Financial Accounting Tutorial



## About the Tutorial

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985- 1990. Like Perl, Python source code is also available under the GNU General Public License (GPL). This tutorial gives enough understanding on Python programming language.

## Audience

This tutorial is designed for software programmers who need to learn Python programming language from scratch.

## Prerequisites

You should have a basic understanding of Computer Programming terminologies. A basic understanding of any of the programming languages is a plus.

## Disclaimer & Copyright

Copyright 2014 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at [contact@tutorialspoint.com.](mailto:contact@tutorialspoint.com)

i



## Table of Contents

[**About the Tutorial i**](#_bookmark0)

[**Audience i**](#_bookmark1)

[**Prerequisites i**](#_bookmark2)

[**Disclaimer & Copyright i**](#_bookmark3)

[**Table of Contents ii**](#_bookmark4)

1. [OVERVIEW 1](#_bookmark5)

[**History of Python 1**](#_bookmark6)

[**Python Features 1**](#_bookmark7)

1. [ENVIRONMENT 3](#_bookmark8)

[**Local Environment Setup 3**](#_bookmark9)

[**Getting Python 3**](#_bookmark10)

[**Installing Python 4**](#_bookmark11)

[**Setting up PATH 5**](#_bookmark12)

[**Setting path at Unix/Linux 5**](#_bookmark13)

[**Setting path at Windows 5**](#_bookmark14)

[**Python Environment Variables 6**](#_bookmark15)

[**Running Python 6**](#_bookmark16)

1. [BASIC SYNTAX 9](#_bookmark17)

[**First Python Program 9**](#_bookmark18)

[**Python Identifiers 10**](#_bookmark19)

[**Python Keywords 11**](#_bookmark20)

[**Lines and Indentation 11**](#_bookmark21)

[**Multi-Line Statements 14**](#_bookmark22)

[**Quotation in Python 14**](#_bookmark23)

[Comments in Python 14](#_bookmark24)

[Using Blank Lines 15](#_bookmark25)

[Waiting for the User 15](#_bookmark26)

[Multiple Statements on a Single Line 16](#_bookmark27)

[Multiple Statement Groups as Suites 16](#_bookmark28)

[Command Line Arguments 16](#_bookmark29)

[Accessing Command-Line Arguments 17](#_bookmark30)

[Parsing Command-Line Arguments 18](#_bookmark31)

[getopt.getopt method 18](#_bookmark32)

[Exception getopt.GetoptError 18](#_bookmark33)

1. [VARIABLE TYPES 21](#_bookmark34)

[Assigning Values to Variables 21](#_bookmark35)

[Multiple Assignment 22](#_bookmark36)

[Standard Data Types 22](#_bookmark37)

[Python Numbers 22](#_bookmark38)

[Python Strings 24](#_bookmark39)

[Python Lists 25](#_bookmark40)

[Python Tuples 26](#_bookmark41)

[Python Dictionary 27](#_bookmark42)

[Data Type Conversion 28](#_bookmark43)

1. [BASIC OPERATORS 30](#_bookmark44)

[Types of Operators 30](#_bookmark45)

[Python Arithmetic Operators 30](#_bookmark46)

[Python Comparison Operators 32](#_bookmark47)

[Python Assignment Operators 35](#_bookmark48)

Python Bitwise Operators 38

Python Logical Operators 40

Python Membership Operators 40

Python Identity Operators 42

Python Operators Precedence 44

1. DECISION MAKING 47

If Statement 48

If…else StateŵeŶt 49

The elif Statement 51

Single Statement Suites 53

1. LOOPS 54

While Loop 55

The Infinite Loop 57

Using else Statement with Loops 58

Single Statement Suites 59

For Loop 59

Iterating by Sequence Index 61

Using else Statement with Loops 62

Nested Loops 63

Loop Control Statements 65

Break Statement 66

Continue Statement 68

Pass Statement 69

1. NUMBERS 71

Number Type Conversion 72

Random Number Functions 74

Trigonometric Functions 74

Mathematical Constants 75

1. STRINGS 76

Accessing Values in Strings 76

Updating Strings 76

Escape Characters 77

String Special Operators 78

String Formatting Operator 79

Triple Quotes 81

Unicode String 82

Built-in String Methods 83

1. LISTS 124

Python Lists 124

Accessing Values in Lists 124

Updating Lists 125

Deleting List Elements 126

Basic List Operations 126

Indexing, Slicing, and Matrixes 127

Built-in List Functions and Methods 127

1. TUPLES 141

Accessing Values in Tuples 141

Updating Tuples 142

Deleting Tuple Elements 142

Basic Tuples Operations 143

Indexing, Slicing, and Matrixes 144

No Enclosing Delimiters: 144

Built-in Tuple Functions 145

1. DICTIONARY 151

Accessing Values in Dictionary 151

Updating Dictionary 152

Delete Dictionary Elements 153

Properties of Dictionary Keys 153

Built-in Dictionary Functions and Methods 154

1. DATE AND TIME 169

What is Tick? 169

What is TimeTuple? 169

Getting Current Time 171

Getting Formatted Time 171

Getting Calendar for a Month 172

The time Module 172

The calendar Module 189

Other Modules and Functions 191

1. FUNCTIONS 192

Defining a Function 192

Calling a Function 193

Passing by Reference Versus Passing by Value 194

Function Arguments 195

Required Arguments 195

Keyword Arguments 196

Default Arguments 198

Variable Length Arguments 198

The Anonymous Functions 200

The return Statement 201

Scope of Variables 202

Global vs. Local variables: 202

1. MODULES 204

The import Statement 204

The from...import Statement 205

The from...import \* Statement: 205

Locating Modules: 205

The PYTHONPATH Variable 206

Namespaces and Scoping 206

The dir( ) Function 207

The globals() and locals() Functions 208

The reload() Function 208

Packages in Python 209

1. FILES I/O 211

Printing to the Screen 211

Reading Keyboard Input 211

The raw\_input Function 211

The input Function 212

Opening and Closing Files 212

The open Function 212

The file Object Attributes 214

The close() Method 215

Reading and Writing Files 216

The write() Method 216

The read() Method 217

File Positions 218

Renaming and Deleting Files 219

The rename() Method 219

The remove() Method 219

Directories in Python 220

The mkdir() Method 220

The chdir() Method 221

The getcwd() Method 221

The rmdir() Method 222

File and Directory Related Methods 222

1. EXCEPTIONS 248

What is Exception? 251

Handling an Exception 251

The *except* Clause with No Exceptions 253

The *except* Clause with Multiple Exceptions 254

The try-finally Clause 255

Argument of an Exception 256

Raising an Exception 257

User-Defined Exceptions 258

1. CLASSES AND OBJECTS 260

Overview of OOP Terminology 260

Creating Classes 261

Creating Instance Objects 262

Accessing Attributes 262

Built-In Class Attributes 264

Destroying Objects (Garbage Collection) 266

Class Inheritance 268

Overriding Methods 270

Base Overloading Methods 271

Overloading Operators 271

Data Hiding 272

1. REGULAR EXPRESSIONS 274

The match Function 274

The search Function 275

Matching Versus Searching 277

Search and Replace 278

Regular-Expression Modifiers: Option Flags 279

Regular-Expression Patterns 279

Regular-Expression Examples 282

Grouping with Parentheses 284

Backreferences 284

1. CGI PROGRAMMING 287

What is CGI? 287

Web Browsing 287

CGI Architecture 288

Web Server Support and Configuration 288

First CGI Program 289

HTTP Header 290

CGI Environment Variables 291

GET and POST Methods 292

Passing Information using GET method 292

Simple URL Example : Get Method 293

Simple FORM Example: GET Method 294

Passing Information Using POST Method 295

Passing Checkbox Data to CGI Program 296

Passing Radio Button Data to CGI Program 298

Passing Text Area Data to CGI Program 299

Passing Drop Down Box Data to CGI Program 301

Using Cookies in CGI 302

How It Works? 302

Setting up Cookies 303

Retrieving Cookies 304

File Upload Example 305

How To Raise a "File Download" Dialog Box? 307

1. DATABASE ACCESS 308

What is MySQLdb? 308

How do I Install MySQLdb? 309

Database Connection 309

Creating Database Table 311

INSERT Operation 312

READ Operation 314

Update Operation 316

DELETE Operation 317

Performing Transactions 319

COMMIT Operation 319

ROLLBACK Operation 320

Disconnecting Database 320

Handling Errors 320

1. NETWORK PROGRAMMING 322

What is Sockets? 322

The socket Module 323

Server Socket Methods 323

Client Socket Methods 324

General Socket Methods 324

A Simple Server 324

A Simple Client 325

Python Internet modules 326

Further Readings 327

1. SENDING EMAIL 328

Sending an HTML e-mail using Python 330

Sending Attachments as an E-mail 331

1. MULTITHREADING 334

Starting a New Thread 334

The Threading Module 336

Creating Thread Using Threading Module 336

Synchronizing Threads 339

Multithreaded Priority Queue 341

1. XML PROCESSING 345

What is XML? 345

XML Parser Architectures and APIs: 345

Parsing XML with SAX APIs 347

The make\_parser Method 347

The parse Method 348

The parseString Method 348

Parsing XML with DOM APIs 352

1. GUI PROGRAMMING 355

Tkinter Programming 355

Tkinter Widgets 356

Geometry Management 431

1. FURTHER EXTENSIONS 437

Pre-Requisites for Writing Extensions 437

First Look at a Python Extension 437

The Header File Python.h 438

The C Functions 438

The Method Mapping Table 439

The Initialization Function 440

Building and Installing Extensions 442

Importing Extensions 442

Passing Function Parameters 443

The PyArg\_ParseTuple Function 444

Returning Values 446

The Py\_BuildValue Function 447

xii



# OVERVIEW

Python

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

* **Python is Interpreted:** Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
* **Python is Interactive:** You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
* **Python is Object-Oriented:** Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
* **Python is a Beginner's Language:** Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

## 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.

Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, Unix shell, and other scripting languages.

Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).

Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.

## Python Features

Python's features include:

* **Easy-to-learn:** Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.

Python

* **Easy-to-read:** Python code is more clearly defined and visible to the eyes.
* **Easy-to-maintain:** Python's source code is fairly easy-to-maintain.
* **A broad standard library:** Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
* **Interactive Mode:** Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
* **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
* **Extendable:** You can add low-level 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 shell scripting.

Apart from the above-mentioned features, Python has a big list of good features, few are listed below:

* IT supports functional and structured programming methods as well as OOP.
* It can be used as a scripting language or can be compiled to byte-code for building large applications.
* It provides very high-level dynamic data types and supports dynamic type checking.
* IT supports automatic garbage collection.
* It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

# ENVIRONMENT

Python

Python is available on a wide variety of platforms including Linux and Mac OS X. Let's understand how to set up our Python environment.

## Local Environment Setup

Open a terminal window and type "python" to find out if it is already installed and which version is installed.

* + Unix (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX, etc.)

 Win 9x/NT/2000

* + Macintosh (Intel, PPC, 68K)
  + OS/2
  + DOS (multiple versions)
  + PalmOS
  + Nokia mobile phones
  + Windows CE
  + Acorn/RISC OS
  + BeOS
  + Amiga
  + VMS/OpenVMS
  + QNX
  + VxWorks
  + Psion
  + Python has also been ported to the Java and .NET virtual machines

## Getting Python

The most up-to-date and current source code, binaries, documentation, news, etc., is available on the official website of Python: <http://www.python.org/>.

You can download Python documentation from [www.python.org/doc/](http://www.python.org/doc/). The documentation is available in HTML, PDF, and PostScript formats.

**Installing Python**

Python distribution is available for a wide variety of platforms. You need to download only the binary code applicable for your platform and install Python.

If the binary code for your platform is not available, you need a C compiler to compile the source code manually. Compiling the source code offers more flexibility in terms of choice of features that you require in your installation.

Here is a quick overview of installing Python on various platforms:

### Unix and Linux Installation

Here are the simple steps to install Python on Unix/Linux machine.

* Open a Web browser and go to [**http://www.python.org/download/**](http://www.python.org/download/).
* Follow the link to download zipped source code available for Unix/Linux.
* Download and extract files.
* Editing the *Modules/Setup* file if you want to customize some options.
* **run** ./configure script
* make
* make install

This installs Python at standard location */usr/local/bin* and its libraries at

*/usr/local/lib/pythonXX* where XX is the version of Python.

### Windows Installation

Here are the steps to install Python on Windows machine.

* Open a Web browser and go to [**http://www.python.org/download/**](http://www.python.org/download/)
* Follow the link for the Windows installer *python-XYZ.msi* file where XYZ is the version you need to install.
* To use this installer *python-XYZ.msi*, the Windows system must support Microsoft Installer 2.0. Save the installer file to your local machine and then run it to find out if your machine supports MSI.
* Run the downloaded file. This brings up the Python install wizard, which is really easy to use. Just accept the default settings, wait until the install is finished, and you are done.

### Macintosh Installation

Recent Macs come with Python installed, but it may be several years out of date. See <http://www.python.org/download/mac/>for instructions on getting the current

version along with extra tools to support development on the Mac. For older Mac OS's before Mac OS X 10.3 (released in 2003), MacPython is available.

Jack Jansen maintains it and you can have full access to the entire documentation at his website - [http://www.cwi.nl/~jack/macpython.html](http://www.cwi.nl/%7Ejack/macpython.html). You can find complete installation details for Mac OS installation.

## Setting up PATH

Programs and other executable files can be in many directories, so operating systems provide a search path that lists the directories that the OS searches for executables.

The path is stored in an environment variable, which is a named string maintained by the operating system. This variable contains information available to the command shell and other programs.

The **path** variable is named as PATH in Unix or Path in Windows (Unix is case- sensitive; Windows is not).

In Mac OS, the installer handles the path details. To invoke the Python interpreter from any particular directory, you must add the Python directory to your path.

## Setting path at Unix/Linux

To add the Python directory to the path for a particular session in Unix:

* **In the csh shell:** type setenv PATH "$PATH:/usr/local/bin/python" and press Enter.
* **In the bash shell (Linux):** type export ATH="$PATH:/usr/local/bin/python" and press Enter.
* **In the sh or ksh shell:** type PATH="$PATH:/usr/local/bin/python" and press Enter.
* **Note:** /usr/local/bin/python is the path of the Python directory

## Setting path at Windows

To add the Python directory to the path for a particular session in Windows: **At the command prompt:** type path %path%;C:\Python and press Enter. **Note:** C:\Python is the path of the Python directory

## Python Environment Variables

Here are important environment variables, which can be recognized by Python:

|  |  |
| --- | --- |
| **Variable** | **Description** |
| PYTHONPATH | It has a role similar to PATH. This variable tells the Python interpreter where to locate the module files imported into a program. It should include the Python source library directory and the directories containing Python source code. PYTHONPATH is sometimes preset by the Python installer. |
| PYTHONSTARTUP | It contains the path of an initialization file containing Python source code. It is executed every time you start the interpreter. It is named as .pythonrc.py in Unix and it contains commands that load utilities or modify PYTHONPATH. |
| PYTHONCASEOK | It is used in Windows to instruct Python to find the first case- insensitive match in an import statement. Set this variable to any value to activate it. |
| PYTHONHOME | It is an alternative module search path. It is usually embedded in the PYTHONSTARTUP or PYTHONPATH directories to make switching module libraries easy. |

## Running Python

There are three different ways to start Python:

### Interactive Interpreter

You can start Python from Unix, DOS, or any other system that provides you a command-line interpreter or shell window.

Enter **python** the command line.

Start coding right away in the interactive interpreter.

$python

# Unix/Linux

or

python%

# Unix/Linux

or

C:>python

# Windows/DOS

Here is the list of all the available command line options:

|  |  |
| --- | --- |
| **Option** | **Description** |
| -d | It provides debug output. |
| -O | It generates optimized bytecode (resulting in .pyo files). |
| -S | Do not run import site to look for Python paths on startup. |
| -v | verbose output (detailed trace on import statements). |
| -X | disable class-based built-in exceptions (just use strings); obsolete starting with version 1.6. |
| -c cmd | run Python script sent in as cmd string |
| file | run Python script from given file |

### Script from the Command-line

A Python script can be executed at command line by invoking the interpreter on your application, as in the following:

|  |  |  |
| --- | --- | --- |
| $python script.py | # Unix/Linuxor |  |
| python% script.py Windows/DOS | # Unix/Linuxor C:>python script.py | # |

**Note:** Be sure the file permission mode allows execution.

### Integrated Development Environment

You can run Python from a Graphical User Interface (GUI) environment as well, if you have a GUI application on your system that supports Python.

* + **Unix:** IDLE is the very first Unix IDE for Python.
  + **Windows:** PythonWin is the first Windows interface for Python and is an IDE with a GUI.
* **Macintosh:** The Macintosh version of Python along with the IDLE IDE is available from the main website, downloadable as either MacBinary or BinHex'd files.

If you are not able to set up the environment properly, then you can take help from your system admin. Make sure the Python environment is properly set up and working perfectly fine.

**Note:** All the examples given in subsequent chapters are executed with Python 2.4.3 version available on CentOS flavor of Linux.

We already have set up Python Programming environment online, so that you can execute all the available examples online at the same time when you are learning theory. Feel free to modify any example and execute it online.

# BASIC SYNTAX

Python

The Python language has many similarities to Perl, C, and Java. However, there are some definite differences between the languages.

## First Python Program

Let us execute programs in different modes of programming.

### Interactive Mode Programming:

Invoking the interpreter without passing a script file as a parameter brings up the following prompt:

$ python

Python 2.4.3 (#1, Nov 11 2010, 13:34:43)

[GCC 4.1.2 20080704 (Red Hat 4.1.2-48)] on linux2

Type "help", "copyright", "credits" or "license" for more information.

>>>

Type the following text at the Python prompt and press the Enter:

>>> print "Hello, Python!";

If you are running new version of Python, then you need to use print statement with parenthesis as in **print ("Hello, Python!");**. However in Python version 2.4.3, this produces the following result:

Hello, Python!

### Script Mode Programming

Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

Let us write a simple Python program in a script. Python files have extension **.py**. Type the following source code in a test.py file:

print "Hello, Python!";

We assume that you have Python interpreter set in PATH variable. Now, try to run this program as follows:

$ python test.py

This produces the following result:

Hello, Python!

Let us try another way to execute a Python script. Here is the modified test.py file:

#!/usr/bin/python

print "Hello, Python!";

We assume that you have Python interpreter available in /usr/bin directory. Now, try to run this program as follows:

$ chmod +x test.py

# This is to make file executable

$./test.py

This produces the following result:

Hello, Python!

## Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module, or other object. An identifier starts with a letter A to Z or a to z, or an underscore (\_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Here are naming conventions for Python identifiers:

* + Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
  + Starting an identifier with a single leading underscore indicates that the identifier is private.
* Starting an identifier with two leading underscores indicates a strongly private identifier.
* If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

## Python Keywords

The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

|  |  |  |
| --- | --- | --- |
| And | exec | Not |
| Assert | finally | or |
| Break | for | pass |
| Class | from | print |
| Continue | global | raise |
| def | if | return |
| del | import | try |
| elif | in | while |
| else | is | with |
| except | lambda | yield |

## Lines and Indentation

Python provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. For example:

if True:

print "True" else:

print "False"

However, the following block generates an error:

if True:

print "Answer" print "True"

else:

print "Answer" print "False"

Thus, in Python all the continuous lines indented with same number of spaces would form a block. The following example has various statement blocks:

**Note:** Do not try to understand the logic at this point of time. Just make sure you understood various blocks even if they are without braces.

#!/usr/bin/python

import sys

try:

# open file stream

file = open(file\_name, "w") except IOError:

print "There was an error writing to", file\_name

sys.exit()

print "Enter '", file\_finish, print "' When finished"

while file\_text != file\_finish: file\_text = raw\_input("Enter text: ") if file\_text == file\_finish:

# close the file file.close break

file.write(file\_text) file.write("\n")

file.close()

file\_name = raw\_input("Enter filename: ") if len(file\_name) == 0:

print "Next time please enter something" sys.exit()

try:

file = open(file\_name, "r") except IOError:

print "There was an error reading file" sys.exit()

file\_text = file.read() file.close()

print file\_text

## Multi-Line Statements

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example:

total = item\_one + \

item\_two + \ item\_three

Statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example:

days = ['Monday', 'Tuesday', 'Wednesday',

'Thursday', 'Friday']

## Quotation in Python

Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes are used to span the string across multiple lines. For example, all the following are legal:

word = 'word'

sentence = "This is a sentence." paragraph = """This is a paragraph. It is

made up of multiple lines and sentences."""

## Comments in Python

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.

#!/usr/bin/python

# First comment

print "Hello, Python!"; # second comment

This produces the following result:

Hello, Python!

You can type a comment on the same line after a statement or expression:

name = "Madisetti" # This is again comment

You can comment multiple lines as follows:

# This is a comment.

# This is a comment, too.

# This is a comment, too.

# I said that already.

## Using Blank Lines

A line containing only whitespace, possibly with a comment, is known as a blank line and Python totally ignores it.

In an interactive interpreter session, you must enter an empty physical line to terminate a multiline statement.

## Waiting for the User

The following line of the program displays the prompt, the statement saying “Press the enter key to exit”, and waits for the user to take action:

#!/usr/bin/python

raw\_input("\n\nPress the enter key to exit.")

Here, "\n\n" is used to create two new lines before displaying the actual line. Once the user presses the key, the program ends. This is a nice trick to keep a console window open until the user is done with an application.

## Multiple Statements on a Single Line

The semicolon ( ; ) allows multiple statements on the single line given that neither statement starts a new code block. Here is a sample snip using the semicolon:

import sys; x = 'foo'; sys.stdout.write(x + '\n')

## Multiple Statement Groups as Suites

A group of individual statements, which make a single code block are called **suites** in Python. Compound or complex statements, such as if, while, def, and class require a header line and a suite.

Header lines begin the statement (with the keyword) and terminate with a colon ( :

) and are followed by one or more lines which make up the suite. For example:

if expression : suite

elif expression : suite

else :

suite

## Command Line Arguments

Many programs can be run to provide you with some basic information about how they should be run. Python enables you to do this with -h:

$ python -h

usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ... Options and arguments (and corresponding environment variables):

-c cmd : program passed in as string (terminates option list)

-d

: debug output from parser (also PYTHONDEBUG=x)

-E

: ignore environment variables (such as PYTHONPATH)

-h

: print this help message and exit

[ etc. ]

You can also program your script in such a way that it should accept various options.

## Accessing Command-Line Arguments

Python provides a **getopt** module that helps you parse command-line options and arguments.

$ python test.py arg1 arg2 arg3

The Python **sys** module provides access to any command-line arguments via the **sys.argv**. This serves two purposes:

* + sys.argv is the list of command-line arguments.
  + len(sys.argv) is the number of command-line arguments. Here sys.argv[0] is the program i.e. script name.

### Example

Consider the following script test.py:

#!/usr/bin/python

import sys

print 'Number of arguments:', len(sys.argv), 'arguments.' print 'Argument List:', str(sys.argv)

Now run above script as follows:

$ python test.py arg1 arg2 arg3

This produces the following result:

Number of arguments: 4 arguments.

Argument List: ['test.py', 'arg1', 'arg2', 'arg3']

**NOTE:** As mentioned above, first argument is always script name and it is also being counted in number of arguments.

## Parsing Command-Line Arguments

Python provided a **getopt** module that helps you parse command-line options and arguments. This module provides two functions and an exception to enable command line argument parsing.

## getopt.getopt method

This method parses command line options and parameter list. Following is simple syntax for this method:

getopt.getopt(args, options[, long\_options])

Here is the detail of the parameters:

* + **args**: This is the argument list to be parsed.
  + **options**: This is the string of option letters that the script wants to recognize, with options that require an argument should be followed by a colon (:).
  + **long\_options**: This is optional parameter and if specified, must be a list of strings with the names of the long options, which should be supported. Long options, which require an argument should be followed by an equal sign ('='). To accept only long options, options should be an empty string.
  + This method returns value consisting of two elements: the first is a list of **(option, value)** pairs. The second is the list of program arguments left after the option list was stripped.
  + Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options (e.g., '-x') or two hyphens for long options (e.g., '--long-option').

## Exception getopt.GetoptError:

This is raised when an unrecognized option is found in the argument list or when an option requiring an argument is given none.

The argument to the exception is a string indicating the cause of the error. The attributes **msg** and **opt** give the error message and related option.

### Example

Consider we want to pass two file names through command line and we also want to give an option to check the usage of the script. Usage of the script is as follows:

usage: test.py -i <inputfile> -o <outputfile>

Here is the following script to test.py:

#!/usr/bin/python

import sys, getopt

def main(argv): inputfile = '' outputfile = '' try:

opts, args = getopt.getopt(argv,"hi:o:",["ifile=","ofile="]) except getopt.GetoptError:

print 'test.py -i <inputfile> -o <outputfile>' sys.exit(2)

for opt, arg in opts: if opt == '-h':

print 'test.py -i <inputfile> -o <outputfile>' sys.exit()

elif opt in ("-i", "--ifile"): inputfile = arg

elif opt in ("-o", "--ofile"): outputfile = arg

print 'Input file is "', inputfile

print 'Output file is "', outputfile

if name == " main ": main(sys.argv[1:])

Now, run above script as follows:

$ test.py -h

usage: test.py -i <inputfile> -o <outputfile>

$ test.py -i BMP -o

usage: test.py -i <inputfile> -o <outputfile>

$ test.py -i inputfile Input file is " inputfile Output file is "

# VARIABLE TYPES

Python

Variables are nothing but reserved memory locations to store values. This means when you create a variable, you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

## Assigning Values to Variables

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. For example:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| miles | = | 1000.0 | # | A | floating point |
| name | = | "John" | # | A | string |

Here, 100, 1000.0, and "John" are the values assigned to *counter*, *miles,* and *name* variables respectively. This produces the following result:

#!/usr/bin/python

counter = 100

# An integer assignment

print counter

print miles print name

100

1000.0

John

## Multiple Assignment

Python allows you to assign a single value to several variables simultaneously. For example:

a = b = c = 1

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example:

a, b, c = 1, 2, "john"

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

## Standard Data Types

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types:

* + Numbers
  + String
  + List
  + Tuple
  + Dictionary

## Python Numbers

Number data types store numeric values. Number objects are created when you assign a value to them. For example:

var1 = 1

var2 = 10

You can also delete the reference to a number object by using the **del** statement. The syntax of the del statement is:

del var1[,var2[,var3[....,varN]]]]

You can delete a single object or multiple objects by using the del statement. For example:

del var

del var\_a, var\_b

Python supports four different numerical types:

* + int (signed integers)
  + long (long integers, they can also be represented in octal and hexadecimal)
  + float (floating point real values)
  + complex (complex numbers)

### Examples

Here are some examples of numbers:

|  |  |  |  |
| --- | --- | --- | --- |
| **int** | **long** | **Float** | **complex** |
| 10 | 51924361L | 0.0 | 3.14j |
| 100 | -0x19323L | 15.20 | 45.j |
| -786 | 0122L | -21.9 | 9.322e-36j |
| 080 | 0xDEFABCECBDAECBFBAEl | 32.3+e18 | .876j |
| -0490 | 535633629843L | -90. | -.6545+0J |
| -0x260 | -052318172735L | -32.54e100 | 3e+26J |
| 0x69 | -4721885298529L | 70.2-E12 | 4.53e-7j |

* + Python allows you to use a lowercase L with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.
  + A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x is the real part and b is the imaginary part of the complex number.

## Python Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:] ) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

The plus (+) sign is the string concatenation operator and the asterisk (\*) is the repetition operator. For example:

#!/usr/bin/python

str = 'Hello World!'

print str

# Prints complete string

print str[0]

# Prints first character of the string

print str[2:5]

# Prints characters starting from 3rd to 5th

print str[2:]

# Prints string starting from 3rd character

print str \* 2

# Prints string two times

print str + "TEST" # Prints concatenated string

This will produce the following result:

Hello World! H

llo

llo World!

Hello World!Hello World!

Hello World!TEST

## Python Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (\*) is the repetition operator. For example:

#!/usr/bin/python

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]

tinylist = [123, 'john']

print list

# Prints complete list

print list[0]

# Prints first element of the list

print list[1:3]

# Prints elements starting from 2nd till 3rd

print list[2:]

# Prints elements starting from 3rd element

print tinylist \* 2 # Prints list two times

print list + tinylist # Prints concatenated lists

This produces the following result:

['abcd', 786, 2.23, 'john', 70.200000000000003]

abcd

[786, 2.23]

[2.23, 'john', 70.200000000000003]

[123, 'john', 123, 'john']

['abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john']

## Python Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

The main differences between lists and tuples are: Lists are enclosed in brackets ( [

] ) and their elements and size can be changed, while tuples are enclosed in parentheses ( ( ) ) and cannot be updated. Tuples can be thought of as **read- only** lists. For example:

#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )

tinytuple = (123, 'john')

print tuple

# Prints complete list

print tuple[0]

# Prints first element of the list

print tuple[1:3]

# Prints elements starting from 2nd till 3rd

print tuple[2:]

# Prints elements starting from 3rd element

print tinytuple \* 2 # Prints list two times

print tuple + tinytuple # Prints concatenated lists

This produces the following result:

('abcd', 786, 2.23, 'john', 70.200000000000003)

abcd

(786, 2.23)

(2.23, 'john', 70.200000000000003)

(123, 'john', 123, 'john')

('abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john')

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists:

#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]

tuple[2] = 1000

# Invalid syntax with tuple

list[2] = 1000

# Valid syntax with list

## Python Dictionary

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). For example:

#!/usr/bin/python

dict = {}

dict['one'] = "This is one"

dict[2]

= "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}

print dict['one']

# Prints value for 'one' key

print dict[2]

# Prints value for 2 key

print tinydict

# Prints complete dictionary

print tinydict.keys() # Prints all the keys

print tinydict.values() # Prints all the values

This produces the following result:

This is one This is two

{'dept': 'sales', 'code': 6734, 'name': 'john'} ['dept', 'code', 'name']

['sales', 6734, 'john']

Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.

## Data Type Conversion

Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type name as a function.

There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

|  |  |
| --- | --- |
| **Function** | **Description** |
| int(x [,base]) | Converts x to an integer. base specifies the base if x is a string. |
| long(x [,base] ) | Converts x to a long integer. base specifies the base if x is a string. |
| float(x) | Converts x to a floating-point number. |
| complex(real [,imag]) | Creates a complex number. |
| str(x) | Converts object x to a string representation. |
| repr(x) | Converts object x to an expression string. |
| eval(str) | Evaluates a string and returns an object. |
| tuple(s) | Converts s to a tuple. |

|  |  |  |  |
| --- | --- | --- | --- |
| list(s) | Converts s to a list. | | |
| set(s) | Converts s to a set. | | |
| dict(d) | Creates a dictionary. d must be a sequence tuples. | of | (key,value) |
| frozenset(s) | Converts s to a frozen set. | | |
| chr(x) | Converts an integer to a character. | | |
| unichr(x) | Converts an integer to a Unicode character. | | |
| ord(x) | Converts a single character to its integer value. | | |
| hex(x) | Converts an integer to a hexadecimal string. | | |
| oct(x) | Converts an integer to an octal string. | | |

# BASIC OPERATORS

Python

Operators are the constructs which can manipulate the value of operands.

Consider the expression *4 + 5 = 9.* Here, 4 and 5 are called operands and + is called operator.

## Types of Operators

Python language supports the following types of operators.

* + - Arithmetic Operators
    - Comparison (Relational) Operators
    - Assignment Operators
    - Logical Operators
    - Bitwise Operators
    - Membership Operators
    - Identity Operators

Let us have a look on all operators one by one.

## Python Arithmetic Operators

Assume variable a holds 10 and variable b holds 20, then:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + Addition | Adds values on either side of the operator. | a + b = 30 |
| - Subtraction | Subtracts right hand operand from left hand operand. | a – b = -10 |
| \* Multiplication | Multiplies values on either side of the operator | a \* b = 200 |

|  |  |  |
| --- | --- | --- |
| / Division | Divides left hand operand by right hand operand | b / a = 2 |
| % Modulus | Divides left hand operand by right hand operand and returns remainder | b % a = 0 |
| \*\* Exponent | Performs exponential (power) calculation on operators | a\*\*b =10 to the power 20 |
| // | Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. | 9//2 = 4 and 9.0//2.0 = 4.0 |

### Example

Assume variable a holds 10 and variable b holds 20, then:

#!/usr/bin/python

a = 21

b = 10

c = 0

c = a + b

print "Line 1 - Value of c is ", c c = a - b

print "Line 2 - Value of c is ", c c = a \* b

print "Line 3 - Value of c is ", c c = a / b

print "Line 4 - Value of c is ", c

c = a % b

print "Line 5 - Value of c is ", c

a = 2

b = 3

c = a\*\*b

print "Line 6 - Value of c is ", c

a = 10

b = 5

c = a//b

print "Line 7 - Value of c is ", c

When you execute the above program, it produces the following result:

Line 1 - Value of c is 31 Line 2 - Value of c is 11 Line 3 - Value of c is 210 Line 4 - Value of c is 2 Line 5 - Value of c is 1 Line 6 - Value of c is 8

Line 7 - Value of c is 2

## Python Comparison Operators

These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators.

Assume variable a holds 10 and variable b holds 20, then:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |

|  |  |  |
| --- | --- | --- |
| == | If the values of two operands are equal, then the condition becomes true. | (a == b) is not true. |
| != | If values of two operands are not equal, then condition becomes true. | (a != b) is true. |
| <> | If values of two operands are not equal, then condition becomes true. | (a <> b) is true. This is similar to != operator. |
| > | If the value of left operand is greater than the value of right operand, then condition becomes true. | (a > b) is not true. |
| < | If the value of left operand is less than the value of right operand, then condition becomes true. | (a < b) is true. |
| >= | If the value of left operand is greater than or equal to the value of right operand, then condition becomes true. | (a >= b) is not true. |
| <= | If the value of left operand is less than or equal to the value of right operand, then condition becomes true. | (a <= b) is true. |

### Example

Assume variable a holds 10 and variable b holds 20, then:

#!/usr/bin/python

a = 21

b = 10

c = 0

if ( a == b ):

print "Line 1 - a is equal to b"

else:

print "Line 1 - a is not equal to b"

if ( a != b ):

print "Line 2 - a is not equal to b" else:

print "Line 2 - a is equal to b"

if ( a <> b ):

print "Line 3 - a is not equal to b" else:

print "Line 3 - a is equal to b"

if ( a < b ):

print "Line 4 - a is less than b" else:

print "Line 4 - a is not less than b"

if ( a > b ):

print "Line 5 - a is greater than b" else:

print "Line 5 - a is not greater than b"

a = 5;

b = 20;

if ( a <= b ):

print "Line 6 - a is either less than or equal to b" else:

print "Line 6 - a is neither less than nor equal to b"

if ( b >= a ):

print "Line 7 - b is either greater than or equal to b" else:

print "Line 7 - b is neither greater than nor equal to b"

When you execute the above program it produces the following result:

Line 1 - a is not equal to b Line 2 - a is not equal to b Line 3 - a is not equal to b Line 4 - a is not less than b Line 5 - a is greater than b

Line 6 - a is either less than or equal to b

Line 7 - b is either greater than or equal to b

## Python Assignment Operators

Assume variable a holds 10 and variable b holds 20, then:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Assigns values from right side operands to left side operand | c = a + b assigns value of a + b into c |
| +=  Add AND | It adds right operand to the left operand and assign the result to left operand | c += a is equivalent to c = c + a |

|  |  |  |
| --- | --- | --- |
| -=  Subtract AND | It subtracts right operand from the left operand and assign the result to left operand | c -= a is equivalent to c = c - a |
| \*=  Multiply AND | It multiplies right operand with the left operand and assign the result to left operand | c \*= a is equivalent to c = c \* a |
| /=  Divide AND | It divides left operand with the right operand and assign the result to left operand | c /= a is equivalent to c = c / a |
| %=  Modulus AND | It takes modulus using two operands and assign the result to left operand | c %= a is equivalent to c = c % a |
| \*\*=  Exponent AND | Performs exponential (power) calculation on operators and assign value to the left operand | c \*\*= a is equivalent to c = c \*\* a |
| //=  Floor Division | It performs floor division on operators and assign value to the left operand | c //= a is equivalent to c = c // a |

### Example

Assume variable a holds 10 and variable b holds 20, then:

#!/usr/bin/python

a = 21

b = 10

c = 0

c = a + b

print "Line 1 - Value of c is ", c

c += a

print "Line 2 - Value of c is ", c

c \*= a

print "Line 3 - Value of c is ", c

c /= a

print "Line 4 - Value of c is ", c

c = 2 c %= a

print "Line 5 - Value of c is ", c

c \*\*= a

print "Line 6 - Value of c is ", c

c //= a

print "Line 7 - Value of c is ", c

When you execute the above program, it produces the following result:

Line 1 - Value of c is 31 Line 2 - Value of c is 52 Line 3 - Value of c is 1092 Line 4 - Value of c is 52 Line 5 - Value of c is 2

Line 6 - Value of c is 2097152

Line 7 - Value of c is 99864