# Python - Overview

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, and 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.
- **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.

## Python - Environment Setup

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 https://www.python.org/

You can download Python documentation from https://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 https://www.python.org/downloads/.
- 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 https://www.python.org/downloads/.
- 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. 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 "$PATH:/usr/local/bin/python" and press Enter.
- **In the bash shell (Linux)** − type export PATH="$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 −

| Sr.No. | Variable & Description |
|---|---|
| 1 | **PYTHONPATH**<br>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. |
| 2 | **PYTHONSTARTUP**<br>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. |
| 3 | **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. |
| 4 | **PYTHONHOME**<br><br>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 −

| Sr.No. | Option & Description |
|---|---|
| 1 | **-d**<br><br>It provides debug output. |
| 2 | **-O**<br><br>It generates optimized bytecode (resulting in .pyo files). |
| 3 | **-S**<br><br>Do not run import site to look for Python paths on startup. |
| 4 | **-v** |

| | |
|---|---|
| | verbose output (detailed trace on import statements). |
| 5 | **-X**<br><br>disable class-based built-in exceptions (just use strings); obsolete starting with version 1.6. |
| 6 | **-c cmd**<br><br>run Python script sent in as cmd string |
| 7 | **file**<br><br>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/Linux

or

python% script.py # Unix/Linux

or

C: >python script.py # Windows/DOS
```

**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.

## Python - Basic Syntax

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 would 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.

## Reserved Words

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.
```

Following triple-quoted string is also ignored by Python interpreter and can be used as a multiline comments:

```
'''
This is a multiline
comment.
'''
```

## 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. Command Line Arguments is an advanced topic and should be studied a bit later once you have gone through rest of the Python concepts.

## Python - Variable Types

Variables are nothing but reserved memory locations to store values. This means that 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 −

```
#!/usr/bin/python

counter = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "John"       # A string

print counter
print miles
print name
```

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

```
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 and y are the real numbers and j is the imaginary unit.

## 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 produce the following result −

```
['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 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 produce the following result −

```
('abcd', 786, 2.23, 'john', 70.2)
abcd
(786, 2.23)
(2.23, 'john', 70.2)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.2, 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 produce 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.

| Sr.No. | Function & Description |
|---|---|
| 1 | **int(x [,base])**<br><br>Converts x to an integer. base specifies the base if x is a string. |
| 2 | **long(x [,base] )**<br><br>Converts x to a long integer. base specifies the base if x is a string. |
| 3 | **float(x)**<br><br>Converts x to a floating-point number. |
| 4 | **complex(real [,imag])**<br><br>Creates a complex number. |
| 5 | **str(x)**<br><br>Converts object x to a string representation. |

| | |
|---|---|
| 6 | **repr(x)**<br><br>Converts object x to an expression string. |
| 7 | **eval(str)**<br><br>Evaluates a string and returns an object. |
| 8 | **tuple(s)**<br><br>Converts s to a tuple. |
| 9 | **list(s)**<br><br>Converts s to a list. |
| 10 | **set(s)**<br><br>Converts s to a set. |
| 11 | **dict(d)**<br><br>Creates a dictionary. d must be a sequence of (key,value) tuples. |
| 12 | **frozenset(s)**<br><br>Converts s to a frozen set. |
| 13 | **chr(x)**<br><br>Converts an integer to a character. |
| 14 | **unichr(x)**<br><br>Converts an integer to a Unicode character. |
| 15 | **ord(x)**<br><br>Converts a single character to its integer value. |
| 16 | **hex(x)**<br><br>Converts an integer to a hexadecimal string. |

|  |  |
|---|---|
| 17 | **oct(x)**<br><br>Converts an integer to an octal string. |

## Python - Basic Operators

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 Operator

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 −

[ Show Example ]

| 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. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) − | 9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0 |

## 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 −

[ Show Example ]

| **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. |

## Python Assignment Operators

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

[ Show Example ]

| 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 / ac /= 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 |

## Python Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows −

a = 0011 1100

b = 0000 1101

-----------------

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a  = 1100 0011

There are following Bitwise operators supported by Python language

[ Show Example ]

| Operator | Description | Example |
|---|---|---|
| & Binary AND | Operator copies a bit to the result if it exists in both operands | (a & b) (means 0000 1100) |

| | | |
|---|---|---|
| \| Binary OR | It copies a bit if it exists in either operand. | (a \| b) = 61 (means 0011 1101) |
| ^ Binary XOR | It copies the bit if it is set in one operand but not both. | (a ^ b) = 49 (means 0011 0001) |
| ~ Binary Ones Complement | It is unary and has the effect of 'flipping' bits. | (~a ) = -61 (means 1100 0011 in 2's complement form due to a signed binary number. |
| << Binary Left Shift | The left operands value is moved left by the number of bits specified by the right operand. | a << 2 = 240 (means 1111 0000) |
| >> Binary Right Shift | The left operands value is moved right by the number of bits specified by the right operand. | a >> 2 = 15 (means 0000 1111) |

## Python Logical Operators

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then

[ Show Example ]

| Operator | Description | Example |
|---|---|---|
| and Logical AND | If both the operands are true then condition becomes true. | (a and b) is true. |
| or Logical OR | If any of the two operands are non-zero then condition becomes true. | (a or b) is true. |

| | | |
|---|---|---|
| not Logical NOT | Used to reverse the logical state of its operand. | Not(a and b) is false. |

Used to reverse the logical state of its operand.

## Python Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below −

[ Show Example ]

| **Operator** | **Description** | **Example** |
|---|---|---|
| in | Evaluates to true if it finds a variable in the specified sequence and false otherwise. | x in y, here in results in a 1 if x is a member of sequence y. |
| not in | Evaluates to true if it does not finds a variable in the specified sequence and false otherwise. | x not in y, here not in results in a 1 if x is not a member of sequence y. |

## Python Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators explained below −

[ Show Example ]

| **Operator** | **Description** | **Example** |
|---|---|---|
| is | Evaluates to true if the variables on either side of the operator point to the same object and false otherwise. | x is y, here **is** results |

| | | |
|---|---|---|
| | | in 1 if id(x) equals id(y). |
| is not | Evaluates to false if the variables on either side of the operator point to the same object and true otherwise. | x is not y, here **is not** results in 1 if id(x) is not equal to id(y). |

## Python Operators Precedence

The following table lists all operators from highest precedence to lowest.

[ Show Example ]

| Sr.No. | Operator & Description |
|---|---|
| 1 | ******<br><br>Exponentiation (raise to the power) |
| 2 | **~ + -**<br><br>Complement, unary plus and minus (method names for the last two are +@ and -@) |
| 3 | *** / % //**<br><br>Multiply, divide, modulo and floor division |
| 4 | **+ -**<br><br>Addition and subtraction |
| 5 | **>> <<**<br><br>Right and left bitwise shift |
| 6 | **&**<br><br>Bitwise 'AND' |
| 7 | **^ \|** |

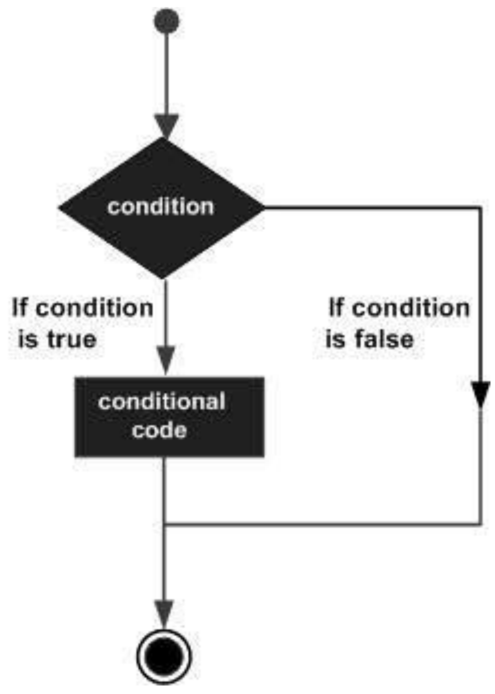
|  |  |
|---|---|
|  | Bitwise exclusive `OR' and regular `OR' |
| 8 | **<= < > >=**<br><br>Comparison operators |
| 9 | **<> == !=**<br><br>Equality operators |
| 10 | **= %= /= //= -= += *= **=**<br><br>Assignment operators |
| 11 | **is is not**<br><br>Identity operators |
| 12 | **in not in**<br><br>Membership operators |
| 13 | **not or and**<br><br>Logical operators |

## Python - Decision Making

Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.

Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome. You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.

Following is the general form of a typical decision making structure found in most of the programming languages −



Python programming language assumes any **non-zero** and **non-null** values as TRUE, and if it is either **zero** or **null**, then it is assumed as FALSE value.

Python programming language provides following types of decision making statements. Click the following links to check their detail.

| Sr.No. | Statement & Description |
|---|---|
| 1 | **if statements**<br><br>An **if statement** consists of a boolean expression followed by one or more statements. |
| 2 | **if...else statements**<br><br>An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is FALSE. |
| 3 | **nested if statements** |

| | |
|---|---|
| | You can use one **if** or **else if** statement inside another **if** or **else if**statement(s). |

Let us go through each decision making briefly −

## Single Statement Suites

If the suite of an **if** clause consists only of a single line, it may go on the same line as the header statement.

Here is an example of a **one-line if** clause −

```
#!/usr/bin/python


var = 100

if ( var == 100 ) : print "Value of expression is 100"

print "Good bye!"
```

When the above code is executed, it produces the following result −

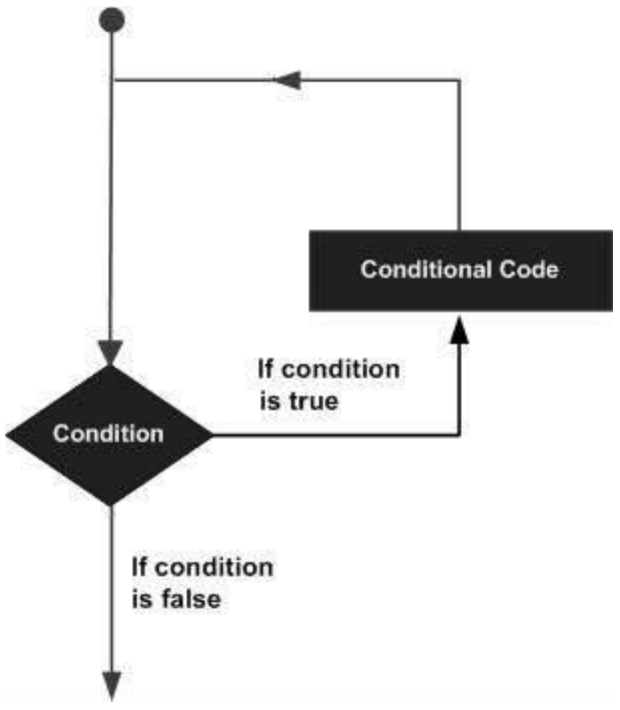
```
Value of expression is 100
Good bye!
```

## Python – Loops

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement −



Python programming language provides following types of loops to handle looping requirements.

| Sr.No. | Loop Type & Description |
|---|---|
| 1 | **while loop**<br>Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body. |
| 2 | **for loop**<br>Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. |

| | |
|---|---|
| 3 | **nested loops**<br><br>You can use one or more loop inside any another while, for or do..while loop. |

## Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Python supports the following control statements. Click the following links to check their detail.

Let us go through the loop control statements briefly

| Sr.No. | Control Statement & Description |
|---|---|
| 1 | **break statement**<br><br>Terminates the loop statement and transfers execution to the statement immediately following the loop. |
| 2 | **continue statement**<br><br>Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |
| 3 | **pass statement**<br><br>The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute. |

## Python - Numbers

Number data types store numeric values. They are immutable data types, means that changing the value of a number data type results in a newly allocated object.

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)** − They are often called just integers or ints, are positive or negative whole numbers with no decimal point.
- **long (long integers )** − Also called longs, they are integers of unlimited size, written like integers and followed by an uppercase or lowercase L.
- **float (floating point real values)** − Also called floats, they represent real numbers and are written with a decimal point dividing the integer and fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 (2.5e2 = 2.5 x $10^2$ = 250).
- **complex (complex numbers)** − are of the form a + bJ, where a and b are floats and J (or j) represents the square root of -1 (which is an imaginary number). The real part of the number is a, and the imaginary part is b. Complex numbers are not used much in Python programming.

### 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 a + bj, where a is the real part and b is the imaginary part of the complex number.

## Number Type Conversion

Python converts numbers internally in an expression containing mixed types to a common type for evaluation. But sometimes, you need to coerce a number explicitly from one type to another to satisfy the requirements of an operator or function parameter.

- Type **int(x)** to convert x to a plain integer.
- Type **long(x)** to convert x to a long integer.
- Type **float(x)** to convert x to a floating-point number.
- Type **complex(x)** to convert x to a complex number with real part x and imaginary part zero.
- Type **complex(x, y)** to convert x and y to a complex number with real part x and imaginary part y. x and y are numeric expressions

## Mathematical Functions

Python includes following functions that perform mathematical calculations.

| Sr.No. | Function & Returns ( description ) |
|---|---|

| | |
|---|---|
| 1 | **abs(x)**<br><br>The absolute value of x: the (positive) distance between x and zero. |
| 2 | **ceil(x)**<br>The ceiling of x: the smallest integer not less than x |
| 3 | **cmp(x, y)**<br>-1 if x < y, 0 if x == y, or 1 if x > y |
| 4 | **exp(x)**<br>The exponential of x: $e^x$ |
| 5 | **fabs(x)**<br>The absolute value of x. |
| 6 | **floor(x)**<br>The floor of x: the largest integer not greater than x |
| 7 | **log(x)**<br>The natural logarithm of x, for x> 0 |
| 8 | **log10(x)**<br>The base-10 logarithm of x for x> 0. |
| 9 | **max(x1, x2,...)**<br>The largest of its arguments: the value closest to positive infinity |
| 10 | **min(x1, x2,...)**<br>The smallest of its arguments: the value closest to negative infinity |
| 11 | **modf(x)**<br>The fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float. |
| 12 | **pow(x, y)**<br>The value of x**y. |
| 13 | **round(x [,n])** |

| | |
|---|---|
| | **x** rounded to n digits from the decimal point. Python rounds away from zero as a tie-breaker: round(0.5) is 1.0 and round(-0.5) is -1.0. |
| 14 | **sqrt(x)**<br>The square root of x for x > 0 |

## Random Number Functions

Random numbers are used for games, simulations, testing, security, and privacy applications. Python includes following functions that are commonly used.

| Sr.No. | Function & Description |
|---|---|
| 1 | **choice(seq)**<br>A random item from a list, tuple, or string. |
| 2 | **randrange ([start,] stop [,step])**<br>A randomly selected element from range(start, stop, step) |
| 3 | **random()**<br>A random float r, such that 0 is less than or equal to r and r is less than 1 |
| 4 | **seed([x])**<br>Sets the integer starting value used in generating random numbers. Call this function before calling any other random module function. Returns None. |
| 5 | **shuffle(lst)**<br>Randomizes the items of a list in place. Returns None. |
| 6 | **uniform(x, y)**<br>A random float r, such that x is less than or equal to r and r is less than y |

## Trigonometric Functions

Python includes following functions that perform trigonometric calculations.

| Sr.No. | Function & Description |
|---|---|
| 1 | **acos(x)** |

| | |
|---|---|
| | Return the arc cosine of x, in radians. |
| 2 | **asin(x)**<br>Return the arc sine of x, in radians. |
| 3 | **atan(x)**<br>Return the arc tangent of x, in radians. |
| 4 | **atan2(y, x)**<br>Return atan(y / x), in radians. |
| 5 | **cos(x)**<br>Return the cosine of x radians. |
| 6 | **hypot(x, y)**<br>Return the Euclidean norm, sqrt(x*x + y*y). |
| 7 | **sin(x)**<br>Return the sine of x radians. |
| 8 | **tan(x)**<br>Return the tangent of x radians. |
| 9 | **degrees(x)**<br>Converts angle x from radians to degrees. |
| 10 | **radians(x)**<br>Converts angle x from degrees to radians. |

## Mathematical Constants

The module also defines two mathematical constants −

| Sr.No. | Constants & Description |
|---|---|
| 1 | **pi**<br>The mathematical constant pi. |
| 2 | **e** |

| | |
|---|---|
| | The mathematical constant e. |

## Python - Strings

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. Creating strings is as simple as assigning a value to a variable. For example −

```
var1 = 'Hello World!'

var2 = "Python Programming"
```

## Accessing Values in Strings

Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring. For example −

```
#!/usr/bin/python



var1 = 'Hello World!'

var2 = "Python Programming"



print "var1[0]: ", var1[0]

print "var2[1:5]: ", var2[1:5]
```

When the above code is executed, it produces the following result −

```
var1[0]:  H
var2[1:5]:  ytho
```

## Updating Strings

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether. For example −

```
#!/usr/bin/python
```

```
var1 = 'Hello World!'

print "Updated String :- ", var1[:6] + 'Python'
```

When the above code is executed, it produces the following result −

```
Updated String :-  Hello Python
```

## Escape Characters

Following table is a list of escape or non-printable characters that can be represented with backslash notation.

An escape character gets interpreted; in a single quoted as well as double quoted strings.

| Backslash notation | Hexadecimal character | Description |
|---|---|---|
| \a | 0x07 | Bell or alert |
| \b | 0x08 | Backspace |
| \cx | | Control-x |
| \C-x | | Control-x |
| \e | 0x1b | Escape |
| \f | 0x0c | Formfeed |
| \M-\C-x | | Meta-Control-x |
| \n | 0x0a | Newline |
| \nnn | | Octal notation, where n is in the range 0.7 |
| \r | 0x0d | Carriage return |

| | | |
|---|---|---|
| \s | 0x20 | Space |
| \t | 0x09 | Tab |
| \v | 0x0b | Vertical tab |
| \x | | Character x |
| \xnn | | Hexadecimal notation, where n is in the range 0.9, a.f, or A.F |

## String Special Operators

Assume string variable **a** holds 'Hello' and variable **b** holds 'Python', then −

| **Operator** | **Description** | **Example** |
|---|---|---|
| + | Concatenation - Adds values on either side of the operator | a + b will give HelloPython |
| * | Repetition - Creates new strings, concatenating multiple copies of the same string | a*2 will give -HelloHello |
| [] | Slice - Gives the character from the given index | a[1] will give e |
| [ : ] | Range Slice - Gives the characters from the given range | a[1:4] will give ell |
| in | Membership - Returns true if a character exists in the given string | H in a will give 1 |
| not in | Membership - Returns true if a character does not exist in the given string | M not in a will give 1 |
| r/R | Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with | print r'\n' prints \n and |

| | | |
|---|---|---|
| | the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark. | print R'\n'prints \n |
| % | Format - Performs String formatting | See at next section |

## String Formatting Operator

One of Python's coolest features is the string format operator %. This operator is unique to strings and makes up for the pack of having functions from C's printf() family. Following is a simple example −

```
#!/usr/bin/python


print "My name is %s and weight is %d kg!" % ('Zara', 21)
```

When the above code is executed, it produces the following result −

```
My name is Zara and weight is 21 kg!
```

Here is the list of complete set of symbols which can be used along with % −

| Format Symbol | Conversion |
|---|---|
| %c | character |
| %s | string conversion via str() prior to formatting |
| %i | signed decimal integer |
| %d | signed decimal integer |
| %u | unsigned decimal integer |
| %o | octal integer |

| | |
|---|---|
| %x | hexadecimal integer (lowercase letters) |
| %X | hexadecimal integer (UPPERcase letters) |
| %e | exponential notation (with lowercase 'e') |
| %E | exponential notation (with UPPERcase 'E') |
| %f | floating point real number |
| %g | the shorter of %f and %e |
| %G | the shorter of %f and %E |

Other supported symbols and functionality are listed in the following table −

| **Symbol** | **Functionality** |
|---|---|
| * | argument specifies width or precision |
| - | left justification |
| + | display the sign |
| <sp> | leave a blank space before a positive number |
| # | add the octal leading zero ( '0' ) or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used. |
| 0 | pad from left with zeros (instead of spaces) |
| % | '%%' leaves you with a single literal '%' |
| (var) | mapping variable (dictionary arguments) |

|  |  |
|---|---|
| m.n. | m is the minimum total width and n is the number of digits to display after the decimal point (if appl.) |

## Triple Quotes

Python's triple quotes comes to the rescue by allowing strings to span multiple lines, including verbatim NEWLINEs, TABs, and any other special characters.

The syntax for triple quotes consists of three consecutive **single or double**quotes.

```
#!/usr/bin/python


para_str = """this is a long string that is made up of

several lines and non-printable characters such as

TAB ( \t ) and they will show up that way when displayed.

NEWLINEs within the string, whether explicitly given like

this within the brackets [ \n ], or just a NEWLINE within

the variable assignment will also show up.

"""

print para_str
```

When the above code is executed, it produces the following result. Note how every single special character has been converted to its printed form, right down to the last NEWLINE at the end of the string between the "up." and closing triple quotes. Also note that NEWLINEs occur either with an explicit carriage return at the end of a line or its escape code (\n) −

```
this is a long string that is made up of
several lines and non-printable characters such as
TAB (     ) and they will show up that way when displayed.
NEWLINEs within the string, whether explicitly given like
this within the brackets [
 ], or just a NEWLINE within
the variable assignment will also show up.
```

Raw strings do not treat the backslash as a special character at all. Every character you put into a raw string stays the way you wrote it −

```
#!/usr/bin/python
```

```
print 'C:\\nowhere'
```

When the above code is executed, it produces the following result −

```
C:\nowhere
```

Now let's make use of raw string. We would put expression in **r'expression'**as follows −

```
#!/usr/bin/python


print r'C:\\nowhere'
```

When the above code is executed, it produces the following result −

```
C:\\nowhere
```

## Unicode String

Normal strings in Python are stored internally as 8-bit ASCII, while Unicode strings are stored as 16-bit Unicode. This allows for a more varied set of characters, including special characters from most languages in the world. I'll restrict my treatment of Unicode strings to the following −

```
#!/usr/bin/python


print u'Hello, world!'
```

When the above code is executed, it produces the following result −

```
Hello, world!
```

As you can see, Unicode strings use the prefix u, just as raw strings use the prefix r.

## Built-in String Methods

Python includes the following built-in methods to manipulate strings −

| Sr.No. | Methods with Description |
|---|---|
| 1 | **capitalize()** |

| | |
|---|---|
| | Capitalizes first letter of string |
| 2 | **center(width, fillchar)**<br><br>Returns a space-padded string with the original string centered to a total of width columns. |
| 3 | **count(str, beg= 0,end=len(string))**<br><br>Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given. |
| 4 | **decode(encoding='UTF-8',errors='strict')**<br><br>Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding. |
| 5 | **encode(encoding='UTF-8',errors='strict')**<br><br>Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'. |
| 6 | **endswith(suffix, beg=0, end=len(string))**<br><br>Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise. |
| 7 | **expandtabs(tabsize=8)**<br><br>Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided. |
| 8 | **find(str, beg=0 end=len(string))**<br><br>Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise. |
| 9 | **index(str, beg=0, end=len(string))**<br><br>Same as find(), but raises an exception if str not found. |
| 10 | **isalnum()** |

| | |
|---|---|
| | Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise. |
| 11 | **isalpha()**<br><br>Returns true if string has at least 1 character and all characters are alphabetic and false otherwise. |
| 12 | **isdigit()**<br><br>Returns true if string contains only digits and false otherwise. |
| 13 | **islower()**<br><br>Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise. |
| 14 | **isnumeric()**<br><br>Returns true if a unicode string contains only numeric characters and false otherwise. |
| 15 | **isspace()**<br><br>Returns true if string contains only whitespace characters and false otherwise. |
| 16 | **istitle()**<br><br>Returns true if string is properly "titlecased" and false otherwise. |
| 17 | **isupper()**<br><br>Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise. |
| 18 | **join(seq)**<br><br>Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string. |
| 19 | **len(string)**<br><br>Returns the length of the string |
| 20 | **ljust(width[, fillchar])** |

| | |
|---|---|
| | Returns a space-padded string with the original string left-justified to a total of width columns. |
| 21 | **lower()**<br><br>Converts all uppercase letters in string to lowercase. |
| 22 | **lstrip()**<br><br>Removes all leading whitespace in string. |
| 23 | **maketrans()**<br><br>Returns a translation table to be used in translate function. |
| 24 | **max(str)**<br><br>Returns the max alphabetical character from the string str. |
| 25 | **min(str)**<br><br>Returns the min alphabetical character from the string str. |
| 26 | **replace(old, new [, max])**<br><br>Replaces all occurrences of old in string with new or at most max occurrences if max given. |
| 27 | **rfind(str, beg=0,end=len(string))**<br><br>Same as find(), but search backwards in string. |
| 28 | **rindex( str, beg=0, end=len(string))**<br><br>Same as index(), but search backwards in string. |
| 29 | **rjust(width,[, fillchar])**<br><br>Returns a space-padded string with the original string right-justified to a total of width columns. |
| 30 | **rstrip()**<br><br>Removes all trailing whitespace of string. |

| | |
|---|---|
| 31 | **split(str="", num=string.count(str))**<br><br>Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given. |
| 32 | **splitlines( num=string.count('\n'))**<br><br>Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed. |
| 33 | **startswith(str, beg=0,end=len(string))**<br><br>Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise. |
| 34 | **strip([chars])**<br><br>Performs both lstrip() and rstrip() on string. |
| 35 | **swapcase()**<br><br>Inverts case for all letters in string. |
| 36 | **title()**<br><br>Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase. |
| 37 | **translate(table, deletechars="")**<br><br>Translates string according to translation table str(256 chars), removing those in the del string. |
| 38 | **upper()**<br><br>Converts lowercase letters in string to uppercase. |
| 39 | **zfill (width)**<br><br>Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero). |
| 40 | **isdecimal()** |

| | |
|---|---|
| | Returns true if a unicode string contains only decimal characters and false otherwise. |

## Python - Lists

The most basic data structure in Python is the **sequence**. Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.

Python has six built-in types of sequences, but the most common ones are lists and tuples, which we would see in this tutorial.

There are certain things you can do with all sequence types. These operations include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

## Python Lists

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

Creating a list is as simple as putting different comma-separated values between square brackets. For example −

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"]
```

Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

## Accessing Values in Lists

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example −

```
#!/usr/bin/python



list1 = ['physics', 'chemistry', 1997, 2000];

list2 = [1, 2, 3, 4, 5, 6, 7 ];

print "list1[0]: ", list1[0]

print "list2[1:5]: ", list2[1:5]
```

When the above code is executed, it produces the following result −

```
list1[0]:  physics
list2[1:5]:  [2, 3, 4, 5]
```

## Updating Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method. For example −

```
#!/usr/bin/python


list = ['physics', 'chemistry', 1997, 2000];

print "Value available at index 2 : "

print list[2]

list[2] = 2001;

print "New value available at index 2 : "

print list[2]
```

**Note** − append() method is discussed in subsequent section.

When the above code is executed, it produces the following result −

```
Value available at index 2 :
1997
New value available at index 2 :
2001
```

## Delete List Elements

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know. For example −

```
#!/usr/bin/python


list1 = ['physics', 'chemistry', 1997, 2000];

print list1

del list1[2];

print "After deleting value at index 2 : "
print list1
```

When the above code is executed, it produces following result −

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :
['physics', 'chemistry', 2000]
```

**Note** − remove() method is discussed in subsequent section.

## Basic List Operations

Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

In fact, lists respond to all of the general sequence operations we used on strings in the prior chapter.

| Python Expression | Results | Description |
|---|---|---|
| len([1, 2, 3]) | 3 | Length |
| [1, 2, 3] + [4, 5, 6] | [1, 2, 3, 4, 5, 6] | Concatenation |
| ['Hi!'] * 4 | ['Hi!', 'Hi!', 'Hi!', 'Hi!'] | Repetition |
| 3 in [1, 2, 3] | True | Membership |
| for x in [1, 2, 3]: print x, | 1 2 3 | Iteration |

## Indexing, Slicing, and Matrixes

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

Assuming following input −

```
L = ['spam', 'Spam', 'SPAM!']
```

| Python Expression | Results | Description |
|---|---|---|
| L[2] | SPAM! | Offsets start at zero |

|  |  |  |
|---|---|---|
| L[-2] | Spam | Negative: count from the right |
| L[1:] | ['Spam', 'SPAM!'] | Slicing fetches sections |

## Built-in List Functions & Methods

Python includes the following list functions −

| Sr.No. | Function with Description |
|---|---|
| 1 | **cmp(list1, list2)**<br><br>Compares elements of both lists. |
| 2 | **len(list)**<br><br>Gives the total length of the list. |
| 3 | **max(list)**<br><br>Returns item from the list with max value. |
| 4 | **min(list)**<br><br>Returns item from the list with min value. |
| 5 | **list(seq)**<br><br>Converts a tuple into list. |

Python includes following list methods

| Sr.No. | Methods with Description |
|---|---|
| 1 | **list.append(obj)**<br><br>Appends object obj to list |
| 2 | **list.count(obj)**<br><br>Returns count of how many times obj occurs in list |

| | |
|---|---|
| 3 | **list.extend(seq)**<br><br>Appends the contents of seq to list |
| 4 | **list.index(obj)**<br><br>Returns the lowest index in list that obj appears |
| 5 | **list.insert(index, obj)**<br><br>Inserts object obj into list at offset index |
| 6 | **list.pop(obj=list[-1])**<br><br>Removes and returns last object or obj from list |
| 7 | **list.remove(obj)**<br><br>Removes object obj from list |
| 8 | **list.reverse()**<br><br>Reverses objects of list in place |
| 9 | **list.sort([func])**<br><br>Sorts objects of list, use compare func if given |

## Python - Tuples

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also. For example −

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5 );
tup3 = "a", "b", "c", "d";
```

The empty tuple is written as two parentheses containing nothing −

```
tup1 = ();
```

To write a tuple containing a single value you have to include a comma, even though there is only one value −

```
tup1 = (50,);
```

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

## Accessing Values in Tuples

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example −

```
#!/usr/bin/python


tup1 = ('physics', 'chemistry', 1997, 2000);

tup2 = (1, 2, 3, 4, 5, 6, 7 );

print "tup1[0]: ", tup1[0];

print "tup2[1:5]: ", tup2[1:5];
```

When the above code is executed, it produces the following result −

```
tup1[0]:  physics
tup2[1:5]:  [2, 3, 4, 5]
```

## Updating Tuples

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples as the following example demonstrates −

```
#!/usr/bin/python

tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');

# Following action is not valid for tuples
# tup1[0] = 100;

# So let's create a new tuple as follows
tup3 = tup1 + tup2;
print tup3;
```

When the above code is executed, it produces the following result −

```
(12, 34.56, 'abc', 'xyz')
```

## Delete Tuple Elements

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the **del** statement. For example −

```
#!/usr/bin/python

tup = ('physics', 'chemistry', 1997, 2000);
print tup;
del tup;
print "After deleting tup : ";
print tup;
```

This produces the following result. Note an exception raised, this is because after **del tup** tuple does not exist any more −

```
('physics', 'chemistry', 1997, 2000)
After deleting tup :
Traceback (most recent call last):
  File "test.py", line 9, in <module>
    print tup;
NameError: name 'tup' is not defined
```

## Basic Tuples Operations

Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.

In fact, tuples respond to all of the general sequence operations we used on strings in the prior chapter −

| Python Expression | Results | Description |
|---|---|---|
| len((1, 2, 3)) | 3 | Length |
| (1, 2, 3) + (4, 5, 6) | (1, 2, 3, 4, 5, 6) | Concatenation |
| ('Hi!',) * 4 | ('Hi!', 'Hi!', 'Hi!', 'Hi!') | Repetition |
| 3 in (1, 2, 3) | True | Membership |
| for x in (1, 2, 3): print x, | 1 2 3 | Iteration |

## Indexing, Slicing, and Matrixes

Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings. Assuming following input −

```
L = ('spam', 'Spam', 'SPAM!')
```

| Python Expression | Results | Description |
|---|---|---|
| L[2] | 'SPAM!' | Offsets start at zero |

|  |  |  |
|---|---|---|
| L[-2] | 'Spam' | Negative: count from the right |
| L[1:] | ['Spam', 'SPAM!'] | Slicing fetches sections |

## No Enclosing Delimiters

Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., default to tuples, as indicated in these short examples −

```
#!/usr/bin/python


print 'abc', -4.24e93, 18+6.6j, 'xyz';

x, y = 1, 2;

print "Value of x , y : ", x,y;
```

When the above code is executed, it produces the following result −

```
abc -4.24e+93 (18+6.6j) xyz
Value of x , y : 1 2
```

## Built-in Tuple Functions

Python includes the following tuple functions −

| Sr.No. | Function with Description |
|---|---|
| 1 | **cmp(tuple1, tuple2)**<br><br>Compares elements of both tuples. |
| 2 | **len(tuple)**<br><br>Gives the total length of the tuple. |
| 3 | **max(tuple)**<br><br>Returns item from the tuple with max value. |
| 4 | **min(tuple)** |

| | |
|---|---|
| | Returns item from the tuple with min value. |
| 5 | **tuple(seq)**<br><br>Converts a list into tuple. |

## Python - Dictionary

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

## Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value. Following is a simple example −

```
#!/usr/bin/python


dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

print "dict['Name']: ", dict['Name']

print "dict['Age']: ", dict['Age']
```

When the above code is executed, it produces the following result −

```
dict['Name']:  Zara
dict['Age']:  7
```

If we attempt to access a data item with a key, which is not part of the dictionary, we get an error as follows −

```
#!/usr/bin/python


dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

print "dict['Alice']: ", dict['Alice']
```

When the above code is executed, it produces the following result −

```
dict['Alice']:
Traceback (most recent call last):
  File "test.py", line 4, in <module>
    print "dict['Alice']: ", dict['Alice'];
KeyError: 'Alice'
```

## Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown below in the simple example −

```
#!/usr/bin/python


dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

dict['Age'] = 8; # update existing entry

dict['School'] = "DPS School"; # Add new entry


print "dict['Age']: ", dict['Age']

print "dict['School']: ", dict['School']
```

When the above code is executed, it produces the following result −

```
dict['Age']:  8
dict['School']:  DPS School
```

## Delete Dictionary Elements

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.

To explicitly remove an entire dictionary, just use the **del** statement. Following is a simple example −

```
#!/usr/bin/python


dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

del dict['Name']; # remove entry with key 'Name'

dict.clear();     # remove all entries in dict

del dict ;        # delete entire dictionary


print "dict['Age']: ", dict['Age']

print "dict['School']: ", dict['School']
```

This produces the following result. Note that an exception is raised because after **del dict** dictionary does not exist any more −

```
dict['Age']:
Traceback (most recent call last):
  File "test.py", line 8, in <module>
    print "dict['Age']: ", dict['Age'];
TypeError: 'type' object is unsubscriptable
```

**Note** − del() method is discussed in subsequent section.

## Properties of Dictionary Keys

Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.

There are two important points to remember about dictionary keys −

**(a)** More than one entry per key not allowed. Which means no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins. For example −

```
#!/usr/bin/python


dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'}

print "dict['Name']: ", dict['Name']
```

When the above code is executed, it produces the following result −

```
dict['Name']:  Manni
```

**(b)** Keys must be immutable. Which means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed. Following is a simple example −

```
#!/usr/bin/python


dict = {['Name']: 'Zara', 'Age': 7}

print "dict['Name']: ", dict['Name']
```

When the above code is executed, it produces the following result −

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    dict = {['Name']: 'Zara', 'Age': 7};
```

```
TypeError: unhashable type: 'list'
```

## Built-in Dictionary Functions & Methods

Python includes the following dictionary functions −

| Sr.No. | Function with Description |
|---|---|
| 1 | **cmp(dict1, dict2)**<br><br>Compares elements of both dict. |
| 2 | **len(dict)**<br><br>Gives the total length of the dictionary. This would be equal to the number of items in the dictionary. |
| 3 | **str(dict)**<br><br>Produces a printable string representation of a dictionary |
| 4 | **type(variable)**<br><br>Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type. |

Python includes following dictionary methods −

| Sr.No. | Methods with Description |
|---|---|
| 1 | **dict.clear()**<br><br>Removes all elements of dictionary *dict* |
| 2 | **dict.copy()**<br><br>Returns a shallow copy of dictionary *dict* |
| 3 | **dict.fromkeys()**<br><br>Create a new dictionary with keys from seq and values *set* to *value*. |
| 4 | **dict.get(key, default=None)** |

| | |
|---|---|
| | For *key* key, returns value or default if key not in dictionary |
| 5 | **dict.has_key(key)**<br><br>Returns *true* if key in dictionary *dict*, *false* otherwise |
| 6 | **dict.items()**<br><br>Returns a list of *dict*'s (key, value) tuple pairs |
| 7 | **dict.keys()**<br><br>Returns list of dictionary dict's keys |
| 8 | **dict.setdefault(key, default=None)**<br><br>Similar to get(), but will set dict[key]=default if *key* is not already in dict |
| 9 | **dict.update(dict2)**<br><br>Adds dictionary *dict2*'s key-values pairs to *dict* |
| 10 | **dict.values()**<br><br>Returns list of dictionary *dict*'s values |

# Python - Functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called *user-defined functions.*

## Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses ( ( ) ).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

## Syntax

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

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

## Example

The following function takes a string as input parameter and prints it on standard screen.

```
def printme( str ):

   "This prints a passed string into this function"

   print str
   return
```

## Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printme() function −

```
#!/usr/bin/python


# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print str
   return;


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

When the above code is executed, it produces the following result −

```
I'm first call to user defined function!
Again second call to the same function
```

## Pass by reference vs value

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example −

```
#!/usr/bin/python


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

   mylist.append([1,2,3,4]);

   print "Values inside the function: ", mylist

   return


# Now you can call changeme function

mylist = [10,20,30];

changeme( mylist );

print "Values outside the function: ", mylist
```

Here, we are maintaining reference of the passed object and appending values in the same object. So, this would produce the following result −

```
Values inside the function:  [10, 20, 30, [1, 2, 3, 4]]
Values outside the function:  [10, 20, 30, [1, 2, 3, 4]]
```

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

```
#!/usr/bin/python


# Function definition is here

def changeme( mylist ):

   "This changes a passed list into this function"

   mylist = [1,2,3,4]; # This would assig new reference in mylist

   print "Values inside the function: ", mylist

   return


# Now you can call changeme function

mylist = [10,20,30];

changeme( mylist );

print "Values outside the function: ", mylist
```

The parameter *mylist* is local to the function changeme. Changing mylist within the function does not affect *mylist*. The function accomplishes nothing and finally this would produce the following result −

```
Values inside the function:  [1, 2, 3, 4]
Values outside the function:  [10, 20, 30]
```

## Function Arguments

You can call a function by using the following types of formal arguments −

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

## Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function *printme()*, you definitely need to pass one argument, otherwise it gives a syntax error as follows −

```
#!/usr/bin/python


# Function definition is here

def printme( str ):

   "This prints a passed string into this function"

   print str

   return;


# Now you can call printme function

printme()
```

When the above code is executed, it produces the following result −

```
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    printme();
```

```
TypeError: printme() takes exactly 1 argument (0 given)
```

## Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the *printme()* function in the following ways −

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print str
   return;

# Now you can call printme function
printme( str = "My string")
```

When the above code is executed, it produces the following result −

```
My string
```

The following example gives more clear picture. Note that the order of parameters does not matter.

```
#!/usr/bin/python

# Function definition is here
def printinfo( name, age ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;


# Now you can call printinfo function

printinfo( age=50, name="miki" )
```

When the above code is executed, it produces the following result −

```
Name:  miki
Age  50
```

## Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed −

```
#!/usr/bin/python


# Function definition is here

def printinfo( name, age = 35 ):

   "This prints a passed info into this function"

   print "Name: ", name

   print "Age ", age

   return;


# Now you can call printinfo function

printinfo( age=50, name="miki" )

printinfo( name="miki" )
```

When the above code is executed, it produces the following result −

```
Name:  miki
Age  50
Name:  miki
Age  35
```

## Variable-length arguments

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

Syntax for a function with non-keyword variable arguments is this −

```
def functionname([formal_args,] *var_args_tuple ):
   "function_docstring"
   function_suite
   return [expression]
```

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

```
#!/usr/bin/python


# Function definition is here

def printinfo( arg1, *vartuple ):

   "This prints a variable passed arguments"

   print "Output is: "

   print arg1

   for var in vartuple:

      print var

   return;


# Now you can call printinfo function

printinfo( 10 )

printinfo( 70, 60, 50 )
```

When the above code is executed, it produces the following result −

```
Output is:
10
Output is:
70
60
50
```

## The *Anonymous* Functions

These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You can use the *lambda* keyword to create small anonymous functions.

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

## Syntax

The syntax of *lambda* functions contains only a single statement, which is as follows −

```
lambda [arg1 [,arg2,.....argn]]:expression
```

Following is the example to show how *lambda* form of function works −

```
#!/usr/bin/python


# Function definition is here

sum = lambda arg1, arg2: arg1 + arg2;


# Now you can call sum as a function

print "Value of total : ", sum( 10, 20 )

print "Value of total : ", sum( 20, 20 )
```

When the above code is executed, it produces the following result −

```
Value of total :  30
Value of total :  40
```

## The *return* Statement

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

All the above examples are not returning any value. You can return a value from a function as follows −

```
#!/usr/bin/python


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


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

When the above code is executed, it produces the following result −

```
Inside the function :  30
Outside the function :  30
```

## Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python −

- Global variables
- Local variables

## Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Following is a simple example −

```
#!/usr/bin/python


total = 0; # This is global variable.

# Function definition is here

def sum( arg1, arg2 ):

   # Add both the parameters and return them."

   total = arg1 + arg2; # Here total is local variable.

   print "Inside the function local total : ", total

   return total;


# Now you can call sum function

sum( 10, 20 );

print "Outside the function global total : ", total
```

When the above code is executed, it produces the following result −

```
Inside the function local total :  30
Outside the function global total :  0
```

# Python - Object Oriented

Python has been an object-oriented language since it existed. Because of this, creating and using classes and objects are downright easy. This chapter helps you become an expert in using Python's object-oriented programming support.

If you do not have any previous experience with object-oriented (OO) programming, you may want to consult an introductory course on it or at least a tutorial of some sort so that you have a grasp of the basic concepts.

However, here is small introduction of Object-Oriented Programming (OOP) to bring you at speed −

## Overview of OOP Terminology

- **Class** − A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- **Class variable** − A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- **Data member** − A class variable or instance variable that holds data associated with a class and its objects.
- **Function overloading** − The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
- **Instance variable** − A variable that is defined inside a method and belongs only to the current instance of a class.
- **Inheritance** − The transfer of the characteristics of a class to other classes that are derived from it.
- **Instance** − An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Instantiation** − The creation of an instance of a class.
- **Method** − A special kind of function that is defined in a class definition.
- **Object** − A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.
- **Operator overloading** − The assignment of more than one function to a particular operator.

## Creating Classes

The *class* statement creates a new class definition. The name of the class immediately follows the keyword *class* followed by a colon as follows −

```
class ClassName:
   'Optional class documentation string'
   class_suite
```

- The class has a documentation string, which can be accessed via *ClassName.__doc__*.
- The *class_suite* consists of all the component statements defining class members, data attributes and functions.

## Example

Following is the example of a simple Python class −

```
class Employee:

   'Common base class for all employees'

   empCount = 0


   def __init__(self, name, salary):

      self.name = name

      self.salary = salary

      Employee.empCount += 1


   def displayCount(self):

     print "Total Employee %d" % Employee.empCount


   def displayEmployee(self):

      print "Name : ", self.name,  ", Salary: ", self.salary
```

- The variable *empCount* is a class variable whose value is shared among all instances of a this class. This can be accessed as *Employee.empCount* from inside the class or outside the class.
- The first method *__init__()* is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.

- You declare other class methods like normal functions with the exception that the first argument to each method is *self*. Python adds the *self* argument to the list for you; you do not need to include it when you call the methods.

## Creating Instance Objects

To create instances of a class, you call the class using class name and pass in whatever arguments its *__init__* method accepts.

```
"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
```

## Accessing Attributes

You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows −

```
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

Now, putting all the concepts together −

```
#!/usr/bin/python

class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
    print "Name : ", self.name,  ", Salary: ", self.salary


"This would create first object of Employee class"

emp1 = Employee("Zara", 2000)

"This would create second object of Employee class"

emp2 = Employee("Manni", 5000)

emp1.displayEmployee()

emp2.displayEmployee()

print "Total Employee %d" % Employee.empCount
```

When the above code is executed, it produces the following result −

```
Name :  Zara ,Salary:  2000
Name :  Manni ,Salary:  5000
Total Employee 2
```

You can add, remove, or modify attributes of classes and objects at any time −

```
emp1.age = 7  # Add an 'age' attribute.
emp1.age = 8  # Modify 'age' attribute.
del emp1.age  # Delete 'age' attribute.
```

Instead of using the normal statements to access attributes, you can use the following functions −

- The **getattr(obj, name[, default])** − to access the attribute of object.
- The **hasattr(obj,name)** − to check if an attribute exists or not.
- The **setattr(obj,name,value)** − to set an attribute. If attribute does not exist, then it would be created.
- The **delattr(obj, name)** − to delete an attribute.

```
hasattr(emp1, 'age')    # Returns true if 'age' attribute exists
getattr(emp1, 'age')    # Returns value of 'age' attribute
setattr(emp1, 'age', 8) # Set attribute 'age' at 8
delattr(empl, 'age')    # Delete attribute 'age'
```

## Built-In Class Attributes

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute −

- **__dict__** − Dictionary containing the class's namespace.
- **__doc__** − Class documentation string or none, if undefined.
- **__name__** − Class name.

- **__module__** − Module name in which the class is defined. This attribute is "__main__" in interactive mode.
- **__bases__** − A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

For the above class let us try to access all these attributes −

```
#!/usr/bin/python


class Employee:

   'Common base class for all employees'

   empCount = 0


   def __init__(self, name, salary):

      self.name = name

      self.salary = salary

      Employee.empCount += 1


   def displayCount(self):

     print "Total Employee %d" % Employee.empCount


   def displayEmployee(self):

      print "Name : ", self.name,  ", Salary: ", self.salary


print "Employee.__doc__:", Employee.__doc__

print "Employee.__name__:", Employee.__name__

print "Employee.__module__:", Employee.__module__

print "Employee.__bases__:", Employee.__bases__

print "Employee.__dict__:", Employee.__dict__
```

When the above code is executed, it produces the following result −

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: ()
Employee.__dict__: {'__module__': '__main__', 'displayCount':
<function displayCount at 0xb7c84994>, 'empCount': 2,
'displayEmployee': <function displayEmployee at 0xb7c8441c>,
'__doc__': 'Common base class for all employees',
'__init__': <function __init__ at 0xb7c846bc>}
```

## Destroying Objects (Garbage Collection)

Python deletes unneeded objects (built-in types or class instances) automatically to free the memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed Garbage Collection.

Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of aliases that point to it changes.

An object's reference count increases when it is assigned a new name or placed in a container (list, tuple, or dictionary). The object's reference count decreases when it's deleted with *del*, its reference is reassigned, or its reference goes out of scope. When an object's reference count reaches zero, Python collects it automatically.

```
a = 40      # Create object <40>
b = a       # Increase ref. count  of <40>
c = [b]     # Increase ref. count  of <40>

del a       # Decrease ref. count  of <40>
b = 100     # Decrease ref. count  of <40>
c[0] = -1   # Decrease ref. count  of <40>
```

You normally will not notice when the garbage collector destroys an orphaned instance and reclaims its space. But a class can implement the special method *__del__()*, called a destructor, that is invoked when the instance is about to be destroyed. This method might be used to clean up any non memory resources used by an instance.

### Example

This __del__() destructor prints the class name of an instance that is about to be destroyed −

```
#!/usr/bin/python


class Point:

   def __init__( self, x=0, y=0):

      self.x = x
```

```
     self.y = y

   def __del__(self):

     class_name = self.__class__.__name__

     print class_name, "destroyed"


pt1 = Point()

pt2 = pt1

pt3 = pt1

print id(pt1), id(pt2), id(pt3) # prints the ids of the obejcts

del pt1

del pt2

del pt3
```

When the above code is executed, it produces following result −

```
3083401324 3083401324 3083401324
Point destroyed
```

**Note** − Ideally, you should define your classes in separate file, then you should import them in your main program file using *import* statement.

## Class Inheritance

Instead of starting from scratch, you can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name.

The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class. A child class can also override data members and methods from the parent.

### Syntax

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name −

```
class SubClassName (ParentClass1[, ParentClass2, ...]):

   'Optional class documentation string'

   class_suite
```

### Example

```
#!/usr/bin/python

class Parent:        # define parent class
   parentAttr = 100
   def __init__(self):
      print "Calling parent constructor"

   def parentMethod(self):
      print 'Calling parent method'

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print "Parent attribute :", Parent.parentAttr

class Child(Parent): # define child class
   def __init__(self):
      print "Calling child constructor"

   def childMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.childMethod()      # child calls its method
c.parentMethod()     # calls parent's method
c.setAttr(200)       # again call parent's method
c.getAttr()          # again call parent's method
```

When the above code is executed, it produces the following result −

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

Similar way, you can drive a class from multiple parent classes as follows −

```
class A:        # define your class A
.....

class B:         # define your class B
.....

class C(A, B):   # subclass of A and B
.....
```

You can use issubclass() or isinstance() functions to check a relationships of two classes and instances.

- The **issubclass(sub, sup)** boolean function returns true if the given subclass **sub** is indeed a subclass of the superclass **sup**.
- The **isinstance(obj, Class)** boolean function returns true if *obj* is an instance of class *Class* or is an instance of a subclass of Class

## Overriding Methods

You can always override your parent class methods. One reason for overriding parent's methods is because you may want special or different functionality in your subclass.

### Example

```
#!/usr/bin/python


class Parent:        # define parent class

  def myMethod(self):

     print 'Calling parent method'


class Child(Parent): # define child class

  def myMethod(self):

     print 'Calling child method'


c = Child()          # instance of child

c.myMethod()         # child calls overridden method
```

When the above code is executed, it produces the following result −

```
Calling child method
```

## Base Overloading Methods

Following table lists some generic functionality that you can override in your own classes −

| Sr.No. | Method, Description & Sample Call |
|---|---|
| 1 | **__init__ ( self [,args...] )**<br><br>Constructor (with any optional arguments)<br><br>Sample Call : *obj = className(args)* |
| 2 | **__del__( self )**<br><br>Destructor, deletes an object<br><br>Sample Call : *del obj* |
| 3 | **__repr__( self )**<br><br>Evaluable string representation<br><br>Sample Call : *repr(obj)* |
| 4 | **__str__( self )**<br><br>Printable string representation<br><br>Sample Call : *str(obj)* |
| 5 | **__cmp__ ( self, x )**<br><br>Object comparison<br><br>Sample Call : *cmp(obj, x)* |

## Overloading Operators

Suppose you have created a Vector class to represent two-dimensional vectors, what happens when you use the plus operator to add them? Most likely Python will yell at you.

You could, however, define the *__add__* method in your class to perform vector addition and then the plus operator would behave as per expectation −

### Example

```
#!/usr/bin/python

class Vector:
   def __init__(self, a, b):
      self.a = a
      self.b = b

   def __str__(self):
      return 'Vector (%d, %d)' % (self.a, self.b)

   def __add__(self,other):
      return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print v1 + v2
```

When the above code is executed, it produces the following result −

```
Vector(7,8)
```

## Data Hiding

An object's attributes may or may not be visible outside the class definition. You need to name attributes with a double underscore prefix, and those attributes then are not be directly visible to outsiders.

### Example

```
#!/usr/bin/python
```

```
class JustCounter:
  __secretCount = 0


  def count(self):
    self.__secretCount += 1
    print self.__secretCount


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

When the above code is executed, it produces the following result −

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

Python protects those members by internally changing the name to include the class name. You can access such attributes as *object._className__attrName*. If you would replace your last line as following, then it works for you −

```
.........................
print counter._JustCounter__secretCount
```

When the above code is executed, it produces the following result −

```
1
2
2
```

# Python Sets

## Set

A set is a collection which is unordered and unindexed. In Python sets are written with curly brackets.

### Example

Create a Set:

```
thisset = {“apple”, “banana”, “cherry”}

print(thisset)
```

**Note:** Sets are unordered, so the items will appear in a random order.

## Access Items

You cannot access items in a set by referring to an index, since sets are unordered the items has no index.

But you can loop through the set items using a `for` loop, or ask if a specified value is present in a set, by using the `in` keyword.

### Example

Loop through the set, and print the values:

```
thisset = {“apple”, “banana”, “cherry”}


for x in thisset:
  print(x)
```

### Example

Check if "banana" is present in the set:

```
thisset = {“apple”, “banana”, “cherry”}
```

```
print("banana" in thisset)
```

## Change Items

Once a set is created, you cannot change its items, but you can add new items.

## Add Items

To add one item to a set use the add() method.

To add more than one item to a set use the update() method.

### Example

Add an item to a set, using the add() method:

```
thisset = {“apple”, “banana”, “cherry”}


thisset.add("orange")

print(thisset)
```

### Example

Add multiple items to a set, using the update() method:

```
thisset = {“apple”, “banana”, “cherry”}


thisset.update(["orange", "mango", "grapes"])

print(thisset)
```

## Get the Length of a Set

To determine how many items a set has, use the `len()` method.

### Example

Get the number of items in a set:

```
thisset = {“apple”, “banana”, “cherry”}


print(len(thisset))
```

## Remove Item

To remove an item in a set, use the `remove()`, or the `discard()` method.

### Example

Remove "banana" by using the `remove()` method:

```
thisset = {“apple”, “banana”, “cherry”}


thisset.remove("banana")

print(thisset)
```

**Note:** If the item to remove does not exist, `remove()` will raise an error.

### Example

Remove "banana" by using the `discard()` method:

```
thisset = {“apple”, “banana”, “cherry”}


thisset.discard("banana")
```

```
print(thisset)
```

**Note:** If the item to remove does not exist, discard() will **NOT** raise an error.

You can also use the pop(), method to remove an item, but this method will remove the *last* item. Remember that sets are unordered, so you will not know what item that gets removed.

The return value of the pop() method is the removed item.

## Example

Remove the last item by using the pop() method:

```
thisset = {“apple”, “banana”, “cherry”}

x = thisset.pop()

print(x)

print(thisset)
```

**Note:** Sets are *unordered*, so when using the pop() method, you will not know which item that gets removed.

## Example

The clear() method empties the set:

```
thisset = {“apple”, “banana”, “cherry”}

thisset.clear()

print(thisset)
```

## Example

The del keyword will delete the set completely:

```
thisset = {“apple”, “banana”, “cherry”}
```

```
del thisset

print(thisset)
```

## The set() Constructor

It is also possible to use the set() constructor to make a set.

### Example

Using the set() constructor to make a set:

```
thisset = set(("apple", "banana", "cherry")) # note the double round-brackets
print(thisset)
```

## Set Methods

Python has a set of built-in methods that you can use on sets.

| Method | Description |
|---|---|
| add() | Adds an element to the set |
| clear() | Removes all the elements from the set |
| copy() | Returns a copy of the set |
| difference() | Returns a set containing the difference between two or more sets |
| difference_update() | Removes the items in this set that are also included in another, specified set |
| discard() | Remove the specified item |

| | |
|---|---|
| intersection() | Returns a set, that is the intersection of two other sets |
| intersection_update() | Removes the items in this set that are not present in other, specified set(s) |
| isdisjoint() | Returns whether two sets have a intersection or not |
| issubset() | Returns whether another set contains this set or not |
| issuperset() | Returns whether this set contains another set or not |
| pop() | Removes an element from the set |
| remove() | Removes the specified element |
| symmetric_difference() | Returns a set with the symmetric differences of two sets |
| symmetric_difference_update() | inserts the symmetric differences from this set and another |
| union() | Return a set containing the union of sets |
| update() | Update the set wit the union of this set and others |