigmod.cosine(45)
print 'The tanget of given number is', samplepack.trigmod.tangent(45)
```

The above screen shot is the consume file which consumes the package members using fully qualified naming conventions. The package search path is by default only in the current directory and default search location. If the package body and source exists in different location, we can use the same mechanism to define the path as we did with module file using either Platform style or Program style definitions.

Setting up Package Search Path

The package search path is by default only in the current directory and default search location. If the package body and source exists in different location, we can use the same mechanism to define the path as we did with module file using either Platform style or Program style definitions.

Package search path is the locations where python interpreter would search for the importing package. If the user defined package exist anywhere outside this default search path, then we should include the exact location within the package search path. Earlier we had created both the Package body file and Package use file in the same directory which would not be the case in real time scenarios. Packages can be made available to access anywhere across the storage repository. If the Package body and the Package use file does not exist in the same directory, then it is necessary to include the package search path in the following ways.

There are two ways to set the module search path.

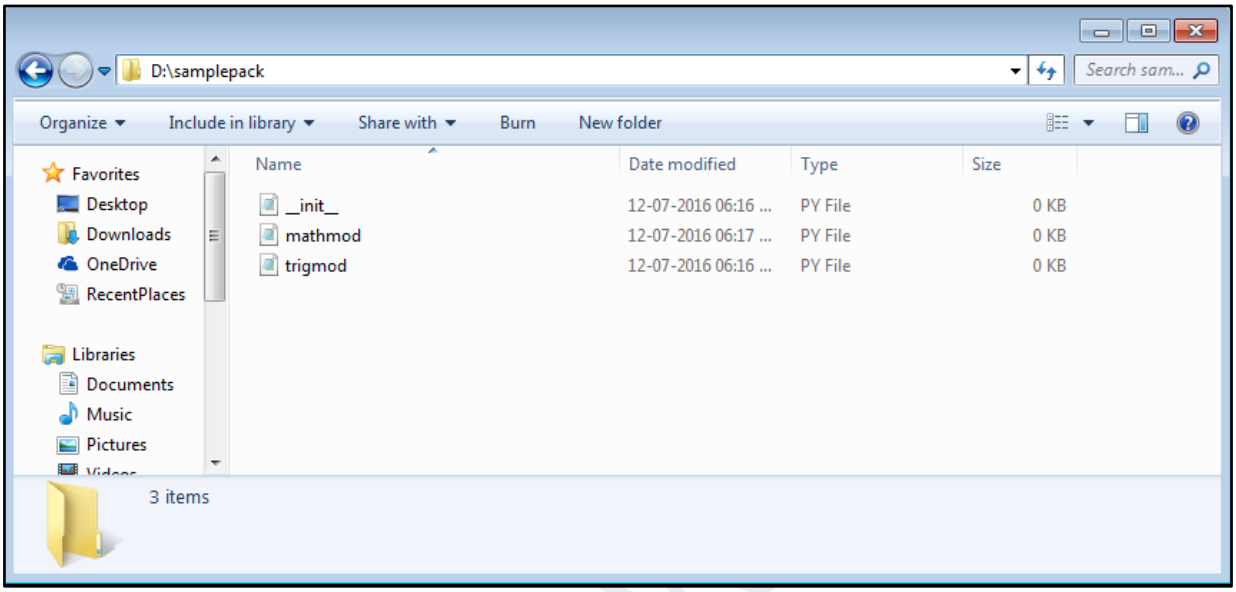
1. **Platform level**
2. **Script level**

To define the package search path at the platform level, we create a Python environment variable called **PYTHONPATH** and include the package search path to look for the defined modules within it.

To define the package search path at the script level the python provides a built-in list object inside **sys** module called **path**. This list object will be used by Python to search

Draft Copy Version -1

the package path when import statement is defined with module use file. We have to import sys module and append the new search path into the **path** object. It is left to programmer's choice to decide defining style of module search path.



The above screen shot demonstrates the contents of the directory, where **samplepack** store in local drive "**D:**" which is created as a package and other module files will be inside this folder. **samplepack** folder contains three files __init__.py which is a package constructor file and two other are the module files namely mathmod.py and trigmod.py. The contents of all the files are shown in the following screenshots.

Package constructor file __init__.py contents;

```
#create package constructor file __init__.py
#this file should be present for every directory

import mathmod
import trigmod
```

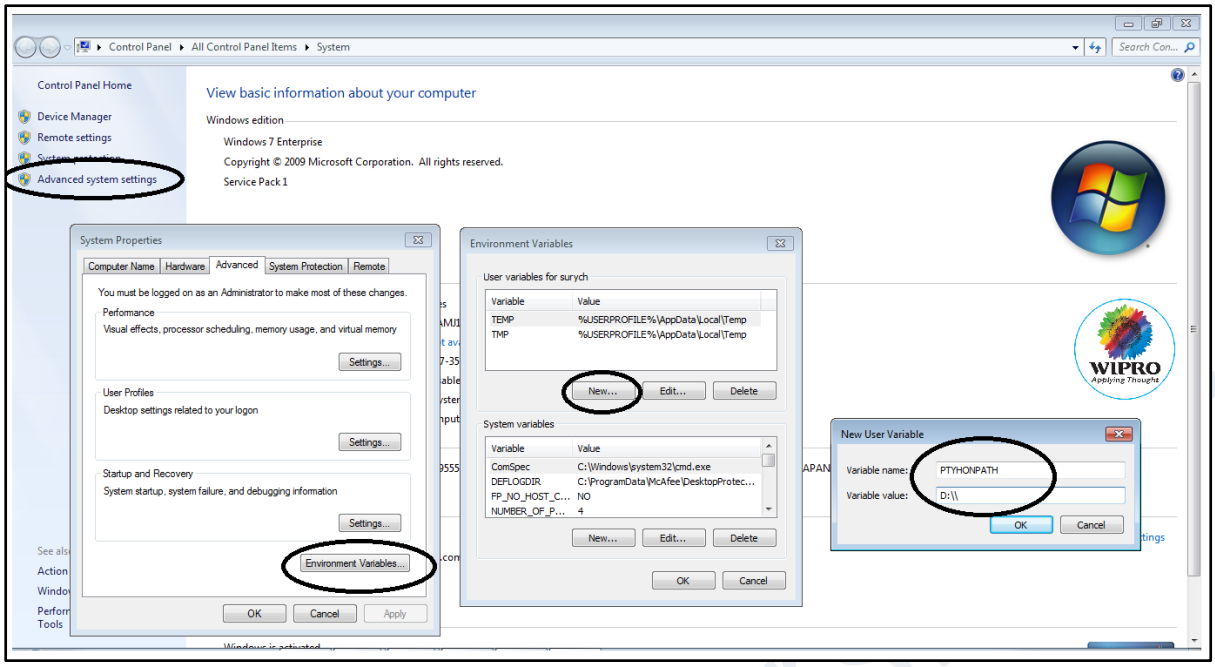
trigmod.py module file contents;

```
#Create a trigmod.py module file which
#defines basic trignometric functionalities

from math import sin, cos, tan

def sine(x):
    return sin(x)

def cosine(x):
    return cos(x)

def tangent(x):
    return tan(x)
```

mathmod.py module file contents;

```
#create a file mathmod.py and
#add the following functionalities

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

def sub(x,y) :
    return x-y

def mul(x,y) :
    return x * y

def div(x,y) :
    return x / y
```

Create another file which is the package source file in any other place outside drive "**D:**" in our case, it is created in drive "**C:**". Let us implement platform level access style to include the package contents to packuse file. Platform level definition of **PYTHONPATH** is used to define the package search path. **PYTHONPATH** variable is recommended to be created within the **USER SECTION** of environment variable since the variable is only used by current user and becomes appropriate to individuals.




```
#create another file mathsrc.py module
#source to consume package file samplepack
import samplepack

print 'The addition of two numbers is', samplepack.mathmod.add(10,20)
print 'The subtraction of two number is', samplepack.mathmod.sub(10,20)
print 'The multiplication of two number is', samplepack.mathmod.mul(10,20)
print 'The division of two number is', samplepack.mathmod.div(20,10)

print 'The sine of given number is', samplepack.trigmod.sine(45)
print 'The cosine of given number is', samplepack.trigmod.cosine(45)
print 'The tanget of given number is', samplepack.trigmod.tangent(45)
```

The above screen shot is the consume file which consumes the package members using fully qualified naming conventions. This program runs only after setting the **PYTHONPATH** or as appropriate describing path location.

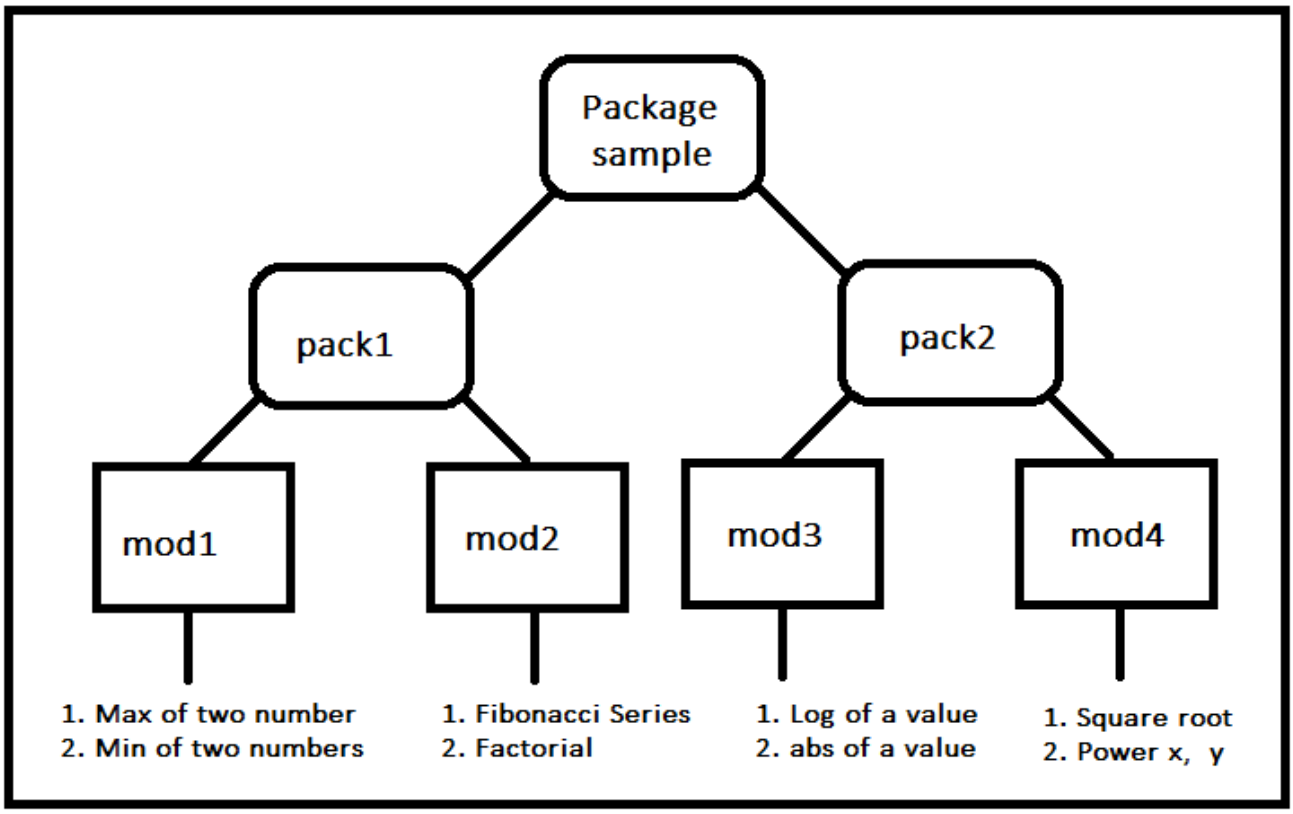
Script level Path inclusion can be done by defining the drive where the package exist before the package is imported. The following line added to the above program prior

importing the package "**samplpack**" should would fine without need of creating the environment variable **PYTHONPATH**.

```
from sys import path
path.append("D://")
```

Draft Copy Version - 1

Try It Out



Draft Copy

Chapter 8 – Exception Handling

Introduction

Errors are the events that are triggered during the execution of the program, which are unexpected and abnormal. If error occurs, then it halts the execution of the program any further, checks if any exception handling blocks are defined, if so it then transfers the execution control to the matching exception in the block and then terminate the program. If exception handling block is not defined in the program, then the control is transferred to built-in exception list where the language himself will take care of executing appropriate exception handler.

Exceptions are mechanism to deal with errors which are encountered in the program during the execution of the program. Errors can be categorized as;

1. **Compile time Errors**
2. **Run time Errors**
3. **Logical Errors**

Compile time errors are those type of errors which are occurred during the compilation of the program. Programmers cannot live with these errors. Most likely cause of the compile time error is syntax errors. These errors have to be fixed to continue the execution of the program. Compile time errors can be identified and fixed by knowing the programming language correctly and defining correct syntax.

Draft Copy Version - 1

Run time Errors are those which are occurred during the execution of the program. Programmers can live with these errors still being there. Most likely cause of Run time errors would be wrong / bad / inappropriate data representation, the program will have no challenges for compilation but would encounter error only if bad data. Run time errors can be identified and fixed by defining appropriate test cases to check the data validation.

Logical Errors are those which are also occurred during the execution of the program but **erroneous** situations will not be visible like other errors. Most likely cause of Logical errors is due to wrong / bad coding practices. Logical errors can be identified and fixed by defining appropriate use cases which are functionality test checks.

Built-in Exceptions

Built-in Exceptions in Python provides a number of classes which are used to deal with some of the predefined exceptions. **Built-in Exceptions** are those which are defined by the language himself and are implicitly executed when the expression triggers the error. List of Built-in Exceptions and their likely causes are mentioned below;

AssertionError	Triggered when assert statement fails
AttributeError	Triggered when attribute assignment or reference fails
EOFError	Triggered when the input() functions hits end-of-file condition
FloatingPointError	Triggered when a floating point operation fails

Draft Copy Version - 1

GeneratorExit	Triggered when a generator's close()method is called
ImportError	Triggered when the imported module is not found
IndexError	Triggered when index of a sequence is out of range
KeyError	Triggered when a key is not found in a dictionary
KeyboardInterrupt	Triggered when the user hits interrupt key (Ctrl+c or delete)
MemoryError	Triggered when an operation runs out of memory
NameError	Triggered when a variable is not found in local or global scope
NotImplementedError	Triggered by abstract methods
OSError	Triggered when system operation causes system related error
OverflowError	Triggered when result of an arithmetic operation is too large to be represented
ReferenceError	Triggered when a weak reference proxy is used to access a garbage collected referent
RuntimeError	Triggered when an error does not fall under any other category
StopIteration	Triggered by next() function to indicate that there is no further item to be returned by iterator
SyntaxError	Triggered by parser when syntax error is encountered

IndentationError	Triggered when there is incorrect indentation
TabError	Triggered when indentation consists of inconsistent tabs and spaces
SystemError	Triggered when interpreter detects internal error
SystemExit	Triggered by sys.exit() function
TypeError	Triggered when a function or operation is applied to an object of incorrect type
UnboundLocalError	Triggered when a local variable is referenced before its assignment
UnicodeError	Triggered when a Unicode-related encoding or decoding error occurs
UnicodeEncodeError	Triggered when a Unicode-related error occurs during encoding
UnicodeDecodeError	Triggered when a Unicode-related error occurs during decoding
UnicodeTranslateError	Triggered when a Unicode-related error occurs during translating
ValueError	Triggered when a function gets argument of correct type but improper value
ZeroDivisionError	Triggered when second operand of division or modulo operation is zero

The classes are case sensitive and has to be dealt with appropriate case representations.

Exception Handling

In python we deal with only **Runtime errors** and we call them as **Runtime exceptions**. Exceptions in python can be handled with **try** and **except** keyword block definitions. If exceptions are handled in the program, the program anticipates the defined type of errors and if encountered will halt the execution and transfer the control to the appropriate exception handler and smoothly terminate the program. It is a good practice to deal with errors by defining exceptional handling.

There are two component of Exception;

a. **Exception handle**
b. **Exception message**

Exception handle is one through which the exception is referenced. In python every exception handle is a class.

Exception message is the information provided to understand the cause of the error.

Handling exception is not a mechanism to solve the problem but to make the program termination smooth and as anticipated.

Some of the best practices or Guidelines for Exception handling;

1. Define only those likely exceptions which are tentative in the program
2. Define most likely exceptions on the top and less likely exceptions at the bottom
3. Define an exception object to capture the associated error message

Draft Copy Version - 1

4. Define non-technical and simple text message to display appropriate information to users regarding the errors
5. Define default exception handler to deal all undefined errors in the programs
6. Define nesting of exceptions do deal specific and generic exception and control of exception.
7. Group the exceptions to make it simple and provide more code clarity
8. Define program closure blocks to deal with program closure statements

There are two types of exceptions.

1. **Built-in Exceptions**
2. **User Defined Exceptions**

Built-in Exceptions are those which are defined by the language himself and are implicitly executed when the expression triggers the error.

User defined exceptions are those which are defined by the user based on the business requirements and will be triggered based on the business scenarios. We would be not covering user defined exceptions due to other pre-requisites required.

```
#! /usr/bin/python
# Program without exception handling
x = input("Enter first number : ")
y = input("Enter second number : ")
print "The values stored are ", x,y
res = x / y
print "The result of two divided numbers are ",res
```

Draft Copy Version -1

The above program executes successfully with no challenges, as long as the values provided are appropriate. If the values are not compatible, then error will be triggered as the case if second number is given with zero.

Defining Exceptions:

```
#! /usr/bin/python
try:
    x = input("Enter any first number : ")
    y = input("Enter any second number : ")
    print "The values given are ", x,y
    res = x / y
    print "The result of two divided numbers are “, res
    print "I am the last statement and I have executed:"
except ZeroDivisionError, e:
    print "Error Encountered, you are trying to divide a number by zero"
    print "Standard error message: ",e
```

The above program will execute successfully the data is appropriate and If the given data is mismatch or the second number given with zero, When the division expression statement in program tries to evaluates it encounter error and finds and matches the exception handle and execute the statements defined of the exception block and terminates.

Draft Copy Version - 1

Default exception

Default exception block will handle all the unhandled exceptions in the program and has to be defined only at the end of all exceptions.

```
#! /usr/bin/python
try:
    x = input("Enter first number")
    y = input("Enter second number")
    print "The values stored are ", x,y
    res = x / y
    print "The result of two divided numbers are “, res
except ZeroDivisionError, e:
    print "You are trying to divide by zero"
except NameError, e:
    print "You are trying to access a label which is not defined"
except Exception, e:
    print "Default Exception handled: ", e
```

Program closure statements

Program closure statements are those which will get executed just before terminating the program. There are two types of program closure statements as blocks.

1. **finally**
2. **else**

Draft Copy Version - 1

finally block will get executed anyhow, whether if any error is thrown or not in the program and then the defined finally block will get executed.

else block will get executed only if there is no error in the program and the else block gets executed. The program closures should be only defined along with try block only.

```
#! /usr/bin/python
try:
   x = input("Enter first number");
   y = input("Enter second number")
   print "The values stored are ", (x,y)
   res = x / y
   print "The result of two divided numbers are ",res
except ZeroDivisionError:
      print "External: You are trying to divide by zero"
except NameError, e:
   print "External: Error with the value given to program"
else:
   print "Else: I execute only if the error is not thrown in program"
finally:
   print "Finally: I will be executed all the times, irrespective of error thrown or not”
```

Grouping of exceptions would be recommended to make the program to have more code clarity and the code representations better.

Draft Copy Version - 1

```
#! /usr/bin/python
try:
    x = input("Enter first number : ");
    y = input("Enter second number : ")
    print "The values stored are ", x,y
    res = x / y
    print "The result of two divided numbers are ",res
except (ZeroDivisionError, TypeError, NameError), e:
    print "Error occured : ", e
except Exception,e:
    print 'Default Exception caught: ', e
```

Draft Copy Version - 1

Try it Out

1. Write a program to demonstrate commonly encountered exceptions with list by reading values into it and also define the default exception other than common exceptions
2. Write a program to demonstrate nesting of exceptions and the common errors when defining nested exceptions and display trace statements
3. Write a program to implement program closure statements and list out common real time scenarios

Draft Copy Version - 1

Chapter 9 – Command Line Arguments

Introduction

Command line arguments is one of the mechanism to read values or input values from user beforehand of the program execution. This would be the recommended way to input values as it would reduce the overheads caused during inputting values in interactive mode.

Most of the programming languages do support command line arguments and python also supports it. The values given at the command line is stored in a default list label object called "**path**" which is inside sys module. Programmer has to import this object and access the values given at the command line. The syntax of providing values at the command line would be;

Command Line Arguments

Command line syntax: <**command**> [-**options**] <**ProgramName**> [<**arguments**>]

Command in the above syntax is the executing utility

Options in the above syntax are used to fine tune program execution

ProgramName in the above syntax is the file which contains the instructions to be executed

Arguments in the above syntax are the necessary values provided in hand to program

Draft Copy Version - 1

The values given in command line would be treated by default as a raw text, hence it needs to be type casted to an appropriate type specification for evaluating any given expression within the program. There are exclusive functions in python to do the type casting in python and also generic function to deal with it. Python also provides many other sophisticated packages / modules to deal with command line arguments like argparse, cli, getopt etc. The getopt module is a parser for command line options whose API is designed to be familiar to users of the C getopt() function. The argparse module makes it easy to write user-friendly flexible way to deal with command-line operations and I would highly recommend this module for programmers.

```
#! /usr/bin/python
import sys
print "Total parameter given at command line :",len(sys.argv)
print "The given values at command line is : ", sys.argv
print "The first parameter in command  line is :",sys.argv[0]
print "The second parameter in command  line is :",sys.argv[1]
print "The type of the value in 1st index is : ", type(sys.argv[1])
$
$
$ python cla0.py 10 20
Total parameter given at command line : 3
The given values at command line is :  ['cla0.py', '10', '20']
The first parameter in command  line is : cla0.py
The second parameter in command  line is : 10
The type of the value in 1st index is :  <type 'str'>
```

The above program demonstrates command line parameters which has to be given at the command line using a space separated values and if there are multiple word representations then “Quotes” can be used to group this value as one / single. argv will hold always file name at 0th index and remaining values in the subsequent indexes. Any type of

value provided in command line will be stored internally as a string which has to be type converted.

Type casting

The values given in command line would be treated by default as a raw text, hence it needs to be type casted to an appropriate type specification for evaluating any given expression within the program.

There are exclusive functions in Python to do the type casting and also generic function to deal with it. Python also provides many other sophisticated packages / modules to deal with command line arguments like argparse, cli, getopt etc. The getopt module is a parser for command line options whose API is designed to be familiar to users of the C getopt() function. The argparse module makes it easy to write user-friendly flexible way to deal with command-line operations and I would highly recommend this module for programmers.

```
#! /usr/bin/python
import sys
print "Total parameter given at command line :",len(sys.argv)
print "The given values at command line is : ", sys.argv
print "The first parameter in command  line is :",sys.argv[0]
print "The second parameter in command  line is :",sys.argv[1]
print "The type of the value in 1st index is : ", type(sys.argv[1])
$
$
$ python cla0.py 10 20
Total parameter given at command line : 3
The given values at command line is :  ['cla0.py', '10', '20']
The first parameter in command  line is : cla0.py
The second parameter in command  line is : 10
The type of the value in 1st index is :  <type 'str'>
```

The above program demonstrates command line parameters which has to be given at the command line using a space separated values and if there are multiple word representations then “Quotes” can be used to group this value as one / single. argv will hold always file name at 0th index and remaining values in the subsequent indexes.

```
#! /usr/bin/python
import sys

print "Total parameter given at command line :",len(sys.argv)
print "The given values at command line : ", sys.argv
print "First parameter in commandline :",sys.argv[0], type(sys.argv[0])
print "Second parameter in commandline :",sys.argv[1],type(sys.argv[1])
print "Third parameter in commandline :",sys.argv[2],type(sys.argv[2])
print 'Resultant value of all parameters' , sys.argv[1] + sys.argv[2]
$
$ python cla00.py 10 20
Total parameter given at command line : 3
The given values at command line :  ['cla00.py', '10', '20']
First parameter in commandline : cla00.py <type 'str'>
Second parameter in commandline : 10 <type 'str'>
Third parameter in commandline : 20 <type 'str'>
Resultant value of all parameters 1020
```

Any type of value provided in command line will be stored internally as a string which has to be type converted. We can use int(), float(), str(), bool, eval() etc., for type conversion.

```
#! /usr/bin/python

import sys

print 'Sum arguments doing typecasting using exclusive function'

print "Sum of two arguments :", int(sys.argv[1]) + int(sys.argv[2])

print 'Sum arguments using eval'

print "Sum of two arguments :", eval(sys.argv[1]) + eval(sys.argv[2])
```

Draft Copy

As mentioned earlier the above program demonstrates the type conversion using exclusive function and also generic function.

```
#! /usr/bin/python
import sys
print "The length of Command line arguments :", len(sys.argv)
print "The given values at command line is : ", sys.argv
i=1
res=0
while(i!=len(sys.argv)):
   print "sys.argv[%d] : %s" % (i, sys.argv[i])
   if (sys.argv[i].isdigit()):
       res += int(sys.argv[i])
   i+=1
print "Sum of all values given in command line is ", res
```

```
$ python cla21.py 10 a 20 b 30 c 40 c 50 d
The length of Command line arguments : 11
The given values at command line is :  ['cla21.py', '10', 'a', '20', 'b', '30
', 'c', '40', 'c', '50', 'd']
sys.argv[1] : 10
sys.argv[2] : a
sys.argv[3] : 20
sys.argv[4] : b
sys.argv[5] : 30
sys.argv[6] : c
sys.argv[7] : 40
sys.argv[8] : c
sys.argv[9] : 50
sys.argv[10] : d
Sum of all values given in command line is  150
```

The above program reads any number or different types of values and only chooses int type of value from the given to perform sum operations.

Try it Out

1. Write a program to read any number of values in command line arguments and sum up all the values using for loop
2. Write a program to define the necessary functions within the program to handle the basic mathematical operations given in the command line. Eg: a) ./do.py add 10 20 b) ./do.py mul 10 2

Draft Copy Version - 1

Chapter 10 – File Handling

Introduction

A file is physical entity in filesystem of the secondary storage device to store some data permanently within computer even after computer is turned off. We have come across various types of files stored in computer, like your program files, audio files, video files, text files etc., Python provides an easy way to store, access and manipulate these files. Generally, we divide files in two categories, text file and binary file. Text files are simple text whereas the binary files contain binary data which is only readable by computer.

File handling is a mechanism to store, retrieve and manipulate the program generated data permanently into storage device and can be loaded back to program by populating data into various data structure object for internal purposes.

Python has many built-in functions to perform file operations. **open ()**, for opening a file on disk. open returns a file object, which has methods and attributes for getting information about and manipulating the opened file.

There are two key file operations to be performed before the data is either to be stored or retrieved.

1. **Read Operation**
2. **Write Operation**

Read operation will load the associated given physical file contents into temporary area reading the contents from the secondary storage to primary storage, which we

Draft Copy Version - 1

can use within the python program. Read operation does not allow the user to make modifications to the opened file or the data associated to it.

Write operations will dump / write the program temporary data contents from temporary memory to secondary storage device as a physical file which was associated during the opening and will support any manipulation of the contents in the physical file.

Before the file operations are done, File has to be associated to the program for either of the said operations. Open function is used to specify the file handle for the required operations. Open function will create a handle / association of the physical file to the temporary memory.

filehandle = open ('filename', 'mode', [***buffering***])

The open function takes three parameters,

a. Filename
b. Mode
c. Buffering

Filename: The name of the file which needs to be associated to the program. b. Mode of the file opening to restrict the type of operation.

Mode: Mode is an optional argument. The type of operation intended on the file to be specified. If the file needs to only read / open the contents to be used without any changes, then open in 'r', If the file has to be opened with write / modification operations possible then we open with write mode. But write mode will erase the previous contents

Draft Copy Version - 1

of the file. The file can be also opened with append mode 'a', which will allow users to append the new contents to be written into the file at the end of the current file without deleting the previous contents.

Buffering: Buffering argument is optional which specifies the file's desired buffer size: 0 means unbuffered, 1 means line buffered, any other positive value means use a buffer of (approximately) that size (in bytes). A negative buffering means to use the system default, which is usually line buffered for tty devices and fully buffered for other files. If omitted, the system default is used.

Closing of file handle which was opened for operations makes the file to terminate the handle associated with the program to safely terminate the connection and write back any contents permanently to storage device.

```
filehandle.close()
```

In Python "**file**" is a class which contains many methods to deal with file handling operations

File handling Operations

File when successfully opened with any associated handle will return the file attributes like file name, file mode etc. The default mode is read only, i.e. if we do not provide any mode it will open the file as read only.

Draft Copy Version - 1

```
#! /usr/bin/python

#Sample script for file handling

fd = open ('wipro.txt','r')

print 'The file handle opened at : ', fd
print 'The file opened for operations is : ', fd.name
print 'The status of the file whether closed : ', fd.closed
print 'The mode of file opened : ', fd.mode
fd.close()
print 'The status of the file after closing : ', fd.closed
```

In the above program file 'wipro.txt' is opened for read mode. If successful it will return the file handle and will be able to read the contents using the file handle as a reference.

- **fd** returns the reference associated to the file handle
- **fd.name** this provides the name associated to the file handle for file operations
- **fd.closed** returns Boolean value based on if file is opened or closed
- **fd.mode** returns the mode associated to file handle

When file operations are done it is mostly expected to trigger I/O Errors and hence it would be recommended to deal with I/O Exceptions which would handle any associated errors.

It is recommended to define exceptions when any file operations are performed on a file. Python deals file handling exception using IOError exception.

Draft Copy Version 1

```
#! /usr/bin/python
#Sample script for file handling
try:
    fd = open ('wipro.txt','r')
    print 'The file opened for operations is : ', fd.name
    print 'The status of the file is : ', fd.closed
    print 'The mode of file opened is : ', fd.mode
except IOError, e:
    print 'The file you are trying to open does not exist'
    print 'Error : ' , e
except Exception, e:
    print 'Error Occured : ', e
else:
    fd.close()
```

Reading file contents: Python file class provides various methods to read the file contents. Some of them are discussed here. There is an internal file pointer which moves from the beginning of the file to the end of file and will be able to read the contents using various methods. The file pointer references can be move using three position;

a. **Beginning of file**
b. **Current pointer position**
c. **End of file**

Beginning of file is the start of the file which referenced with unique number 0. **Current pointer position** is the position of the file pointer if it has been moved anywhere within file which is referenced as 1. **End of file** is the end position of the file which is referenced as 2.

Draft Copy Version

File read methods:

1. read()
2. readline()
3. readlines()

read () is a method which is used to read file contents either as complete file if no parameter given or read file contents with specified number of bytes from current position if any value specified.

readline () is a method which is used to read the file contents line by line and returns the line as a string literal.

readlines () is a method which is used to read the file contents line by line similar to **readline** () method, but it will return the lines into a list object, which means each line goes as individual element value of the list and hence every line comes into lists as individual elements.

```
#! /usr/bin/python
#Sample script for file handling
try:
    fd = open ('wipro.txt','r')
    print 'The file opened for operations is : ', fd.name
    print 'The status of the file is : ', fd.closed
    print 'The mode of file opened is : ', fd.mode
    print 'Enabling read operation for file : '
    print "Read operation on the file : ", fd.read(10)
    print "Read Line Operation on the file : ", fd.readline()
except IOError, e:
    print 'The file you are trying to open does not exist'
    print 'Error : ' , e
except Exception, e:
    print 'Error Occured : ', e
else:
    fd.close()
```

Python provides to change the contents of the file by specifying write mode while opening the file for file operations. If the file mode is specified as write it will create a new file if the file does not exist and if the file exist it will **overwrite** the previous contents of the file with the new contents. If you would want to retain the previous contents and append the new contents from the end of current file, then we can open the file with append mode.

```
#! /usr/bin/python
#Sample script for file handling

fd = open ('Wipro.txt','w')

print 'The file opened for operations is ', fd.name
print 'The status of the file is ', fd.closed
print 'The mode of file opened is ', fd.mode

print 'Enabling write operation for file'

text = "This line has been added from file operation\nNext line"

fd.write(text)

fd.close()
```

We can position the file pointer and display the position of the file pointer using seek and tell methods.

seek () method is used to position the file pointer at any position in the file using the reference position.

tell () method is used to display the position of the file pointer in terms of bytes from the beginning of the file.

```
#! /usr/bin/python

#Sample script for file handling

fd = open ('wipro.txt','r')

print 'Enabling read operation for file : '
print "Read operation on the file first 10 characters : ",fd.read(11)
print 'The file pointer is in : ', fd.tell(), 'position.'
fd.seek(-6,2) # fd.seek(numberofbytes,reference 0,1,2)
print 'The file pointer after reset line is : ', fd.readline()
fd.close()
```

File modes:

Mode	Description
r	Open file in read mode and is the default mode if not specified.
w	Open file in write mode, will overwrite the previous contents if file exist
a	Open file in append mode, will append the contents to existing file at end

There are numerous methods available in python for file handling operations. Some of the commonly used file methods are mentioned;

Method	Description
close()	Close the file handle which was opened

Draft Copy Version

fileno()	Returns the file descriptor associated to the file
read()	Reads complete file if no parameter given
read(n)	Reads 'n' bytes from the current position
readline()	Reads the current file pointer line
readlines()	Reads lines into a list object
seek(offset, bytes)	Moves the file pointer to given bytes position using offset reference
tell()	Displays the current position of file pointer from beginning in bytes
write()	Writes the data into the file stream
writelines()	Writes set of lines into file stream

File handling Modules

Python provides various API for file stream operations to perform file associate operations. Some of them have been demonstrated.

os Module: The built-in OS module provided by Python is a way of using operating system dependent functionalities. The various functions that the OS module provides allows users to interface with the underlying operating system on which Python is installed on be any that Windows, Unix / Linux or Mac etc., Python OS module can be used to perform tasks such as retrieve name of present working directory, changing current working directory, checking if certain files or directories exist at a location, creating new directories, deleting existing files or directories, walking through a

Draft Copy Version - 1

directory and performing operations on every file in the directory that satisfies some user-defined criteria, and a lot more.

shutil Module: The shutil module offers a number of high-level operations on files and collections of files. In particular, functions are provided which support file copying and removal. For operations on individual files, It would be easier to use os module.

glob Module: The glob module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell, although results are returned in arbitrary order. No tilde expansion is done, but *, ?, and character ranges expressed with [] will be correctly matched.

subprocess Module: The subprocess module allows user to spawn new processes, connect to their input / output / error pipes, and obtain their return codes. This module intends to replace several older modules and functions.

Draft Copy Version - 1

Try it Out

1. Write a Python program to read an entire text file into the program
2. Write a Python program to read only first “n” lines of a file into the program
3. Write a Python program to read only last “n” lines of a file into the program
4. Write a Python program to read a given file contents line by line and store it into a list object in the program
5. Write a Python program to write a list contents into a file

Draft Copy Version - 1