PYTHON 3

tutorialspoint
SIMPLYEASYLEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia https://twitter.com/tutorialspoint

About the Tutorial

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985 – 1990. Like Perl, Python source code is also available under the GNU General Public License (GPL). Python is named after a TV Show called 'Monty Python's Flying Circus' and not after Python-the snake.

Python 3.0 was released in 2008. Although this version is supposed to be backward incompatibles, later on many of its important features have been backported to be compatible with the version 2.7. This tutorial gives enough understanding on Python 3 version programming language. Please refer to this link for our Python 2 tutorial.

Audience

This tutorial is designed for software programmers who want to upgrade their Python skills to Python 3. This tutorial can also be used to learn Python programming language from scratch.

Prerequisites

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

Execute Python Programs

For most of the examples given in this tutorial you will find **Try it** option, so just make use of it and enjoy your learning.

Try the following example using **Try it** option available at the top right corner of the below sample code box –

```
#!/usr/bin/python3

print ("Hello, Python!")
```

Copyright & Disclaimer

Table of Contents

Python 3 – Basic Tutorial

1. Python 3 – What is New?

The __future__ module

Python 3.x introduced some Python 2-incompatible keywords and features that can be imported via the in-built __future__ module in Python 2. It is recommended to use __future__ imports, if you are planning Python 3.x support for your code.

For example, if we want Python 3.x's integer division behavior in Python 2, add the following import statement.

```
from __future__ import division
```

The print Function

Most notable and most widely known change in Python 3 is how the **print** function is used. Use of parenthesis () with print function is now mandatory. It was optional in Python 2.

```
print "Hello World" #is acceptable in Python 2
print ("Hello World") # in Python 3, print must be followed by ()
```

The print() function inserts a new line at the end, by default. In Python 2, it can be suppressed by putting ',' at the end. In Python 3, "end=' '" appends space instead of newline.

```
print x,           # Trailing comma suppresses newline in Python 2
print(x, end=" ")  # Appends a space instead of a newline in Python 3
```

Reading Input from Keyboard

Python 2 has two versions of input functions, **input()** and **raw_input()**. The input() function treats the received data as string if it is included in quotes '' or "", otherwise the data is treated as number.

In Python 3, raw_input() function is deprecated. Further, the received data is always treated as string.

```
In Python 2
>>> x=input('something:')
something:10 #entered data is treated as number
>>> x
10
>>> x=input('something:')
something:'10' #eentered data is treated as string
```

```
>>> x
'10'
>>> x=raw_input("something:")
something:10 #entered data is treated as string even without ''
>>> x
'10'
>>> x=raw_input("something:")
something:'10' #entered data treated as string including ''
>>> x
"'10'"
In Python 3
>>> x=input("something:")
something:10
>>> x
'10'
>>> x=input("something:")
something:'10' #entered data treated as string with or without ''
>>> x
"'10'"
>>> x=raw_input("something:") # will result NameError
Traceback (most recent call last):
  File "", line 1, in

    x=raw_input("something:")
NameError: name 'raw_input' is not defined
```

Integer Division

In Python 2, the result of division of two integers is rounded to the nearest integer. As a result, 3/2 will show 1. In order to obtain a floating-point division, numerator or denominator must be explicitly used as float. Hence, either 3.0/2 or 3/2.0 or 3.0/2.0 will result in 1.5

Python 3 evaluates 3 / 2 as 1.5 by default, which is more intuitive for new programmers.

Unicode Representation

Python 2 requires you to mark a string with a **u** if you want to store it as Unicode.

Python 3 stores strings as Unicode, by default. We have Unicode (utf-8) strings, and 2 byte classes: byte and byte arrays.

xrange() Function Removed

In Python 2 range() returns a list, and xrange() returns an object that will only generate the items in the range when needed, saving memory.

In Python 3, the range() function is removed, and xrange() has been renamed as range(). In addition, the range() object supports slicing in Python 3.2 and later .

raise exceprion

Python 2 accepts both notations, the 'old' and the 'new' syntax; Python 3 raises a SyntaxError if we do not enclose the exception argument in parenthesis.

```
raise IOError, "file error" #This is accepted in Python 2
raise IOError("file error") #This is also accepted in Python 2
raise IOError, "file error" #syntax error is raised in Python 3
raise IOError("file error") #this is the recommended syntax in Python 3
```

Arguments in Exceptions

In Python 3, arguments to exception should be declared with 'as' keyword.

```
except Myerror, err: # In Python2
except Myerror as err: #In Python 3
```

next() Function and .next() Method

In Python 2, next() as a method of generator object, is allowed. In Python 2, the next() function, to iterate over generator object, is also accepted. In Python 3, however, next(0 as a generator method is discontinued and raises **AttributeError.**

```
gen = (letter for letter in 'Hello World') # creates generator object
next(my_generator) #allowed in Python 2 and Python 3
my_generator.next() #allowed in Python 2. raises AttributeError in Python 3
```

2to3 Utility

Along with Python 3 interpreter, 2to3.py script is usually installed in tools/scripts folder. It reads Python 2.x source code and applies a series of fixers to transform it into a valid Python 3.x code.

```
Here is a sample Python 2 code (area.py):
def area(x,y=3.14):
    a=y*x*x
    print a
    return a
```

```
a=area(10)
print "area",a
To convert into Python 3 version:
$2to3 -w area.py
Converted code :
def area(x,y=3.14): # formal parameters
    a=y*x*x
    print (a)
    return a
a=area(10)
print("area",a)
```

2. Python 3 – 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 whereas the other languages use punctuations. 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 1.0 was released in November 1994. In 2000, Python 2.0 was released. Python 2.7.11 is the latest edition of Python 2.
- Meanwhile, Python 3.0 was released in 2008. Python 3 is not backward compatible with Python 2. The emphasis in Python 3 had been on the removal of duplicate programming constructs and modules so that "There should be one -- and preferably only one -- obvious way to do it." Python 3.5.1 is the latest version of Python 3.

Python Features

Python's features include-

- **Easy-to-learn:** Python has few keywords, simple structure, and a clearly defined syntax. This allows a 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. A 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.

3. Python 3 – Environment Setup

Try it Option Online

We have set up the Python Programming environment online, so that you can compile and execute all the available examples online. It will give you the confidence in what you are reading and will enable you to verify the programs with different options. Feel free to modify any example and execute it online.

Try the following example using our online compiler available at CodingGround

```
#!/usr/bin/python3
print ("Hello, Python!")
```

For most of the examples given in this tutorial, you will find a **Try it** option on our website code sections, at the top right corner that will take you to the online compiler. Just use it and enjoy your learning.

Python 3 is available for Windows, Mac OS and most of the flavors of Linux operating system. Even though Python 2 is available for many other OSs, Python 3 support either has not been made available for them or has been dropped.

Local Environment Setup

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

Getting Python

Windows platform

Binaries of latest version of Python 3 (Python 3.5.1) are available on this download page

The following different installation options are available.

- Windows x86-64 embeddable zip file
- Windows x86-64 executable installer
- Windows x86-64 web-based installer
- Windows x86 embeddable zip file
- Windows x86 executable installer
- Windows x86 web-based installer

Note: In order to install Python 3.5.1, minimum OS requirements are Windows 7 with SP1. For versions 3.0 to 3.4.x, Windows XP is acceptable.

Linux platform

Different flavors of Linux use different package managers for installation of new packages.

On Ubuntu Linux, Python 3 is installed using the following command from the terminal.

```
$sudo apt-get install python3-minimal
```

Installation from source

```
Download Gzipped source tarball from Python's download URL:
https://www.python.org/ftp/python/3.5.1/Python-3.5.1.tgz
Extract the tarball
tar xvfz Python-3.5.1.tgz
Configure and Install:
cd Python-3.5.1
./configure --prefix=/opt/python3.5.1
make
sudo make install
```

Mac OS

Download Mac OS installers from this URL:https://www.python.org/downloads/mac-osx/

- Mac OS X 64-bit/32-bit installer : python-3.5.1-macosx10.6.pkg
- Mac OS X 32-bit i386/PPC installer : python-3.5.1-macosx10.5.pkg

Double click this package file and follow the wizard instructions to install.

The most up-to-date and current source code, binaries, documentation, news, etc., is available on the official website of Python:

Python Official Website : http://www.python.org/

You can download Python documentation from the following site. The documentation is available in HTML, PDF and PostScript formats.

Python Documentation Website : www.python.org/doc/

Setting up PATH

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

The important features are-

- 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 "$PATH:/usr/local/bin/python3" and press Enter.
- **In the bash shell (Linux):** type export PATH="$PATH:/usr/local/bin/python3" and press Enter.
- **In the sh or ksh shell:** type PATH="$PATH:/usr/local/bin/python3" and press Enter.

Note: /usr/local/bin/python3 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 are recognized by Python-

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

Running Python

There are three different ways to start Python-

(1) Interactive Interpreter

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

Enter **python** the command line.

Start coding right away in the interactive interpreter.

```
$python              # Unix/Linux
or
python%              # Unix/Linux
or
C:>python            # Windows/DOS
```

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

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

file	run Python script from given file

(2) Script from the Command-line

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

```
$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.

(3) 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 the help of 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 3.4.1 version available on Windows 7 and Ubuntu Linux.

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

4. Python 3 – 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 the 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 3.3.2 (default, Dec 10 2013, 11:35:01)
[GCC 4.6.3] on Linux
Type "help", "copyright", "credits", or "license" for more information.
>>>
On Windows:
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on
win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

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

```
>>> print ("Hello, Python!")
```

If you are running the older version of Python (Python 2.x), use of parenthesis as **inprint** function is optional. 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 the extension**.py**. Type the following source code in a test.py file-

```
print ("Hello, Python!")
```

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

On Linux

```
$ python test.py
```

This produces the following result-

```
Hello, Python!
```

On Windows

```
C:\Python34>Python test.py
```

This produces the following result-

```
Hello, Python!
```

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

```
#!/usr/bin/python3
print ("Hello, Python!")
```

We assume that you have Python interpreter available in the /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 strong 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 constants or variables or any other identifier names. All the Python keywords contain lowercase letters only.

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

Lines and Indentation

Python does not use 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 the 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 the various blocks even if they are without braces.

```
#!/usr/bin/python3
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 = 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, however, allows the use of the line continuation character (\) to denote that the line should continue. For example-

```
total = item_one + \
        item_two + \
        item_three
```

The 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 is the beginning of a comment. All characters after the #, up to the end of the physical line, are part of the comment and the Python interpreter ignores them.

```
#!/usr/bin/python3
```

```
# 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
```

Python does not have multiple-line commenting feature. You have to comment each line individually as follows-

```
# This is a comment.
# This is a comment, too.
# This is a comment, too.
# I said that already.
```

Using Blank Lines

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

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

Waiting for the User

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

```
#!/usr/bin/python3
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 a single line given that no 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

Groups 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 advance topic. Let us understand it.

Command Line Arguments

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

```
$ python test.py arg1 arg2 arg3
```

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

- **sys.argv** is the list of command-line arguments.
- **len(sys.argv)** is the number of command-line arguments.

Here sys.argv[0] is the program i.e. the script name.

Example

Consider the following script **test.py-**

```
#!/usr/bin/python3
import sys
print ('Number of arguments:', len(sys.argv), 'arguments.')
print ('Argument List:', str(sys.argv))
```

Now run the above script as follows –

```
$ python test.py arg1 arg2 arg3
```

This produces the following result-

```
Number of arguments: 4 arguments.
Argument List: ['test.py', 'arg1', 'arg2', 'arg3']
```

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

Parsing Command-Line Arguments

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

getopt.getopt method

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

```
getopt.getopt(args, options, [long_options])
```

Here is the detail of the parameters-

- **args**: This is the argument list to be parsed.
- **options**: This is the string of option letters that the script wants to recognize, with options that require an argument should be followed by a colon (:).
- **long_options**: This is an optional parameter and if specified, must be a list of strings with the names of the long options, which should be supported. Long options, which require an argument should be followed by an equal sign ('='). To accept only long options, options should be an empty string.
- This method returns a value consisting of two elements- the first is a list of **(option, value)** pairs, the second is a list of program arguments left after the option list was stripped.

- Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options (e.g., '-x') or two hyphens for long options (e.g., '--long-option').

Exception getopt.GetoptError

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

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

Example

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

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

Here is the following script to test.py-

```
#!/usr/bin/python3
import sys, getopt
def main(argv):
   inputfile = ''
   outputfile = ''
   try:
      opts, args = getopt.getopt(argv,"hi:o:",["ifile=","ofile="])
   except getopt.GetoptError:
      print ('test.py -i <inputfile> -o <outputfile>')
      sys.exit(2)
   for opt, arg in opts:
      if opt == '-h':
         print ('test.py -i <inputfile> -o <outputfile>')
         sys.exit()
      elif opt in ("-i", "--ifile"):
         inputfile = arg
      elif opt in ("-o", "--ofile"):
         outputfile = arg
   print ('Input file is "', inputfile)
   print ('Output file is "', outputfile)
if __name__ == "__main__":
   main(sys.argv[1:])
```

Now, run the above script as follows-

```
$ test.py -h
usage: test.py -i <inputfile> -o <outputfile>
$ test.py -i BMP -o
usage: test.py -i <inputfile> -o <outputfile>
$ test.py -i inputfile -o outputfile
Input file is " inputfile
Output file is " outputfile
```

5. Python 3 – Variable Types

Variables are nothing but reserved memory locations to store values. It means that when you create a variable, you reserve some space in the 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 the 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/python3
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 the 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 the 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 three different numerical types –

- int (signed integers)
- float (floating point real values)
- complex (complex numbers)

All integers in Python 3 are represented as long integers. Hence, there is no separate number type as long.

Examples

Here are some examples of numbers-

int	float	complex
10	0.0	3.14j
100	15.20	45.j
-786	-21.9	9.322e-36j
080	32.3+e18	.876j
-0490	-90.	-.6545+0J
-0x260	-32.54e100	3e+26J
0x69	70.2-E12	4.53e-7j

A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x and y are 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 either pair 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 to the end.

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

```
#!/usr/bin/python3
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 of the differences 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/python3
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']
print (list)          # Prints complete list
print (list[0])       # Prints first element of the list
print (list[1:3])     # Prints elements starting from 2nd till 3rd
print (list[2:])      # Prints elements starting from 3rd element
print (tinylist * 2)  # Prints list two times
print (list + tinylist) # Prints concatenated lists
```

This produces the following result-

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

Python Tuples

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

The main difference between lists and tuples is- 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/python3
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')
print (tuple)           # Prints complete tuple
print (tuple[0])        # Prints first element of the tuple
print (tuple[1:3])      # Prints elements starting from 2nd till 3rd
print (tuple[2:])       # Prints elements starting from 3rd element
print (tinytuple * 2)   # Prints tuple two times
print (tuple + tinytuple) # Prints concatenated tuple
```

This produces the following result-

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

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

```
#!/usr/bin/python3
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/python3
dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"
tinydict = {'name': 'john','code':6734, 'dept': 'sales'}
print (dict['one'])       # Prints value for 'one' key
print (dict[2])           # Prints value for 2 key
print (tinydict)          # Prints complete dictionary
print (tinydict.keys())   # Prints all the keys
print (tinydict.values()) # Prints all the values
```

This produces the following result-

```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john']
```

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

Data Type Conversion

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

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

Function	Description
int(x [,base])	Converts x to an integer. The base specifies the base if x is a string.
float(x)	Converts x to a floating-point number.
complex(real [,imag])	Creates a complex number.
str(x)	Converts object x to a string representation.
repr(x)	Converts object x to an expression string.
eval(str)	Evaluates a string and returns an object.
tuple(s)	Converts s to a tuple.
list(s)	Converts s to a list.
set(s)	Converts s to a set.
dict(d)	Creates a dictionary. d must be a sequence of (key,value) tuples.
frozenset(s)	Converts s to a frozen set.
chr(x)	Converts an integer to a character.
unichr(x)	Converts an integer to a Unicode character.
ord(x)	Converts a single character to its integer value.
hex(x)	Converts an integer to a hexadecimal string.
oct(x)	Converts an integer to an octal string.

6. Python 3 – 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 the 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 at all the operators one by one.

Python Arithmetic Operators

Assume variable **a** holds the value 10 and variable **b** holds the value 21, then-

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 31
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -11
* Multiplication	Multiplies values on either side of the operator	a * b = 210
/ Division	Divides left hand operand by right hand operand	b / a = 2.1
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 1
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed.	9//2 = 4 and 9.0//2.0 = 4.0

Example

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

```
#!/usr/bin/python3
a = 21
b = 10
c = 0
c = a + b
print ("Line 1 - Value of c is ", c)

c = a - b
print ("Line 2 - Value of c is ", c )

c = a * b
print ("Line 3 - Value of c is ", c)

c = a / b
print ("Line 4 - Value of c is ", c )

c = a % b
print ("Line 5 - Value of c is ", c)

a = 2
b = 3
c = a**b
print ("Line 6 - Value of c is ", c)

a = 10
b = 5
c = a//b
print ("Line 7 - Value of c is ", c)
```

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

```
Line 1 - Value of c is  31
Line 2 - Value of c is  11
Line 3 - Value of c is  210
Line 4 - Value of c is  2.1
Line 5 - Value of c is  1
Line 6 - Value of c is  8
Line 7 - Value of c is  2
```

Python Comparison Operators

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

Assume variable a holds the value 10 and variable b holds the value 20, then-

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

Example

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

```
#!/usr/bin/python3
a = 21
b = 10
if ( a == b ):
   print ("Line 1 - a is equal to b")
else:
```

```
   print ("Line 1 - a is not equal to b")

if ( a != b ):
   print ("Line 2 - a is not equal to b")
else:
   print ("Line 2 - a is equal to b")

if ( a < b ):
   print ("Line 3 - a is less than b" )
else:
   print ("Line 3 - a is not less than b")

if ( a > b ):
   print ("Line 4 - a is greater than b")
else:
   print ("Line 4 - a is not greater than b")

a,b=b,a #values of a and b swapped. a becomes 10, b becomes 21

if ( a <= b ):
   print ("Line 5 - a is either less than or equal to  b")
else:
   print ("Line 5 - a is neither less than nor equal to  b")

if ( b >= a ):
   print ("Line 6 - b is either greater than  or equal to b")
else:
   print ("Line 6 - b is neither greater than  nor equal to b")
```

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

```
Line 1 - a is not equal to b
Line 2 - a is not equal to b
Line 3 - a is not less than b
Line 4 - a is greater than b
Line 5 - a is either less than or equal to  b
Line 6 - b is either greater than  or equal to b
```

Python Assignment Operators

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

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

-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / 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

Example

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

```
#!/usr/bin/python3

a = 21
b = 10
c = 0

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

c += a
print ("Line 2 - Value of c is ", c )

c *= a
print ("Line 3 - Value of c is ", c )
```

```
c /= a
print ("Line 4 - Value of c is ", c )

c  = 2
c %= a
print ("Line 5 - Value of c is ", c)

c **= a
print ("Line 6 - Value of c is ", c)

c //= a
print ("Line 7 - Value of c is ", c)
```

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

```
Line 1 - Value of c is  31
Line 2 - Value of c is  52
Line 3 - Value of c is  1092
Line 4 - Value of c is  52.0
Line 5 - Value of c is  2
Line 6 - Value of c is  2097152
Line 7 - Value of c is  99864
```

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

Pyhton's built-in function bin() can be used to obtain binary representation of an integer number.

The following Bitwise operators are supported by Python language-

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 operand's value is moved left by the number of bits specified by the right operand.	a << = 240 (means 1111 0000)
>> Binary Right Shift	The left operand's value is moved right by the number of bits specified by the right operand.	a >> = 15 (means 0000 1111)

Example

```
#!/usr/bin/python3

a = 60            # 60 = 0011 1100
b = 13            # 13 = 0000 1101
print ('a=',a,':',bin(a),'b=',b,':',bin(b))
c = 0

c = a & b;        # 12 = 0000 1100
print ("result of AND is ", c,':',bin(c))

c = a | b;        # 61 = 0011 1101
print ("result of OR is ", c,':',bin(c))
```

```
c = a ^ b;        # 49 = 0011 0001
print ("result of EXOR is ", c,':',bin(c))

c = ~a;           # -61 = 1100 0011
print ("result of COMPLEMENT is ", c,':',bin(c))

c = a << 2;       # 240 = 1111 0000
print ("result of LEFT SHIFT is ", c,':',bin(c))

c = a >> 2;       # 15 = 0000 1111
print ("result of RIGHT SHIFT is ", c,':',bin(c))
```

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

```
a= 60 : 0b111100 b= 13 : 0b1101
result of AND is  12 : 0b1100
result of OR is  61 : 0b111101
result of EXOR is  49 : 0b110001
result of COMPLEMENT is  -61 : -0b111101
result of LEFT SHIFT is  240 : 0b11110000
result of RIGHT SHIFT is  15 : 0b111
```

Python Logical Operators

The following logical operators are supported by Python language. Assume variable a holds True and variable b holds False then-

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is False.
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 True.

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-

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 find 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.

Example

```
#!/usr/bin/python3

a = 10
b = 20
list = [1, 2, 3, 4, 5 ]

if ( a in list ):
   print ("Line 1 - a is available in the given list")
else:
   print ("Line 1 - a is not available in the given list")

if ( b not in list ):
   print ("Line 2 - b is not available in the given list")
else:
   print ("Line 2 - b is available in the given list")

c=b/a
if ( c in list ):
   print ("Line 3 - a is available in the given list")
else:
print ("Line 3 - a is not available in the given list")
```

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

```
Line 1 - a is not available in the given list
Line 2 - b is not available in the given list
Line 3 - a is available in the given list
```

Python Identity Operators

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

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).

Example

```
#!/usr/bin/python3

a = 20
b = 20
print ('Line 1','a=',a,':',id(a), 'b=',b,':',id(b))

if ( a is b ):
   print ("Line 2 - a and b have same identity")
else:
   print ("Line 2 - a and b do not have same identity")

if ( id(a) == id(b) ):
   print ("Line 3 - a and b have same identity")
else:
   print ("Line 3 - a and b do not have same identity")
```

```
b = 30
print ('Line 4','a=',a,':',id(a), 'b=',b,':',id(b))

if ( a is not b ):
   print ("Line 5 - a and b do not have same identity")
else:
   print ("Line 5 - a and b have same identity")
```

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

```
Line 1 a= 20 : 1594701888 b= 20 : 1594701888
Line 2 - a and b have same identity
Line 3 - a and b have same identity
Line 4 a= 20 : 1594701888 b= 30 : 1594702048
Line 5 - a and b do not have same identity
```

Python Operators Precedence

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

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Ccomplement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^ |	Bitwise exclusive `OR' and regular `OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Operator precedence affects the evaluation of an an expression.

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because the operator * has higher precedence than +, so it first multiplies 3*2 and then is added to 7.

Here, the operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom.

Example

```
#!/usr/bin/python3

a = 20
b = 10
c = 15
d = 5

print ("a:%d b:%d c:%d d:%d" % (a,b,c,d ))
e = (a + b) * c / d       #( 30 * 15 ) / 5
print ("Value of (a + b) * c / d is ",  e)

e = ((a + b) * c) / d     # (30 * 15 ) / 5
print ("Value of ((a + b) * c) / d is ",  e)

e = (a + b) * (c / d)    # (30) * (15/5)
print ("Value of (a + b) * (c / d) is ",  e)

e = a + (b * c) / d      #  20 + (150/5)
print ("Value of a + (b * c) / d is ",  e)
```

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

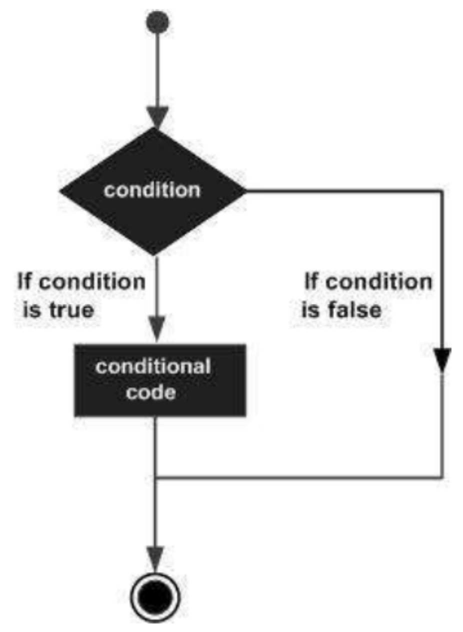
```
a:20 b:10 c:15 d:5
Value of (a + b) * c / d is  90.0
Value of ((a + b) * c) / d is  90.0
Value of (a + b) * (c / d) is  90.0
Value of a + (b * c) / d is  50.0
```

7. Python 3 – Decision Making

Decision-making is the anticipation of conditions occurring during the execution of a program and specified actions taken according to the conditions.

Decision structures evaluate multiple expressions, which produce TRUE or FALSE as the outcome. You need to determine which action to take and which statements to execute if the 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 any **zero** or **null values** as FALSE value.

Python programming language provides the following types of decision-making statements.

Statement	Description
if statements	An if statement consists of a Boolean expression followed by one or more statements.
if...else statements	An if statement can be followed by an optional else statement, which executes when the boolean expression is FALSE.

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 statement quickly.

IF Statement

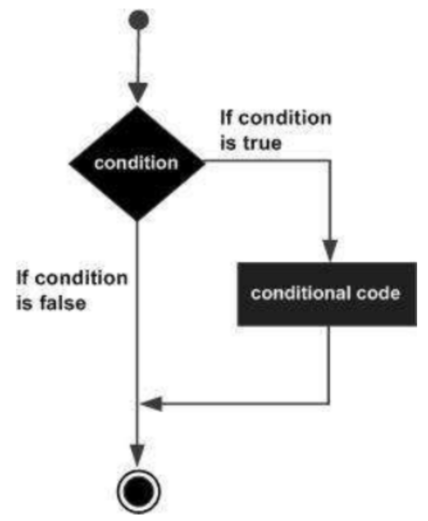
The IF statement is similar to that of other languages. The **if** statement contains a logical expression using which the data is compared and a decision is made based on the result of the comparison.

Syntax

```
if expression:
   statement(s)
```

If the boolean expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed. In Python, statements in a block are uniformly indented after the : symbol. If boolean expression evaluates to FALSE, then the first set of code after the end of block is executed.

Flow Diagram



Example

```
#!/usr/bin/python3
var1 = 100
if var1:
   print ("1 - Got a true expression value")
   print (var1)
```

```
var2 = 0
if var2:
   print ("2 - Got a true expression value")
   print (var2)
print ("Good bye!")
```

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

```
1 - Got a true expression value
100
Good bye!
```

IF...ELIF...ELSE Statements

An **else** statement can be combined with an **if** statement. An **else** statement contains a block of code that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.

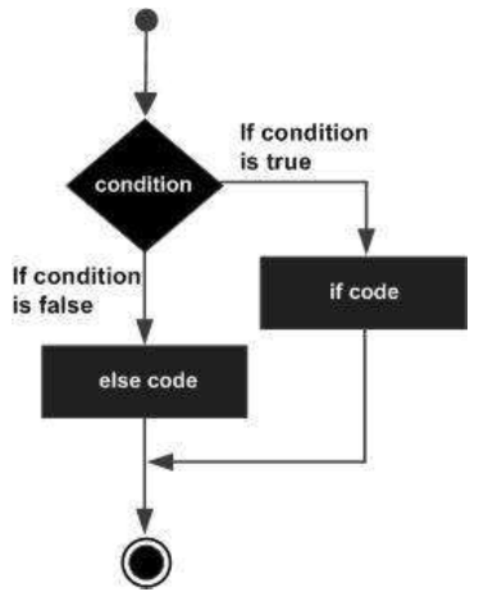
The else statement is an optional statement and there could be at the most only one **else** statement following **if**.

Syntax

The syntax of the **if...else** statement is-

```
if expression:
   statement(s)
else:
   statement(s)
```

Flow Diagram



Example

```
#!/usr/bin/python3
amount=int(input("Enter amount: "))
if amount<1000:
    discount=amount*0.05
    print ("Discount",discount)
else:
    discount=amount*0.10
    print ("Discount",discount)

print ("Net payable:",amount-discount)
```

In the above example, discount is calculated on the input amount. Rate of discount is 5%, if the amount is less than 1000, and 10% if it is above 10000. When the above code is executed, it produces the following result-

```
Enter amount: 600
Discount 30.0
Net payable: 570.0
Enter amount: 1200
Discount 120.0
```

```
Net payable: 1080.0
```

The elif Statement

The **elif** statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.

Similar to the **else**, the **elif** statement is optional. However, unlike **else**, for which there can be at the most one statement, there can be an arbitrary number of **elif** statements following an **if**.

Syntax

```
if expression1:
   statement(s)
elif expression2:
   statement(s)
elif expression3:
   statement(s)
else:
   statement(s)
```

Core Python does not provide switch or case statements as in other languages, but we can use if..elif...statements to simulate switch case as follows-

Example

```
#!/usr/bin/python3
amount=int(input("Enter amount: "))

if amount<1000:
    discount=amount*0.05
    print ("Discount",discount)
elif amount<5000:
    discount=amount*0.10
    print ("Discount",discount)
else:
    discount=amount*0.15
    print ("Discount",discount)
print ("Net payable:",amount-discount)
```

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

```
Enter amount: 600
Discount 30.0
Net payable: 570.0

Enter amount: 3000
Discount 300.0
Net payable: 2700.0

Enter amount: 6000
Discount 900.0
Net payable: 5100.0
```

Nested IF Statements

There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested **if** construct.

In a nested **if** construct, you can have an **if...elif...else** construct inside another **if...elif...else** construct.

Syntax

The syntax of the nested if...elif...else construct may be-

```
if expression1:
   statement(s)
   if expression2:
      statement(s)
   elif expression3:
      statement(s)
   else
      statement(s)
elif expression4:
   statement(s)
else:
   statement(s)
```

Example

```
# !/usr/bin/python3
num=int(input("enter number"))
```

```
if num%2==0:
    if num%3==0:
        print ("Divisible by 3 and 2")
    else:
        print ("divisible by 2 not divisible by 3")
else:
    if num%3==0:
        print ("divisible by 3 not divisible by 2")
    else:
        print  ("not Divisible by 2 not divisible by 3")
```

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

```
enter number8
divisible by 2 not divisible by 3

enter number15
divisible by 3 not divisible by 2

enter number12
Divisible by 3 and 2

enter number5
not Divisible by 2 not divisible by 3
```

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/python3
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-

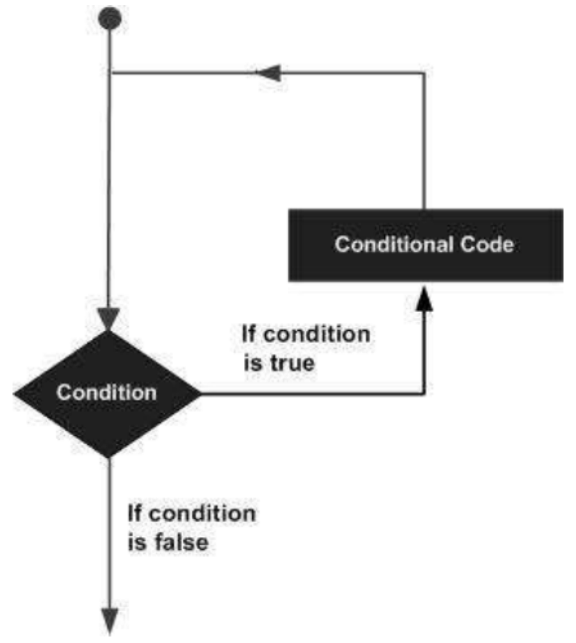
```
Value of expression is 100
Good bye!
```

8. Python 3 – 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 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 the following types of loops to handle looping requirements.

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
for loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

nested loops	You can use one or more loop inside any another while, or for loop.

while Loop Statements

A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

The syntax of a **while** loop in Python programming language is-

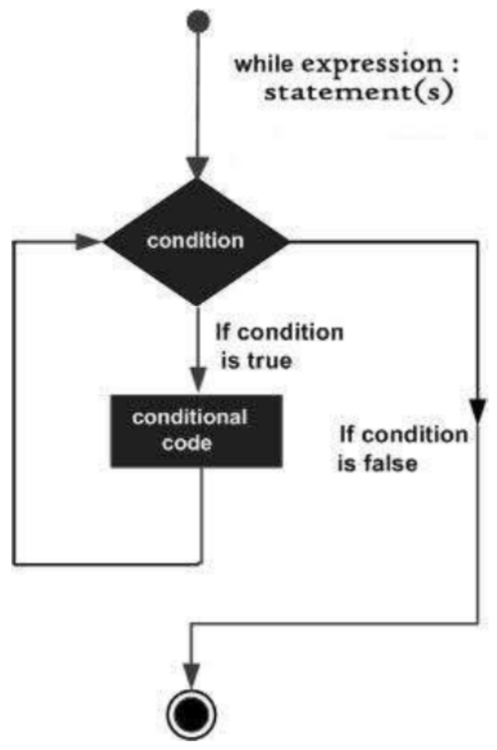
```
while expression:
   statement(s)
```

Here, **statement(s)** may be a single statement or a block of statements with uniform indent. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

Flow Diagram



Here, a key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

```
#!/usr/bin/python3

count = 0
while (count < 9):
   print ('The count is:', count)
   count = count + 1

print ("Good bye!")
```

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

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
Good bye!
```

The block here, consisting of the print and increment statements, is executed repeatedly until count is no longer less than 9. With each iteration, the current value of the index count is displayed and then increased by 1.

The Infinite Loop

A loop becomes infinite loop if a condition never becomes FALSE. You must be cautious when using while loops because of the possibility that this condition never resolves to a FALSE value. This results in a loop that never ends. Such a loop is called an infinite loop.

An infinite loop might be useful in client/server programming where the server needs to run continuously so that client programs can communicate with it as and when required.

```
#!/usr/bin/python3
var = 1
while var == 1 :  # This constructs an infinite loop
   num = int(input("Enter a number  :"))
   print ("You entered: ", num)
print ("Good bye!")
```

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

```
Enter a number  :20
You entered:  20
Enter a number  :29
You entered:  29
Enter a number  :3
You entered:  3
Enter a number  :11
You entered:  11
Enter a number  :22
You entered:  22
Enter a number  :Traceback (most recent call last):
  File "examples\test.py", line 5, in
    num = int(input("Enter a number  :"))
KeyboardInterrupt
```

The above example goes in an infinite loop and you need to use CTRL+C to exit the program.

Using else Statement with Loops

Python supports having an **else** statement associated with a loop statement.

- If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.

- If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

The following example illustrates the combination of an else statement with a while statement that prints a number as long as it is less than 5, otherwise the else statement gets executed.

```
#!/usr/bin/python3
count = 0
while count < 5:
   print (count, " is  less than 5")
   count = count + 1
else:
   print (count, " is not less than 5")
```

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

```
0 is less than 5
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
5 is not less than 5
```

Single Statement Suites

Similar to the **if** statement syntax, if your **while** clause consists only of a single statement, it may be placed on the same line as the while header.

Here is the syntax and example of a **one-line while** clause-

```
#!/usr/bin/python3
flag = 1
while (flag): print ('Given flag is really true!')
print ("Good bye!")
```

The above example goes into an infinite loop and you need to press CTRL+C keys to exit.

for Loop Statements

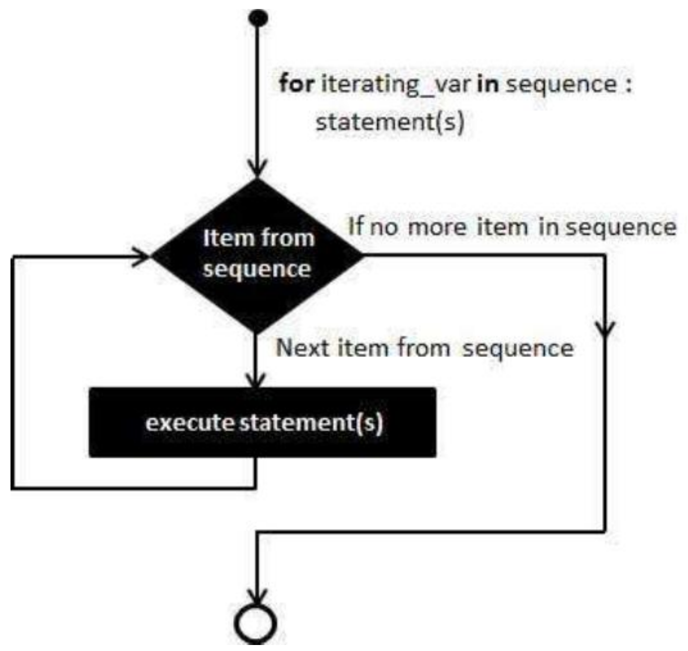
The for statement in Python has the ability to iterate over the items of any sequence, such as a list or a string.

Syntax

```
for iterating_var in sequence:
   statements(s)
```

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable *iterating_var*. Next, the statements block is executed. Each item in the list is assigned to *iterating_var*, and the statement(s) block is executed until the entire sequence is exhausted.

Flow Diagram



The range() function

The built-in function range() is the right function to iterate over a sequence of numbers. It generates an iterator of arithmetic progressions.

```
>>> range(5)
range(0, 5)
>>> list(range(5))
[0, 1, 2, 3, 4]
```

range() generates an iterator to progress integers starting with 0 upto n-1. To obtain a list object of the sequence, it is typecasted to list(). Now this list can be iterated using the for statement.

```
>>> for var in list(range(5)):
     print (var)
```

This will produce the following output.

```
0
1
2
3
4
```

Example

```
#!/usr/bin/python3
for letter in 'Python':     # traversal of a string sequence
   print ('Current Letter :', letter)
print()
fruits = ['banana', 'apple',  'mango']
for fruit in fruits:        # traversal of List sequence
   print ('Current fruit :', fruit)

print ("Good bye!")
```

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

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n

Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Iterating by Sequence Index

An alternative way of iterating through each item is by index offset into the sequence itself. Following is a simple example-

```
#!/usr/bin/python3
fruits = ['banana', 'apple',  'mango']
for index in range(len(fruits)):
   print ('Current fruit :', fruits[index])
print ("Good bye!")
```

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

```
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Here, we took the assistance of the len() built-in function, which provides the total number of elements in the tuple as well as the range() built-in function to give us the actual sequence to iterate over.

Using else Statement with Loops

Python supports having an else statement associated with a loop statement.

- If the **else** statement is used with a **for** loop, the **else** block is executed only if for loops terminates normally (and not by encountering break statement).
- If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

The following example illustrates the combination of an else statement with a **for** statement that searches for even number in given list.

```
#!/usr/bin/python3
numbers=[11,33,55,39,55,75,37,21,23,41,13]
for num in numbers:
    if num%2==0:
        print ('the list contains an even number')
        break
else:
    print ('the list doesnot contain even number')
```

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

```
the list does not contain even number
```

Nested loops

Python programming language allows the use of one loop inside another loop. The following section shows a few examples to illustrate the concept.

Syntax

```
for iterating_var in sequence:
   for iterating_var in sequence:
      statements(s)
   statements(s)
```

The syntax for a nested while loop statement in Python programming language is as follows-

```
while expression:
   while expression:
      statement(s)
   statement(s)
```

A final note on loop nesting is that you can put any type of loop inside any other type of loop. For example a **for** loop can be inside a while loop or vice versa.

Example

The following program uses a nested-for loop to display multiplication tables from 1-10.

```
#!/usr/bin/python3
import sys
```

```
for i in range(1,11):
    for j in range(1,11):
        k=i*j
        print (k, end=' ')
    print()
```

The print() function inner loop has **end=' '** which appends a space instead of default newline. Hence, the numbers will appear in one row.

Last print() will be executed at the end of inner for loop.

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

```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

Loop Control Statements

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

Python supports the following control statements.

Control Statement	Description
break statement	Terminates the loop statement and transfers execution to the statement immediately following the loop.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
pass statement	The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

Let us go through the loop control statements briefly.

break statement

The **break** statement is used for premature termination of the current loop. After abandoning the loop, execution at the next statement is resumed, just like the traditional break statement in C.

The most common use of break is when some external condition is triggered requiring a hasty exit from a loop. The **break** statement can be used in both *while* and *for* loops.

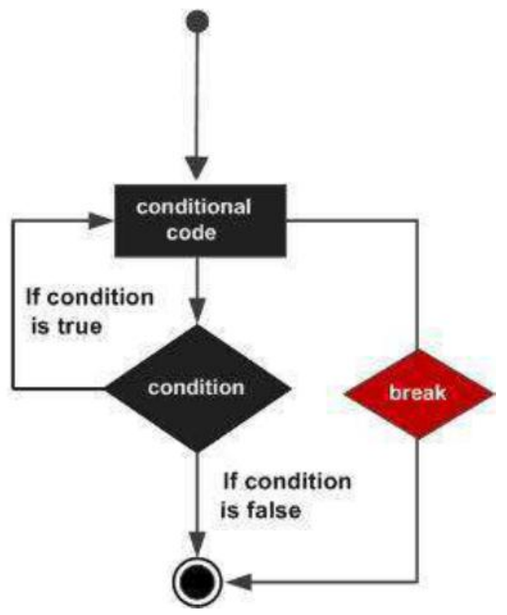
If you are using nested loops, the break statement stops the execution of the innermost loop and starts executing the next line of the code after the block.

Syntax

The syntax for a **break** statement in Python is as follows-

```
break
```

Flow Diagram



Example

```
#!/usr/bin/python3
for letter in 'Python':     # First Example
   if letter == 'h':
      break
   print ('Current Letter :', letter)

var = 10                    # Second Example
while var > 0:
   print ('Current variable value :', var)
   var = var -1
   if var == 5:
      break

print ("Good bye!")
```

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

```
Current Letter : P
Current Letter : y
Current Letter : t
```

```
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!
```

The following program demonstrates the use of break in a for loop iterating over a list. User inputs a number, which is searched in the list. If it is found, then the loop terminates with the 'found' message.

```
#!/usr/bin/python3
no=int(input('any number: '))
numbers=[11,33,55,39,55,75,37,21,23,41,13]
for num in numbers:
    if num==no:
        print ('number found in list')
        break
else:
    print ('number not found in list')
```

The above program will produce the following output-

```
any number: 33
number found in list
any number: 5
number not found in list
```

continue Statement

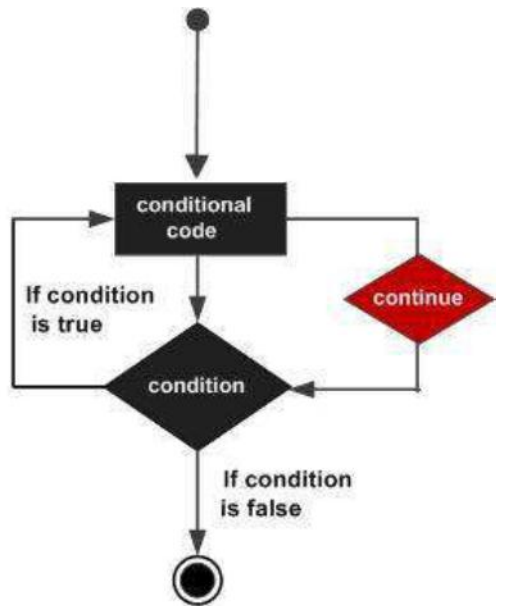
The **continue** statement in Python returns the control to the beginning of the current loop. When encountered, the loop starts next iteration without executing the remaining statements in the current iteration.

The **continue** statement can be used in both *while* and *for* loops.

Syntax

```
continue
```

Flow Diagram



Example

```
#!/usr/bin/python3

for letter in 'Python':     # First Example
   if letter == 'h':
      continue
   print ('Current Letter :', letter)

var = 10                    # Second Example
while var > 0:
   var = var -1
   if var == 5:
      continue
   print ('Current variable value :', var)
print ("Good bye!")
```

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

```
Current Letter : P
```

```
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Current variable value : 0
Good bye!
```

pass Statement

It is used when a statement is required syntactically but you do not want any command or code to execute.

The **pass** statement is a *null* operation; nothing happens when it executes. The **pass** statement is also useful in places where your code will eventually go, but has not been written yet i.e. in stubs).

Syntax

```
pass
```

Example

```
#!/usr/bin/python3

for letter in 'Python':
   if letter == 'h':
      pass
      print ('This is pass block')
   print ('Current Letter :', letter)

print ("Good bye!")
```

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

```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
Good bye!
```

Iterator and Generator

Iterator is an object, which allows a programmer to traverse through all the elements of a collection, regardless of its specific implementation. In Python, an iterator object implements two methods, **iter()** and **next().**

String, List or Tuple objects can be used to create an Iterator.

```
list=[1,2,3,4]
it = iter(list) # this builds an iterator object
print (next(it)) #prints next available element in iterator
Iterator object can be traversed using regular for statement
!usr/bin/python3
for x in it:
   print (x, end=" ")
or using next() function
while True:
   try:
      print (next(it))
   except StopIteration:
      sys.exit() #you have to import sys module for this
```

A **generator** is a function that produces or yields a sequence of values using yield method.

When a generator function is called, it returns a generator object without even beginning execution of the function. When the next() method is called for the first time, the function starts executing, until it reaches the yield statement, which returns the yielded value. The yield keeps track i.e. remembers the last execution and the second next() call continues from previous value.

The following example defines a generator, which generates an iterator for all the Fibonacci numbers.

```
!usr/bin/python3
```

```
import sys
def fibonacci(n): #generator function
    a, b, counter = 0, 1, 0
    while True:
        if (counter > n):
            return
        yield a
        a, b = b, a + b
        counter += 1
f = fibonacci(5) #f is iterator object

while True:
   try:
      print (next(f), end=" ")
   except StopIteration:
      sys.exit()
```

9. Python 3 – Numbers

Number data types store numeric values. They are immutable data types. This means, 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 different numerical types-

- **int (signed integers)**: They are often called just integers or **ints**. They are positive or negative whole numbers with no decimal point. Integers in Python 3 are of unlimited size. Python 2 has two integer types - int and long. There is no **'long integer'** in Python 3 anymore.

- **float (floating point real values)** : Also called floats, they represent real numbers and are written with a decimal point dividing the integer and the 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.

It is possible to represent an integer in hexa-decimal or octal form.

```
>>> number = 0xA0F #Hexa-decimal
>>> number
2575

>>> number=0o37 #Octal
>>> number
```

```
31
```

Examples

Here are some examples of numbers.

int	float	complex
10	0.0	3.14j
100	15.20	45.j
-786	-21.9	9.322e-36j
080	32.3+e18	.876j
-0490	-90.	-.6545+0J
-0x260	-32.54e100	3e+26J
0x69	70.2-E12	4.53e-7j

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. 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 the following functions that perform mathematical calculations.

Function	Returns (Description)
abs(x)	The absolute value of x: the (positive) distance between x and zero.
ceil(x)	The ceiling of x: the smallest integer not less than x.
cmp(x, y)	-1 if x < y, 0 if x == y, or 1 if x > y. **Deprecated** in Python 3; Instead use **return (x>y)-(x<y).**
exp(x)	The exponential of x: e^x
fabs(x)	The absolute value of x.
floor(x)	The floor of x: the largest integer not greater than x.
log(x)	The natural logarithm of x, for x> 0.
log10(x)	The base-10 logarithm of x for x> 0.
max(x1, x2,...)	The largest of its arguments: the value closest to positive infinity.
min(x1, x2,...)	The smallest of its arguments: the value closest to negative infinity.
modf(x)	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.
pow(x, y)	The value of x**y.
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.
sqrt(x)	The square root of x for x > 0.

Let us learn about these functions in detail.

Number abs() Method

Description

The **abs()** method returns the absolute value of x i.e. the positive distance between x and zero.

Syntax

Following is the syntax for abs() method-

```
abs( x )
```

Parameters

x - This is a numeric expression.

Return Value

This method returns the absolute value of x.

Example

The following example shows the usage of the abs() method.

```
#!/usr/bin/python3
print ("abs(-45) : ", abs(-45))
print ("abs(100.12) : ", abs(100.12))
```

When we run the above program, it produces the following result-

```
abs(-45) :  45
abs(100.12) :  100.12
```

Number ceil() Method

Description

The **ceil()** method returns the ceiling value of x i.e. the smallest integer not less than x.

Syntax

Following is the syntax for the **ceil()** method-

```
import math
math.ceil( x )
```

Note: This function is not accessible directly, so we need to import math module and then we need to call this function using the math static object.

Parameters

x - This is a numeric expression.

Return Value

This method returns the smallest integer not less than x.

Example

The following example shows the usage of the ceil() method.

```
#!/usr/bin/python3
import math   # This will import math module
print ("math.ceil(-45.17) : ", math.ceil(-45.17))
print ("math.ceil(100.12) : ", math.ceil(100.12))
print ("math.ceil(100.72) : ", math.ceil(100.72))
print ("math.ceil(math.pi) : ", math.ceil(math.pi))
```

When we run the above program, it produces the following result-

```
math.ceil(-45.17) :  -45
math.ceil(100.12) :  101
math.ceil(100.72) :  101
math.ceil(math.pi) :  4
```

Number exp() Method

Description

The **exp()** method returns exponential of x: e^x.

Syntax

Following is the syntax for the exp() method-

```
import math
math.exp( x )
```

Note: This function is not accessible directly. Therefore, we need to import the math module and then we need to call this function using the math static object.

Parameters

X - This is a numeric expression.

Return Value

This method returns exponential of x: ex.

Example

The following example shows the usage of exp() method.

```
#!/usr/bin/python3
import math   # This will import math module
print ("math.exp(-45.17) : ", math.exp(-45.17))
print ("math.exp(100.12) : ", math.exp(100.12))
print ("math.exp(100.72) : ", math.exp(100.72))
print ("math.exp(math.pi) : ", math.exp(math.pi))
```

When we run the above program, it produces the following result-

```
math.exp(-45.17) :  2.4150062132629406e-20
math.exp(100.12) :  3.0308436140742566e+43
math.exp(100.72) :  5.522557130248187e+43
math.exp(math.pi) :  23.140692632779267
```

Number fabs() Method

Description

The **fabs()** method returns the absolute value of x. Although similar to the abs() function, there are differences between the two functions. They are-

- abs() is a built in function whereas fabs() is defined in math module.
- fabs() function works only on float and integer whereas abs() works with complex number also.

Syntax

Following is the syntax for the fabs() method-

```
import math
math.fabs( x )
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This is a numeric value.

Return Value

This method returns the absolute value of x.

Example

The following example shows the usage of the fabs() method.

```
#!/usr/bin/python3
import math   # This will import math module
print ("math.fabs(-45.17) : ", math.fabs(-45.17))
print ("math.fabs(100.12) : ", math.fabs(100.12))
print ("math.fabs(100.72) : ", math.fabs(100.72))
print ("math.fabs(math.pi) : ", math.fabs(math.pi))
```

When we run the above program, it produces following result-

```
math.fabs(-45.17) :  45.17
math.fabs(100) :  100.0
math.fabs(100.72) :  100.72
math.fabs(math.pi) :  3.141592653589793
```

Number floor() Method

Description

The **floor()** method returns the floor of **x** i.e. the largest integer not greater than x.

Syntax

Following is the syntax for the **floor()** method-

```
import math
math.floor( x )
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This is a numeric expression.

Return Value

This method returns the largest integer not greater than x.

Example

The following example shows the usage of the floor() method.

```
#!/usr/bin/python3
import math   # This will import math module
print ("math.floor(-45.17) : ", math.floor(-45.17))
print ("math.floor(100.12) : ", math.floor(100.12))
print ("math.floor(100.72) : ", math.floor(100.72))
print ("math.floor(math.pi) : ", math.floor(math.pi))
```

When we run the above program, it produces the following result-

```
math.floor(-45.17) :  -46
math.floor(100.12) :  100
math.floor(100.72) :  100
math.floor(math.pi) :  3
```

Number log() Method

Description

The **log()** method returns the natural logarithm of x, for x > 0.

Syntax

Following is the syntax for the **log()** method-

```
import math
math.log( x )
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This is a numeric expression.

Return Value

This method returns natural logarithm of x, for x > 0.

Example

The following example shows the usage of the log() method.

```
#!/usr/bin/python3
import math   # This will import math module
print ("math.log(100.12) : ", math.log(100.12))
print ("math.log(100.72) : ", math.log(100.72))
print ("math.log(math.pi) : ", math.log(math.pi))
```

When we run the above program, it produces the following result-

```
math.log(100.12) :  4.6063694665635735
math.log(100.72) :  4.612344389736092
math.log(math.pi) :  1.1447298858494002
```

Number log10() Method

Description

The **log10()** method returns base-10 logarithm of x for x > 0.

Syntax

Following is the syntax for **log10()** method-

```
import math
math.log10( x )
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This is a numeric expression.

Return Value

This method returns the base-10 logarithm of x for x > 0.

Example

The following example shows the usage of the **log10()** method.

```
#!/usr/bin/python3
import math   # This will import math module
print ("math.log10(100.12) : ", math.log10(100.12))
print ("math.log10(100.72) : ", math.log10(100.72))
print ("math.log10(119) : ", math.log10(119))
print ("math.log10(math.pi) : ", math.log10(math.pi))
```

When we run the above program, it produces the following result-

```
math.log10(100.12) :  2.0005208409361854
math.log10(100.72) :  2.003115717099806
math.log10(119) :  2.0755469613925306
math.log10(math.pi) :  0.49714987269413385
```

Number max() Method

Description

The **max()** method returns the largest of its arguments i.e. the value closest to positive infinity.

Syntax

Following is the syntax for **max()** method-

```
max( x, y, z, .... )
```

Parameters

- **x** - This is a numeric expression.
- **y** - This is also a numeric expression.
- **z** - This is also a numeric expression.

Return Value

This method returns the largest of its arguments.

Example

The following example shows the usage of the max() method.

```
#!/usr/bin/python3
print ("max(80, 100, 1000) : ", max(80, 100, 1000))
print ("max(-20, 100, 400) : ", max(-20, 100, 400))
print ("max(-80, -20, -10) : ", max(-80, -20, -10))
print ("max(0, 100, -400) : ", max(0, 100, -400))
```

When we run the above program, it produces the following result-

```
max(80, 100, 1000) :  1000
max(-20, 100, 400) :  400
max(-80, -20, -10) :  -10
max(0, 100, -400) :  100
```

Number min() Method

Description

The method **min()** returns the smallest of its arguments i.e. the value closest to negative infinity.

Syntax

Following is the syntax for the **min()** method-

```
min( x, y, z, .... )
```

Parameters

- **x** - This is a numeric expression.
- **y** - This is also a numeric expression.
- **z** - This is also a numeric expression.

Return Value

This method returns the smallest of its arguments.

Example

The following example shows the usage of the **min()** method.

```
#!/usr/bin/python3
print ("min(80, 100, 1000) : ", min(80, 100, 1000))
print ("min(-20, 100, 400) : ", min(-20, 100, 400))
print ("min(-80, -20, -10) : ", min(-80, -20, -10))
print ("min(0, 100, -400) : ", min(0, 100, -400))
```

When we run the above program, it produces the following result-

```
min(80, 100, 1000) :  80
min(-20, 100, 400) :  -20
min(-80, -20, -10) :  -80
min(0, 100, -400) :  -400
```

Number modf() Method

Description

The **modf()** method returns 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.

Syntax

Following is the syntax for the **modf()** method-

```
import math
math.modf( x )
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This is a numeric expression.

Return Value

This method returns the fractional and integer parts of x in a two-item tuple. Both the parts have the same sign as x. The integer part is returned as a float.

Example

The following example shows the usage of the **modf()** method.

```
#!/usr/bin/python3
import math   # This will import math module
print ("math.modf(100.12) : ", math.modf(100.12))
print ("math.modf(100.72) : ", math.modf(100.72))
print ("math.modf(119) : ", math.modf(119))
print ("math.modf(math.pi) : ", math.modf(math.pi))
```

When we run the above program, it produces the following result-

```
math.modf(100.12) :  (0.12000000000000455, 100.0)
math.modf(100.72) :  (0.7199999999999989, 100.0)
math.modf(119) :  (0.0, 119.0)
math.modf(math.pi) :  (0.14159265358979312, 3.0)
```

Number pow() Method

Return Value

This method returns the value of x^y.

Example

The following example shows the usage of the **pow()** method.

```
#!/usr/bin/python3
import math   # This will import math module
print ("math.pow(100, 2) : ", math.pow(100, 2))
print ("math.pow(100, -2) : ", math.pow(100, -2))
print ("math.pow(2, 4) : ", math.pow(2, 4))
print ("math.pow(3, 0) : ", math.pow(3, 0))
```

When we run the above program, it produces the following result-

```
math.pow(100, 2) :  10000.0
math.pow(100, -2) :  0.0001
math.pow(2, 4) :  16.0
math.pow(3, 0) :  1.0
```

Number round() Method

Description

round() is a built-in function in Python. It returns x rounded to n digits from the decimal point.

Syntax

Following is the syntax for the round() method-

```
round( x [, n]  )
```

Parameters

- **x** - This is a numeric expression.
- **n** - Represents number of digits from decimal point up to which x is to be rounded. Default is 0.

Return Value

This method returns x rounded to n digits from the decimal point.

Example

The following example shows the usage of **round()** method.

```
#!/usr/bin/python3
print ("round(70.23456) : ", round(70.23456))
print ("round(56.659,1) : ", round(56.659,1))
print ("round(80.264, 2) : ", round(80.264, 2))
print ("round(100.000056, 3) : ", round(100.000056, 3))
print ("round(-100.000056, 3) : ", round(-100.000056, 3))
```

When we run the above program, it produces the following result-

```
round(70.23456) :  70
round(56.659,1) :  56.7
round(80.264, 2) :  80.26
round(100.000056, 3) :  100.0
round(-100.000056, 3) :  -100.0
```

Number sqrt() Method

Description

The **sqrt()** method returns the square root of x for x > 0.

Syntax

Following is the syntax for **sqrt()** method-

```
import math
math.sqrt( x )
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This is a numeric expression.

Return Value

This method returns square root of x for x > 0.

Example

The following example shows the usage of sqrt() method.

```
#!/usr/bin/python3
import math   # This will import math module
print ("math.sqrt(100) : ", math.sqrt(100))
print ("math.sqrt(7) : ", math.sqrt(7))
print ("math.sqrt(math.pi) : ", math.sqrt(math.pi))
```

When we run the above program, it produces the following result-

```
math.sqrt(100) :  10.0
math.sqrt(7) :  2.6457513110645907
math.sqrt(math.pi) :  1.7724538509055159
```

Random Number Functions

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

Function	Description
choice(seq)	A random item from a list, tuple, or string.
randrange ([start,] stop [,step])	A randomly selected element from range(start, stop, step).
random()	A random float r, such that 0 is less than or equal to r and r is less than 1.
seed([x])	Sets the integer starting value used in generating random numbers. Call this function before calling any other random module function. Returns None.
shuffle(lst)	Randomizes the items of a list in place. Returns None.
uniform(x, y)	A random float r, such that x is less than or equal to r and r is less than y.

Number choice() Method

Description

The **choice()** method returns a random item from a list, tuple, or string.

Syntax

Following is the syntax for **choice()** method-

```
choice( seq )
```

Note: This function is not accessible directly, so we need to import the random module and then we need to call this function using the random static object.

Parameters

seq - This could be a list, tuple, or string...

Return Value

This method returns a random item.

Example

The following example shows the usage of the **choice()** method.

```
#!/usr/bin/python3
import random
print ("returns a random number from range(100) : ",random.choice(range(100)))
print ("returns random element from list [1, 2, 3, 5, 9]) : ", random.choice([1,
2, 3, 5, 9]))
print ("returns random character from string 'Hello World' : ",
random.choice('Hello World'))
```

When we run the above program, it produces a result similar to the following-

```
returns a random number from range(100) :  19
returns random element from list [1, 2, 3, 5, 9]) :  9
returns random character from string 'Hello World' :  r
```

Number randrange() Method

Description

The **randrange()** method returns a randomly selected element from range(start, stop, step).

Syntax

Following is the syntax for the **randrange()** method-

```
randrange ([start,] stop [,step])
```

Note: This function is not accessible directly, so we need to import the random module and then we need to call this function using the random static object.

Parameters

- **start** - Start point of the range. This would be included in the range. Default is 0.
- **stop** - Stop point of the range. This would be excluded from the range.
- **step** - Value with which number is incremented. Default is 1.

Return Value

This method returns a random item from the given range.

Example

The following example shows the usage of the randrange() method.

```
#!/usr/bin/python3
import random
# randomly select an odd number between 1-100
print ("randrange(1,100, 2) : ", random.randrange(1, 100, 2))
# randomly select a number between 0-99
print ("randrange(100) : ", random.randrange(100))
```

When we run the above program, it produces the following result-

```
randrange(1,100, 2) :  83
randrange(100) :  93
```

Number random() Method

Description

The **random()** method returns a random floating point number in the range [0.0, 1.0].

Syntax

Following is the syntax for the random() method-

```
random ( )
```

Note: This function is not accessible directly, so we need to import the random module and then we need to call this function using the random static object.

Parameters

NA

Return Value

This method returns a random float r, such that 0.0 <= r <= 1.0

Example

The following example shows the usage of the **random()** method.

```
#!/usr/bin/python3
import random
# First random number
print ("random() : ", random.random())
# Second random number
print ("random() : ", random.random())
```

When we run the above program, it produces the following result-

```
random() :  0.281954791393
random() :  0.309090465205
```

Number seed() Method

Description

The **seed()** method initializes the basic random number generator. Call this function before calling any other random module function.

Syntax

Following is the syntax for the seed() method-

```
seed ([x], [y])
```

Note: This function initializes the basic random number generator.

Parameters

- **x -** This is the seed for the next random number. If omitted, then it takes system time to generate the next random number. If x is an int, it is used directly.
- **Y -** This is version number (default is 2). str, byte or byte array object gets converted in int. Version 1 used hash() of x.

Return Value

This method does not return any value.

Example

The following example shows the usage of the seed() method.

```
#!/usr/bin/python3
import random
random.seed()
print ("random number with default seed", random.random())
random.seed(10)
print ("random number with int seed", random.random())
random.seed("hello",2)
print ("random number with string seed", random.random())
```

When we run above program, it produces following result-

```
random number with default seed 0.2524977842762465
random number with int seed 0.5714025946899135
random number with string seed 0.3537754404730722
```

Number shuffle() Method

Description

The **shuffle()** method randomizes the items of a list in place.

Syntax

Following is the syntax for the **shuffle()** method-

```
shuffle (lst,[random])
```

Note: This function is not accessible directly, so we need to import the shuffle module and then we need to call this function using the random static object.

Parameters

- **lst** - This could be a list or tuple.
- **random** - This is an optional 0 argument function returning float between 0.0 - 1.0. Default is None.

Return Value

This method returns reshuffled list.

Example

The following example shows the usage of the shuffle() method.

```
#!/usr/bin/python3
import random
list = [20, 16, 10, 5];
random.shuffle(list)
print ("Reshuffled list : ",  list)
random.shuffle(list)
print ("Reshuffled list : ",  list)
```

When we run the above program, it produces the following result-

```
Reshuffled list :  [16, 5, 10, 20]
reshuffled list :  [20, 5, 10, 16]
```

Number uniform() Method

Description

The **uniform()** method returns a random float r, such that x is less than or equal to r and r is less than y.

Syntax

Following is the syntax for the **uniform()** method-

```
uniform(x, y)
```

Note: This function is not accessible directly, so we need to import the uniform module and then we need to call this function using the random static object.

Parameters

- **x** - Sets the lower limit of the random float.
- **y** - Sets the upper limit of the random float.

Return Value

This method returns a floating point number r such that x <=r < y.

Example

The following example shows the usage of the uniform() method.

```
#!/usr/bin/python3
import random
```

```
print ("Random Float uniform(5, 10) : ",  random.uniform(5, 10))
print ("Random Float uniform(7, 14) : ",  random.uniform(7, 14))
```

Let us run the above program. This will produce the following result-

```
Random Float uniform(5, 10) :  5.52615217015
Random Float uniform(7, 14) :  12.5326369199
```

Trigonometric Functions

Python includes the following functions that perform trigonometric calculations.

Function	Description
acos(x)	Return the arc cosine of x, in radians.
asin(x)	Return the arc sine of x, in radians.
atan(x)	Return the arc tangent of x, in radians.
atan2(y, x)	Return atan(y / x), in radians.
cos(x)	Return the cosine of x radians.
hypot(x, y)	Return the Euclidean norm, sqrt(x*x + y*y).
sin(x)	Return the sine of x radians.
tan(x)	Return the tangent of x radians.
degrees(x)	Converts angle x from radians to degrees.
radians(x)	Converts angle x from degrees to radians.

Number acos() Method

Description

The **acos()** method returns the arc cosine of x in radians.

Syntax

Following is the syntax for acos() method-

```
acos(x)
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This must be a numeric value in the range -1 to 1. If x is greater than 1 then it will generate 'math domain error'.

Return Value

This method returns arc cosine of x, in radians.

Example

The following example shows the usage of the acos() method.

```
#!/usr/bin/python3
import math
print ("acos(0.64) : ",  math.acos(0.64))
print ("acos(0) : ",  math.acos(0))
print ("acos(-1) : ",  math.acos(-1))
print ("acos(1) : ",  math.acos(1))
```

When we run the above program, it produces the following result-

```
acos(0.64) :  0.876298061168
acos(0) :  1.57079632679
acos(-1) :  3.14159265359
acos(1) :  0.0
```

Number asin() Method

Description

The **asin()** method returns the arc sine of x (in radians).

Syntax

Following is the syntax for the asin() method-

```
asin(x)
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function usingthe math static object.

Parameters

x - This must be a numeric value in the range -1 to 1. If x is greater than 1 then it will generate 'math domain error'.

Return Value

This method returns arc sine of x, in radians.

Example

The following example shows the usage of the asin() method.

```
#!/usr/bin/python3
import math
print ("asin(0.64) : ",  math.asin(0.64))
print ("asin(0) : ",  math.asin(0))
print ("asin(-1) : ",  math.asin(-1))
print ("asin(1) : ",  math.asin(1))
```

When we run the above program, it produces the following result-

```
asin(0.64) :  0.694498265627
asin(0) :  0.0
asin(-1) :  -1.57079632679
asin(1) :  1.5707963267
```

Number atan() Method

Description

The atan() method returns the arc tangent of x, in radians.

Syntax

Following is the syntax for atan() method-

```
atan(x)
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This must be a numeric value.

Return Value

This method returns arc tangent of x, in radians.

Example

The following example shows the usage of the atan() method.

```
#!/usr/bin/python3
import math
print ("atan(0.64) : ",  math.atan(0.64))
print ("atan(0) : ",  math.atan(0))
print ("atan(10) : ",  math.atan(10))
print ("atan(-1) : ",  math.atan(-1))
print ("atan(1) : ",  math.atan(1))
```

When we run the above program, it produces the following result-

```
atan(0.64) :  0.569313191101
atan(0) :  0.0
atan(10) :  1.4711276743
atan(-1) :  -0.785398163397
atan(1) :  0.785398163397
```

Number atan2() Method

Description

The **atan2()** method returns atan(y / x), in radians.

Syntax

Following is the syntax for atan2() method-

```
atan2(y, x)
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

- **y** - This must be a numeric value.
- **x** - This must be a numeric value.

Return Value

This method returns atan(y / x), in radians.

Example

The following example shows the usage of atan2() method.

```
#!/usr/bin/python3
import math
print ("atan2(-0.50,-0.50) : ",  math.atan2(-0.50,-0.50))
print ("atan2(0.50,0.50) : ",  math.atan2(0.50,0.50))
print ("atan2(5,5) : ",  math.atan2(5,5))
print ("atan2(-10,10) : ",  math.atan2(-10,10))
print ("atan2(10,20) : ",  math.atan2(10,20))
```

When we run the above program, it produces the following result-

```
atan2(-0.50,-0.50) :  -2.35619449019
atan2(0.50,0.50) :  0.785398163397
atan2(5,5) :  0.785398163397
atan2(-10,10) :  -0.785398163397
atan2(10,20) :  0.463647609001
```

Number cos() Method

Description

The **cos()** method returns the cosine of x radians.

Syntax

Following is the syntax for **cos()** method-

```
cos(x)
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This must be a numeric value.

Return Value

This method returns a numeric value between -1 and 1, which represents the cosine of the angle.

Example

The following example shows the usage of cos() method.

```
#!/usr/bin/python3
import math
print ("cos(3) : ",  math.cos(3))
print ("cos(-3) : ",  math.cos(-3))
print ("cos(0) : ",  math.cos(0))
print ("cos(math.pi) : ",  math.cos(math.pi))
print ("cos(2*math.pi) : ",  math.cos(2*math.pi))
```

When we run the above program, it produces the following result-

```
cos(3) :  -0.9899924966
cos(-3) :  -0.9899924966
cos(0) :  1.0
cos(math.pi) :  -1.0
cos(2*math.pi) :  1.0
```

Number hypot() Method

Description

The method hypot() return the Euclidean norm, sqrt(x*x + y*y). This is length of vector from origin to point (x,y)

Syntax

Following is the syntax for hypot() method-

```
hypot(x, y)
```

Note: This function is not accessible directly, so we need to import math module and then we need to call this function using math static object.

Parameters

- **x** - This must be a numeric value.
- **y** - This must be a numeric value.

Return Value

This method returns Euclidean norm, sqrt(x*x + y*y).

Example

The following example shows the usage of hypot() method.

```
#!/usr/bin/python3
import math
print ("hypot(3, 2) : ",  math.hypot(3, 2))
print ("hypot(-3, 3) : ",  math.hypot(-3, 3))
print ("hypot(0, 2) : ",  math.hypot(0, 2))
```

When we run the above program, it produces the following result-

```
hypot(3, 2) :  3.60555127546
hypot(-3, 3) :  4.24264068712
hypot(0, 2) :  2.0
```

Number sin() Method

Description

The **sin()** method returns the sine of x, in radians.

Syntax

Following is the syntax for sin() method-

```
sin(x)
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This must be a numeric value.

Return Value

This method returns a numeric value between -1 and 1, which represents the sine of the parameter x.

Example

The following example shows the usage of sin() method.

```
#!/usr/bin/python3
import math
print ("sin(3) : ",  math.sin(3))
print ("sin(-3) : ",  math.sin(-3))
print ("sin(0) : ",  math.sin(0))
```

```
print ("sin(math.pi) : ",  math.sin(math.pi))
print ("sin(math.pi/2) : ",  math.sin(math.pi/2))
```

When we run the above program, it produces the following result-

```
sin(3) :  0.14112000806
sin(-3) :  -0.14112000806
sin(0) :  0.0
sin(math.pi) :  1.22460635382e-16
sin(math.pi/2) :  1
```

Number tan() Method

Description

The **tan()** method returns the tangent of x radians.

Syntax

Following is the syntax for tan() method.

```
tan(x)
```

Note: This function is not accessible directly, so we need to import math module and then we need to call this function using math static object.

Parameters

x - This must be a numeric value.

Return Value

This method returns a numeric value between -1 and 1, which represents the tangent of the parameter x.

Example

The following example shows the usage of tan() method.

```
#!/usr/bin/python3
import math
print ("(tan(3) : ",  math.tan(3))
print ("tan(-3) : ",  math.tan(-3))
print ("tan(0) : ",  math.tan(0))
print ("tan(math.pi) : ",  math.tan(math.pi))
print ("tan(math.pi/2) : ",  math.tan(math.pi/2))
print ("tan(math.pi/4) : ",  math.tan(math.pi/4))
```

When we run the above program, it produces the following result-

```
print ("(tan(3) : ",  math.tan(3))
print ("tan(-3) : ",  math.tan(-3))
print ("tan(0) : ",  math.tan(0))
print ("tan(math.pi) : ",  math.tan(math.pi))
print ("tan(math.pi/2) : ",  math.tan(math.pi/2))
print ("tan(math.pi/4) : ",  math.tan(math.pi/4))
```

Number degrees() Method

Description

The **degrees()** method converts angle x from radians to degrees..

Syntax

Following is the syntax for degrees() method-

```
degrees(x)
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This must be a numeric value.

Return Value

This method returns the degree value of an angle.

Example

The following example shows the usage of degrees() method.

```
#!/usr/bin/python3
import math
print ("degrees(3) : ",  math.degrees(3))
print ("degrees(-3) : ",  math.degrees(-3))
print ("degrees(0) : ",  math.degrees(0))
print ("degrees(math.pi) : ",  math.degrees(math.pi))
print ("degrees(math.pi/2) : ",  math.degrees(math.pi/2))
```

```
print ("degrees(math.pi/4) : ",  math.degrees(math.pi/4))
```

When we run the above program, it produces the following result-

```
degrees(3) :  171.88733853924697
degrees(-3) :  -171.88733853924697
degrees(0) :  0.0
degrees(math.pi) :  180.0
degrees(math.pi/2) :  90.0
degrees(math.pi/4) :  45.0
```

Number radians() Method

Description

The **radians()** method converts angle x from degrees to radians.

Syntax

Following is the syntax for radians() method-

```
radians(x)
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This must be a numeric value.

Return Value

This method returns radian value of an angle.

Example

The following example shows the usage of radians() method.

```
#!/usr/bin/python3
import math
print ("radians(3) : ",  math.radians(3))
print ("radians(-3) : ",  math.radians(-3))
print ("radians(0) : ",  math.radians(0))
print ("radians(math.pi) : ",  math.radians(math.pi))
print ("radians(math.pi/2) : ",  math.radians(math.pi/2))
print ("radians(math.pi/4) : ",  math.radians(math.pi/4))
```

When we run the above program, it produces the following result-

```
radians(3) :  0.0523598775598
radians(-3) :  -0.0523598775598
radians(0) :  0.0
radians(math.pi) :  0.0548311355616
radians(math.pi/2) :  0.0274155677808
radians(math.pi/4) :  0.0137077838904
```

Mathematical Constants

The module also defines two mathematical constants-

Constants	Description
pi	The mathematical constant pi.
e	The mathematical constant e.

10. Python 3 – 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/python3
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/python3
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 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 and print R'\n'prints \n

%	Format - Performs String formatting	See 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/python3
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/python3

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/python3
print ('C:\\nowhere')
```

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

```
C:\nowhere
```

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

```
#!/usr/bin/python3
```

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

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

```
C:\\nowhere
```

Unicode String

In Python 3, all strings are represented in Unicode. In Python 2 are stored internally as 8-bit ASCII, hence it is required to attach 'u' to make it Unicode. It is no longer necessary now.

Built-in String Methods

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

S. No.	Methods with Description
1	**capitalize()** Capitalizes first letter of string
2	**center(width, fillchar)** Returns a string padded with *fillchar* with the original string centered to a total of *width* columns.
3	**count(str, beg= 0,end=len(string))** 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')** Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding.
5	**encode(encoding='UTF-8',errors='strict')** 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))**

	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)** 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))** 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))** 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()** Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
12	**isdigit()** Returns true if the string contains only digits and false otherwise.
13	**islower()** Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
14	**isnumeric()** Returns true if a unicode string contains only numeric characters and false otherwise.

15	**isspace()** Returns true if string contains only whitespace characters and false otherwise.
16	**istitle()** Returns true if string is properly "titlecased" and false otherwise.
17	**isupper()** Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
18	**join(seq)** Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.
19	**len(string)** 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()** Converts all uppercase letters in string to lowercase.
22	**lstrip()** Removes all leading whitespace in string.
23	**maketrans()** Returns a translation table to be used in translate function.

24	**max(str)** Returns the max alphabetical character from the string str.
25	**min(str)** Returns the min alphabetical character from the string str.
26	**replace(old, new [, max])** 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))** Same as find(), but search backwards in string.
28	**rindex(str, beg=0, end=len(string))** Same as index(), but search backwards in string.
29	**rjust(width,[, fillchar])** Returns a space-padded string with the original string right-justified to a total of width columns.
30	**rstrip()** Removes all trailing whitespace of string.
31	**split(str="", num=string.count(str))** 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'))** 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))** 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])** Performs both lstrip() and rstrip() on string
35	**swapcase()** Inverts case for all letters in string.
36	**title()** Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.
37	**translate(table, deletechars="")** Translates string according to translation table str(256 chars), removing those in the del string.
38	**upper()** Converts lowercase letters in string to uppercase.
39	**zfill (width)** 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.

String capitalize() Method

It returns a copy of the string with only its first character capitalized.

Syntax

```
str.capitalize()
```

Parameters

NA

Return Value

string

Example

```
#!/usr/bin/python3
str = "this is string example....wow!!!"
print ("str.capitalize() : ", str.capitalize())
```

Result

```
str.capitalize() :  This is string example....wow!!!
```

String center() Method

The method center() returns centered in a string of length width. Padding is done using the specified fillchar. Default filler is a space.

Syntax

```
str.center(width[, fillchar])
```

Parameters

- width - This is the total width of the string.
- fillchar - This is the filler character.

Return Value

This method returns a string that is at least width characters wide, created by padding the string with the character fillchar (default is a space).

Example

The following example shows the usage of the center() method.

```
#!/usr/bin/python3
str = "this is string example....wow!!!"
```

```
print ("str.center(40, 'a') : ", str.center(40, 'a'))
```

Result

```
str.center(40, 'a') :  aaaathis is string example....wow!!!aaaa
```

String count() Method

Description

The **count()** method returns the number of occurrences of substring sub in the range [start, end]. Optional arguments start and end are interpreted as in slice notation.

Syntax

```
str.count(sub, start= 0,end=len(string))
```

Parameters

- **sub -** This is the substring to be searched.
- **start** - Search starts from this index. First character starts from 0 index. By default search starts from 0 index.
- **end** - Search ends from this index. First character starts from 0 index. By default search ends at the last index.

Return Value

Centered in a string of length width.

Example

```
#!/usr/bin/python3
str="this is string example....wow!!!"
sub='i'
print ("str.count('i') : ", str.count(sub))
sub='exam'
print ("str.count('exam', 10, 40) : ", str.count(sub,10,40))
```

Result

```
str.count('i') :  3
str.count('exam', 4, 40) :
```

String decode() Method

Description

The **decode()** method decodes the string using the codec registered for encoding. It defaults to the default string encoding.

Syntax

```
Str.decode(encoding='UTF-8',errors='strict')
```

Parameters

- **encoding** - This is the encodings to be used. For a list of all encoding schemes please visit: Standard Encodings.

- **errors** - This may be given to set a different error handling scheme. The default for errors is 'strict', meaning that encoding errors raise a UnicodeError. Other possible values are 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace' and any other name registered via codecs.register_error()..

Return Value

Decoded string.

Example

```
#!/usr/bin/python3
Str = "this is string example....wow!!!";
Str = Str.encode('base64','strict');
print "Encoded String: " + Str
print "Decoded String: " + Str.decode('base64','strict')
```

Result

```
Encoded String:  b'dGhpcyBpcyBzdHJpbmcgZXhhbXBsZS4uLi53b3chISE='
Decoded String:  this is string example....wow!!!
```

String encode() Method

Description

The **encode()** method returns an encoded version of the string. Default encoding is the current default string encoding. The errors may be given to set a different error handling scheme.

Syntax

```
str.encode(encoding='UTF-8',errors='strict')
```

Parameters

- **encoding -** This is the encodings to be used. For a list of all encoding schemes please visit: Standard Encodings.

- **errors -** This may be given to set a different error handling scheme. The default for errors is 'strict', meaning that encoding errors raise a UnicodeError. Other possible values are 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace' and any other name registered via codecs.register_error().

Return Value

Decoded string.

Example

```
#!/usr/bin/python3
import base64
Str = "this is string example....wow!!!"
Str=base64.b64encode(Str.encode('utf-8',errors='strict'))
print ("Encoded String: " , Str)
```

Result

```
Encoded String:  b'dGhpcyBpcyBzdHJpbmcgZXhhbXBsZS4uLi53b3chISE='
```

String endswith() Method

Description

It returns True if the string ends with the specified suffix, otherwise return False optionally restricting the matching with the given indices start and end.

Syntax

```
str.endswith(suffix[, start[, end]])
```

Parameters

- **suffix -** This could be a string or could also be a tuple of suffixes to look for.

- **start** - The slice begins from here.

- **end** - The slice ends here.

Return Value

TRUE if the string ends with the specified suffix, otherwise FALSE.

Example

```
#!/usr/bin/python3
Str='this is string example....wow!!!'
suffix='!!'
print (Str.endswith(suffix))
print (Str.endswith(suffix,20))
suffix='exam'
print (Str.endswith(suffix))
print (Str.endswith(suffix, 0, 19))
```

Result

```
True
True
False
True
```

String expandtabs() Method

Description

The **expandtabs()** method returns a copy of the string in which the tab characters ie. '\t' are expanded using spaces, optionally using the given tabsize (default 8)..

Syntax

```
str.expandtabs(tabsize=8)
```

Parameters

tabsize - This specifies the number of characters to be replaced for a tab character '\t'.

Return Value

This method returns a copy of the string in which tab characters i.e., '\t' have been expanded using spaces.

Example

```
#!/usr/bin/python3
str = "this is\tstring example....wow!!!"
print ("Original string: " + str)
print ("Defualt exapanded tab: " +  str.expandtabs())
print ("Double exapanded tab: " +  str.expandtabs(16))
```

Result

```
Original string: this is        string example....wow!!!
Defualt exapanded tab:         this is string example....wow!!!
Double exapanded tab: this is         string example....wow!!!
```

String find() Method

Description

The find() method determines if the string str occurs in string, or in a substring of string if the starting index beg and ending index end are given.

Syntax

```
str.find(str, beg=0 end=len(string))
```

Parameters

- **str** - This specifies the string to be searched.
- **beg** - This is the starting index, by default its 0.
- **end** - This is the ending index, by default its equal to the lenght of the string.

Return Value

Index if found and -1 otherwise.

Example

```
#!/usr/bin/python3
str1 = "this is string example....wow!!!"
str2 = "exam";
print (str1.find(str2))
print (str1.find(str2, 10))
print (str1.find(str2, 40))
```

Result

```
15
15
-1
```

String index() Method

Description

The index() method determines if the string str occurs in string or in a substring of string, if the starting index beg and ending index end are given. This method is same as find(), but raises an exception if sub is not found.

Syntax

```
str.index(str, beg=0 end=len(string))
```

Parameters

- **str** - This specifies the string to be searched.
- **beg** - This is the starting index, by default its 0.
- **end** - This is the ending index, by default its equal to the length of the string.

Return Value

Index if found otherwise raises an exception if str is not found.

Example

```
#!/usr/bin/python3
str1 = "this is string example....wow!!!"
str2 = "exam";
print (str1.index(str2))
print (str1.index(str2, 10))
print (str1.index(str2, 40))
```

Result

```
15
```

```
15
Traceback (most recent call last):
  File "test.py", line 7, in
    print (str1.index(str2, 40))
ValueError: substring not found
shell returned 1
```

String isalnum() Method

Description

The **isalnum()** method checks whether the string consists of alphanumeric characters.

Syntax

Following is the syntax for isalnum() method-

```
str.isa1num()
```

Parameters

NA

Return Value

This method returns true if all the characters in the string are alphanumeric and there is at least one character, false otherwise.

Example

The following example shows the usage of isalnum() method.

```
#!/usr/bin/python3
str = "this2016"  # No space in this string
print (str.isalnum())
str = "this is string example....wow!!!"
print (str.isalnum())
```

When we run the above program, it produces the following result-

```
True
False
```

String isalpha() Method

Description

The **isalpha()** method checks whether the string consists of alphabetic characters only.

Syntax

Following is the syntax for islpha() method-

```
str.isalpha()
```

Parameters

NA

Return Value

This method returns true if all the characters in the string are alphabetic and there is at least one character, false otherwise.

Example

The following example shows the usage of isalpha() method.

```
#!/usr/bin/python3
str = "this";  # No space & digit in this string
print (str.isalpha())
str = "this is string example....wow!!!"
print (str.isalpha())
```

Result

```
True
False
```

String isdigit() Method

Description

The method isdigit() checks whether the string consists of digits only.

Syntax

Following is the syntax for isdigit() method-

```
str.isdigit()
```

Parameters

NA

Return Value

This method returns true if all characters in the string are digits and there is at least one character, false otherwise.

Example

The following example shows the usage of isdigit() method.

```
#!/usr/bin/python3
str = "123456";  # Only digit in this string
print (str.isdigit())
str = "this is string example....wow!!!"
print (str.isdigit())
```

Result

```
True
False
```

String islower() Method

Description

The **islower()** method checks whether all the case-based characters (letters) of the string are lowercase.

Syntax

Following is the syntax for islower() method-

```
str.islower()
```

Parameters

NA

Return Value

This method returns true if all cased characters in the string are lowercase and there is at least one cased character, false otherwise.

Example

The following example shows the usage of islower() method.

```
#!/usr/bin/python3
str = "THIS is string example....wow!!!"
print (str.islower())
str = "this is string example....wow!!!"
print (str.islower())
```

Result

```
False
True
```

String isnumeric() Method

Description

The **isnumeric()** method checks whether the string consists of only numeric characters. This method is present only on unicode objects.

Note: Unlike Python 2, all strings are represented in Unicode in Python 3. Given below is an example illustrating it.

Syntax

Following is the syntax for isnumeric() method-

```
str.isnumeric()
```

Parameters

NA

Return Value

This method returns true if all characters in the string are numeric, false otherwise.

Example

The following example shows the usage of isnumeric() method.

```
#!/usr/bin/python3
str = "this2016"
print (str.isnumeric())
str = "23443434"
```

```
print (str.isnumeric())
```

Result

```
False
True
```

String isspace() Method

Description

The **isspace()** method checks whether the string consists of whitespace..

Syntax

Following is the syntax for isspace() method-

```
str.isspace()
```

Parameters

NA

Return Value

This method returns true if there are only whitespace characters in the string and there is at least one character, false otherwise.

Example

The following example shows the usage of isspace() method.

```
#!/usr/bin/python3
str = "       "
print (str.isspace())
str = "This is string example....wow!!!"
print (str.isspace())
```

Result

```
True
False
```

String istitle() Method

Description

The **istitle()** method checks whether all the case-based characters in the string following non-casebased letters are uppercase and all other case-based characters are lowercase.

Syntax

Following is the syntax for istitle() method-

```
str.istitle()
```

Parameters

NA

Return Value

This method returns true if the string is a titlecased string and there is at least one character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones. It returns false otherwise.

Example

The following example shows the usage of istitle() method.

```
#!/usr/bin/python3
str = "This Is String Example...Wow!!!"
print (str.istitle())
str = "This is string example....wow!!!"
print (str.istitle())
```

Result

```
True
False
```

String isupper() Method

Description

The **isupper()** method checks whether all the case-based characters (letters) of the string are uppercase.

Syntax

Following is the syntax for isupper() method-

```
str.isupper()
```

Parameters

NA

Return Value

This method returns true if all the cased characters in the string are uppercase and there is at least one cased character, false otherwise.

Example

The following example shows the usage of isupper() method.

```
#!/usr/bin/python3
str = "THIS IS STRING EXAMPLE....WOW!!!"
print (str.isupper())
str = "THIS is string example....wow!!!"
print (str.isupper())
```

Result

```
True
False
```

String join() Method

Description

The **join()** method returns a string in which the string elements of sequence have been joined by str separator.

Syntax

Following is the syntax for join() method-

```
str.join(sequence)
```

Parameters

sequence - This is a sequence of the elements to be joined.

Return Value

This method returns a string, which is the concatenation of the strings in the sequence **seq**. The separator between elements is the string providing this method.

Example

The following example shows the usage of join() method.

```
#!/usr/bin/python3
s = "-"
seq = ("a", "b", "c") # This is sequence of strings.
print (s.join( seq ))
```

Result

```
a-b-c
```

String len() Method

Description

The **len()** method returns the length of the string.

Syntax

Following is the syntax for len() method –

```
len( str )
```

Parameters

NA

Return Value

This method returns the length of the string.

Example

The following example shows the usage of len() method.

```
#!/usr/bin/python3
str = "this is string example....wow!!!"
print ("Length of the string: ", len(str))
```

Result

```
Length of the string:  32
```

String ljust() Method

Description

The method ljust() returns the string left justified in a string of length width. Padding is done using the specified fillchar (default is a space). The original string is returned if width is less than len(s).

Syntax

Following is the syntax for ljust() method –

```
str.ljust(width[, fillchar])
```

Parameters

- **width** - This is string length in total after padding.
- **fillchar** - This is filler character, default is a space.

Return Value

This method returns the string left justified in a string of length width. Padding is done using the specified fillchar (default is a space). The original string is returned if width is less than len(s).

Example

The following example shows the usage of ljust() method.

```
#!/usr/bin/python3
str = "this is string example....wow!!!"
print str.ljust(50, '*')
```

Result

```
this is string example....wow!!!******************
```

String lower() Method

Description

The method lower() returns a copy of the string in which all case-based characters have been lowercased.

Syntax

Following is the syntax for lower() method –

```
str.lower()
```

Parameters

NA

Return Value

This method returns a copy of the string in which all case-based characters have been lowercased.

Example

The following example shows the usage of lower() method.

```
#!/usr/bin/python3
str = "THIS IS STRING EXAMPLE....WOW!!!"
print (str.lower())
```

Result

```
this is string example....wow!!!
```

String lstrip() Method

Description

The **lstrip()** method returns a copy of the string in which all chars have been stripped from the beginning of the string (default whitespace characters).

Syntax

Following is the syntax for lstrip() method-

```
str.lstrip([chars])
```

Parameters

chars - You can supply what chars have to be trimmed.

Return Value

This method returns a copy of the string in which all chars have been stripped from the beginning of the string (default whitespace characters).

Example

The following example shows the usage of lstrip() method.

```
#!/usr/bin/python3
str = "     this is string example....wow!!!"
print (str.lstrip())
str = "*****this is string example....wow!!!*****"
print (str.lstrip('*'))
```

Result

```
this is string example....wow!!!
this is string example....wow!!!*****
```

String maketrans() Method

Description

The **maketrans()** method returns a translation table that maps each character in the intabstring into the character at the same position in the outtab string. Then this table is passed to the translate() function.

Note: Both intab and outtab must have the same length.

Syntax

Following is the syntax for maketrans() method-

```
str.maketrans(intab, outtab]);
```

Parameters

- **intab** - This is the string having actual characters.
- **outtab** - This is the string having corresponding mapping character.

Return Value

This method returns a translate table to be used translate() function.

Example

The following example shows the usage of maketrans() method. Under this, every vowel in a string is replaced by its vowel position –

```
#!/usr/bin/python3
intab = "aeiou"
```

```
outtab = "12345"
trantab = str.maketrans(intab, outtab)
str = "this is string example....wow!!!"
print (str.translate(trantab))
```

Result

```
th3s 3s str3ng 2x1mpl2....w4w!!!
```

String max() Method

Description

The **max()** method returns the max alphabetical character from the string str.

Syntax

Following is the syntax for max() method-

```
max(str)
```

Parameters

str - This is the string from which max alphabetical character needs to be returned.

Return Value

This method returns the max alphabetical character from the string str.

Example

The following example shows the usage of max() method.

```
#!/usr/bin/python3
str = "this is a string example....really!!!"
print ("Max character: " + max(str))
str = "this is a string example....wow!!!"
print ("Max character: " + max(str))
```

Result

```
Max character: y
Max character: x
```

String min() Method

Description

The **min()** method returns the min alphabetical character from the string str.

Syntax

Following is the syntax for min() method-

```
min(str)
```

Parameters

str - This is the string from which min alphabetical character needs to be returned.

Return Value

This method returns the max alphabetical character from the string str.

Example

The following example shows the usage of min() method.

```
#!/usr/bin/python3
str = "www.tutorialspoint.com"
print ("Min character: " + min(str))
str = "TUTORIALSPOINT"
print ("Min character: " + min(str))
```

Result

```
Min character: .
Min character: A
```

String replace() Method

Description

The **replace()** method returns a copy of the string in which the occurrences of old have been replaced with new, optionally restricting the number of replacements to max.

Syntax

Following is the syntax for replace() method-

```
str.replace(old, new[, max])
```

Parameters

- **old** - This is old substring to be replaced.
- **new** - This is new substring, which would replace old substring.
- **max** - If this optional argument max is given, only the first count occurrences are replaced.

Return Value

This method returns a copy of the string with all occurrences of substring old replaced by new. If the optional argument max is given, only the first count occurrences are replaced.

Example

The following example shows the usage of replace() method.

```
#!/usr/bin/python3
str = "this is string example....wow!!! this is really string"
print (str.replace("is", "was"))
print (str.replace("is", "was", 3))
```

Result

```
thwas was string example....wow!!! thwas was really string
thwas was string example....wow!!! thwas is really string
```

String rfind() Method

Description

The **rfind()** method returns the last index where the substring str is found, or -1 if no such index exists, optionally restricting the search to string[beg:end].

Syntax

Following is the syntax for rfind() method-

```
str.rfind(str, beg=0 end=len(string))
```

Parameters

- **str** - This specifies the string to be searched.
- **beg** - This is the starting index, by default its 0.
- **end** - This is the ending index, by default its equal to the length of the string.

Return Value

This method returns last index if found and -1 otherwise.

Example

The following example shows the usage of rfind() method.

```
#!/usr/bin/python3
str1 = "this is really a string example....wow!!!"
str2 = "is"
print (str1.rfind(str2))
print (str1.rfind(str2, 0, 10))
print (str1.rfind(str2, 10, 0))
print (str1.find(str2))
print (str1.find(str2, 0, 10))
print (str1.find(str2, 10, 0))
```

Result

```
5
5
-1
2
2
-1
```

String rindex() Method

Description

The **rindex()** method returns the last index where the substring str is found, or raises an exception if no such index exists, optionally restricting the search to string[beg:end].

Syntax

Following is the syntax for rindex() method-

```
str.rindex(str, beg=0 end=len(string))
```

Parameters

- **str** - This specifies the string to be searched.

- **beg** - This is the starting index, by default its 0.
- **len** - This is ending index, by default its equal to the length of the string.

Return Value

This method returns last index if found otherwise raises an exception if str is not found.

Example

The following example shows the usage of rindex() method.

```
#!/usr/bin/python3
str1 = "this is really a string example....wow!!!"
str2 = "is"
print (str1.rindex(str2))
print (str1.rindex(str2,10))
```

Result

```
5
Traceback (most recent call last):
  File "test.py", line 5, in
    print (str1.rindex(str2,10))
ValueError: substring not found
```

String rjust() Method

Description

The **rjust()** method returns the string right justified in a string of length width. Padding is done using the specified fillchar (default is a space). The original string is returned if width is less than len(s).

Syntax

Following is the syntax for rjust() method-

```
str.rjust(width[, fillchar])
```

Parameters

- **width** - This is the string length in total after padding.
- **fillchar** - This is the filler character, default is a space.

Return Value

This method returns the string right justified in a string of length width. Padding is done using the specified fillchar (default is a space). The original string is returned if the width is less than len(s).

Example

The following example shows the usage of rjust() method.

```
#!/usr/bin/python3
str = "this is string example....wow!!!"
print (str.rjust(50, '*'))
```

Result

```
******************this is string example....wow!!!
```

String rstrip() Method

Description

The **rstrip()** method returns a copy of the string in which all chars have been stripped from the end of the string (default whitespace characters).

Syntax

Following is the syntax for rstrip() method-

```
str.rstrip([chars])
```

Parameters

chars - You can supply what chars have to be trimmed.

Return Value

This method returns a copy of the string in which all chars have been stripped from the end of the string (default whitespace characters).

Example

The following example shows the usage of rstrip() method.

```
#!/usr/bin/python3
str = "     this is string example....wow!!!     "
print (str.rstrip())
str = "*****this is string example....wow!!!*****"
```

```
print (str.rstrip('*'))
```

Result

```
     this is string example....wow!!!
*****this is string example....wow!!!
```

String split() Method

Description

The **split()** method returns a list of all the words in the string, using str as the separator (splits on all whitespace if left unspecified), optionally limiting the number of splits to num.

Syntax

Following is the syntax for split() method-

```
str.split(str="", num=string.count(str)).
```

Parameters

- **str** - This is any delimeter, by default it is space.
- **num** - this is number of lines to be made

Return Value

This method returns a list of lines.

Example

The following example shows the usage of split() method.

```
#!/usr/bin/python3
str = "this is string example....wow!!!"
print (str.split( ))
print (str.split('i',1))
print (str.split('w'))
```

Result

```
['this', 'is', 'string', 'example....wow!!!']
['th', 's is string example....wow!!!']
['this is string example....', 'o', '!!!']
```

String splitlines() Method

Description

The **splitlines()** method returns a list with all the lines in string, optionally including the line breaks (if num is supplied and is true).

Syntax

Following is the syntax for splitlines() method-

```
str.splitlines( num=string.count('\n'))
```

Parameters

num - This is any number, if present then it would be assumed that the line breaks need to be included in the lines.

Return Value

This method returns true if found matching with the string otherwise false.

Example

The following example shows the usage of splitlines() method.

```
#!/usr/bin/python3
str = "this is \nstring example....\nwow!!!"
print (str.splitlines( ))
```

Result

```
['this is ', 'string example....', 'wow!!!']
```

String startswith() Method

Description

The **startswith()** method checks whether the string starts with str, optionally restricting the matching with the given indices start and end.

Syntax

Following is the syntax for startswith() method-

```
str.startswith(str, beg=0,end=len(string));
```

Parameters

- **str** - This is the string to be checked.
- **beg** - This is the optional parameter to set start index of the matching boundary.
- **end** - This is the optional parameter to set start index of the matching boundary.

Return Value

This method returns true if found matching with the string otherwise false.

Example

The following example shows the usage of startswith() method.

```
#!/usr/bin/python3
str = "this is string example....wow!!!"
print (str.startswith( 'this' ))
print (str.startswith( 'string', 8 ))
print (str.startswith( 'this', 2, 4 ))
```

Result

```
True
True
False
```

String strip() Method

Description

The **strip()** method returns a copy of the string in which all chars have been stripped from the beginning and the end of the string (default whitespace characters).

Syntax

Following is the syntax for strip() method –

```
str.strip([chars]);
```

Parameters

chars - The characters to be removed from beginning or end of the string.

Return Value

This method returns a copy of the string in which all the chars have been stripped from the beginning and the end of the string.

Example

The following example shows the usage of strip() method.

```
#!/usr/bin/python3
str = "*****this is string example....wow!!!*****"
print (str.strip( '*' ))
```

Result

```
this is string example....wow!!!
```

String swapcase() Method

Description

The **swapcase()** method returns a copy of the string in which all the case-based characters have had their case swapped.

Syntax

Following is the syntax for swapcase() method-

```
str.swapcase();
```

Parameters

NA

Return Value

This method returns a copy of the string in which all the case-based characters have had their case swapped.

Example

The following example shows the usage of swapcase() method.

```
#!/usr/bin/python3
str = "this is string example....wow!!!"
print (str.swapcase())
str = "This Is String Example....WOW!!!"
print (str.swapcase())
```

Result

```
THIS IS STRING EXAMPLE....WOW!!!
tHIS iS sTRING eXAMPLE....wow!!!
```

String title() Method

Description

The **title()** method returns a copy of the string in which first characters of all the words are capitalized.

Syntax

Following is the syntax for title() method-

```
str.title();
```

Parameters

NA

Return Value

This method returns a copy of the string in which first characters of all the words are capitalized.

Example

The following example shows the usage of title() method.

```
#!/usr/bin/python3
str = "this is string example....wow!!!"
print (str.title())
```

Result

```
This Is String Example....Wow!!!
```

String translate() Method

Description

The method translate() returns a copy of the string in which all the characters have been translated using table (constructed with the maketrans() function in the string module), optionally deleting all characters found in the string deletechars.

Syntax

Following is the syntax for translate() method-

```
str.translate(table[, deletechars]);
```

Parameters

- **table** - You can use the maketrans() helper function in the string module to create a translation table.
- **deletechars** - The list of characters to be removed from the source string.

Return Value

This method returns a translated copy of the string.

Example

The following example shows the usage of translate() method. Under this, every vowel in a string is replaced by its vowel position.

```
#!/usr/bin/python3
from string import maketrans   # Required to call maketrans function.
intab = "aeiou"
outtab = "12345"
trantab = maketrans(intab, outtab)
str = "this is string example....wow!!!";
print (str.translate(trantab))
```

Result

```
th3s 3s str3ng 2x1mpl2....w4w!!!
```

Following is the example to delete 'x' and 'm' characters from the string-

```
#!/usr/bin/python3
from string import maketrans   # Required to call maketrans function.
intab = "aeiouxm"
outtab = "1234512"
trantab = maketrans(intab, outtab)
str = "this is string example....wow!!!";
print (str.translate(trantab))
```

Result

```
th3s 3s str3ng 21pl2....w4w!!!
```

String upper() Method

Description

The **upper()** method returns a copy of the string in which all case-based characters have been uppercased.

Syntax

Following is the syntax for upper() method –

```
str.upper()
```

Parameters

NA

Return Value

This method returns a copy of the string in which all case-based characters have been uppercased.

Example

The following example shows the usage of upper() method.

```
#!/usr/bin/python3
str = "this is string example....wow!!!"
print ("str.upper : ",str.upper())
```

Result

```
str.upper :  THIS IS STRING EXAMPLE....WOW!!!
```

String zfill() Method

Description

The **zfill()** method pads string on the left with zeros to fill width.

Syntax

Following is the syntax for zfill() method-

```
str.zfill(width)
```

Parameters

width - This is final width of the string. This is the width which we would get after filling zeros.

Return Value

This method returns padded string.

Example

The following example shows the usage of zfill() method.

```
#!/usr/bin/python3
str = "this is string example....wow!!!"
print ("str.zfill : ",str.zfill(40))
print ("str.zfill : ",str.zfill(50))
```

Result

```
str.zfill :  00000000this is string example....wow!!!
str.zfill :  000000000000000000this is string example....wow!!!
```

String isdecimal() Method

Description

The **isdecimal()** method checks whether the string consists of only decimal characters. This method are present only on unicode objects.

Note: Unlike in Python 2, all strings are represented as Unicode in Python 3. Given Below is an example illustrating it.

Syntax

Following is the syntax for isdecimal() method-

```
str.isdecimal()
```

Parameters

NA

Return Value

This method returns true if all the characters in the string are decimal, false otherwise.

Example

The following example shows the usage of isdecimal() method.

```
#!/usr/bin/python3
str = "this2016"
print (str.isdecimal())
str = "23443434"
print (str.isdecimal())
```

Result

```
False
True
```

11. Python 3 – 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 the 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 the 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 the 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/python3
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/python3
list = ['physics', 'chemistry', 1997, 2000]
print ("Value available at index 2 : ", list[2])
list[2] = 2001
print ("New value available at index 2 : ", list[2])
```

Note: The append() method is discussed in the 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. You can use the remove() method if you do not know exactly which items to delete. For example-

```
#!/usr/bin/python3
list = ['physics', 'chemistry', 1997, 2000]
print (list)
del list[2]
print ("After deleting value at index 2 : ", list)
```

When the above code is executed, it produces the 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,end=' ')	1 2 3	Iteration

Indexing, Slicing and Matrixes

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

Assuming the following input-

```
L=['C++'', 'Java', 'Python']
```

Python Expression	Results	Description
L[2]	'Python'	Offsets start at zero
L[-2]	'Java'	Negative: count from the right
L[1:]	['Java', 'Python']	Slicing fetches sections

Built-in List Functions & Methods

Python includes the following list functions-

SN	Function with Description
1	**cmp(list1, list2)** No longer available in Python 3.

2	**len(list)** Gives the total length of the list.
3	**max(list)** Returns item from the list with max value.
4	**min(list)** Returns item from the list with min value.
5	**list(seq)** Converts a tuple into list.

Let us understand the use of these functions.

List len() Method

Description

The **len()** method returns the number of elements in the list.

Syntax

Following is the syntax for len() method-

```
len(list)
```

Parameters

list - This is a list for which, number of elements are to be counted.

Return Value

This method returns the number of elements in the list.

Example

The following example shows the usage of len() method.

```
#!/usr/bin/python3
list1 = ['physics', 'chemistry', 'maths']
print (len(list1))
list2=list(range(5)) #creates list of numbers between 0-4
print (len(list2))
```

When we run above program, it produces following result-

```
3
5
```

List max() Method

Description

The **max()** method returns the elements from the list with maximum value.

Syntax

Following is the syntax for max() method-

```
max(list)
```

Parameters

list - This is a list from which max valued element are to be returned.

Return Value

This method returns the elements from the list with maximum value.

Example

The following example shows the usage of max() method.

```
#!/usr/bin/python3
list1, list2 = ['C++','Java', 'Python'], [456, 700, 200]
print ("Max value element : ", max(list1))
print ("Max value element : ", max(list2))
```

When we run above program, it produces following result-

```
Max value element :  Python
Max value element :  700
```

List min() Method

Description

The method min() returns the elements from the list with minimum value.

Syntax

Following is the syntax for min() method-

```
min(list)
```

Parameters

list - This is a list from which min valued element is to be returned.

Return Value

This method returns the elements from the list with minimum value.

Example

```
The following example shows the usage of min() method.
#!/usr/bin/python3
list1, list2 = ['C++','Java', 'Python'], [456, 700, 200]
print ("min value element : ", min(list1))
print ("min value element : ", min(list2))
```

When we run above program, it produces following result-

```
min value element :  C++
min value element :  200
```

List list() Method

Description

The **list()** method takes sequence types and converts them to lists. This is used to convert a given tuple into list.

Note: Tuple are very similar to lists with only difference that element values of a tuple can not be changed and tuple elements are put between parentheses instead of square bracket. This function also converts characters in a string into a list.

Syntax

Following is the syntax for list() method-

```
list( seq )
```

Parameters

seq - This is a tuple or string to be converted into list.

Return Value

This method returns the list.

Example

The following example shows the usage of list() method.

```
#!/usr/bin/python3
aTuple = (123, 'C++', 'Java', 'Python')
list1 = list(aTuple)
print ("List elements : ", list1)
str="Hello World"
list2=list(str)
print ("List elements : ", list2)
```

When we run above program, it produces following result-

```
List elements :  [123, 'C++', 'Java', 'Python']
List elements :  ['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']
```

Python includes the following list methods-

SN	Methods with Description
1	**list.append(obj)** Appends object obj to list
2	**list.count(obj)** Returns count of how many times obj occurs in list
3	**list.extend(seq)** Appends the contents of seq to list
4	**list.index(obj)** Returns the lowest index in list that obj appears
5	**list.insert(index, obj)** Inserts object obj into list at offset index

6	**list.pop(obj=list[-1])** Removes and returns last object or obj from list
7	**list.remove(obj)** Removes object obj from list
8	**list.reverse()** Reverses objects of list in place
9	**list.sort([func])** Sorts objects of list, use compare func if given

List append() Method

Description

The **append()** method appends a passed obj into the existing list.

Syntax

Following is the syntax for append() method-

```
list.append(obj)
```

Parameters

obj - This is the object to be appended in the list.

Return Value

This method does not return any value but updates existing list.

Example

The following example shows the usage of append() method.

```
#!/usr/bin/python3
list1 = ['C++', 'Java', 'Python']
list1.append('C#')
print ("updated list : ", list1)
```

When we run the above program, it produces the following result-

```
updated list :  ['C++', 'Java', 'Python', 'C#']
```

List count() Method

Description

The **count()** method returns count of how many times obj occurs in list.

Syntax

Following is the syntax for count() method-

```
list.count(obj)
```

Parameters

obj - This is the object to be counted in the list.

Return Value

This method returns count of how many times obj occurs in list.

Example

The following example shows the usage of count() method.

```
#!/usr/bin/python3
aList = [123, 'xyz', 'zara', 'abc', 123];
print ("Count for 123 : ", aList.count(123))
print ("Count for zara : ", aList.count('zara'))
```

When we run the above program, it produces the following result-

```
Count for 123 :  2
Count for zara :  1
```

List extend() Method

Description

The **extend()** method appends the contents of seq to list.

Syntax

Following is the syntax for extend() method-

```
list.extend(seq)
```

Parameters

seq - This is the list of elements

Return Value

This method does not return any value but adds the content to an existing list.

Example

The following example shows the usage of extend() method.

```
#!/usr/bin/python3
list1 = ['physics', 'chemistry', 'maths']
list2=list(range(5)) #creates list of numbers between 0-4
list1.extend('Extended List :', list2)
print (list1)
```

When we run the above program, it produces the following result-

```
Extended List :  ['physics', 'chemistry', 'maths', 0, 1, 2, 3, 4]
```

List index() Method

Description

The **index()** method returns the lowest index in list that obj appears.

Syntax

Following is the syntax for index() method-

```
list.index(obj)
```

Parameters

obj - This is the object to be find out.

Return Value

This method returns index of the found object otherwise raises an exception indicating that the value is not found.

Example

The following example shows the usage of index() method.

```
#!/usr/bin/python3
list1 = ['physics', 'chemistry', 'maths']
print ('Index of chemistry', list1.index('chemistry'))
print ('Index of C#', list1.index('C#'))
```

When we run the above program, it produces the following result-

```
Index of chemistry 1
Traceback (most recent call last):
  File "test.py", line 3, in
    print ('Index of C#', list1.index('C#'))
ValueError: 'C#' is not in list
```

List insert() Method

Description

The **insert() method** inserts object obj into list at offset index.

Syntax

Following is the syntax for insert() method-

```
list.insert(index, obj)
```

Parameters

- **index** - This is the Index where the object obj need to be inserted.
- **obj** - This is the Object to be inserted into the given list.

Return Value

This method does not return any value but it inserts the given element at the given index.

Example

The following example shows the usage of insert() method.

```
#!/usr/bin/python3
list1 = ['physics', 'chemistry', 'maths']
list1.insert(1, 'Biology')
print ('Final list : ', list1)
```

When we run the above program, it produces the following result-

```
Final list :  ['physics', 'Biology', 'chemistry', 'maths']
```

List pop() Method

Description

The **pop()** method removes and returns last object or obj from the list.

Syntax

Following is the syntax for pop() method-

```
list.pop(obj=list[-1])
```

Parameters

obj - This is an optional parameter, index of the object to be removed from the list.

Return Value

This method returns the removed object from the list.

Example

The following example shows the usage of pop() method.

```
#!/usr/bin/python3
list1 = ['physics', 'Biology', 'chemistry', 'maths']
list1.pop()
print ("list now : ", list1)
list1.pop(1)
print ("list now : ", list1)
```

When we run the above program, it produces the following result-

```
list now :  ['physics', 'Biology', 'chemistry']
list now :  ['physics', 'chemistry']
```

List remove() Method

Parameters

obj - This is the object to be removed from the list.

Return Value

This method does not return any value but removes the given object from the list.

Example

The following example shows the usage of remove() method.

```
#!/usr/bin/python3
list1 = ['physics', 'Biology', 'chemistry', 'maths']
list1.remove('Biology')
print ("list now : ", list1)
list1.remove('maths')
print ("list now : ", list1)
```

When we run the above program, it produces the following result-

```
list now :  ['physics', 'chemistry', 'maths']
list now :  ['physics', 'chemistry']
```

List reverse() Method

Description

The **reverse()** method reverses objects of list in place.

Syntax

Following is the syntax for reverse() method-

```
list.reverse()
```

Parameters

NA

Return Value

This method does not return any value but reverse the given object from the list.

Example

The following example shows the usage of reverse() method.

```
#!/usr/bin/python3
list1 = ['physics', 'Biology', 'chemistry', 'maths']
list1.reverse()
print ("list now : ", list1)
```

When we run above program, it produces following result-

```
list now :  ['maths', 'chemistry', 'Biology', 'physics']
```

List sort() Method

Description

The **sort()** method sorts objects of list, use compare function if given.

Syntax

Following is the syntax for sort() method-

```
list.sort([func])
```

Parameters

NA

Return Value

This method does not return any value but reverses the given object from the list.

Example

The following example shows the usage of sort() method.

```
#!/usr/bin/python3
list1 = ['physics', 'Biology', 'chemistry', 'maths']
list1.sort()
print ("list now : ", list1)
```

When we run the above program, it produces the following result-

```
list now :  ['Biology', 'chemistry', 'maths', 'physics']
```

12. Python 3 – Tuples

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The main difference between the tuples and the lists is that the tuples cannot be changed unlike lists. 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 the value available at that index. For example-

```
#!/usr/bin/python3
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 the existing tuples to create new tuples as the following example demonstrates.

```
#!/usr/bin/python3

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/python3
tup = ('physics', 'chemistry', 1997, 2000);
print (tup)
del tup;
print "After deleting tup : "
print tup
```

This produces the following result.

Note: An exception is 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 previous 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, end=' ')	1 2 3	Iteration

Indexing, Slicing, and Matrixes

Since tuples are sequences, indexing and slicing work the same way for tuples as they do for strings, assuming the following input-

```
T=('C++', 'Java', 'Python')
```

Python Expression	Results	Description
T[2]	'Python'	Offsets start at zero
T[-2]	'Java'	Negative: count from the right
T[1:]	('Java', 'Python')	Slicing fetches sections

No Enclosing Delimiters

No enclosing Delimiters is 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.

Built-in Tuple Functions

Python includes the following tuple functions-

SN	Function with Description
1	cmp(tuple1, tuple2) No longer available in Python 3.
2	len(tuple) Gives the total length of the tuple.
3	max(tuple) Returns item from the tuple with max value.
4	min(tuple) Returns item from the tuple with min value.
5	tuple(seq) Converts a list into tuple.

Tuple len() Method

Description

The **len()** method returns the number of elements in the tuple.

Syntax

Following is the syntax for len() method-

```
len(tuple)
```

Parameters

tuple - This is a tuple for which number of elements to be counted.

Return Value

This method returns the number of elements in the tuple.

Example

The following example shows the usage of len() method.

```
#!/usr/bin/python3
tuple1, tuple2 = (123, 'xyz', 'zara'), (456, 'abc')
print ("First tuple length : ", len(tuple1))
print ("Second tuple length : ", len(tuple2))
```

When we run above program, it produces following result-

```
First tuple length :  3
Second tuple length :  2
```

Tuple max() Method

Description

The **max()** method returns the elements from the tuple with maximum value.

Syntax

Following is the syntax for max() method-

```
max(tuple)
```

Parameters

tuple - This is a tuple from which max valued element to be returned.

Return Value

This method returns the elements from the tuple with maximum value.

Example

The following example shows the usage of max() method.

```
#!/usr/bin/python3
tuple1, tuple2 = ('maths', 'che', 'phy', 'bio'), (456, 700, 200)
print ("Max value element : ", max(tuple1))
print ("Max value element : ", max(tuple2))
```

When we run the above program, it produces the following result-

```
Max value element :  phy
Max value element :  700
```

Tuple min() Method

Description

The **min()** method returns the elements from the tuple with minimum value.

Syntax

Following is the syntax for min() method-

```
min(tuple)
```

Parameters

tuple - This is a tuple from which min valued element is to be returned.

Return Value

This method returns the elements from the tuple with minimum value.

Example

The following example shows the usage of min() method.

```
#!/usr/bin/python3
tuple1, tuple2 = ('maths', 'che', 'phy', 'bio'), (456, 700, 200)
print ("min value element : ", min(tuple1))
print ("min value element : ", min(tuple2))
```

When we run the above program, it produces the following result-

```
min value element :  bio
min value element :  200
```

Tuple tuple() Method

Description

The **tuple()** method converts a list of items into tuples.

Syntax

Following is the syntax for tuple() method-

```
tuple( seq )
```

Parameters

seq - This is a tuple to be converted into tuple.

Return Value

This method returns the tuple.

Example

The following example shows the usage of tuple() method.

```
#!/usr/bin/python3
list1= ['maths', 'che', 'phy', 'bio']
tuple1=tuple(list1)
print ("tuple elements : ", tuple1)
```

When we run the above program, it produces the following result-

```
tuple elements :  ('maths', 'che', 'phy', 'bio')
```

13. Python 3 – 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/python3

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 a part of the dictionary, we get an error as follows-

```
#!/usr/bin/python3

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

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

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

```
dict['Zara']:
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 in a simple example given below.

```
#!/usr/bin/python3

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/python3

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: An exception is raised because after **del dict,** the dictionary does not exist anymore.

```
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: The 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 is not allowed. This means no duplicate key is allowed. When duplicate keys are encountered during assignment, the last assignment wins. For example-

```
#!/usr/bin/python3

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. This 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/python3

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: list objects are unhashable
```

Built-in Dictionary Functions & Methods

Python includes the following dictionary functions-

SN	Functions with Description
1	**cmp(dict1, dict2)** No longer available in Python 3.
2	**len(dict)** Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.
3	**str(dict)** Produces a printable string representation of a dictionary.
4	**type(variable)** Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type.

Dictionary len() Method

DescriptionThe method len() **gives the total length of the dictionary. This would be equal to the number of items in the dictionary.**

Syntax

Following is the syntax for len() method-

```
len(dict)
```

Parameters

dict - This is the dictionary, whose length needs to be calculated.

Return Value

This method returns the length.

Example

The following example shows the usage of len() method.

```
#!/usr/bin/python3
```

```
dict = {'Name': 'Manni', 'Age': 7, 'Class': 'First'}
print ("Length : %d" % len (dict))
```

When we run the above program, it produces the following result-

```
Length : 3
```

Dictionary str() Method

Description

The method **str()** produces a printable string representation of a dictionary.

Syntax

Following is the syntax for str() method –

```
str(dict)
```

Parameters

dict - This is the dictionary.

Return Value

This method returns string representation.

Example

The following example shows the usage of str() method.

```
#!/usr/bin/python3
dict = {'Name': 'Manni', 'Age': 7, 'Class': 'First'}
print ("Equivalent String : %s" % str (dict))
```

When we run the above program, it produces the following result-

```
Equivalent String : {'Name': 'Manni', 'Age': 7, 'Class': 'First'}
```

Dictionary type() Method

Description

The method **type()** returns the type of the passed variable. If passed variable is dictionary then it would return a dictionary type.

Syntax

Following is the syntax for type() method-

```
type(dict)
```

Parameters

dict - This is the dictionary.

Return Value

This method returns the type of the passed variable.

Example

The following example shows the usage of type() method.

```
#!/usr/bin/python3
dict = {'Name': 'Manni', 'Age': 7, 'Class': 'First'}
print ("Variable Type : %s" %  type (dict))
```

When we run the above program, it produces the following result-

```
Variable Type : <type 'dict'>
```

Python includes the following dictionary methods-

SN	Methods with Description
1	**dict.clear()** Removes all elements of dictionary *dict.*
2	**dict.copy()** Returns a shallow copy of dictionary *dict.*
3	**dict.fromkeys()** 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)** Removed, use the **in** operation instead.
6	**dict.items()** Returns a list of *dict*'s (key, value) tuple pairs.
7	**dict.keys()** Returns list of dictionary dict's keys.
8	**dict.setdefault(key, default=None)** Similar to get(), but will set dict[key]=default if *key* is not already in dict.
9	**dict.update(dict2)** Adds dictionary *dict2*'s key-values pairs to *dict.*
10	**dict.values()** Returns list of dictionary *dict*'s values.

Dictionary clear() Method

Description

The method **clear()** removes all items from the dictionary.

Syntax

Following is the syntax for clear() method-

```
dict.clear()
```

Parameters

NA

Return Value

This method does not return any value.

Example

The following example shows the usage of clear() method.

```
#!/usr/bin/python3
```

```
dict = {'Name': 'Zara', 'Age': 7}
print ("Start Len : %d" %  len(dict))
dict.clear()
print ("End Len : %d" %  len(dict))
```

When we run the above program, it produces the following result-

```
Start Len : 2
End Len : 0
```

Dictionary copy() Method

Description

The method **copy()** returns a shallow copy of the dictionary.

Syntax

Following is the syntax for copy() method-

```
dict.copy()
```

Parameters

NA

Return Value

This method returns a shallow copy of the dictionary.

Example

The following example shows the usage of copy() method.

```
#!/usr/bin/python3
dict1 = {'Name': 'Manni', 'Age': 7, 'Class': 'First'}
dict2 = dict1.copy()
print ("New Dictionary : ",dict2)
```

When we run the above program, it produces following result-

```
New dictionary :  {'Name': 'Manni', 'Age': 7, 'Class': 'First'}
```

Dictionary fromkeys() Method

Description

The method fromkeys() creates a new dictionary with keys from seq and values set to value.

Syntax

Following is the syntax for fromkeys() method-

```
dict.fromkeys(seq[, value]))
```

Parameters

- **seq** - This is the list of values which would be used for dictionary keys preparation.
- **value** - This is optional, if provided then value would be set to this value

Return Value

This method returns the list.

Example

The following example shows the usage of fromkeys() method.

```
#!/usr/bin/python3
seq = ('name', 'age', 'sex')
dict = dict.fromkeys(seq)
print ("New Dictionary : %s" %  str(dict))
dict = dict.fromkeys(seq, 10)
print ("New Dictionary : %s" %  str(dict))
```

When we run the above program, it produces the following result-

```
New Dictionary : {'age': None, 'name': None, 'sex': None}
New Dictionary : {'age': 10, 'name': 10, 'sex': 10}
```

Dictionary get() Method

Description

The method **get()** returns a value for the given key. If the key is not available then returns default value None.

Syntax

Following is the syntax for get() method-

```
dict.get(key, default=None)
```

Parameters

- **key** - This is the Key to be searched in the dictionary.
- **default** - This is the Value to be returned in case key does not exist.

Return Value

This method returns a value for the given key. If the key is not available, then returns default value as None.

Example

The following example shows the usage of get() method.

```
#!/usr/bin/python3
dict = {'Name': 'Zara', 'Age': 27}
print ("Value : %s" %  dict.get('Age'))
print ("Value : %s" %  dict.get('Sex', "NA"))
```

When we run the above program, it produces the following result-

```
Value : 27
Value : NA
```

Dictionary items() Method

Description

The method items() returns a list of dict's (key, value) tuple pairs.

Syntax

Following is the syntax for items() method-

```
dict.items()
```

Parameters

NA

Return Value

This method returns a list of tuple pairs.

Example

The following example shows the usage of items() method.

```
#!/usr/bin/python
dict = {'Name': 'Zara', 'Age': 7}
print ("Value : %s" %  dict.items())
```

When we run the above program, it produces the following result-

```
Value : [('Age', 7), ('Name', 'Zara')]
```

Dictionary keys() Method

Description

The method **keys()** returns a list of all the available keys in the dictionary.

Syntax

Following is the syntax for keys() method-

```
dict.keys()
```

Parameters

NA

Return Value

This method returns a list of all the available keys in the dictionary.

Example

The following example shows the usage of keys() method.

```
#!/usr/bin/python3
dict = {'Name': 'Zara', 'Age': 7}
print ("Value : %s" %  dict.keys())
```

When we run the above program, it produces the following result-

```
Value : ['Age', 'Name']
```

Dictionary setdefault() Method

Description

The method setdefault() is similar to get(), but will set dict[key]=default if the key is not already in dict.

Syntax

Following is the syntax for setdefault() method-

```
dict.setdefault(key, default=None)
```

Parameters

- **key** - This is the key to be searched.
- **default** - This is the Value to be returned in case key is not found.

Return Value

This method returns the key value available in the dictionary and if given key is not available then it will return provided default value.

Example

The following example shows the usage of setdefault() method.

```
#!/usr/bin/python3
dict = {'Name': 'Zara', 'Age': 7}
print ("Value : %s" %  dict.setdefault('Age', None))
print ("Value : %s" %  dict.setdefault('Sex', None))
print (dict)
```

When we run the above program, it produces the following result-

```
Value : 7
Value : None
{'Name': 'Zara', 'Sex': None, 'Age': 7}
```

Dictionary update() Method

Description

The method **update()** adds dictionary dict2's key-values pairs in to dict. This function does not return anything.

Syntax

Following is the syntax for update() method-

```
dict.update(dict2)
```

Parameters

dict2 - This is the dictionary to be added into dict.

Return Value

This method does not return any value.

Example

The following example shows the usage of update() method.

```
#!/usr/bin/python3
dict = {'Name': 'Zara', 'Age': 7}
dict2 = {'Sex': 'female' }
dict.update(dict2)
print ("updated dict : ", dict)
```

When we run the above program, it produces the following result-

```
updated dict :  {'Sex': 'female', 'Age': 7, 'Name': 'Zara'}
```

Dictionary values() Method

Description

The method **values()** returns a list of all the values available in a given dictionary.

Syntax

Following is the syntax for values() method-

```
dict.values()
```

Parameters

NA

Return Value

This method returns a list of all the values available in a given dictionary.

Example

The following example shows the usage of values() method.

```
#!/usr/bin/python3
dict = {'Sex': 'female', 'Age': 7, 'Name': 'Zara'}
```

```
print ("Values : ",  list(dict.values()))
```

When we run above program, it produces following result-

```
Values :  ['female', 7, 'Zara']
```