Python - 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).

# Audience

Software programmers who need to learn Python programming language from scratch.

# Prerequisites

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

# Python Overview

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

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

## **History of Python**

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

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

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

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

## **Python Features**

Python's features include:

* **Easy-to-learn:** Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
* **Easy-to-read:** Python code is more clearly defined and visible to the eyes.
* **Easy-to-maintain:** Python's source code is fairly easy-to-maintain.
* **A broad standard library:** Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
* **Interactive Mode:**Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
* **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
* **Extendable:** You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
* **Databases:** Python provides interfaces to all major commercial databases.
* **GUI Programming:** Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
* **Scalable:** Python provides a better structure and support for large programs than shell scripting.

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

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

# Python - Environment Setup

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

## **Local Environment Setup**

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

* Unix (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX, etc.)
* Win 9x/NT/2000
* Macintosh (Intel, PPC, 68K)
* OS/2
* DOS (multiple versions)
* PalmOS
* Nokia mobile phones
* Windows CE
* Acorn/RISC OS
* BeOS
* Amiga
* VMS/OpenVMS
* QNX
* VxWorks
* Psion
* Python has also been ported to the Java and .NET virtual machines

## **Getting Python**

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

You can download Python documentation from <https://www.python.org/doc/> .

The documentation is available in HTML, PDF, and PostScript formats.

## **Installing Python**

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

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

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

### **Unix and Linux Installation**

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

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

This installs Python at standard location */usr/local/bin* and its libraries at */usr/local/lib/pythonXX* where XX is the version of Python.

### **Windows Installation**

Here are the steps to install Python on Windows machine.

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

### **Macintosh Installation**

Recent Macs come with Python installed, but it may be several years out of date. See [http://www.python.org/download/mac/](https://www.python.org/download/mac/) for instructions on getting the current version along with extra tools to support development on the Mac. For older Mac OS's before Mac OS X 10.3 (released in 2003), MacPython is available.

Jack Jansen maintains it and you can have full access to the entire documentation at his website − <http://www.cwi.nl/~jack/macpython.html>. You can find complete installation details for Mac OS installation.

## **Setting up PATH**

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

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

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

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

## **Setting path at Unix/Linux**

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

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

## **Setting path at Windows**

To add Python directory to the path for a particular session in Windows −

**At the command prompt** − type path **echo %path%** and press Enter.

Microsoft Windows [Version 10.0.15063]

(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\haryadav>cd ..

C:\Users>cd..

C:\>chdir C:\Program Files\Python36

C:\Program Files\Python36>python

Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32

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

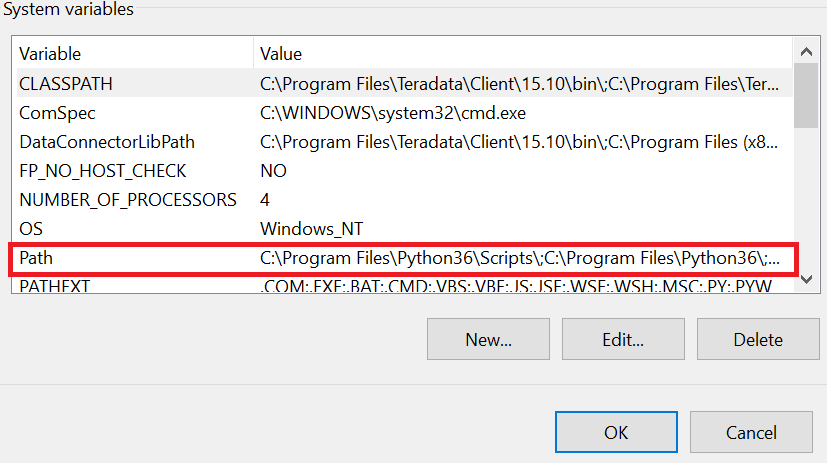
Welcome to Python!

Hari Yadav

>>>

Or

Simply, In my case environment variable is already set to



Microsoft Windows [Version 10.0.10586]

(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\haryadav>python --version

**Python 3.6.2**

C:\Users\haryadav>python

Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32

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

>>>import sys

>>>print(sys.version);

3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)]

>>>import platform

>>>platform.python\_version();

'3.6.2'

>>> **help()**

Welcome to Python 3.6's help utility!

If this is your first time using Python, you should definitely check out the tutorial on the Internet at http://docs.python.org/3.6/tutorial/.

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

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

help>quit or simply type enter

## **Python Environment Variables**

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

|  |  |
| --- | --- |
| **S.No.** | **Variable & Description** |
| 1 | **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. |
| 2 | **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. |
| 3 | **PYTHONCASEOK**  It is used in Windows to instruct Python to find the first case-insensitive match in an import statement. Set this variable to any value to activate it. |
| 4 | **PYTHONHOME**  It is an alternative module search path. It is usually embedded in the PYTHONSTARTUP or PYTHONPATH directories to make switching module libraries easy. |

**How to set the environment variable PYTHONSTARTUP**

Create a file named **startupImports.py** and write the below code in it.

import os

cls = lambda: os.system('cls')

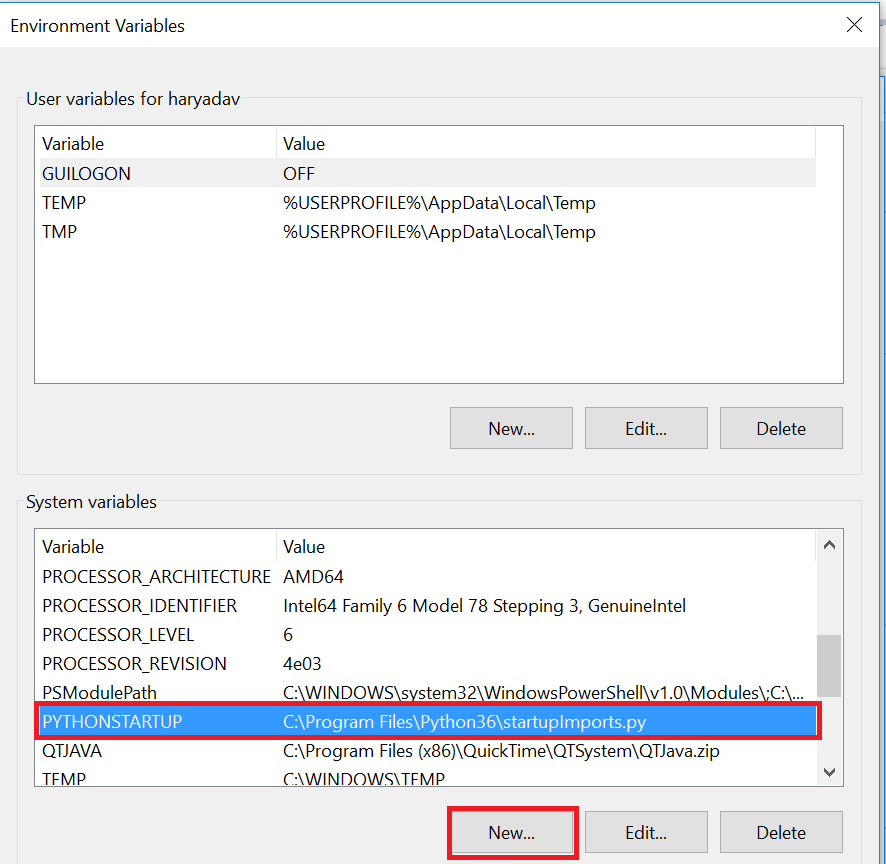
print('Welcome to Python')

print('Hari Yadav')

Copy this file in the directory where python binaries are installed. In my case it is:

C:\Program Files\Python36

Insert Environment Variables **PYTHONSTARTUP**

****

Open command prompt and login to python.

C:\Users\haryadav>python

Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32

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

Welcome to Python

Hari Yadav

>>>

**Or if file present at different location.**

C:\Users\haryadav> python -i D:\Users\haryadav\Desktop\Python\startupImports.py

Welcome to Python

Hari Yadav

>>>

## **Running Python**

There are three different ways to start Python −

### **Interactive Interpreter**

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

Enter **python** the command line.

Start coding right away in the interactive interpreter.

$python # Unix/Linux

or

python% # Unix/Linux

or

C:> python # Windows/DOS

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

|  |  |
| --- | --- |
| **S.No.** | **Option & Description** |
| 1 | **-d**  It provides debug output. |
| 2 | **-O**  It generates optimized bytecode (resulting in .pyo files). |
| 3 | **-S**  Do not run import site to look for Python paths on startup. |
| 4 | **-v**  verbose output (detailed trace on import statements). |
| 5 | **-X**  disable class-based built-in exceptions (just use strings); obsolete starting with version 1.6. |
| 6 | **-c cmd**  run Python script sent in as cmd string |
| 7 | **file**  run Python script from given file |

### **Script from the Command-line**

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

$python script.py # Unix/Linux

or

python% script.py # Unix/Linux

or

C: >python script.py # Windows/DOS

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

### **Integrated Development Environment**

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

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

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

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

# Python Basic Syntax

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

## **First Python Program**

Let us execute programs in different modes of programming.

### **Interactive Mode Programming**

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

$ python

Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32

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

Welcome to Python

Hari Yadav

>>>

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

>>> print "Hello, Python!"

If you are running new version of Python, then you would need to use print statement with parenthesis as in **print ("Hello, Python!");**.

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

Hello, Python!

### **Script Mode Programming**

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

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

print ("Hello, Python!")

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

$ python test.py

This produces the following result:

Hello, Python!

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

#!/usr/bin/python

print ("Hello, Python!")

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

$ chmod +x test.py # This is to make file executable

$./test.py

This produces the following result −

Hello, Python!

## **Python Identifiers**

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

Python does not allow punctuation characters such as @, $, and % within identifiers.

Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Here are naming conventions for Python identifiers −

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

## **Reserved Words**

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

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

## **Lines and Indentation**

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

The number of spaces in the indentation is variable, but all statements within the block must be indented with the same amount space. 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")

C:\>python -i D:\Users\haryadav\Desktop\Python\test.txt

File "D:\Users\haryadav\Desktop\Python\test.txt", line 6

print("False")

^

IndentationError: unindent does not match any outer indentation level

>>>exit()

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

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

#!/usr/bin/python

import sys

try:

# open file stream

file = open(file\_name, "w")

except IOError:

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

sys.exit()

print "Enter '", file\_finish,

print "' When finished"

while file\_text != file\_finish:

file\_text = raw\_input("Enter text: ")

if file\_text == file\_finish:

# close the file

file.close

break

file.write(file\_text)

file.write("\n")

file.close()

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

if len(file\_name) == 0:

print "Next time please enter something"

sys.exit()

try:

file = open(file\_name, "r")

except IOError:

print "There was an error reading file"

sys.exit()

file\_text = file.read()

file.close()

print file\_text

## **Multi-Line Statements**

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

total = 100 + \

200 + \

300 + \

400

print(total)

1000

name = 'hari','ram','dev', \

'mark', \

'shyam', \

'irshad'

print(name)

('hari', 'ram', 'dev', 'mark', 'shyam', 'irshad')

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

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

'Thursday', 'Friday']

print(days)

months ={'January', 'February', 'March', 'April', 'May', 'June', 'July', 'August',

'September', 'October', 'November', 'December'}

print(months)

print(sorted(months))

## **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 = 'India'

sentence = "India is our country. It’s capital is New Delhi"

paragraph = """India is our country. It has

29 states and seven union territories."""

print(word)

print(sentence)

print(paragraph)

## **Comments in Python**

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

#!/usr/bin/python

# First comment

Print("Hello, Python!") # second comment

This produces the following result −

Hello, Python!

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

name = "Capgemini" # This is again comment

You can comment multiple lines as follows −

# This is a comment.

# This is a comment, too.

# This is a comment, too.

# I said that already.

## **Using Blank Lines**

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

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

## **Waiting for the User Input**

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

#!/usr/bin/python

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

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

**NOTE:**

In Python 3.x, input() replaces raw\_input(), for input from the console. It returns the user's response as string, so when an int or a float is needed, it is necessary to convert the returned value from the str type using int() or float().

Python provides the function **input()**. input has an optional parameter, which is the prompt string.

If the input function is called, the program flow will be stopped until the user has given an input and has ended the input with the return key. The text of the optional parameter, i.e. the prompt, will be printed on the screen.

The input of the user will be interpreted. If the user e.g. puts in an integer value, the input function returns this integer value. If the user on the other hand inputs a list, the function will return a list.

>>> name = input('Enter Your Name: ')

Enter Your Name: Hari Yadav

>>> print('Hello ', name)

Hello Hari Yadav

Put these code in a script file input\_test.txt

fname = input('\n\nEnter Your First Name: ')

mname = input('\n\nEnter Your Middle Name: ')

lname = input('\n\nEnter Your Last Name: ')

print('\n\nYour name is :', fname, mname, lname)

C:\Program Files\Python36>python D:\Users\haryadav\Desktop\Python\input\_test.txt

Enter Your First Name: Hari

Enter Your Middle Name: Shankar

Enter Your Last Name: Yadav

Your name is : Hari Shankar Yadav

Put these code in a script file input\_test.txt

name = input("What's your name? ")

age = input("Your age? ")

print(name, type(name))

print(age, type(age))

colours = input("Your favourite colours? ")

print(colours)

print(colours, type(colours))

C:\>python -i D:\Users\haryadav\Desktop\Python\input\_test.txt

What's your name? Hari

Your age? 35

Hari <class 'str'>

35 <class 'str'>

Your favourite colours? ["Red","Blue",'Green','Yellow',"Pink"]

["Red","Blue",'Green','Yellow',"Pink"]

["Red","Blue",'Green','Yellow',"Pink"] <class 'str'>

## **Multiple Statements on a Single Line**

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

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

## **Multiple Statement Groups as Suites**

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

A compound statement consists of one or more ‘clauses.’ **A clause consists of a header and a ‘suite.’** The clause headers of a particular compound statement are all at the **same indentation level.** Each clause header begins with a uniquely identifying keyword and ends with a colon. A suite is a group of statements controlled by a clause. A suite can be one or more semicolon-separated simple statements on the same line as the header, following the header’s colon, or it can be one or more indented statements on subsequent lines. Only the latter form of a suite can contain nested compound statements; the following is illegal, mostly because it wouldn’t be clear to which if clause a following else clause would belong:

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 : # This is header line

suite # This is suite line and it should be indented.

elif expression :

suite

else :

suite

**NOTE: if , elif, else keyword should be lined up and suite line should be indented.**

v\_marks = int(input("Enter your score :"))

if (v\_marks<50) :

print("He is fail.");

print("He should work hard.");

elif (v\_marks>=50 and v\_marks<60) :

print("He stood 2nd."); print("He need improvments.");

elif (v\_marks>=60 and v\_marks<80) :

print("He stood 1st.");

else :

print("Outstanding.");

print("Well done.");

# Output

C:\>python -i D:\Users\haryadav\Desktop\Python\input\_test.txt

He is fail.

He should work hard.

>>> exit()

C:\>python -i D:\Users\haryadav\Desktop\Python\input\_test.txt

He stood 2nd.

He need improvments.

>>> exit()

C:\>python -i D:\Users\haryadav\Desktop\Python\input\_test.txt

He stood 1st.

>>> exit()

C:\>python -i D:\Users\haryadav\Desktop\Python\input\_test.txt

Outstanding.

Well done.

>>> exit()

x = 10 ; y = 20 ; z = 30

if x < y < z: print(x); print(y); print(z);

...

10

20

30

if z < y < x: print(x); print(y); print(z);

...

# No output because condition is false.

The following is illegal, mostly because it wouldn’t be clear to which if clause a following else clause would belong:

**NOTE: if , elif, else keyword should be lined up and suite line should be indented.**

if z < y < x: print(x); print(y); print(z); else: print("else execution");

File "<stdin>", line 1

if z < y < x: print(x); print(y); print(z); **else:** print("else execution");

**^**

SyntaxError: invalid syntax

if z < y < x:

... print(x);

... print(y);

... print(z);

... else:

... print("else execution");

File "<stdin>", line 6

print("else execution");

^

IndentationError: expected an indented block

if z < y < x:

... print(x);

... print(y);

... print(z);

... else:

... print("else execution");

...

else execution

## **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.

# Python Variable Types

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

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

## **Assigning Values to Variables**

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

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

v\_counter = 100 # An integer assignment

v\_miles = 1000.550 # A floating point

v\_name = "Dev" # A string

print (v\_counter, "\t", type(v\_counter))

print (v\_miles ,"\t", type(v\_miles))

print (v\_name ,"\t", type(v\_name))

Here, 100, 1000.55 and "Dev" are the values assigned to v\_*counter*, v\_*miles*, and v\_*name* variables, respectively. This produces the following result −

100 <class 'int'>

1000.55 <class 'float'>

Dev <class 'str'>

**# Print word separated by tab.**

>>> print ("Hello World\t","hi");

Hello World hi

>>> print ("Hello World","hi");

Hello World hi

## **Multiple Assignment**

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

a = b = c = 1

print(a, b, c)

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

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

print(a, b, c)

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

## **Standard Data Types**

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

Python has many native datatypes. Here are the important ones:

* **Booleans** are either True or False.
* **Numbers** can be integers (1 and 2), floats (1.1 and 1.2), fractions (1/2 and 2/3), or even complex numbers. Long is no longger supported in python 3.
* **Strings** are sequences of Unicode characters, e.g. an html document.
* **Bytes** and **byte** **arrays**, e.g. a jpeg image file.
* **Lists** are ordered sequences of values.
* **Tuples** are ordered, immutable sequences of values.
* **Dictionaries** are unordered bags of key-value pairs.
* **Sets** are unordered bags of values.

# [In-memory size of a Python structure](https://stackoverflow.com/questions/1331471/in-memory-size-of-a-python-structure)

import sys

import decimal

d = {

"int": 0,

"float": 0.0,

"dict": dict(),

"set": set(),

"tuple": tuple(),

"list": list(),

"str": "a",

"unicode": u"a",

"decimal": decimal.Decimal(0),

"object": object()}

for k, v in d.items():

print( k, '\t', '\t', sys.getsizeof(v))

int 24

float 24

dict 240

set 224

tuple 48

list 64

str 50

unicode 50

decimal 104

object 16

Difference between len() and sys.getsizeof() methods in python?

len() Return the length (the number of items) of an object. The argument may be a sequence (string, tuple or list) or a mapping (dictionary).

getsizeof() Return the size of an object in bytes. The object can be any type of object. Python string objects are not simple sequences of characters, 1 byte per character.

var='hari';

sys.getsizeof(var);

53

len(var)

4

var='h';

sys.getsizeof(var);

50

len(var)

1

vnum = 0

sys.getsizeof(vnum);

24

vnum=1

sys.getsizeof(vnum);

28

## **Python Numbers**

# Integer Objects

All integers are implemented as “long” integer objects of **arbitrary size**. Python has arbitrary precision integers so there is no true fixed maximum. You're only limited by available memory.

# Can Integer Operations Overflow in Python?

# Integer representations

# Integers are typically represented in memory as a base-2 bit pattern, and in python the built-in function bin can be used to inspect that:

bin(19)

'0b10011'

# If the number of bits used is fixed, the range of integers that can be represented would be fixed and can potentially overflow. That is the case for many languages such as C/C++.

# In python, integers have arbitrary precision and therefore we can represent an arbitrarily large range of integers (only limited by memory available).

# Here output is curtailed to two pages.

2 \*\* 200000

998005181847120956085934630921350542004176561034622082912657095919580749322328100903449168845945002918040561977246529938402087852178538096905146451834083282509402726791838696368444875295871898027402085079821331102776321213781388328995964738487754679597723583152631732547027718472560917480736968482313697603312282296190443169454169954802264133669700776559374383231932554850366449334600827153320453983118896114991575072864292096458416956403761646426772121248411624699569184283605656152971917522359236022075643304719888008355171721328916628686308164118068245148691429932600883813736312620045302987710970822800520962649211225839431479235046980809413304371297959042825781931136432350165028081254333893091680495211821999268242801853405742914839963932704131035543917473091149694788080143296781307733796745930664767138381337677082663308190711563085748488061849401457620560269103109796838192439478767736981477186772343800792411539379587023823986268442688667268891003377419079266485084537282267731158859020385011241419463687167902320809035024379659744292650039268759567382096655687566426016323719865038199357508708828966729391089498571708284794505518791885423774732615107304584368178319948215523282772329294197495377161379899179468084135125646358027352359058780926958354535964559036962955581016727588479652430770300047696633333706930736982543013671917977937539838236383396171006485675780030583551000355738714585645500142030899617822378548926869544888848297038974108738952452423512342944246297842428173654461372952180083810938746164721677381777825723297689380165124485961542236434714829672148922126123867148953042173646075766114697824521246503846872572821425467834473681880443890256971952836274226207704075762012912060998099519075194454271066493986571584878600703201294607436850098829204510196175216020606237158351752284514654811620300250831292505936116506745776244025821511400366952601952462282361442951572222488204579667716764568432125191958985988015882402597196676230474763020966267010308001000480651729616295828611989458526809601700416386518726151592084170709288903599472331124693278176915679093714760851284964475856470306169060854519083870942149738608622581384861863114748809629012229884078704486012888179238710978675577550888823356907735606054585798313640892583012706626076995074264341272242882908284288864997801730335816629949204315439298021629770986039943674982883383809135351899965897057639080681987992654478751047103780615686857259237855521529380447897280140979472166003572643267700973174766258931947097506227821163325594902789557283002354897977820517036309877692691301463708977972669601423493844107954142881826918454420252520185960499311428515157116865890274198596968782459266992100610888086748169138782647574241834062976103327058413948619856372700233716074439907175628752746588534161293287654419927535430390924334678326436146672330379440204787919861321870023349557927739786516697953035996981170362803063719724615886901745521847034382152200541318117286052623067624393366332447142654739433968994145112554468891618790887338289121520365885384281733278178180665579814936568058220774803393249421545519036682541715999606317392243690576416266777138352298286560797662132535312647699259456049383884231737375893970217087909919189053836151608192580748259833079392277484646636529602712121420800715902028396325562174778763298049495639098530102425075118333322153474624255630016289991606560589839321726495099719850917758818556073409821206163802474077656887091125351059299382344784307667622729165416413568082588420839119030938105526143479355127829617273549483700992848311504623313866663294681240559305656501417129557441585966445325473097062070819726083635445682191919395774415630328614764529543431454780480214840506995992886543886121161012558501741512817695727144593539459579579969178096886897475449943430469457030603834709742783120321484426279434706125287121435834606639536433904603565826910469718992394376799376088099921805744804212499395437116014811835691713855499858525957645193101946026291635475236558154312481760658258111985019271365648743038870183566314849443868007071380353414204523366178186129288403367319458063135979274333666739715253064365277448274400113427464107359939774410069480943414567741975665092933891185608367399730592478146984992115510944954370813329536236155320982075931090460931312177886773973069235215965168138543846408787010963008701632822373409794722532372318660420251806172871386377569035833636946275674368649186375802457092464036432277823984879592792662624417932094155987835148629604710238486074434797289496641963207124956668870183128890598117921683650368206753302235246923555329998906974887578790840748546245774412613005389973266796441687650786586349382364945314756111151598927314730858659506106986221234093387699520351367458956682674606107231626167137350271478504919200160341722910898201222608830990861762988720702434899396636107958113463637521810175540674254315597525506013820261569113780895100534050732742127966592345435818085582694306976848223585665141172437642624618305360201646250019504153545532184923270770571407434660684840753380309550306620927569037935606277952436028535924538140280389693256516518078344740169950157145297660005105877738618967057434788165276561254283825749039212656443222071115950713399005023635764603998707834583726926154453191323442857314504777963887202472319618735085434803351839256513623845608948166840020969077356368829281140630198639404186671163330219048797536596095894515534088611078767555349245600810456086169110099293322619046363369366742058395234384521579479032179803995159432154188938121345238837711499609563447217033933499732666408726420905632612241440229450597506843386714409464614942390425245103703967155598265069597508734988402525962257284195240338351583733903706911644107757490196386451853253237689468790314842053612751204219196471717786024654594696380335942562910150411998043512769359337062566773307772119059113978607006966678527518507957463667350659115929171731941434920511681251468668087654346043585659398692860655017251627228778274121462425076187250973800000915576676241951422819527979775476810237617481730846201321753840118310221545904754070123830817170364688854948437384997228445618983489811173222999562742727061203357657963000756248295335301690148285433584802650058167930310529529959885027857484789561354221569269207016106601971257672748168066379040056144030477206943092469230040901978747282735864532153657034177468394395842685837329460320196851731328715800018781504962392190210485687049409066558958073204876046912481084899717761379767768408693032633853683275812709471328340793417594443602696379274951607546556152260458240963513674708294085845623139832224843763156453020816357368766899902674784628335034790468941209236041779988220963875607796592643909721810261318401362587581542050693147654942663671797432401769155390296738682265177454521653328095411224480331260702455779648169967785367164378914834397483554870432846648176166397289080514744817855822854469428687681039865122793527807950496749321139900066564494584281312125346232701644337140629021710007245083678766075530173685797333883904124846254182373088265269823554273785287700481020121338506883297556684315712195412534261272031046350229000269465340168972372728438609114425949602799018486395474149238978683985039731574352656135028607485585326255539948897559893195964573142248354702505806423966948144414352577224059851808184284926256841919662984135526589278934514380319508457740248687792938982041923643771813447256335780734927588810249190858658289519538341964314296834271928990800027984366419559692002471049919751064121252226539949152070130543605435752845797872214480359419219290898200109850645410034000128428248484152279351905775789474353961835874218860542060158366327074798604318007846666958315210317525980417236946736846248047852367881681500317467197795030433428950868582962617767982438362531758944660361832924229707921862481778738597256583755547248581049385109729456453607247675587584383151483163203040752144955509577856824864957340346875703651240981589572309658622451320991198548588001975256128700771758790861998861695632554540971801382553496443738119315097203771082633701692925514498416745520418905876404667902390957748611770734490856483477914444526097751001714797945238845282134700646294703665299539141558119424203230396841497503826812358023288483734876597925720344152304620598403246404968114321143211861950895432151500150675986278770250796151565968258747098070022306529083930719527285878836474286110102858301499945264779091598042803956031619601332684104705017156987473898358203487955039088638703220032247679254070858361842197719552066565182740950026562385620115752154988916478209049750104812365443745602963476049418523806975549549521376894810285225528226295089080786783806224146806700380049208215142508260786296227927216919140729150245748398197502479116256144825256444348113145269866074128890531640518587114858708896712021742991352689822512478294587418072422287917667307046043850741273250610634050953687850554149457234445393870979043340401128511528888309519250560818500106402072313452110239467931593823583849534101449158256111391133342662770013523551465406781863107072610291317589643796258033080758120232392864763822675195570182509153962017240526799743504556151394552677060489817993402794828072269755509827904322670226723751060179869900092899017051354690930179418970729953545989954978006656690020237980257077255815232829042346502223241052962837072102965453054676279993251235617499811651071952557810585362461905795107959005778213224555645977473058203058954554295574570711769976367887404892904198238078132365087180404712243861721437460127951411187207837334483643792813505075919374738008366151939255936473526827477600519256134883337982424679986579993328692548703160930208793989613242803489250327271592207807445174588651318109626015886986572757652663029381467875066355846589928227147640677301220015390624169475646589424846110995203044029212400979266326264199650830708616935528991070908547072517476709203262614439238243384314373725853367877830663809091901565732967862145869498564072966412467667948679930716417244847177169359888953914205650712448786810941098097917812020507112854964083306148836628522241167796921685443460830326453994906841835781370424200197723581994583555726106130499810272244930784639427407017662715518701605437369710207218352044620194063752950712369272593591358628683708506736794795894477920542003783075381077494136398281913428987296779694518401116684938475413796651877927882237421793591034013400862573271598659273418719320881705549604616213038228809049117351101442647932156725087486678713198046212438627663408274123028471024202375740363363961349390237653919617503006800980079698540900190190028577555980618260157867940814652525761110096897130489920881149182566251732183127450414883431426677256371704560393009047126822615571693035875265

# Can integers overflow in python?

# Short answers:

# No

# if the operations are done in pure python, because python integers have arbitrary precision

# Yes

# if the operations are done in the pydata stack (numpy/pandas), because they use C-style fixed-precision integers.

# Arbitrary precision

# So how do python integers achieve arbitrary precision?

# In python 2, there are actually two integers types: int and long, where int is the C-style fixed-precision integer and long is the arbitrary-precision integer. Operations are automatically promoted to long if int is not sufficient, so there's no risk of overflowing. In python 3, int is the only integer type, long is obsoleted and it is arbitrary-precision.

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

var1 = 20

var2 = 40

var3 = 60

var4 = 70

var5 = 80

print(var1, '\t', var2, '\t', var3, '\t', var4, '\t', var5)

20 40 60 70 80

**Viewing all defined variables**

dir() will give you the list of in scope variables:

globals() will give you a dictionary of global variables

locals() will give you a dictionary of local variables

>>> dir()

['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_cached\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'cls', 'name', 'os', 'value', 'var1', 'var2', 'var3', 'var4', 'var5', 'vars']

>>> globals()

{'\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None, '\_\_package\_\_': None, '\_\_loader\_\_': <\_frozen\_importlib\_external.SourceFileLoader object at 0x0000023858E3F0B8>, '\_\_spec\_\_': None, '\_\_annotations\_\_': {}, '\_\_builtins\_\_': <module 'builtins' (built-in)>, '\_\_cached\_\_': None, 'os': <module 'os' from 'C:\\Program Files\\Python36\\lib\\os.py'>, 'cls': <function <lambda> at 0x0000023858DB3E18>, 'name': 'var5', 'value': 80, 'var1': 20, 'var2': 40, 'var3': 60, 'var4': 70, 'var5': 80, 'vars': 'vars'}

>>> locals()

{'\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None, '\_\_package\_\_': None, '\_\_loader\_\_': <\_frozen\_importlib\_external.SourceFileLoader object at 0x0000023858E3F0B8>, '\_\_spec\_\_': None, '\_\_annotations\_\_': {}, '\_\_builtins\_\_': <module 'builtins' (built-in)>, '\_\_cached\_\_': None, 'os': <module 'os' from 'C:\\Program Files\\Python36\\lib\\os.py'>, 'cls': <function <lambda> at 0x0000023858DB3E18>, 'name': 'var5', 'value': 80, 'var1': 20, 'var2': 40, 'var3': 60, 'var4': 70, 'var5': 80, 'vars': 'vars'}

>>> locals('%var%')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: locals() takes no arguments (1 given)

import sys, pprint

sys.displayhook = pprint.pprint

locals()

{'\_\_annotations\_\_': {},

'\_\_builtins\_\_': <module 'builtins' (built-in)>,

'\_\_cached\_\_': None,

'\_\_doc\_\_': None,

'\_\_loader\_\_': <\_frozen\_importlib\_external.SourceFileLoader object at 0x0000023858E3F0B8>,

'\_\_name\_\_': '\_\_main\_\_',

'\_\_package\_\_': None,

'\_\_spec\_\_': None,

'cls': <function <lambda> at 0x0000023858DB3E18>,

'name': 'var5',

'os': <module 'os' from 'C:\\Program Files\\Python36\\lib\\os.py'>,

'pprint': <module 'pprint' from 'C:\\Program Files\\Python36\\lib\\pprint.py'>,

'sys': <module 'sys' (built-in)>,

'value': 80,

'var1': 20,

'var2': 40,

'var3': 60,

'var4': 70,

'var5': 80,

'vars': 'vars'}

for vars in dir():

if vars.startswith("var"):

print(vars)

var1

var2

var3

var4

var5

vars

for vars in dir():

print(vars)

\_\_annotations\_\_

\_\_builtins\_\_

\_\_cached\_\_

\_\_doc\_\_

\_\_loader\_\_

\_\_name\_\_

\_\_package\_\_

\_\_spec\_\_

cls

name

os

value

var1

var2

var3

var4

var5

vars

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 var1

del var2, var5

>>> print(var1, '\t', var2, '\t', var3, '\t', var4, '\t', var5)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'var1' is not defined

>>> print(var3, '\t', var4)

60 70

Python supports four different numerical types −

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

**NOTE:** The long() function is no longer supported by Python 3. It only has one built-in integral type, named int; but it behaves mostly like the old long type. So you just need to use int() built-in function in python-3.x.

import sys

sys.maxsize

9223372036854775807

sys.maxint

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: module 'sys' has no attribute 'maxint'

vnum=9223372036854775807999999999999999999999999999999999999999999999999999999999999999999999999999;

vnum

9223372036854775807999999999999999999999999999999999999999999999999999999999999999999999999999

sys.getsizeof(sys.maxsize);

36

sys.getsizeof(vnum);

68

### **Examples**

Here are some examples of numbers −

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

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

v\_num = 99999;

print(v\_num, type(v\_num))

99999 <class 'int'>

v\_num = '99999';

print(v\_num, type(v\_num))

99999 <class 'str'>

v\_num = float(99999);

print(v\_num, type(v\_num))

99999.0 <class 'float'>

del v\_num

**v\_num = long(5.5)**

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

**NameError: name 'long' is not defined**

v\_comp = complex(2,3)

print(v\_comp, type(v\_comp))

(2+3j) <class 'complex'>

v\_comp.real

2.0

v\_comp.imag

3.0

v\_comp.conjugate() # Change the sign of imaginary part.

(2-3j)

Several built-in functions support complex numbers:

abs(3 + 4j)

5.0

pow(3 + 4j, 2)

(-7+24j)

Complex number manipulations:

x = complex(1,2)

print x

(1+2j)

y = complex(3,4)

print y

(3+4j)

z = x+y

print z

(4+6j)

z = x-y

print(z)

(-2-2j)

z = x\*y

print z

(-5+10j)

z = x/y

print z

(0.44+0.08j)

print x.conjugate()

(1-2j)

print x.real

1.0

print x.imag

2.0

print x>y

Traceback (most recent call last):

File "<pyshell#149>", line 1, in <module>

print x>y

TypeError: no ordering relation is defined for complex numbers

print x==y

False

## **Python Strings**

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

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

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

print(str[-5:-2]) # Extract from rear end

This will produce the following result −

Hello World!

H

llo

llo World!

Hello World!Hello World!

Hello World!TEST

orl

**Update and delete from string**

print(str)

Hello World!

str[5] = '-'

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'str' object does not support item assignment

del str[2:5]

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'str' object does not support item deletion

## **Python Lists**

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

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

#!/usr/bin/python

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

tinylist = [123, 'john']

print(list) # Prints complete list

print(list[0]) # Prints first element of the list

print(list[1:3]) # Prints elements starting from 2nd till 3rd

print(list[2:]) # Prints elements starting from 3rd element

print(list[-2:]) # Prints elements starting from 2nd element from rear in forward direction

print(list[-3:-1]) # Prints elements starting from 3rd element till 1st from rear

print(tinylist \* 2) # Prints list two times

print(list + tinylist) # Prints concatenated lists

list[0] = 'Hari' # Update list element

print(list[0]) # Prints updated first element of the list

print(list) # Prints complete list

This produce the following result −

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

abcd

[786, 2.23]

[2.23, 'john', 70.2]

['john', 70.2]

[2.23, 'john']

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

['abcd', 786, 2.23, 'john', 70.2, 123, 'john']

Hari

['Hari', 786, 2.23, 'john', 70.2]

**Extend the list.**

list.append(['proj01','payroll']) **# Appends object at end.**

print(list)

['Hari', 786, 2.23, 'john', 70.2, ['proj01', 'payroll']]

print(list[6])

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

IndexError: list index out of range

print(list[5])

['proj01', 'payroll']

print(list[4])

70.2

list.extend(['proj02','HR']) **# Extends list by appending elements from the iterable.**

print(list)

['Hari', 786, 2.23, 'john', 70.2, ['proj01', 'payroll'], 'proj02', 'HR']

print(list[6], list[7])

proj02 HR

**# Delete elements from the list.**

del list[7]

print(list)

['Hari', 786, 2.23, 'john', 70.2, ['proj01', 'payroll'], 'proj02']

del list[5:6]

print(list)

['Hari', 786, 2.23, 'john', 70.2, 'proj02']

del list[3:6]

print(list)

['Hari', 786, 2.23]

## **Python Tuples**

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

The main differences between lists and tuples are:

* Lists are enclosed in brackets ( [ ] ) and their elements and size can be changed.
* Tuples are enclosed in parentheses ( ( ) ) and cannot be updated. Tuples can be thought of as **read-only** lists.

#!/usr/bin/python

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

tinytuple = (123, 'john')

print(tuple) # Prints complete list

print(tuple[0]) # Prints first element of the list

print(tuple[1:3]) # Prints elements starting from 2nd till 3rd

print(tuple[2:]) # Prints elements starting from 3rd element

print(tinytuple \* 2) # Prints list two times

print(tuple + tinytuple) # Prints concatenated lists

This produce the following result −

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

abcd

(786, 2.23)

(2.23, 'john', 70.200000000000003)

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

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

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

#!/usr/bin/python

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

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

print(tuple[3:5])

['john', 70.2]

print(list[3:5])

['john', 70.2]

# Update element in tuple.

tuple[2] = 1000 **# Invalid syntax with tuple**

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'tuple' object does not support item assignment

list[2] = 1000 **# Valid syntax with list**

print(list)

['abcd', 786, 1000, 'john', 70.2]

del tuple[2]

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'tuple' object doesn't support item deletion

del list[2]

print(list)

['abcd', 786, 'john', 70.2]

## **Python Dictionary**

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

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

#!/usr/bin/python

dict = {}

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

dict[2] = "This is two"

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

print(dict['one']) # Prints value for 'one' key

print(dict[2]) # Prints value for 2 key

print(tinydict) # Prints complete dictionary

print(tinydict.keys()) # Prints all the keys

print(tinydict.values()) # Prints all the values

This produce the following result −

This is one

This is two

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

['dept', 'code', 'name']

['sales', 6734, 'john']

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

**Extend and update the dictionaries.**

**# Dictionary extended.**

dict['Three'] = "This is three"

dict[4] = "This is four"

print(dict)

{'one': 'This is one', 2: 'This is two', 'Three': 'This is three', 4: 'This is four'}

**# Dictionary updated.**

dict[4] = "Today is 4th of the month"

print(dict)

{'one': 'This is one', 2: 'This is two', 'Three': 'This is three', 4: 'Today is 4th of the month'}

**# Delete an item from a dictionary**

del dict['Three']

print(dict)

{'one': 'This is one', 2: 'This is two', 4: 'Today is 4th of the month'}

dict.pop(4)

'Today is 4th of the month'

print(dict)

{'one': 'This is one', 2: 'This is two' }

**# Replace or recreate the whole items in a dictionary**

dict = {'EmpNo':114649, 'Ename':'Hari Yadav', 'Phone':'666777', 'State':'MH', 'Pin':556622}

print(dict)

{'EmpNo': 114649, 'Ename': 'Hari Yadav', 'Phone': '666777', 'State': 'MH', 'Pin': 556622}

del dict

## **Data Type Conversion**

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

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

|  |  |
| --- | --- |
| **Function** | **Description** |
| int(x [,base]) | Converts x to an integer. base specifies the base if x is a string.  **# integer**  print("int(123) is:", int(123))  int(123) is: 123  # float  print("int(123.23) is:", int(123.23))  int(123.23) is: 123  # string  print("int('123') is:", int('123'))  int('123') is: 123  Like 123 = 1\*102 +2\*101+3\*100  123 = 100 + 20 + 3  **# binary 0b or 0B**  print("For 1010, int is:", int('1010', 2))  For 1010, int is: 10  Like 1010 = 1\*23 + 0\*22 + 1\*21 + 0\*20  1010 = 8+0+2+0 => 10  print("For 101010, int is:", int('101010', 2))  For 101010, int is: 42  print("For 0b101010, int is:", int('0b101010', 2))  For 0b101010, int is: 42  print("For 0B101010, int is:", int('0B101010', 2))  For 0B101010, int is: 42  **# octal 0o or 0O**  print("For 234, int is:", int('234', 8))  For 234, int is: 156  Like (234)8 = 2\*82 + 3\*81 + 4\*80  (234)8 = 128 + 24 + 4 => 156  print("For 0o234, int is:", int('0o234', 8))  For 0o234, int is: 156  **# hexadecimal**  print("For A, int is:", int('A', 16))  For A, int is: 10  print("For ABC, int is:", int('ABC', 16))  For ABC, int is: 2748  Like (ABC)16 = A\*162 + B\*161 + C\*160  (ABC)16 = 10\*162 + 11\*161 + 12\*160 => 2748  (ABC)16 = 10\*256 + 11\*16 + 12\*1 => 2748  (ABC)16 = 2560 + 176 + 12 => 2748  print("For 0xA, int is:", int('0xA', 16))  For 0xA, int is: 10  print("For 0xABC, int is:", int('0xABC', 16))  For 0xABC, int is: 2748 |
| long(x [,base] ) | Converts x to a long integer. base specifies the base if x is a string. As Python 3 treats all integers as long integer this long() has been removed. (Python 3's int is the same as Python 2's long). there is no distinction between long and int in python3. (long does not exist...)  long('234')  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  NameError: name 'long' is not defined  int(float('234.89'))  234 |
| float(x) | Converts x to a floating-point number.  float('12345')  12345.0  float('12345.55')  12345.55 |
| complex(real [,imag]) | Creates a complex number.  complex(3,5)  (3+5j) |
| str(x) | Converts object x to a string representation.  str(12)  '12'  str(99.99)  '99.99'  str('abc')  'abc'  user = "Dev"  lines = 50  print("Congratulations, " + user + "! You just wrote " + str(lines) + " lines of code.")  Congratulations, Dev! You just wrote 50 lines of code.  lines\_yesterday = "50"  lines\_today = "108"  lines\_more = int(lines\_today) - int(lines\_yesterday)  print(lines\_more)  58 |
| repr(x) | * Converts object x to an expression string. repr() [compute the “official” string representation of an object](http://docs.python.org/reference/datamodel.html#object.__repr__)(a representation that has all information about the object) and str() is used to [compute the “informal” string representation of an object](http://docs.python.org/reference/datamodel.html#object.__str__) (a representation that is useful for printing the object).   import datetime  today = datetime.datetime.now()    # Prints readable format for date-time object  print(str(today))    # prints the official format of date-time object  print(repr(today))  Output:  2016-02-22 19:32:04.078030  datetime.datetime(2016, 2, 22, 19, 32, 4, 78030) |
| eval(str) | Evaluates a string and returns an object.  x = 1  eval('x+1')  2 |
| tuple(s) | Converts s to a tuple.  In Python:   * a [**tuple**](https://www.digitalocean.com/community/tutorials/understanding-tuples-in-python-3) is an immutable ordered sequence of elements contained within parentheses ( ).   sea\_creatures = **[**'shark', 'cuttlefish', 'squid', 'mantis shrimp'**]**  print(type(sea\_creatures))  <class 'list'>  x = tuple(sea\_creatures);  print(x, '\t', type(x));  **(**'shark', 'cuttlefish', 'squid', 'mantis shrimp'**)** <class 'tuple'> |
| list(s) | Converts s to a list.  In Python:   * a [**list**](https://www.digitalocean.com/community/tutorials/understanding-lists-in-python-3) is a mutable ordered sequence of elements that is contained within square brackets [ ].   sea\_creatures = **(**'shark', 'cuttlefish', 'squid', 'mantis shrimp'**)**  print(type(sea\_creatures));  <class 'tuple'>  x = list(sea\_creatures);  print(x, '\t', type(x));  **[**'shark', 'cuttlefish', 'squid', 'mantis shrimp'**]** <class 'list'> |
| set(s) | Converts s to a set.  lang = ("Perl", "Python", "Java")  print(type(lang));  <class 'tuple'>  x = set(lang)  print(x, '\t', type(x));  {'Python', 'Java', 'Perl'} <class 'set'>  We can define sets (since Python2.6) without using the built-in set function. We can use curly braces instead:  x = {"cheap","expensive","inexpensive","economical"}  x  {'economical', 'inexpensive', 'expensive', 'cheap'}  print(x,'\t',type(x))  {'economical', 'inexpensive', 'expensive', 'cheap'} <class 'set'> |
| Immutable Sets | Sets are implemented in a way, which doesn't allow mutable objects. The following example demonstrates that we can include tuples as elements in the sets but cannot include multiple lists as elements (single list is allowed):  cities = set((("Python","Perl"), ("Paris", "Berlin", "London")))  # Here we used tuples.  print(cities)  {('Paris', 'Berlin', 'London'), ('Python', 'Perl')}  # Here we are going to use lists. Single list is allowed as in frozenset.  cities = set((["Python","Perl"], ["Paris", "Berlin", "London"]))  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  TypeError: unhashable type: 'list'  cities = set(["Paris", "Berlin", "London"],["Delhi","Mumbai"])  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  **TypeError: set expected at most 1 arguments, got 2** |
| frozenset(s) | Converts s to a frozen set.  Though sets can't contain mutable objects (list), multiple lists are not allowed single list is allowed, sets are mutable:  cities = set(["Paris", "Berlin", "London"])  cities.add("Rome")  cities  {'London', 'Berlin', 'Rome', 'Paris'}  Frozensets are like sets except that they cannot be changed, i.e. they are immutable:  cities = frozenset(["Paris", "Berlin", "London"])  cities  frozenset({'London', 'Berlin', 'Paris'})  cities.add("Rome")  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  AttributeError: 'frozenset' object has no attribute 'add' |
| Set Operations | add(element)  A method which adds an element, which has to be immutable, to a set.  colours = {"red","green"}  colours.add("yellow")  colours  {'green', 'yellow', 'red'}  colours.add(["black","white"]) # List cannot be a added.  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  TypeError: unhashable type: 'list'  Of course, an element will only be added, if it is not already contained in the set. If it is already contained, the method call has no effect.  colours.add("yellow")  colours  {'yellow', 'green', 'red'} clear colours.clear()  colours  set() copy cities = {"Delhi","Mumbai","Chennai","Kolkata"};  metro = cities.copy();  metro  {'Chennai', 'Kolkata', 'Mumbai', 'Delhi'}  metro1 = cities;  metro1  {'Chennai', 'Kolkata', 'Mumbai', 'Delhi'} difference() This method returns the difference of two or more sets as a new set.  x = {"a","b","c","d","e"}  y = {"b","c"}  z = {"c","d"}  x.difference(y)  {'a', 'e', 'd'}  x  {'a', 'c', 'e', 'd', 'b'}  x.difference(y).difference(z)  {'a', 'e'}    Instead of using the method difference, we can use the operator "-":  x - y  {'a', 'e', 'd'}  x - y - z  {'a', 'e'} difference\_update() The method difference\_update removes all elements of another set from this set. x.difference\_update(y) is the same as "x = x - y"  x = {"a","b","c","d","e"}  y = {"b","c"}  x.difference\_update(y)  x  {'a', 'e', 'd'}    x = {"a","b","c","d","e"}  y = {"b","c"}  x = x - y  x  {'a', 'e', 'd'} discard(el) An element el will be removed from the set, if it is contained in the set. If el is not a member of the set, nothing will be done.  x = {"a","b","c","d","e"}  x.discard("a")  x  {'c', 'b', 'e', 'd'}  x.discard("z")  x  {'c', 'b', 'e', 'd'}    x["b"]  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  TypeError: 'set' object is not subscriptable  del x["b"]  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  TypeError: 'set' object does not support item deletion remove(el) works like discard(), but if el is not a member of the set, a KeyError will be raised.  x = {"a","b","c","d","e"}  x.remove("a")  x  {'c', 'b', 'e', 'd'}  x.remove("z")  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  KeyError: 'z' union(s) This method returns the union of two sets as a new set, i.e. all elements that are in either set.  x = {"a","b","c","d","e"}  y = {"c","d","e","f","g"}  x.union(y)  {'d', 'a', 'g', 'c', 'f', 'b', 'e'}  This can be abbreviated with the pipe operator "|":  x = {"a","b","c","d","e"}  y = {"c","d","e","f","g"}  x | y  {'d', 'a', 'g', 'c', 'f', 'b', 'e'} intersection(s) Returns the intersection of the instance set and the set s as a new set. In other words: A set with all the elements which are contained in both sets is returned.  x = {"a","b","c","d","e"}  y = {"c","d","e","f","g"}  x.intersection(y)  {'c', 'e', 'd'}    This can be abbreviated with the ampersand operator "&":  x = {"a","b","c","d","e"}  y = {"c","d","e","f","g"}  x & y  {'c', 'e', 'd'} isdisjoint() This method returns True if two sets have a null intersection.  x = {"a","b","c"}  y = {"c","d","e"}  x.isdisjoint(y)  False    x = {"a","b","c"}  y = {"d","e","f"}  x.isdisjoint(y)  True   issubset() x.issubset(y) returns True, if x is a subset of y. "<=" is an abbreviation for "Subset of" and ">=" for "superset of" and "<" is used to check if a set is a proper subset of a set.  x = {"a","b","c","d","e"}  y = {"c","d"}  x.issubset(y)  False  y.issubset(x)  True  x < y  False  y < x # y is a proper subset of x  True  x < x # a set can never be a proper subset of oneself.  False  x <= x  True issuperset() x.issuperset(y) returns True, if x is a superset of y. ">=" is an abbreviation for "issuperset of" ">" is used to check if a set is a proper superset of a set.  x = {"a","b","c","d","e"}  y = {"c","d"}  x.issuperset(y)  True  x > y  True  x >= y  True  x >= x  True  x > x  False  x.issuperset(x)  True pop() pop() removes and returns an arbitrary set element. The method raises a KeyError if the set is empty  x = {"a","b","c","d","e"}  x.pop()  'a'  x.pop()  'c'  x.pop()  'e'  x.pop()  'd'  x.pop()  'b'  x.pop()  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  KeyError: 'pop from an empty set' |
| dict(d) | Creates a dictionary. d must be a sequence of (key,value) tuples.  state\_population = {"UP":199581477, "MH":112372972, "BR":103804637, "WB": 91347736, "AP": 84665533, "MP":72597565, "TN":72138958, "RJ":68621012, "KA":61130704, "GJ":60383628, "OD":41947358};  print(type(state\_population))  <class 'dict'>  state\_population["UP"]  199581477  state\_population["MH"]  112372972  state\_population["Punjab"] = 27704236;  # Add new element  state\_population["Punjab"] = 27704236;  state\_population  {'UP': 199581477, 'MH': 112372972, 'BR': 103804637, 'WB': 91347736, 'AP': 84665533, 'MP': 72597565, 'TN': 72138958, 'RJ': 68621012, 'KA': 61130704, 'GJ': 60383628, 'OD': 41947358, 'Punjab': 27704236}  # Convert tuple to dictionary.  t = (('a','Apple'), ('b','Ball'), ('c', 'Cat'), ('d', 'Dog'));  print(t, type(t))  (('a', 'Apple'), ('b', 'Ball'), ('c', 'Cat'), ('d', 'Dog')) <class 'tuple'>  x = dict(t);  print(x, type(x))  {'a': 'Apple', 'b': 'Ball', 'c': 'Cat', 'd': 'Dog'} **<class 'dict'>**  y = dict((x, y) for x, y in t);  print(y, type(y))  {'a': 'Apple', 'b': 'Ball', 'c': 'Cat', 'd': 'Dog'} <class 'dict'>  y['a']  'Apple'  y['d']  'Dog'  z = dict(map(reversed, t));  print(z, type(z))  {'Apple': 'a', 'Ball': 'b', 'Cat': 'c', 'Dog': 'd'} <class 'dict'>  # Convert list to dictionary.  x=[('hi','goodbye')]  print(x,type(x));  [('hi', 'goodbye')] <class 'list'>  y = dict(x);  print(y,type(y));  {'hi': 'goodbye'} <class 'dict'> |

Character Values in ASCII



|  |  |
| --- | --- |
| chr(x) | Converts an integer to a character.  The chr() method takes a single parameter, an integer.  The valid range of the integer is from 0 through 1,114,111.  print(chr(97));  print(chr(65));  print(chr(90));  print(chr(1119));  print(chr(63))  a  A  Z  џ  ? |
| unichr(x) | Converts an integer to a Unicode character.  Can't use unichr in Python 3.1  print(unichr(1119));  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  NameError: name 'unichr' is not defined |
| ord(x) | The ord() method returns an integer representing Unicode code point for the given Unicode character.  The ord() method is the inverse of chr().  The ord(x) method takes a single parameter:  x - character string of length 1 whose Unicode code point is to be found.  print(ord('1'))  49  print(ord('5'))  53  print(ord('A'))  65  print(ord('Z'))  90 |
| hex(x) | The hex() function converts an integer number to the corresponding hexadecimal string.  The hex() function takes a single argument.  print(hex(110));  print(hex(111));  print(hex(112));  print(hex(113));  print(hex(114));  print(hex(115));  0x6e  0x6f  0x70  0x71  0x72  0x73  Hexadecimal representation of a float.  print(float.hex(2.5));  0x1.4000000000000p+1  print(float.hex(5.5));  0x1.6000000000000p+2 |
| oct(x) | Converts an integer to an octal string.  print(oct(555));  0o1053  print(oct(555.55));  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  TypeError: 'float' object cannot be interpreted as an integer |

# Python Basic Operators

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

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

## **Types of Operator**

Python language supports the following types of operators.

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

## **Python Arithmetic Operators**

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

a = 10

b = 20

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + Addition | Adds values on either side of the operator. | a + b = 30 |
| - Subtraction | Subtracts right hand operand from left hand operand. | a – b = -10 |
| \* Multiplication | Multiplies values on either side of the operator | a \* b = 200 |
| / Division | Divides left hand operand by right hand operand | b / a = 2 |
| % Modulus | Divides left hand operand by right hand operand and returns remainder | b % a = 0 |
| \*\* Exponent | Performs exponential (power) calculation on operators | a\*\*b =10 to the power 20 |
| // | Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity): | 9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0 |

## **Python Comparison Operators**

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

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

|  |  |  |
| --- | --- | --- |
| **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. |  |
| > | If the value of left operand is greater than the value of right operand, then condition becomes true. | (a > b) is not true. |
| < | If the value of left operand is less than the value of right operand, then condition becomes true. | (a < b) is true. |
| >= | If the value of left operand is greater than or equal to the value of right operand, then condition becomes true. | (a >= b) is not true. |
| <= | If the value of left operand is less than or equal to the value of right operand, then condition becomes true. | (a <= b) is true. |

## **Python Assignment Operators**

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

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Assigns values from right side operands to left side operand | c = a + b assigns value of a + b into c |
| += Add AND | It adds right operand to the left operand and assign the result to left operand | c += a is equivalent to c = c + a |
| -= Subtract AND | It subtracts right operand from the left operand and assign the result to left operand | c -= a is equivalent to c = c - a |
| \*= Multiply AND | It multiplies right operand with the left operand and assign the result to left operand | c \*= a is equivalent to c = c \* a |
| /= Divide AND | It divides left operand with the right operand and assign the result to left operand | c /= a is equivalent to c = c / a |
| %= Modulus AND | It takes modulus using two operands and assign the result to left operand (return remainder) | 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 (return quotient) | c //= a is equivalent to c = c // a  c=58;  c %=a;  print(c);  8  c=58;  c //=a;  print(c);  5 |

## **Python Logical Operators**

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

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| and Logical AND | If both the operands are true then condition becomes true. | (a and b) is true. |
| or Logical OR | If any of the two operands are non-zero then condition becomes true. | (a or b) is true. |
| not Logical NOT | Used to reverse the logical state of its operand. | Not(a and b) is false. |

Example.

x = True;

y = False;

print('x and y is ',x and y);

x and y is False

print('x or y is ',x or y);

x or y is True

print('not x is ',not x);

not x is False

## **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 => 60

b = 0000 1101 => 13

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

a&b = 0000 1100 => 12

a|b = 0011 1101 => 61

a^b = 0011 0001 => 49

~a  = 1100 0011 => -61 (-64+3)

There are following Bitwise operators 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)  a = 60;  b = 13;  x = a&b;  print(x);  12 |
| | Binary OR | It copies a bit if it exists in either operand. | (a | b) = 61 (means 0011 1101)  x = a|b;  print(x);  61 |
| ^ Binary XOR (exclusive OR) | Write the bit as 1 if it is set in one operand otherwise 0 if matches in both operand. | (a ^ b) = 49 (means 0011 0001)  x = a^b;  print(x);  49 |
| ~ 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.  x = ~a;  print(x);  -61 |
| << Binary Left Shift | The left operands value is moved left by the number of bits specified by the right operand. | a << 2 = 240 (means 1111 0000)  x = a << 2 ;  print(x);  240 |
| >> Binary Right Shift | The left operands value is moved right by the number of bits specified by the right operand. | a >> 2 = 15 (means 0000 1111)  x = a >> 2 ;  print(x);  15 |

## **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 finds a variable in the specified sequence and false otherwise. | x not in y, here not in results in a 1 if x is not a member of sequence y. |

### **Example**

a = 10

b = 20

l\_num = [1, 2, 3, 4, 5 ];

if ( a in l\_num ):

print("a is available in the given list");

else:

print("a is not available in the given list");

if ( b not in l\_num ):

print("b is not available in the given list");

else:

print("b is available in the given list");

x = 2

if ( x in l\_num ):

print("x is available in the given list");

else:

print("x is not available in the given list");

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

a is not available in the given list

b is not available in the given list

x is available in the given list

print( a in l\_num );

False

print( b in l\_num );

False

x = 2

print( x in l\_num );

True

## **Python Identity Operators**

Identity operators are used to verify if two variables point to the same memory location or not. Identity operators are of two types:

(1) is

(2) is not.

|  |  |  |
| --- | --- | --- |
| **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**

a = 20

b = 20

if ( a is b ):

print("Line 1 - a and b have same identity")

else:

print("Line 1 - a and b do not have same identity")

if ( id(a) == id(b) ):

print("Line 2 - a and b have same identity")

else:

print("Line 2 - a and b do not have same identity")

b = 30

if ( a is b ):

print("Line 3 - a and b have same identity")

else:

print("Line 3 - a and b do not have same identity")

if ( a is not b ):

print("Line 4 - a and b do not have same identity")

else:

print("Line 4 - a and b have same identity")

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

Line 1 - a and b have same identity

Line 2 - a and b have same identity

Line 3 - a and b do not have same identity

Line 4 - a and b do not have same identity

a = 20

b = 20

id(a)

1539688720

id(b)

1539688720

a = 555

b = 555

id(a)

1390899932912

id(b)

1390901046000

a = 1000

b = 1000

id(a)

1390901046096

id(b)

1390899932912

a = 20

b = 20

id(a)

1539688720

id(b)

1539688720

id(1000)

1390899932912

b = 1000

id(b)

1390899930224

## **Python Operators Precedence**

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

|  |  |
| --- | --- |
| **Operator** | **Description** |
| \*\* | Exponentiation (raise to the power) |
| ~ + - | Complement, unary plus and minus (method names for the last two are +@ and -@)  a = 20  b = 1000  -a  -20  +a  20  -b  -1000  +b  1000  ~a  -21  ~b  -1001 |
| \* / % // | 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 how an expression is evaluated.

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

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

### **Example**

a = 20

b = 10

c = 15

d = 5

e = 0

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 −

Value of (a + b) \* c / d is 90

Value of ((a + b) \* c) / d is 90

Value of (a + b) \* (c / d) is 90

Value of a + (b \* c) / d is 50

**Does Python have a ternary conditional operator?**

**Yes, it was added in version 2.5.**

The syntax is:

a if condition else b

First condition is evaluated, then either a or b is returned based on the Boolean value of condition

If condition evaluates to True a is returned, else b is returned.

'true' if True else 'false'

'true'

'false' if True else 'false'

'false'

'true' if False else 'false'

'false'

'true' if 1 else 'false'

'true'

'true' if 0 else 'false'

'false'

# Python Decision Making

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

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

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



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

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

|  |  |
| --- | --- |
| **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).  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 most one statement, there can be an arbitrary number of **elif** statements following an if. **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) |

var1 = 5555

if var1:

print("1 – Condition evaluated true");

print(var1);

var2 = 0

if var2:

print("2 - Condition evaluated false");

print(var2);

else:

print("Good bye!");

v\_marks = 85

if (v\_marks<50) :

print("He is fail.");

print("He should work hard.");

elif (v\_marks>=50 and v\_marks<60) :

print("He stood 2nd."); print("He need improvments.");

elif (v\_marks>=60 and v\_marks<80) :

print("He stood 1st.");

else :

print("Outstanding.");

print("Well done.");

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

1 - Condition evaluated true

5555

Good bye!

Outstanding.

Well done.

**# Another Example**

var = 100

if var < 200:

print("Expression value is less than 200");

if var == 150:

print("Which is 150");

elif var == 100:

print("Which is 100");

elif var == 50:

print("Which is 50");

elif var < 50:

print("Expression value is less than 50");

else:

print("Could not find true expression");

print("Good bye!"); # This statement is not the of any if.

**# Output**

Expression value is less than 200

Which is 100

>>> print("Good bye!"); # This statement is not the of any if.

Good bye!

## **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 −

var = 100

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

print "Good bye!"

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

Value of expression is 100

Good bye!

# Python Loops

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

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

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



Python programming language provides following types of loops to handle looping requirements. There are two types of loops in Python, **for** and **while**.

|  |  |
| --- | --- |
| **Loop Type** | **Description** |
| [**while loop**](https://www.tutorialspoint.com/python/python_while_loop.htm) | Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body. |
| [**for loop**](https://www.tutorialspoint.com/python/python_for_loop.htm) | Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. |

**Python While Loop**

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.

We generally use this loop when we don't know beforehand, the number of times to iterate.

## Syntax

while test\_expression:

Body of while

**In Python, the body of the while loop is determined through indentation.**

Body starts with indentation and the first unindented line marks the end.

Python interprets any non-zero value as True. None and 0 are interpreted as False.

# Prints out 1,2,3,4,5

count = 1

while count <= 5:

print(count);

count += 1;

print("Good Bye While Loop");

# Output

1

2

3

4

5

Good Bye While Loop

In Python, break and continue statements can alter the flow of a normal loop. The break statement terminates the loop containing it and pass the control of the program to the statement immediately after the body of the loop.

If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.

# Prints out 1,2,3,4,5

count = 1

while True:

print(count)

count += 1

if count > 5:

break;

print("Good Bye While Loop");

# Output

1

2

3

4

5

Good Bye While Loop

# Prints out 1,2,3,4,6,7,8,9,10

count = 1

while (count<=10):

if (count==5):

count=count+1

continue

else:

print(count)

count=count+1

print("Good Bye While Loop")

# Output

1

2

3

4

6

7

8

9

10

Good Bye While Loop

**We use "else" clause in loops.**

When the loop condition of "for" or "while" statement fails then code part in "else" is executed. If break statement is executed inside for loop then the "else" part is skipped. Note that "else" part is executed even if there is a continue statement.

# While loop with else.

# Print out 1,2,3,4,5 and then it prints "count value reached 5"

count=1

while(count<5):

print(count)

count +=1

else:

print("count value reached %d" %(count))

# Output

1

2

3

4

count value reached 5

**Python For Loop**

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

## Syntax of for Loop

for val in sequence:

Body of for

Here, val is the variable that takes the value of the item inside the sequence on each iteration.

Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

for x in (1,2,3,4,5):

print(x)

# Output

1

2

3

4

5

for x in [1,2,3,4,5]:

print(x)

# Output

1

2

3

4

5

for x in ['a','b','c','d','e']:

print(x)

# Output

a

b

c

d

e

for x in range(5):

print(x)

# Output

0

1

2

3

4

for x in range(3, 6):

print(x)

# Output

3

4

5

# range in increment order of 2

for x in range(3, 10, 2):

print(x)

# Output

3

5

7

9

# Prints out only odd numbers - 1,3,5,7,9

for x in range(10):

# Check if x is even

if x % 2 == 0:

**continue;**

print(x);

# Output

1

3

5

7

9

# Prints out 1,2,3,4

for i in range(1, 10):

if(i%5==0):

**break**

print(i)

else:

print("this is not printed because for loop is terminated because of break but not due to fail in condition")

# Output

1

2

3

4

# Program to find the sum of all numbers stored in a list

# List of numbers

numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

sum = 0

for val in numbers:

sum = sum+val;

else:

print("The sum is", sum);

# Output

The sum is 48

**Python Pass Statement**

It is used as a placeholder for future implementation of functions, loops, etc.

In Python programming, pass is a **null** statement. The difference between a comment and pass statement is that the interpreter ignores a comment entirely, pass is not ignored it is executed. However, nothing happens when pass is executed. It results into no operation (NOP).

## Syntax of pass

pass

We generally use it as a placeholder.

Suppose we have a loop or a function that is not implemented yet, but we want to implement it in the future. They cannot have an empty body. The interpreter would complain. So, we use the pass statement to construct a body that does nothing.

# pass is just a placeholder for

# functionality to be added later.

sequence = [1,2,3,4,5]

for val in sequence:

pass

# Python Numbers

# Integer Objects

All integers are implemented as “long” integer objects of **arbitrary size**.

<https://docs.python.org/3/c-api/long.html>

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

Number objects are created when you assign a value to them. For example −

var1 = 1

var2 = 10

id(var1)

1539688112

id(var2)

1539688400

var1 = 100

var2 = 200

**NOTE: Here object is reallocated.**

id(var1)

1539691280

id(var2)

1539694480

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

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

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

del var

del var\_a, var\_b

Python supports four different numerical types −

* **int (signed integers)**: They are often called just integers or ints, are positive or negative whole numbers with no decimal point.
* **long (long integers )**: Also called longs, they are integers of unlimited size, written like integers and followed by an uppercase or lowercase L. No longer supported in Version 3.x.
* **float (floating point real values)** : Also called floats, they represent real numbers and are written with a decimal point dividing the integer and fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 (2.5e2 = 2.5 x 102 = 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.

**What is the range of int in python?**

**Python** has arbitrary precision integers so there **is no true** fixed maximum. You're only limited by available memory. sys.maxint does not even exist in **Python** 3, since **int** and long were unified into a single arbitrary precision **int** type.

v\_num = 9999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999;

print(v\_num);

9999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999

### **Examples**

Here are some examples of numbers

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

## **Number Type Conversion**

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

* Type **int(x)** to convert x to a plain integer.
* Type **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

|  |
| --- |
| float(2);  2.0  float(5.55);  5.55  complex(5)  (5+0j)  complex(5,2)  (5+2j) |

## **Mathematical Functions**

Python includes following functions that perform mathematical calculations.

|  |  |
| --- | --- |
| **Function** | **Returns ( description )** |
| **abs(x)** | The absolute value of x: the (positive) distance between x and zero.  abs(-5);  5 |
| **ceil(x)** | The ceiling of x: the smallest integer not less than x  import math  math.ceil(4.4); 5  math.ceil(4.7); 5 |
| **cmp(x, y)** | -1 if x < y, 0 if x == y, or 1 if x > y Python 3 doesn't have a cmp function. |
| **exp(x)** | The exponential of x: ex  math.exp(1e-5)  1.00001000005 |
| **fabs(x)** | The absolute value of x.  math.fabs(-5.5)  5.5  abs(-5.5)  5.5 |
| **floor(x)** | The floor of x: the largest integer not greater than x  math.floor(4.4); 4  math.floor(4.8); 4 |
| **log(x)** | The natural logarithm of x, for x> 0  math.log(5);  1.6094379124341003 |
| **log10(x)** | The base-10 logarithm of x for x> 0 .  math.log10(5);  0.6989700043360189 |
| **max(x1, x2,...)** | The largest of its arguments: the value closest to positive infinity  max(10,20,66,45,87,55,23,58)  87 |
| **min(x1, x2,...)** | The smallest of its arguments: the value closest to negative infinity  min(10,20,66,45,87,55,23,58)  10 |
| **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.  math.modf(5);  (0.0, 5.0)  math.modf(1,5);  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  TypeError: modf() takes exactly one argument (2 given) |
| **pow(x, y)** | The value of x\*\*y.  math.pow(2,5);  32.0 |
| **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.  round(7676567.6556757,3);  7676567.656  round(7676567.6544757,3);  7676567.654 |
| **sqrt(x)** | The square root of x for x > 0  math.sqrt(16)  4.0 |

## **Random Number Functions**

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

|  |  |
| --- | --- |
| **Function** | **Description** |
| **choice(seq)** | A random item from a list, tuple, or string.  import random;  x=**[**1, 2, 3, 5, 9**]**;  print(type(x))  <class 'list'>  x=**(**1, 2, 3, 5, 9**)**;  print(type(x))  <class 'tuple'>  random.choice([1, 2, 3, 5, 9]);  5  random.choice([1, 2, 3, 5, 9]);  1  random.choice((1, 2, 3, 5, 9));  5  random.choice((1, 2, 3, 5, 9));  2  random.choice('India is my country.');  'd'  random.choice('India is my country.');  'm' |
| **randrange ([start,] stop [,step])** | A randomly selected element from range(start, stop, step)  random.randrange(100, 1000, 2);  334  random.randrange(100, 1000, 2);  434  random.randrange(100, 1000, 2);  624 |
| **random()** | A random float r, such that 0 is less than or equal to r and r is less than 1  print("random() : ", random.random());  random() : 0.0623392041549814  print("random() : ", random.random());  random() : 0.9609234212687972 |
| **seed([x])** | Sets the integer starting value used in generating random numbers. Call this function before calling any other random module function. Returns None.  random.seed( 10 )  print("Random number with seed 10 : ", random.random())  Random number with seed 10 : 0.5714025946899135  random.seed( 10 )  print("Random number with seed 10 : ", random.random())  Random number with seed 10 : 0.5714025946899135  random.seed( 10 )  print("Random number with seed 10 : ", random.random())  Random number with seed 10 : 0.5714025946899135  print("Random number with seed 10 : ", random.random())  Random number with seed 10 : 0.4288890546751146  print("Random number with seed 10 : ", random.random())  Random number with seed 10 : 0.20609823213950174  print("Random number with seed 10 : ", random.random())  Random number with seed 10 : 0.81332125135732  print("Random number with seed 10 : ", random.random())  Random number with seed 10 : 0.8235888725334455 |
| **shuffle(lst)** | Randomizes the items of a list in place. Returns None.  list = [20, 16, 10, 5];  random.shuffle(list);  print(list);  [10, 5, 20, 16]  random.shuffle(list);  print(list);  [5, 16, 10, 20]  random.shuffle(list);  print(list);  [20, 16, 5, 10] |
| **uniform(x, y)** | A random float r, such that x is less than or equal to r and r is less than y  print("Random Float uniform(1, 10) : ", random.uniform(1, 10))  print("Random Float uniform(1, 10) : ", random.uniform(1, 10))  Random Float uniform(1, 10) : 6.418091089533576  Random Float uniform(1, 10) : 8.37482423567478  print("Random Float uniform(7, 10) : ", random.uniform(7, 10))  print("Random Float uniform(7, 10) : ", random.uniform(7, 10))  Random Float uniform(7, 10) : 8.370493453174916  Random Float uniform(7, 10) : 9.057584456423784 |

## **Mathematical Constants**

The module also defines two mathematical constants −

|  |  |
| --- | --- |
| **Constants** | **Description** |
| pi | The mathematical constant pi.  import math  math.pi  3.141592653589793 |
| e | The mathematical constant e.  math.e  2.718281828459045 |

# Python Strings

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

var1 = 'Hello World!'

var2 = "Python Programming"

## **Accessing Values in Strings**

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

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

#!/usr/bin/python

var1 = 'Hello World!'

var2 = "Python Programming"

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

print("var2[1:5]: ", var2[1:5])

print("var2[:10]: ", var2[:10])

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

var1[0]: H

var2[1:5]: ytho

var2[:10]: Python Pro

## **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. You cannot update a character of a string. For example −

var1 = 'Hello World!'

print(var1);

var1 = 'Hello World, Welcome to Python!'

print(var1);

var1 = 'ABC';

var1[2] = 'D'

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'str' object does not support item assignment

## **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 sound generate from your laptop  print('Calling Bell!', '\a');  Calling Bell! |
| \b | 0x08 | Backspace  print('Calling Bell!'+' open the door.');  Calling Bell! open the door.  print('Calling Bell!'+'\b'+' open the door.');  Calling Bell open the door.  print('Calling Bell!'+'\b\b\b\b\b'+' open the door.');  Calling open the door. |
| \e | 0x1b | Escape |
| \f | 0x0c | Formfeed  print('Calling Bell!','\f','Open the door'); |
| \n | 0x0a | Newline  print('Calling Bell!','\n','\bOpen the door');  Calling Bell!  Open the door |
| \nnn |  | Octal notation, where n is in the range 0.7 |
| \r | 0x0d | Carriage return  print('Calling Bell!','\r','Open the door', '\r', 'Guests are welcome.');  Guest are welcome. |
| \s | 0x20 | Space |
| \t | 0x09 | Tab  print('Hello''\t''World');  Hello World |
| \v | 0x0b | Vertical tab |
| \x |  | Character x |
| \xnn |  | Hexadecimal notation, where n is in the range 0.9, a.f, or A.F  print('Hello''\xad''World');  Hello­World  print('Hello''\xae''World');  Hello®World |

## **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  a = 'Hello'  b = 'Python'  a+b  'HelloPython' |
| \* | Repetition - Creates new strings, concatenating multiple copies of the same string | a\*2 will give –HelloHello  a\*3  'HelloHelloHello' |
| [] | 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 (True)  'l' in var1  True  'w' in var1  False |
| not in | Membership - Returns true if a character does not exist in the given string | M not in a will give 1 (True)  'w' not in var1  True |
| 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  print('\n')  print(r'\n')  \n |
| % | Format - Performs String formatting | v\_num = 100;  print('v\_num is: %d' %(v\_num));  v\_num is: 100 |

## **String Formatting Operator**

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

#!/usr/bin/python

print("Heavyweight boxer %s, total win %d/58" % ('Michael Gerard Tyson', 50));

Heavyweight boxer Michael Gerard Tyson, total win 50/58

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

|  |  |
| --- | --- |
| **Format Symbol** | **Conversion** |
| %c | Character  fname='Hari'  lname='Yadav'  print("My name initial is %c%c and weight is %d kg!" % (fname[0],lname[0], 65));  My name initial is HY and weight is 65 kg! |
| %s | string conversion via str() prior to formatting  print("My name is %s" %("Michael Gerard Tyson"));  My name is Michael Gerard Tyson  print("My name is {} {} and surname {} ".format('Michael', 'Gerard', 'Tyson'));  My name is Michael Gerard and surname Tyson  print('%s' % ('test',))  test  print('%10s' % ('test',))  test |
| %i | signed decimal integer |
| %d | signed decimal integer  print("My name initial is %c%c and weight is %d kg!" % (fname[0],lname[0], +65));  My name initial is HY and weight is 65 kg!  print("My name initial is %c%c and weight is %d kg!" % (fname[0],lname[0], -65));  My name initial is HY and weight is -65 kg! |
| %u | unsigned decimal integer  Obsolete type – it is identical to ‘d’. See PEP 237. |
| %o | octal integer  print("My name initial is %c%c and weight is %o kg!" % (fname[0],lname[0], -65));  My name initial is HY and weight is -101 kg!  print("My name initial is %c%c and weight is %o kg!" % (fname[0],lname[0], +65));  My name initial is HY and weight is 101 kg! |
| %x | hexadecimal integer (lowercase letters)  print("My name initial is %c%c and weight is %x kg!" % (fname[0],lname[0], 65));  My name initial is HY and weight is 41 kg! |
| %X | hexadecimal integer (UPPERcase letters) |
| %e | exponential notation (with lowercase 'e')  print("My name initial is %c%c and weight is %e kg!" % (fname[0],lname[0], +65));  My name initial is HY and weight is 6.500000e+01 kg! |
| %E | exponential notation (with UPPERcase 'E')  print("My name initial is %c%c and weight is %E kg!" % (fname[0],lname[0], -65));  My name initial is HY and weight is -6.500000E+01 kg! |
| %f | floating point real number  print("My name initial is %c%c and weight is %f kg!" % (fname[0],lname[0], 65));  My name initial is HY and weight is 65.000000 kg! |

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

|  |  |
| --- | --- |
| **Symbol** | **Functionality** |
| \* | argument specifies width or precision |
| - | left justification  print('%10d' %(837.64)); # right justified.  837  print('%-10d' %(837.64)); # left justified.  837 |
| + | display the sign  print('%010d' %(+837.64));  0000000837  print('%+010d' %(837.64));  +000000837  print('%010d' %(-837.64));  -000000837 |
| <sp> | leave a blank space before a positive number  print('% 010d' %(837.64));  000000837 |
| # | add the octal leading zero ( '0' ) or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used.  print("%#x" % 17 )  0x11  print("%#x" % 18 )  0x12  print("%x" % 17 )  11  print("%x" % 18 )  12 |
| 0 | pad from left with zeros (instead of spaces)  print('%10d' %(837.64));  837  print('%010d' %(837.64));  0000000837 |
| f | Represent the value in floating point.  print('%f' %(837.64759678));  837.647597  print('%.1f' %(837.64759678));  837.6  print('%.2f' %(837.64759678));  837.65  print('%.4f' %(837.64759678));  837.6476  print('%010.2f' %(837.64759678));  0000837.65  print('%010.4f' %(837.64759678));  00837.6476 |
| % | '%%' leaves you with a single literal '%'  print('I have scored %d%%' %(85));  I have scored 85%  print('I have scored %d%% %d%%' %(85,90));  I have scored 85% 90% |

## **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.

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 −

print('C:\\nowhere')

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

C:\nowhere

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

print(r'C:\\nowhere')

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

C:\\nowhere

## **Unicode String**

Normal strings in Python are stored internally as 8-bit ASCII, while Unicode strings are stored as 16-bit Unicode. This allows for a more varied set of characters, including special characters from most languages in the world.

Var1 = 'Hello, world!';

Var2 = u'Hello, world!';

print(u'Hello, world!')

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

Hello, world!

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

## **unicode()** global function[#](http://www.diveintopython3.net/porting-code-to-python-3-with-2to3.html#unicode)

Python 2 had two global functions to coerce objects into strings: unicode() to coerce them into Unicode strings, and str() to coerce them into non-Unicode strings. Python 3 has only one string type so the str() function is all you need. (The unicode() function no longer exists.)

|  |  |  |
| --- | --- | --- |
| **Notes** | **Python 2** | **Python 3** |
|  | unicode(anything) | str(anything) |

## **Built-in String Methods**

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

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **capitalize()** Capitalizes first letter of string  print(str.capitalize('Michael Gerard Tyson'));  Michael gerard tyson |
| 2 | [**center(width, fillchar)**](https://www.tutorialspoint.com/python/string_center.htm)  Returns a space-padded string with the original string centered to a total of width columns.  vname = 'MICHAEL GERARD TYSON';  print(vname.center(30,'x'));  xxxxxMICHAEL GERARD TYSONxxxxx  print(vname.center(30,' '));  MICHAEL GERARD TYSON  print(vname.center(30,' x '));  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  TypeError: The fill character must be exactly one character long |
| 3 | [**count(str, beg= 0,end=len(string))**](https://www.tutorialspoint.com/python/string_count.htm)  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')**](https://www.tutorialspoint.com/python/string_decode.htm)  Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding. |
| 5 | [**encode(encoding='UTF-8',errors='strict')**](https://www.tutorialspoint.com/python/string_encode.htm)  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))**](https://www.tutorialspoint.com/python/string_endswith.htm) 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)**](https://www.tutorialspoint.com/python/string_expandtabs.htm)  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))**](https://www.tutorialspoint.com/python/string_find.htm)  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))**](https://www.tutorialspoint.com/python/string_index.htm)  Same as find(), but raises an exception if str not found. |
| 10 | [**isalnum()**](https://www.tutorialspoint.com/python/string_isalnum.htm)  Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise. |
| 11 | [**isalpha()**](https://www.tutorialspoint.com/python/string_isalpha.htm)  Returns true if string has at least 1 character and all characters are alphabetic and false otherwise. |
| 12 | [**isdigit()**](https://www.tutorialspoint.com/python/string_isdigit.htm)  Returns true if string contains only digits and false otherwise. |
| 13 | [**islower()**](https://www.tutorialspoint.com/python/string_islower.htm)  Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise. |
| 14 | [**isnumeric()**](https://www.tutorialspoint.com/python/string_isnumeric.htm)  Returns true if a unicode string contains only numeric characters and false otherwise. |
| 15 | [**isspace()**](https://www.tutorialspoint.com/python/string_isspace.htm)  Returns true if string contains only whitespace characters and false otherwise. |
| 16 | [**istitle()**](https://www.tutorialspoint.com/python/string_istitle.htm)  Returns true if string is properly "titlecased" and false otherwise. |
| 17 | [**isupper()**](https://www.tutorialspoint.com/python/string_isupper.htm)  Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise. |
| 18 | [**join(seq)**](https://www.tutorialspoint.com/python/string_join.htm)  Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string. |
| 19 | [**len(string)**](https://www.tutorialspoint.com/python/string_len.htm)  Returns the length of the string |
| 20 | [**ljust(width[, fillchar])**](https://www.tutorialspoint.com/python/string_ljust.htm)  Returns a space-padded string with the original string left-justified to a total of width columns. |
| 21 | [**lower()**](https://www.tutorialspoint.com/python/string_lower.htm)  Converts all uppercase letters in string to lowercase.  print(str.lower('Michael Gerard Tyson'));  michael gerard tyson |
| 22 | [**lstrip()**](https://www.tutorialspoint.com/python/string_lstrip.htm)  Removes all leading whitespace in string. |
| 23 | [**maketrans()**](https://www.tutorialspoint.com/python/string_maketrans.htm)  Returns a translation table to be used in translate function. |
| 24 | [**max(str)**](https://www.tutorialspoint.com/python/string_max.htm)  Returns the max alphabetical character from the string str. |
| 25 | [**min(str)**](https://www.tutorialspoint.com/python/string_min.htm)  Returns the min alphabetical character from the string str. |
| 26 | [**replace(old, new [, max])**](https://www.tutorialspoint.com/python/string_replace.htm)  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))**](https://www.tutorialspoint.com/python/string_rfind.htm)  Same as find(), but search backwards in string. |
| 28 | [**rindex( str, beg=0, end=len(string))**](https://www.tutorialspoint.com/python/string_rindex.htm)  Same as index(), but search backwards in string. |
| 29 | [**rjust(width,[, fillchar])**](https://www.tutorialspoint.com/python/string_rjust.htm)  Returns a space-padded string with the original string right-justified to a total of width columns. |
| 30 | [**rstrip()**](https://www.tutorialspoint.com/python/string_rstrip.htm)  Removes all trailing whitespace of string. |
| 31 | [**split(str="", num=string.count(str))**](https://www.tutorialspoint.com/python/string_split.htm)  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'))**](https://www.tutorialspoint.com/python/string_splitlines.htm)  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))**](https://www.tutorialspoint.com/python/string_startswith.htm)  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])**](https://www.tutorialspoint.com/python/string_strip.htm)  Performs both lstrip() and rstrip() on string |
| 35 | [**swapcase()**](https://www.tutorialspoint.com/python/string_swapcase.htm)  Inverts case for all letters in string. |
| 36 | [**title()**](https://www.tutorialspoint.com/python/string_title.htm)  Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase. |
| 37 | [**translate(table, deletechars="")**](https://www.tutorialspoint.com/python/string_translate.htm)  Translates string according to translation table str(256 chars), removing those in the del string. |
| 38 | **upper()** Converts lowercase letters in string to uppercase.  print(str.upper('Michael Gerard Tyson'));  MICHAEL GERARD TYSON |
| 39 | [**zfill (width)**](https://www.tutorialspoint.com/python/string_zfill.htm)  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()**](https://www.tutorialspoint.com/python/string_isdecimal.htm)  Returns true if a unicode string contains only decimal characters and false otherwise. |

# Python Lists

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

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

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

## **Python Lists**

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

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

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

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

list3 = ["a", "b", "c", "d"]

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

## **Accessing Values in Lists**

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

#!/usr/bin/python

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

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

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

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

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

list1[0]: physics

list2[1:5]: [2, 3, 4, 5]

## **Updating Lists**

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

#!/usr/bin/python

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

print "Value available at index 2 : "

print list[2]

list[2] = 2001;

print "New value available at index 2 : "

print list[2]

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

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

Value available at index 2 :

1997

New value available at index 2 :

2001

## **Delete List Elements**

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

#!/usr/bin/python

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

print list1

del list1[2];

print "After deleting value at index 2 : "

print list1

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

['physics', 'chemistry', 1997, 2000]

After deleting value at index 2 :

['physics', 'chemistry', 2000]

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

## **Basic List Operations**

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

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

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

## **Indexing, Slicing, and Matrixes**

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

Assuming following input −

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

|  |  |  |
| --- | --- | --- |
| **Python Expression** | **Results** | **Description** |
| L[2] | 'SPAM!' | Offsets start at zero |
| L[-2] | 'Spam' | Negative: count from the right |
| L[1:] | ['Spam', 'SPAM!'] | Slicing fetches sections |

## **Built-in List Functions & Methods:**

Python includes the following list functions −

|  |  |
| --- | --- |
| **SN** | **Function with Description** |
| 1 | [**cmp(list1, list2)**](https://www.tutorialspoint.com/python/list_cmp.htm)  Compares elements of both lists. |
| 2 | [**len(list)**](https://www.tutorialspoint.com/python/list_len.htm)  Gives the total length of the list. |
| 3 | [**max(list)**](https://www.tutorialspoint.com/python/list_max.htm)  Returns item from the list with max value. |
| 4 | [**min(list)**](https://www.tutorialspoint.com/python/list_min.htm)  Returns item from the list with min value. |
| 5 | [**list(seq)**](https://www.tutorialspoint.com/python/list_list.htm)  Converts a tuple into list. |

Python includes following list methods

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | [**list.append(obj)**](https://www.tutorialspoint.com/python/list_append.htm)  Appends object obj to list |
| 2 | [**list.count(obj)**](https://www.tutorialspoint.com/python/list_count.htm)  Returns count of how many times obj occurs in list |
| 3 | [**list.extend(seq)**](https://www.tutorialspoint.com/python/list_extend.htm)  Appends the contents of seq to list |
| 4 | [**list.index(obj)**](https://www.tutorialspoint.com/python/list_index.htm)  Returns the lowest index in list that obj appears |
| 5 | [**list.insert(index, obj)**](https://www.tutorialspoint.com/python/list_insert.htm)  Inserts object obj into list at offset index |
| 6 | [**list.pop(obj=list[-1])**](https://www.tutorialspoint.com/python/list_pop.htm)  Removes and returns last object or obj from list |
| 7 | [**list.remove(obj)**](https://www.tutorialspoint.com/python/list_remove.htm)  Removes object obj from list |
| 8 | [**list.reverse()**](https://www.tutorialspoint.com/python/list_reverse.htm)  Reverses objects of list in place |
| 9 | [**list.sort([func])**](https://www.tutorialspoint.com/python/list_sort.htm)  Sorts objects of list, use compare func if given |

# Python Tuples

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

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

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

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

tup3 = "a", "b", "c", "d";

The empty tuple is written as two parentheses containing nothing −

tup1 = ();

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

tup1 = (50,);

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

## **Accessing Values in Tuples:**

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

#!/usr/bin/python

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

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

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

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

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

tup1[0]: physics

tup2[1:5]: [2, 3, 4, 5]

## **Updating Tuples**

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

#!/usr/bin/python

tup1 = (12, 34.56);

tup2 = ('abc', 'xyz');

# Following action is not valid for tuples

# tup1[0] = 100;

# So let's create a new tuple as follows

tup3 = tup1 + tup2;

print tup3

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

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

## **Delete Tuple Elements**

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

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

#!/usr/bin/python

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

print tup

del tup;

print "After deleting tup : "

print tup

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

('physics', 'chemistry', 1997, 2000)

After deleting tup :

Traceback (most recent call last):

File "test.py", line 9, in <module>

print tup;

NameError: name 'tup' is not defined

## **Basic Tuples Operations**

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

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

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

## **Indexing, Slicing, and Matrixes**

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

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

|  |  |  |
| --- | --- | --- |
| **Python Expression** | **Results** | **Description** |
| L[2] | 'SPAM!' | Offsets start at zero |
| L[-2] | 'Spam' | Negative: count from the right |
| L[1:] | ['Spam', 'SPAM!'] | Slicing fetches sections |

## **No Enclosing Delimiters**

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

#!/usr/bin/python

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

x, y = 1, 2;

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

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

abc -4.24e+93 (18+6.6j) xyz

Value of x , y : 1 2

## **Built-in Tuple Functions**

Python includes the following tuple functions −

|  |  |
| --- | --- |
| **SN** | **Function with Description** |
| 1 | [**cmp(tuple1, tuple2)**](https://www.tutorialspoint.com/python/tuple_cmp.htm)  Compares elements of both tuples. |
| 2 | [**len(tuple)**](https://www.tutorialspoint.com/python/tuple_len.htm)  Gives the total length of the tuple. |
| 3 | [**max(tuple)**](https://www.tutorialspoint.com/python/tuple_max.htm)  Returns item from the tuple with max value. |
| 4 | [**min(tuple)**](https://www.tutorialspoint.com/python/tuple_min.htm)  Returns item from the tuple with min value. |
| 5 | [**tuple(seq)**](https://www.tutorialspoint.com/python/tuple_tuple.htm)  Converts a list into tuple. |

# Python Dictionary

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

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

## **Accessing Values in Dictionary:**

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

#!/usr/bin/python

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

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

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

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

dict['Name']: Zara

dict['Age']: 7

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

#!/usr/bin/python

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

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

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

dict['Alice']:

Traceback (most recent call last):

File "test.py", line 4, in <module>

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

KeyError: 'Alice'

## **Updating Dictionary**

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

#!/usr/bin/python

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

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

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

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

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

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

dict['Age']: 8

dict['School']: DPS School

## **Delete Dictionary Elements**

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

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

#!/usr/bin/python

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

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

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

del dict ; # delete entire dictionary

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

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

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

dict['Age']:

Traceback (most recent call last):

File "test.py", line 8, in <module>

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

TypeError: 'type' object is unsubscriptable

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

## **Properties of Dictionary Keys**

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

There are two important points to remember about dictionary keys −

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

#!/usr/bin/python

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

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

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

dict['Name']: Manni

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

#!/usr/bin/python

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

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

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

Traceback (most recent call last):

File "test.py", line 3, in <module>

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

TypeError: list objects are unhashable

## **Built-in Dictionary Functions & Methods −**

Python includes the following dictionary functions −

|  |  |
| --- | --- |
| **SN** | **Function with Description** |
| 1 | [**cmp(dict1, dict2)**](https://www.tutorialspoint.com/python/dictionary_cmp.htm)  Compares elements of both dict. |
| 2 | [**len(dict)**](https://www.tutorialspoint.com/python/dictionary_len.htm)  Gives the total length of the dictionary. This would be equal to the number of items in the dictionary. |
| 3 | [**str(dict)**](https://www.tutorialspoint.com/python/dictionary_str.htm)  Produces a printable string representation of a dictionary |
| 4 | [**type(variable)**](https://www.tutorialspoint.com/python/dictionary_type.htm)  Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type. |

Python includes following dictionary methods −

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | [**dict.clear()**](https://www.tutorialspoint.com/python/dictionary_clear.htm)  Removes all elements of dictionary *dict* |
| 2 | [**dict.copy()**](https://www.tutorialspoint.com/python/dictionary_copy.htm)  Returns a shallow copy of dictionary *dict* |
| 3 | [**dict.fromkeys()**](https://www.tutorialspoint.com/python/dictionary_fromkeys.htm)  Create a new dictionary with keys from seq and values *set* to *value*. |
| 4 | [**dict.get(key, default=None)**](https://www.tutorialspoint.com/python/dictionary_get.htm)  For *key* key, returns value or default if key not in dictionary |
| 5 | [**dict.has\_key(key)**](https://www.tutorialspoint.com/python/dictionary_has_key.htm)  Returns *true* if key in dictionary *dict*, *false* otherwise |
| 6 | [**dict.items()**](https://www.tutorialspoint.com/python/dictionary_items.htm)  Returns a list of *dict*'s (key, value) tuple pairs |
| 7 | [**dict.keys()**](https://www.tutorialspoint.com/python/dictionary_keys.htm)  Returns list of dictionary dict's keys |
| 8 | [**dict.setdefault(key, default=None)**](https://www.tutorialspoint.com/python/dictionary_setdefault.htm)  Similar to get(), but will set dict[key]=default if *key* is not already in dict |
| 9 | [**dict.update(dict2)**](https://www.tutorialspoint.com/python/dictionary_update.htm)  Adds dictionary *dict2*'s key-values pairs to *dict* |
| 10 | [**dict.values()**](https://www.tutorialspoint.com/python/dictionary_values.htm)  Returns list of dictionary *dict*'s values |

# Python Date & Time

A Python program can handle date and time in several ways. Converting between date formats is a common chore for computers. Python's time and calendar modules help track dates and times.

## **What is Tick?**

Time intervals are floating-point numbers in units of seconds. Particular instants in time are expressed in seconds since 12:00am, January 1, 1970.

There is a popular **time** module available in Python which provides functions for working with times, and for converting between representations. The function *time.time()* returns the current system time in ticks since 12:00am, January 1, 1970(epoch).

## **Example**

#!/usr/bin/python

import time; # This is required to include time module.

ticks = time.time()

print("Number of ticks since 12:00am, January 1, 1970:", ticks);

This would produce a result something as follows −

Number of ticks since 12:00am, January 1, 1970: 1507888743.720282

Date arithmetic is easy to do with ticks. However, dates before the epoch cannot be represented in this form. Dates in the far future also cannot be represented this way - the cutoff point is sometime in 2038 for UNIX and Windows.

## **What is TimeTuple?**

Many of Python's time functions handle time as a tuple of 9 numbers, as shown below:

|  |  |  |
| --- | --- | --- |
| **Index** | **Field** | **Values** |
| 0 | 4-digit year | 2008 |
| 1 | Month | 1 to 12 |
| 2 | Day | 1 to 31 |
| 3 | Hour | 0 to 23 |
| 4 | Minute | 0 to 59 |
| 5 | Second | 0 to 61 (60 or 61 are leap-seconds) |
| 6 | Day of Week | 0 to 6 (0 is Monday) |
| 7 | Day of year | 1 to 366 (Julian day) |
| 8 | Daylight savings | -1, 0, 1, -1 means library determines DST |

The above tuple is equivalent to **struct\_time** structure. This structure has following attributes:

|  |  |  |
| --- | --- | --- |
| **Index** | **Attributes** | **Values** |
| 0 | tm\_year | 2008 |
| 1 | tm\_mon | 1 to 12 |
| 2 | tm\_mday | 1 to 31 |
| 3 | tm\_hour | 0 to 23 |
| 4 | tm\_min | 0 to 59 |
| 5 | tm\_sec | 0 to 61 (60 or 61 are leap-seconds) |
| 6 | tm\_wday | 0 to 6 (0 is Monday) |
| 7 | tm\_yday | 1 to 366 (Julian day) |
| 8 | tm\_isdst | -1, 0, 1, -1 means library determines DST |

## **Getting current time**

To translate a time instant from a *seconds since the epoch* floating-point value into a time-tuple, pass the floating-point value to a function (e.g., localtime) that returns a time-tuple with all nine items valid.

#!/usr/bin/python

import time;

localtime = time.localtime(time.time())

print("Local current time :", localtime);

This would produce the following result, which could be formatted in any other presentable form −

Local current time : time.struct\_time(tm\_year=2017, tm\_mon=10, tm\_mday=13, tm\_hour=16, tm\_min=2, tm\_sec=8, tm\_wday=4, tm\_yday=286, tm\_isdst=0)

**Getting formatted time**

You can format any time as per your requirement, but simple method to get time in readable format is **asctime()** :

#!/usr/bin/python

import time;

localtime = time.asctime( time.localtime(time.time()) )

print("Local current time :", localtime)

This would produce the following result −

Local current time : Fri Oct 13 16:10:30 2017

## **Getting calendar for a month**

The calendar module gives a wide range of methods to play with yearly and monthly calendars. Here, we print a calendar for a given month ( Jan 2017 )

#!/usr/bin/python

import calendar

cal = calendar.month(2017, 1)

print(cal)

This would produce the following result −

January 2017

Mo Tu We Th Fr Sa Su

1

2 3 4 5 6 7 8

9 10 11 12 13 14 15

16 17 18 19 20 21 22

23 24 25 26 27 28 29

30 31

## **The *time* Module**

There is a popular **time** module available in Python which provides functions for working with times and for converting between representations. Here is the list of all available methods:

|  |  |
| --- | --- |
| **SN** | **Function with Description** |
| 1 | **time.altzone**  The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if daylight is nonzero.  print("time.altzone %d " % time.altzone)  time.altzone -23400 |
| 2 | **time.asctime([tupletime])**  Accepts a time-tuple and returns as character readable 24-character string such as 'Tue Dec 11 18:07:14 2008'.  print(time.gmtime())  time.struct\_time(tm\_year=2017, tm\_mon=10, tm\_mday=13, tm\_hour=10, tm\_min=54, tm\_sec=4, tm\_wday=4, tm\_yday=286, tm\_isdst=0)  # Output in UTC (Coordinated Universal Time (formerly known as Greenwich Mean Time, or GMT).  time.asctime(time.gmtime())  'Fri Oct 13 10:56:04 2017'  time.asctime(time.localtime())  'Fri Oct 13 16:26:18 2017' |
| 3 | **time.clock( )**  Returns the current CPU time as a floating-point number of seconds. To measure computational costs of different approaches, the value of time.clock is more useful than that of time.time(). On Unix and in Windows it returns wall-clock seconds elapsed since the first call to this function, as a floating point number.  time.clock( )  2.1935462828870445  time.clock( )  4.097568009977395  time.clock( )  5.349900923483391 |
| 4 | **time.ctime([secs])**  Like asctime(localtime(secs)) and without arguments is like asctime( )  time.asctime(time.gmtime())  'Fri Oct 13 11:25:41 2017'  time.asctime(time.localtime())  'Fri Oct 13 16:57:43 2017'  time.ctime()  'Fri Oct 13 16:57:52 2017' |
| 5 | **time.gmtime([secs])**  Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the UTC time. Note : t.tm\_isdst is always 0  time.gmtime()  time.struct\_time(tm\_year=2017, tm\_mon=10, tm\_mday=13, tm\_hour=11, tm\_min=31, tm\_sec=33, tm\_wday=4, tm\_yday=286, tm\_isdst=0) |
| 6 | **time.localtime([secs])**  Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the local time (t.tm\_isdst is 0 or 1, depending on whether DST applies to instant secs by local rules).  time.localtime()  time.struct\_time(tm\_year=2017, tm\_mon=10, tm\_mday=13, tm\_hour=17, tm\_min=3, tm\_sec=35, tm\_wday=4, tm\_yday=286, tm\_isdst=0) |
| 7 | **time.mktime(tupletime)**  The method **mktime()** is the inverse function of localtime(). Its argument is the struct\_time or full 9-tuple and it returns a floating point number, for compatibility with time().  If the input value cannot be represented as a valid time, either *OverflowError*or *ValueError* will be raised.  time.localtime()  time.struct\_time(tm\_year=2017, tm\_mon=10, tm\_mday=13, tm\_hour=17, tm\_min=18, tm\_sec=50, tm\_wday=4, tm\_yday=286, tm\_isdst=0)  t =(2017,10,13,17,18,50,4,286,0);  time.mktime(t);  1507895330.0  time.mktime(time.localtime())  1507895862.0 |
| 8 | **time.sleep(secs)**  Suspends the calling thread for secs seconds.  print("Start : %s" % time.ctime())  Start : Fri Oct 13 18:46:57 2017  time.sleep( 5 )  print("End : %s" % time.ctime())  End : Fri Oct 13 18:47:02 2017 |
| 9 | **time.strftime(fmt[,tupletime])**  Accepts an instant expressed as a time-tuple in local time and returns a string representing the instant as specified by string fmt.  Syntax  Following is the syntax for strftime() method:  time.strftime(format[, t])  Format elements   * %a - abbreviated weekday name * %A - full weekday name * %b - abbreviated month name * %B - full month name * %c - preferred date and time representation * %C - century number (the year divided by 100, range 00 to 99) * %d - day of the month (01 to 31) * %D - same as %m/%d/%y * %e - day of the month (1 to 31) * %g - like %G, but without the century * %G - 4-digit year corresponding to the ISO week number (see %V). * %h - same as %b * %H - hour, using a 24-hour clock (00 to 23) * %I - hour, using a 12-hour clock (01 to 12) * %j - day of the year (001 to 366) * %m - month (01 to 12) * %M - minute * %n - newline character * %p - either am or pm according to the given time value * %r - time in a.m. and p.m. notation * %R - time in 24 hour notation * %S - second * %t - tab character * %T - current time, equal to %H:%M:%S * %u - weekday as a number (1 to 7), Monday=1. Warning: In Sun Solaris Sunday=1 * %U - week number of the current year, starting with the first Sunday as the first day of the first week * %V - The ISO 8601 week number of the current year (01 to 53), where week 1 is the first week that has at least 4 days in the current year, and with Monday as the first day of the week * %W - week number of the current year, starting with the first Monday as the first day of the first week * %w - day of the week as a decimal, Sunday=0 * %x - preferred date representation without the time * %X - preferred time representation without the date * %y - year without a century (range 00 to 99) * %Y - year including the century * %Z or %z - time zone or name or abbreviation   t =(2017,10,13,17,18,50,4,286,0);  t = time.mktime(t)  print(time.strftime("%b %d %Y %H:%M:%S", time.gmtime(t)))  Oct 13 2017 11:48:50 |
| 10 | [**time.strptime(str,fmt='%a %b %d %H:%M:%S %Y')**](https://www.tutorialspoint.com/python/time_strptime.htm)  Parses str according to format string fmt and returns the instant in time-tuple format. The method strptime() parses a string representing a time according to a format. The return value is a struct\_time as returned by gmtime() or localtime().  The format parameter uses the same directives as those used by strftime(); it defaults to "%a %b %d %H:%M:%S %Y" which matches the formatting returned by ctime().  If string cannot be parsed according to format, or if it has excess data after parsing, ValueError is raised.  **Syntax**  Following is the syntax for strptime() method:  **time.strptime(string[, format]**  t = time.strptime("30 Nov 2017 16 30 45", "%d %b %Y %H %M %S");  print(t)  time.struct\_time(tm\_year=2017, tm\_mon=11, tm\_mday=30, tm\_hour=16, tm\_min=30, tm\_sec=45, tm\_wday=3, tm\_yday=334, tm\_isdst=-1) |
| 11 | **time.time( )**  Returns the current time instant, a floating-point number of seconds since the epoch.  time.time( )  1507904917.6235135 |
| 12 | **time.tzset()**  Resets the time conversion rules used by the library routines. The environment variable TZ specifies how this is done.  Availability: Unix. |

Let us go through the functions briefly −

There are following two important attributes available with time module:

|  |  |
| --- | --- |
| **SN** | **Attribute with Description** |
| 1 | **time.timezone**  Attribute time.timezone is the offset in seconds of the local time zone (without DST) from UTC (>0 in the Americas; <=0 in most of Europe, Asia, Africa).  time.timezone  -19800 |
| 2 | **time.tzname**  Attribute time.tzname is a pair of locale-dependent strings, which are the names of the local time zone without and with DST, respectively.  time.tzname  ('India Standard Time', 'India Daylight Time') |

## **The *calendar* Module**

The calendar module supplies calendar-related functions, including functions to print a text calendar for a given month or year.

By default, calendar takes Monday as the first day of the week and Sunday as the last one. To change this, call **calendar.setfirstweekday()** function.

Here is a list of functions available with the *calendar* module:

|  |  |
| --- | --- |
| **SN** | **Function with Description** |
| 1 | **calendar.calendar(year,w=2,l=1,c=6)**  Returns a multiline string with a calendar for year year formatted into three columns separated by c spaces. w is the width in characters of each date; each line has length 21\*w+18+2\*c. l is the number of lines for each week.  print(calendar.calendar(2017,w=1,l=1,c=1));  2017  January February March  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 2 3 4 5 6 7 1 2 3 4 1 2 3 4  8 9 10 11 12 13 14 5 6 7 8 9 10 11 5 6 7 8 9 10 11  15 16 17 18 19 20 21 12 13 14 15 16 17 18 12 13 14 15 16 17 18  22 23 24 25 26 27 28 19 20 21 22 23 24 25 19 20 21 22 23 24 25  29 30 31 26 27 28 26 27 28 29 30 31  April May June  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 1 2 3 4 5 6 1 2 3  2 3 4 5 6 7 8 7 8 9 10 11 12 13 4 5 6 7 8 9 10  9 10 11 12 13 14 15 14 15 16 17 18 19 20 11 12 13 14 15 16 17  16 17 18 19 20 21 22 21 22 23 24 25 26 27 18 19 20 21 22 23 24  23 24 25 26 27 28 29 28 29 30 31 25 26 27 28 29 30  30  July August September  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 1 2 3 4 5 1 2  2 3 4 5 6 7 8 6 7 8 9 10 11 12 3 4 5 6 7 8 9  9 10 11 12 13 14 15 13 14 15 16 17 18 19 10 11 12 13 14 15 16  16 17 18 19 20 21 22 20 21 22 23 24 25 26 17 18 19 20 21 22 23  23 24 25 26 27 28 29 27 28 29 30 31 24 25 26 27 28 29 30  30 31  October November December  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 2 3 4 5 6 7 1 2 3 4 1 2  8 9 10 11 12 13 14 5 6 7 8 9 10 11 3 4 5 6 7 8 9  15 16 17 18 19 20 21 12 13 14 15 16 17 18 10 11 12 13 14 15 16  22 23 24 25 26 27 28 19 20 21 22 23 24 25 17 18 19 20 21 22 23  29 30 31 26 27 28 29 30 24 25 26 27 28 29 30  31 |
| 2 | **calendar.firstweekday( )**  Returns the current setting for the weekday that starts each week. By default, when calendar is first imported, this is 0, meaning Monday.  calendar.firstweekday( )  0 |
| 3 | **calendar.isleap(year)**  Returns True if year is a leap year; otherwise, False.  calendar.isleap(2016)  True  calendar.isleap(2017)  False |
| 4 | **calendar.leapdays(y1,y2)**  Returns the total number of leap days in the years within range(y1,y2).  calendar.leapdays(2000,2017)  5 |
| 5 | **calendar.month(year,month,w=2,l=1)**  Returns a multiline string with a calendar for month month of year year, one line per week plus two header lines. w is the width in characters of each date; each line has length 7\*w+6. l is the number of lines for each week.  print(calendar.month(2017,10,w=2,l=1))  October 2017  Su Mo Tu We Th Fr Sa  1 2 3 4 5 6 7  8 9 10 11 12 13 14  15 16 17 18 19 20 21  22 23 24 25 26 27 28  29 30 31 |
| 6 | **calendar.monthcalendar(year,month)**  Returns a list of lists of ints. Each sublist denotes a week. Days outside month month of year year are set to 0; days within the month are set to their day-of-month, 1 and up.  print(calendar.monthcalendar(2017,10))  [[1, 2, 3, 4, 5, 6, 7], [8, 9, 10, 11, 12, 13, 14], [15, 16, 17, 18, 19, 20, 21], [22, 23, 24, 25, 26, 27, 28], [29, 30, 31, 0, 0, 0, 0]] |
| 7 | **calendar.monthrange(year,month)**  Returns two integers. The first one is the code of the weekday for the first day of the month month in year; the second one is the number of days in the month. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 to 12.  >>> print(calendar.calendar(2017,w=1,l=1,c=1));  2017  January February March  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 2 3 4 5 6 7 1 2 3 4 1 2 3 4  8 9 10 11 12 13 14 5 6 7 8 9 10 11 5 6 7 8 9 10 11  15 16 17 18 19 20 21 12 13 14 15 16 17 18 12 13 14 15 16 17 18  22 23 24 25 26 27 28 19 20 21 22 23 24 25 19 20 21 22 23 24 25  29 30 31 26 27 28 26 27 28 29 30 31  April May June  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 1 2 3 4 5 6 1 2 3  2 3 4 5 6 7 8 7 8 9 10 11 12 13 4 5 6 7 8 9 10  9 10 11 12 13 14 15 14 15 16 17 18 19 20 11 12 13 14 15 16 17  16 17 18 19 20 21 22 21 22 23 24 25 26 27 18 19 20 21 22 23 24  23 24 25 26 27 28 29 28 29 30 31 25 26 27 28 29 30  30  calendar.monthrange(2017,1)  (6, 31)  calendar.monthrange(2017,2)  (2, 28)  calendar.monthrange(2017,3)  (2, 31)  calendar.monthrange(2017,4)  (5, 30)  calendar.monthrange(2017,5)  (0, 31)  calendar.monthrange(2017,6)  (3, 30) |
| 8 | **calendar.prcal(year,w=2,l=1,c=6)**  Like print(calendar.calendar(year,w,l,c));  calendar.prcal(2017,w=2,l=1,c=2)  2017  January February March  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 2 3 4 5 6 7 1 2 3 4 1 2 3 4  8 9 10 11 12 13 14 5 6 7 8 9 10 11 5 6 7 8 9 10 11  15 16 17 18 19 20 21 12 13 14 15 16 17 18 12 13 14 15 16 17 18  22 23 24 25 26 27 28 19 20 21 22 23 24 25 19 20 21 22 23 24 25  29 30 31 26 27 28 26 27 28 29 30 31  April May June  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 1 2 3 4 5 6 1 2 3  2 3 4 5 6 7 8 7 8 9 10 11 12 13 4 5 6 7 8 9 10  9 10 11 12 13 14 15 14 15 16 17 18 19 20 11 12 13 14 15 16 17  16 17 18 19 20 21 22 21 22 23 24 25 26 27 18 19 20 21 22 23 24  23 24 25 26 27 28 29 28 29 30 31 25 26 27 28 29 30  30  July August September  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 1 2 3 4 5 1 2  2 3 4 5 6 7 8 6 7 8 9 10 11 12 3 4 5 6 7 8 9  9 10 11 12 13 14 15 13 14 15 16 17 18 19 10 11 12 13 14 15 16  16 17 18 19 20 21 22 20 21 22 23 24 25 26 17 18 19 20 21 22 23  23 24 25 26 27 28 29 27 28 29 30 31 24 25 26 27 28 29 30  30 31  October November December  Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  1 2 3 4 5 6 7 1 2 3 4 1 2  8 9 10 11 12 13 14 5 6 7 8 9 10 11 3 4 5 6 7 8 9  15 16 17 18 19 20 21 12 13 14 15 16 17 18 10 11 12 13 14 15 16  22 23 24 25 26 27 28 19 20 21 22 23 24 25 17 18 19 20 21 22 23  29 30 31 26 27 28 29 30 24 25 26 27 28 29 30  31 |
| 9 | **calendar.prmonth(year,month,w=2,l=1)**  Like print(calendar.month(year,month,w=1,l=1));  print(calendar.month(2017,1,w=1,l=1));  January 2017  Su Mo Tu We Th Fr Sa  1 2 3 4 5 6 7  8 9 10 11 12 13 14  15 16 17 18 19 20 21  22 23 24 25 26 27 28  29 30 31  calendar.prmonth(2017,1,w=2,l=1)  January 2017  Su Mo Tu We Th Fr Sa  1 2 3 4 5 6 7  8 9 10 11 12 13 14  15 16 17 18 19 20 21  22 23 24 25 26 27 28  29 30 31 |
| 10 | **calendar.setfirstweekday(weekday)**  Sets the first day of each week to weekday code weekday. Weekday codes are 0 (Monday) to 6 (Sunday).  import calendar  calendar.firstweekday( )  0  calendar.setfirstweekday(calendar.SUNDAY);  calendar.firstweekday( )  6 |
| 11 | **calendar.timegm(tupletime)**  The inverse of time.gmtime: accepts a time instant in time-tuple form and returns the same instant as a floating-point number of seconds since the epoch.  import calendar  import time  import datetime  d = datetime.datetime(2017, 10, 31)  calendar.timegm(d.timetuple())  1509408000  time.mktime(d.timetuple())  1509388200.0  d = datetime.datetime(2017, 10, 31, 16, 30, 30)  calendar.timegm(d.timetuple())  1509467430  time.mktime(d.timetuple())  1509447630.0 |
| 12 | **calendar.weekday(year,month,day)**  Returns the weekday code for the given date. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 (January) to 12 (December).  print(calendar.month(2017,10,w=2,l=1));  October 2017  Su Mo Tu We Th Fr Sa  1 2 3 4 5 6 7  8 9 10 11 12 13 14  15 16 17 18 19 20 21  22 23 24 25 26 27 28  29 30 31  calendar.weekday(2017,10,3)  1 |

## **Other Modules & Functions:**

If you are interested, then here you would find a list of other important modules and functions to play with date & time in Python:

* [The *datetime* Module](http://docs.python.org/library/datetime.html#module-datetime)
* [The *pytz*Module](http://www.twinsun.com/tz/tz-link.htm)
* [The *dateutil* Module](http://labix.org/python-dateutil)

# Python Functions

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

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

## **Defining a Function**

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

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

## **Syntax**

def function\_name( parameter1, parameter2, parameter3 ):

"function\_docstring"

function\_suite

return [expression]

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

## **Example**

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

def print\_myname( str ):

"This prints a passed string into this function"

print(str)

return

## **Calling a Function**

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

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

#!/usr/bin/python

# Function definition is here

def print\_myname( str ):

"This prints a passed string into this function"

print(str)

return;

# Now you can call printme function

print\_myname("Hari Yadav")

Hari Yadav

## **Pass by reference vs value**

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

#!/usr/bin/python

# Function definition is here

def change\_mylist( mylist ):

"This changes a passed list into this function"

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

print("Values inside the function: ", mylist)

return

# Now you can call change\_mylist function by using a list variable mylist as reference

mylist = [10,20,30];

print("Values before passing in the function: ", mylist)

change\_mylist( mylist );

print("Values after passing in the function: ", mylist)

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

Values before passing in the function: [10, 20, 30]

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

Values after passing in the function: [10, 20, 30, [1, 2, 3, 4]]

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

#!/usr/bin/python

# Function definition is here

def change\_mylist1( mylist ):

"This changes a passed list into this function"

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

print("Values inside the function: ", mylist)

print("Object ID of mylist inside function: ", id(mylist))

return

# Now you can call change\_mylist1 function

mylist = [10,20,30];

print("Object ID of mylist outside function: ", id(mylist))

change\_mylist1( mylist );

print("Values outside the function: ", mylist)

The parameter *mylist* is local to the function change\_mylist1. Changing mylist within the function does not affect *mylist*. The function accomplishes nothing and finally this would produce the following result:

Object ID of mylist outside function: 2301171593544

Values inside the function: [1, 2, 3, 4]

Object ID of mylist inside function: 2301171594056

Values outside the function: [10, 20, 30]

## **Function Arguments**

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

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

## **Required arguments**

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

To call the function *print\_myname()*, you definitely need to pass one argument, otherwise it gives a syntax error as follows –

#!/usr/bin/python

# Function definition is here

def print\_myname( str ):

"This prints a passed string into this function"

print(str);

return;

# Now you can call print\_myname function

print\_myname()

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

Traceback (most recent call last):

File "<stdin>", line 2, in <module>

TypeError: print\_myname() missing 1 required positional argument: 'str'

## **Keyword arguments (parameter named notation)**

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

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

You can also make keyword calls to the *print\_myname()* function in the following ways −

#!/usr/bin/python

# Function definition is here

def print\_myname( str ):

"This prints a passed string into this function"

print(str)

return;

# Now you can call print\_myname function

print\_myname( str = "Michael Gerard Tyson")

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

Michael Gerard Tyson

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

#!/usr/bin/python

# Function definition is here

def personal\_info( name, age ):

"This prints a passed info into this function"

print("Name: ", name)

print("Age ", age)

return;

# Now you can call personal\_info function

personal\_info( age=50, name="Michael Gerard Tyson")

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

Name: Michael Gerard Tyson

Age 50

## **Default arguments**

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

#!/usr/bin/python

# Function definition is here

def personal\_info( name, age, design = "Trainer" ):

"This prints a passed info into this function"

print("Name: ", name)

print("Age ", age)

print("Designation ", design)

return;

# Now you can call personal\_info function

personal\_info( age=50, name="Michael Gerard Tyson" )

personal\_info( age=50, name="Michael Gerard Tyson" , design='Chairman')

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

Name: Michael Gerard Tyson

Age 50

Designation Trainer

Name: Michael Gerard Tyson

Age 50

Designation Chairman

## **Variable-length arguments**

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

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

def function\_name([formal\_args,] \*var\_args\_tuple ):

"function\_docstring"

function\_suite

return [expression]

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

#!/usr/bin/python

# Function definition is here

def student\_marks( name, **\***score\_tuple ):

"This prints a variable passed arguments"

print("Student subject details is: ")

print(name)

for var in score\_tuple:

print(var)

return;

# Now you can call student\_marks function

student\_marks("Dev Singh")

student\_marks ("Sami", 70, 60, 50 )

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

Student subject details is:

Dev Singh

Student subject details is:

Sami

70

60

50

def student\_marks( name, \*score\_tuple, \*class ):

"This prints a variable passed arguments"

print("Student subject details is: ")

print(name, class)

for var in score\_tuple:

print(var)

return;

**NOTE: Only one variable-length arguments can be specified.**

def student\_marks(\*score\_tuple, name):

"This prints a variable passed arguments"

print("Student subject details is: ")

print(name)

for var in score\_tuple:

print(var)

return;

print("Good Bye. . . . . ")

student\_marks("Dev Singh")

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: student\_marks() missing 1 required keyword-only argument: 'name'

student\_marks(name="Dev Singh")

Student subject details is:

Dev Singh

**NOTE: Any statement written after return keyword is not executed.**

student\_marks (70, 60, 50, "Sami" )

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: student\_marks() missing 1 required keyword-only argument: 'name'

student\_marks (70, 60, 50, name="Sami" )

Student subject details is:

Sami

70

60

50

## **The *Anonymous* Functions**

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

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

## **Syntax**

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

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

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

#!/usr/bin/python

# Function definition is here

sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function

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

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

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

Value of total : 30

Value of total : 40

## **The *return* Statement**

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None. Any statement written after return keyword is not executed.

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

#!/usr/bin/python

# Function definition is here

def sum( arg1, arg2 ):

# Add both the parameters and return them."

total = arg1 + arg2

print("Total Inside the function : ", total)

return total;

# Now you can call sum function

v\_total = sum( 10, 20 );

print("Total Outside the function : ", v\_total);

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

Total Inside the function : 30

Total Outside the function : 30

## **Scope of Variables**

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

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

* Global variables
* Local variables

## **Global vs. Local variables**

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

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

#!/usr/bin/python

total = 0; # This is global variable.

# Function definition is here

def sum( arg1, arg2 ):

# Add both the parameters and return them."

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

print("Inside the function local total : ", total)

return total;

# Now you can call sum function

sum( 10, 20 );

print("Outside the function global total : ", total)

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

Inside the function local total : 30

30

Outside the function global total : 0

# Python Modules

Modules refer to a file containing Python statements and definitions like function and class definitions.. A module allows you to logically organize your Python code.

We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code.

We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

Simply, a module is a file consisting of Python code. A module can define functions, classes and variables.

## **Example**

Let’s create a module. Type the following and save it as **example.py**.

# Python Module example

def add(a, b):

"""This program adds two

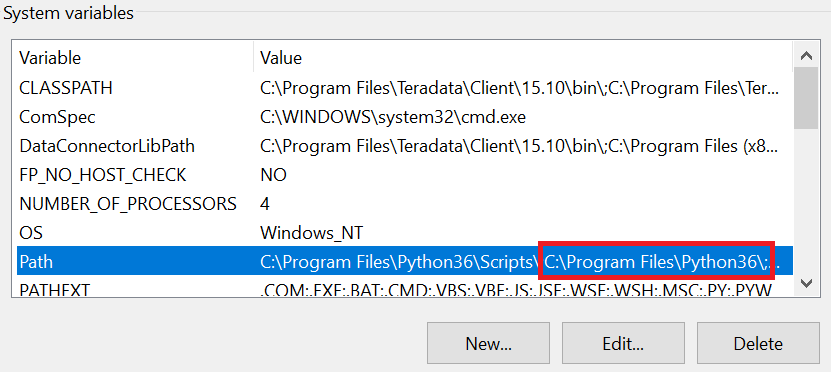
numbers and return the result"""

result = a + b

return result

Here, we have defined a [function](https://www.programiz.com/python-programming/function) **add()** inside a module named example. The function takes in two numbers and returns their sum.

I have copied this python module file **example.py** to **C:\Program Files\Python36** directory because this is the environment variable path.



## How to import modules in Python?

We can import the definitions of a module on the interactive prompt of Python or inside another module.

We use the import keyword to do this. To import our previously defined module example we type the following in the Python prompt.

C:\Users\haryadav>python

Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32

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

Welcome to Python!

Hari Yadav

>>> import example

>>> example.add(50,45.5);

95.5

## Import with renaming

We can import a module by renaming it as follows.

>>> import example as ex

>>> ex.add(50,45.5);

95.5

>>> example.add(50,45.5);

95.5

Using the module name we can access the function using dot (.) operation. A module is loaded only once, regardless of the number of times it is imported.

## Python **from...import** statement

We can import specific function from a module without importing the module as a whole. Here we will add another function in **example.py** file. In such case we don't use the dot operator.

def sub(a, b):

"""This program subtract two

numbers and return the result"""

result = a - b

return result

C:\Users\haryadav>python

Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32

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

Welcome to Python!

Hari Yadav

>>> from example import sub

>>> sub(50,45.5);

4.5

>>> add(50,45.5);

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'add' is not defined

>>> from example import add

>>> add(50,45.5);

95.5

## The **from...import \*** Statement:

It is also possible to import all names from a module into the current namespace by using the following import statement.

**Add this private function in the module example**

def \_income\_tax(income, rate): # private method

"""This program calculate tax and return the result"""

result = (income\*rate)/100

print("Your tax is Rs : ")

return result

C:\Users\haryadav>python

Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32

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

Welcome to Python!

Hari Yadav

>>> import example

>>> example.add(50,45.5);

95.5

>>> example.sub(50,45.5);

4.5

>>> example.\_income\_tax(400000,10)

Your tax is Rs :

40000.0

>>> exit()

This import all object definition except those beginnig with an underscore (private objects), visible in our scope. Objects are access without dot operator (.)

C:\Users\haryadav>python

Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32

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

Welcome to Python!

Hari Yadav

**>>> from example import \***

>>> example.add(50,45.5);

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'example' is not defined

>>> add(50,45.5);

95.5

>>> sub(50,45.5);

4.5

>>> \_income\_tax(400000,10)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

**NameError: name '\_income\_tax' is not defined**

Importing everything with the asterisk (\*) symbol is not a good programming practice. This can lead to duplicate definitions for an identifier.

## **Locating Modules**

When you import a module, the Python interpreter searches for the module in the following sequences −

* The current directory.
* If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
* If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.

The module search path is stored in the system module **sys** as the **sys.path** variable. The **sys.path** variable contains the current directory, PYTHONPATH, and the installation-dependent default.

>>> import sys

>>> sys.path

['', 'C:\\Program Files\\Python36\\python36.zip', 'C:\\Program Files\\Python36\\DLLs', 'C:\\Program Files\\Python36\\lib', 'C:\\Program Files\\Python36', 'C:\\Program Files\\Python36\\lib\\site-packages']

C:\Users\haryadav>echo %PATH%

C:\Program Files\Python36\Scripts\;C:\Program Files\Python36\;C:\Program Files\Teradata\Client\15.10\bin;C:\Program Files (x86)\Teradata\Client\15.10\bin;C:\app\haryadav\product\11.2.0\dbhome\_1\bin;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WINDOWS\System32\WindowsPowerShell\v1.0\;C:\Program Files (x86)\QuickTime\QTSystem\;C:\Program Files\CapgeminiScripts\Support Tools\;C:\Program Files (x86)\Microsoft SQL Server\100\Tools\Binn\;C:\Program Files\Microsoft SQL Server\100\Tools\Binn\;C:\Program Files\Microsoft SQL Server\100\DTS\Binn\;C:\Program Files (x86)\Microsoft SQL Server\100\Tools\Binn\VSShell\Common7\IDE\;C:\Program Files (x86)\Microsoft Visual Studio 9.0\Common7\IDE\PrivateAssemblies\;C:\Program Files (x86)\Microsoft SQL Server\100\DTS\Binn\;C:\Program Files\Microsoft SQL Server\110\DTS\Binn\;C:\Program Files (x86)\Microsoft SQL Server\110\Tools\Binn\;C:\Program Files\Microsoft SQL Server\110\Tools\Binn\;C:\Program Files (x86)\Microsoft SQL Server\110\Tools\Binn\ManagementStudio\;C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\PrivateAssemblies\;C:\Program Files (x86)\Microsoft SQL Server\110\DTS\Binn\;C:\Users\haryadav\AppData\Local\Microsoft\WindowsApps;

import sys

sys.builtin\_module\_names

('\_ast', '\_bisect', '\_blake2', '\_codecs', '\_codecs\_cn', '\_codecs\_hk', '\_codecs\_iso2022', '\_codecs\_jp', '\_codecs\_kr', '\_codecs\_tw', '\_collections', '\_csv', '\_datetime', '\_functools', '\_heapq', '\_imp', '\_io', '\_json', '\_locale', '\_lsprof', '\_md5', '\_multibytecodec', '\_opcode', '\_operator', '\_pickle', '\_random', '\_sha1', '\_sha256', '\_sha3', '\_sha512', '\_signal', '\_sre', '\_stat', '\_string', '\_struct', '\_symtable', '\_thread', '\_tracemalloc', '\_warnings', '\_weakref', '\_winapi', 'array', 'atexit', 'audioop', 'binascii', 'builtins', 'cmath', 'errno', 'faulthandler', 'gc', 'itertools', 'marshal', 'math', 'mmap', 'msvcrt', 'nt', 'parser', 'sys', 'time', 'winreg', 'xxsubtype', 'zipimport', 'zlib')

## **The *PYTHONPATH* Variable:**

The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH.

Here is a typical PYTHONPATH from a Windows system:

C:\Users\haryadav> set PYTHONPATH=c:\python20\lib

C:\Users\haryadav> set PYTHONPATH

PYTHONPATH=c:\python20\lib

C:\Users\haryadav> echo %PYTHONPATH%

c:\python20\lib

# To unset the environment variable

C:\Users\haryadav> set PYTHONPATH=

C:\Users\haryadav> echo %PYTHONPATH%

%PYTHONPATH%

C:\Users\haryadav> set PYTHONPATH

Environment variable PYTHONPATH not defined

And here is a typical PYTHONPATH from a UNIX system:

set PYTHONPATH=/usr/local/lib/python

## **Namespaces and Scoping**

Variables are names (identifiers) that map to objects. A *namespace* is a dictionary of variable names (keys) and their corresponding objects (values).

A Python statement can access variables in a *local namespace* and in the *global namespace*. If a local and a global variable have the same name, the local variable shadows the global variable.

Each function has its own local namespace.

Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local.

Therefore, in order to assign a value to a global variable within a function, you must use the global statement to reference the variable defined outside the function.

The statement ***global VarName*** tells Python that **VarName** is a global variable. Python stops searching the local namespace for the variable.

For example, we define a variable *Money* in the global namespace. Within the function Add*Money*, we assign *Money* a value, therefore Python assumes *Money* as a local variable.

#!/usr/bin/python

Money = 2000

def AddMoney():

# local variable Money is referenced before assignment. It will throw error.

Money = Money + 1

return Money

print(Money)

2000

AddMoney()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 4, in AddMoney

UnboundLocalError: local variable 'Money' referenced before assignment

def AddMoney():

# local variable Money is assigned:

Money = 5000

Money = Money + 1

return Money

AddMoney()

5001

print(Money)

**2000**

def AddMoney():

# Here global variable Money is referenced using global keyword

# local variable Money is commented

global Money

# Money = 5000

Money = Money + 1

return Money

AddMoney()

2001

print(Money)

2001

## **The dir( ) Function**

The dir() built-in function returns a sorted list of strings containing the names defined by a module.

The list contains the names of all the modules, variables and functions that are defined in a module.

#!/usr/bin/python

Add the following lines in the example module which you defined in previous exercise.

# Global variables defined

g\_result = 0

g\_score = 0

**import example**

**content = dir(example)**

**print(content)**

**['\_\_builtins\_\_', '\_\_cached\_\_', '\_\_doc\_\_', '\_\_file\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', '\_income\_tax', 'add', 'g\_result', 'g\_score', 'sub']**

**type(content)**

**<class 'list'>**

Here, the special string variable *\_\_name\_\_* is the module's name, and *\_\_file\_\_*is the filename from which the module was loaded.

## **The *globals()* and *locals()* Functions**

The *globals()* and *locals()* functions can be used to return the names in the global and local namespaces depending on the location from where they are called.

If locals() is called from within a function, it will return all the names that can be accessed locally from that function.

If globals() is called from within a function, it will return all the names that can be accessed globally from that function.

g\_result = 0

g\_score = 0

def currency():

Rupees = 5000

Usd = 5000

Euro= 5000

Ruble=5000

v\_result = Rupees + Usd + Euro + Ruble

print('Local variables are : ', '\n', locals() )

print('Global variables are : ', '\n', globals())

return v\_result

currency()

Local variables are :

{'v\_result': 20000, 'Ruble': 5000, 'Euro': 5000, 'Usd': 5000, 'Rupees': 5000}

Global variables are :

{'\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None, '\_\_package\_\_': None, '\_\_loader\_\_': <\_frozen\_importlib\_external.SourceFileLoader object at 0x0000020EDB74F0B8>, '\_\_spec\_\_': None, '\_\_annotations\_\_': {}, '\_\_builtins\_\_': <module 'builtins' (built-in)>, '\_\_cached\_\_': None, 'os': <module 'os' from 'C:\\Program Files\\Python36\\lib\\os.py'>, 'cls': <function <lambda> at 0x0000020EDB5C3E18>, 'AddMoney': <function AddMoney at 0x0000020EDB8AD8C8>, 'g\_result': 0, 'g\_score': 0, 'currency': <function currency at 0x0000020EDB8AD950>}

20000

The return type of both these functions is dictionary. Therefore, names can be extracted using the keys() function.

def currency():

Rupees = 5000

Usd = 5000

Euro= 5000

Ruble=5000

v\_result = Rupees + Usd + Euro + Ruble

print('Local variables are : ', '\n', locals() )

dict = locals()

print(dict.keys())

print(dict.values())

return v\_result

currency()

Local variables are :

{'v\_result': 20000, 'Ruble': 5000, 'Euro': 5000, 'Usd': 5000, 'Rupees': 5000}

dict\_keys(['v\_result', 'Ruble', 'Euro', 'Usd', 'Rupees'])

dict\_values([20000, 5000, 5000, 5000, 5000])

20000

## **The *reload()* Function**

When the module is imported into a script, the code in the top-level portion of a module is executed only once.

Therefore, if you want to reexecute the top-level code in a module, you can use the *reload()* function. The reload() function imports a previously imported module again. The syntax of the reload() function is this −

reload(module\_name)

Here, *module\_name* is the name of the module you want to reload and not the string containing the module name. For example, to reload *hello* module, do the following −

reload(hello)

reload(example)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'reload' is not defined

**For >= Python3.4:**

import example

import importlib

importlib.reload(example)

<module 'example' from 'C:\\Program Files\\Python36\\example.py'>

**For <= Python3.3:**

import imp

imp.reload(example)

<module 'example' from 'C:\\Program Files\\Python36\\example.py'>

**For Python2.x:**

reload(example)

## **Packages in Python**

A bundle of multiple modules together form a package.

A package is basically a hierarchical directory structure with Python files and a file with the name \_\_init\_\_.py. This means that every directory inside of the Python path, which contains a file named \_\_init\_\_.py, will be treated as a package by Python. It's possible to put several modules into a Package.

Packages are a way of structuring Python’s module namespace by using "dotted module names". A.B stands for a submodule named B in a package named A. Two different packages like P1 and P2 can both have modules with the same name. The submodule A of the package P1 and the submodule A of the package P2 can be totally different.

A package is imported like a "normal" module.

Create a  *Phone* directory where your package will be stored as shown below :-

**NOTE: Run this terminial as administrator if you have privilege issue.**

Microsoft Windows [Version 10.0.15063]

(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\haryadav>cd ..

C:\Users>cd ..

C:\>chdir C:\Program Files\Python36\

C:\Program Files\Python36>mkdir Phone

C:\Program Files\Python36>cd Phone

C:\Program Files\Python36\Phone>

C:\Program Files\Python36\Phone>python

Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32

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

Welcome to Python!

Hari Yadav

Create a file Basicphone*.py* available in *Phone* directory. This file has following line of source code −

def Basicphone():

print("This is Basic Phone")

Similar way, we have another two files having different functions as shown below :−

* *Phone/Isdnphone.py* file having function Isdnphone() code as below:

**def Isdnphone():**

**print("This is ISDN Phone")**

* *Phone/IPphone.py* file having function *IPphone*() code as below:

**def IPphone():**

**print("This is IP Phone")**

Now, create one more file \_\_init\_\_.py in *Phone* directory.

**Phone/\_\_init\_\_.py**

To make all of your functions available when you've imported **Phone**, you need to put explicit import statements in \_\_init\_\_.py as follows −

from Basicphone import Basicphone

from Isdnphone import Isdnphone

from IPphone import IPphone

After you add these lines to \_\_init\_\_.py, you have all of these functions available when you import the Phone package.

#!/usr/bin/python

# Now import your Phone Package.

import Phone

Phone.Basicphone()

Phone.Isdnphone()

Phone.IPphone()

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

This is Basic Phone

This is ISDN Phone

This is IP Phone

In the above example, we have taken example of a single functions in each file, but you can keep multiple functions in your files. You can also define different Python classes in those files and then you can create your packages out of those classes.

# Python Files I/O

This topic covers all the basic I/O functions available in Python. For more functions, please refer to standard Python documentation.

## **Printing to the Screen**

The simplest way to produce output is using the *print* statement where you can pass zero or more expressions separated by commas. This function converts the expressions you pass into a string and writes the result to standard output as follows −

#!/usr/bin/python

print("Python is really a great language,", "it is very interesting?")

This produces the following result on your standard screen −

Python is really a great language,", "it is very interesting?

## **Reading Keyboard Input**

Python provides two built-in functions to read a line of text from standard input, which by default comes from the keyboard. These functions are −

* raw\_input ( prior to Python 3.0)
* input

## **The *raw\_input* Function**

The *raw\_input([prompt])* function reads one line from standard input and returns it as a string (removing the trailing newline).

#!/usr/bin/python

str = raw\_input("Enter your input: ");

print("Received input is : ", str)

This prompts you to enter any string and it would display same string on the screen. When I typed "Hello Python!", its output is like this −

Enter your input: Hello Python

Received input is : Hello Python

## **The *input* Function**

The *input([prompt])* function is equivalent to raw\_input, it assumes the input as character string.

#!/usr/bin/python

str = input("Enter your input: ");

print("Received input is : ", str)

This would produce the following result against the entered input –

Enter your input: Capgemini India

Recieved input is : Capgemini India

**NOTE: Input is treated as character string and arithmetic operation is not possible.**

str = input("Enter your input: ");

Enter your input: 100

v\_result = str + 200

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: must be str, not int

v\_result = int(str) + 200

print('Result is : ', v\_result)

Result is : 300

## **Opening and Closing Files**

Until now, you have been reading and writing to the standard input and output. Now, we will see how to use actual data files.

Python provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation using a **file** object.

## **The *open* Function**

Before you can read or write a file, you have to open it using Python's built-in ***open()*** function. This function creates a **file** object, which would be utilized to call other support methods associated with it.

### **Syntax**

file\_object = open(file\_name [, access\_mode][, buffering])

Here are parameter details:

* **file\_name:** The file\_name argument is a string value that contains the name of the file that you want to access.
* **access\_mode:** The access\_mode determines the mode in which the file has to be opened, i.e. **read, write, append** etc. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read (r).
* **buffering:** If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default(default behavior).

Here is a list of the different modes of opening a file −

|  |  |
| --- | --- |
| **Modes** | **Description** |
| r | Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode. |
| rb | Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode. |
| r+ | Opens a file for both reading and writing. The file pointer placed at the beginning of the file. |
| rb+ | Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file. |
| w | Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing. |
| wb | Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing. |
| w+ | Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing. |
| wb+ | Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing. |
| a | Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing. |
| ab | Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing. |
| a+ | Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing. |
| ab+ | Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing. |

## **The *file* Object Attributes**

Once a file is opened and you have one *file* object, you can get various information related to that file.

Here is a list of all attributes related to file object:

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| file.closed | Returns true if file is closed, false otherwise. |
| file.mode | Returns access mode with which file was opened. |
| file.name | Returns name of the file. |

### **Example**

#!/usr/bin/python

# Open a file

v\_file = open("resume.txt", "w")

print("Name of the file: ", v\_file.name)

print("Closed or not : ", v\_file.closed)

print("Opening mode : ", v\_file.mode)

v\_file.close()

**NOTE: Run this terminial as administrator if you have privilege issue.**

Microsoft Windows [Version 10.0.15063]

(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\haryadav>cd ..

C:\Users>cd ..

C:\>chdir C:\Program Files\Python36\

C:\Program Files\Python36>mkdir fileIO

C:\Program Files\Python36>cd fileIO

C:\Program Files\Python36\fileIO>

C:\Program Files\Python36\fileIO>python

Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32

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

Welcome to Python!

Hari Yadav

v\_file = open("resume.txt", "wb")

print("Name of the file: ", v\_file.name)

print("Closed or not : ", v\_file.closed)

print("Open mode of the file : ", v\_file.mode)

v\_file.close()

print("File closed : ", v\_file.closed)

This produces the following result −

Name of the file: resume.txt

Closed or not : False

Open mode of the file : wb

File closed : True

We create a file object using open() function and get a list of all possible methods that can be used with a file object, using Python built-in dir() function. The dir() function accepts a Python object as an argument and returns a list of attributes and methods related to them.

dir(v\_file)

['\_\_class\_\_', '\_\_del\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_enter\_\_', '\_\_eq\_\_', '\_\_exit\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getstate\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_next\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_checkClosed', '\_checkReadable', '\_checkSeekable', '\_checkWritable', '\_dealloc\_warn', '\_finalizing', 'close', 'closed', 'detach', 'fileno', 'flush', 'isatty', 'mode', 'name', 'raw', 'read', 'read1', 'readable', 'readinto', 'readinto1', 'readline', 'readlines', 'seek', 'seekable', 'tell', 'truncate', 'writable', 'write', 'writelines']

## **The *close()* Method**

The close() method of a *file* object flushes any unwritten information and closes the file object, after which no more writing can be done.

Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.

### **Syntax**

fileObject.close();

### **Example**

#!/usr/bin/python

# Open a file

v\_file = open("resume.txt", "wb")

print("Name of the file: ", v\_file.name)

# Close opend file

v\_file.close()

print("File is closed : ", v\_file.closed)

This produces the following result −

Name of the file: resume.txt

File is closed : True

## **Reading and Writing Files**

The *file* object provides a set of access methods to make our lives easier. We would see how to use ***read()*** and ***write()*** methods to read and write files.

## **The *write()* Method**

The *write()* method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.

The write() method does not add a newline character ('\n') to the end of the string. You have to do it explicitly.

**Syntax**

fileObject.write(string);

Here, passed parameter is the content to be written into the opened file.

### **Example**

#!/usr/bin/python

# Open a file

v\_file = open("resume.txt", "w")

print("Open mode of the file : ", v\_file.mode)

v\_file.write( "Python is a great language.\nYeah its great!!\n");

# Close opend file

v\_file.close()

The above method would create *resume.txt* file and would write given content in that file and finally it would close that file. If you would open this file, it would have following content.

Open mode of the file : w

Python is a great language.

Yeah its great!!

## **The *read()* Method**

The *read()* method reads a string from an open file. It is important to note that Python strings can have binary data. apart from text data.

### **Syntax**

fileObject.read([count]);

Here, passed parameter is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if *count* is missing, then it tries to read as much as possible, maybe until the end of file.

### **Example**

Let's read the data from our file *resume.txt*, which we created above.

#!/usr/bin/python

# Open a file

v\_file = open("resume.txt", "r+")

str = v\_file.read(10);

print("Read String is : ", str)

str = v\_file.read();

print("Read String is : ", str)

v\_file.write( "Python can be installed on both window and linux.\nLinux provide better security!!\n");

v\_file.close()

v\_file = open("resume.txt", "r+")

str = v\_file.read();

print("Read String is : ", str)

# Close opend file

v\_file.close()

# Readlines from the file.

import os

os.getcwd()

'C:\\Program Files\\Python36\\fileIO'

myFileObject = open('C:\\Program Files\\Python36\\fileIO\\resume.txt')

filedata = myFileObject.readlines()

filedata

['Python is a great language.\n', 'Yeah its great!!\n', 'Python can be installed on both window and linux.\n', 'Linux provide better security!!\n']

myFileObject.close()

This produces the following result −

Read String is : Python is

Read String is : a great language.

Yeah its great!!

Read String is : Python is a great language.

Yeah its great!!

Python can be installed on both window and linux.

Linux provide better security!!

## **File Positions**

The ***tell()*** method tells you the current position within the file; in other words, the next read or write will occur at that many bytes from the beginning of the file.

The ***seek(offset[, from])*** method changes the current file position.

The *offset* argument indicates the number of bytes to be moved. The *from* argument specifies the reference position from where the bytes are to be moved.

If *from* is set to:

0 means use the beginning of the file as the reference position.

1 means use the current position as the reference position.

2 means at the end of the file would be taken as the reference position.

### **Example**

#!/usr/bin/python

# Open a file

v\_file = open("resume.txt", "r+")

str = v\_file.read(10);

print("Read String is : ", str)

# Check current position

v\_position = v\_file.tell();

print("Current file position is : ", v\_position)

# Reposition pointer at the beginning once again

v\_position = v\_file.seek(0, 0);

print("Current file position is : ", v\_position)

str = v\_file.read(10);

print("Again Read String is : ", str)

v\_position = v\_file.seek(0, 1);

print("Current file position is now : ", v\_position)

str = v\_file.read(10);

print("Again Read String is : ", str)

v\_position = v\_file.seek(0, 1);

print("Current file position is now : ", v\_position)

# Reposition pointer to the end

v\_position = v\_file.seek(0, 2);

print("Current file position is : ", v\_position)

str = v\_file.read(10);

print("Again Read String is : ", str)

# Reposition pointer back to the begining

v\_position = v\_file.seek(0, 0);

str = v\_file.read(20);

print("Again Read String is : ", str)

# Close opend file

v\_file.close()

## **Renaming and Deleting Files**

Python **os** module provides methods that help you perform file-processing operations, such as renaming and deleting files.

To use this module you need to import it first and then you can call any related functions.

## **The rename() Method**

The *rename()* method takes two arguments, the current filename and the new filename.

### **Syntax**

os.rename(current\_file\_name, new\_file\_name)

### **Example**

Following is the example to rename an existing file *resume.txt*:

#!/usr/bin/python

import os

# Rename a file from resume.txt to Hari\_Yadav\_Resume.txt

os.rename( "resume.txt", "Hari\_Yadav\_Resume.txt" )

os.rename( "Hari\_Yadav\_Resume.txt", "C:\\Program Files\\Python36\\fileIO\\Hari\_Yadav\_Resume\_2.txt"))

os.rename("Hari\_Yadav\_Resume.txt","resume.txt")

## **The *copyfile()* Method**

from shutil import copyfile

copyfile("resume.txt","Hari\_Yadav\_Resume\_1.txt")

'Hari\_Yadav\_Resume\_1.txt'

copyfile("Hari\_Yadav\_Resume\_1.txt","Hari\_Yadav\_Resume\_2.txt")

'Hari\_Yadav\_Resume\_2.txt'

## **The *remove()* Method**

You can use the *remove()* method to delete files by supplying the name of the file to be deleted as the argument.

### **Syntax**

os.remove(file\_name)

### **Example**

#!/usr/bin/python

import os

os.remove("Hari\_Yadav\_Resume\_2.txt")

## **Directories in Python**

All files are contained within various directories, and Python has no problem handling these too. The **os** module has several methods that help you create, remove, and change directories.

## **The *mkdir()* Method**

You can use the *mkdir()* method of the **os** module to create directories in the current directory. You need to supply an argument to this method which contains the name of the directory to be created.

### **Syntax**

os.mkdir("newdir")

### **Example**

Following is the example to create a directory *test* in the current directory −

#!/usr/bin/python

import os

os.mkdir("fileIO\_1")

os.mkdir("fileIO\_2")

## **The *chdir()* Method**

You can use the *chdir()* method to change the current directory. The chdir() method takes an argument, which is the name of the directory that you want to make the current directory.

### **Syntax**

os.chdir("newdir")

### **Example**

Following is the example to go into "/home/newdir" directory −

#!/usr/bin/python

import os

# Changing a directory to "/home/newdir"

os.chdir('C:\\Program Files\\Python36\\fileIO\\fileIO\_1')

os.getcwd()

'C:\\Program Files\\Python36\\fileIO\\fileIO\_1'

os.chdir('C:\\Program Files\\Python36\\fileIO')

os.getcwd()

'C:\\Program Files\\Python36\\fileIO'

# Switch to sub directory

os.chdir("fileIO\_1")

os.getcwd()

'C:\\Program Files\\Python36\\fileIO\\fileIO\_1'

## **The *getcwd()* Method**

The *getcwd()* method displays the current working directory.

### **Syntax**

os.getcwd()

### **Example**

Following is the example to give current directory −

#!/usr/bin/python

import os

# This would give location of the current directory

os.getcwd()

## **The *rmdir()* Method**

The *rmdir()* method deletes the directory, which is passed as an argument in the method.

Before removing a directory, all the contents in it should be removed.

### **Syntax:**

os.rmdir('dirname')

### **Example**

Following is the example to remove "/tmp/test" directory. It is required to give fully qualified name of the directory, otherwise it would search for that directory in the current directory.

#!/usr/bin/python

import os

os.listdir()

['fileIO\_1', 'fileIO\_2', 'Hari\_Yadav\_Resume\_1.txt', 'resume.txt']

os.rmdir("fileIO\_1")

os.rmdir('C:\\Program Files\\Python36\\fileIO\\fileIO\_2')

os.listdir()

['Hari\_Yadav\_Resume\_1.txt', 'resume.txt']

## **File & Directory Related Methods**

There are three important sources, which provide a wide range of utility methods to handle and manipulate files & directories on Windows and Unix operating systems. They are as follows −

File Object Methods: The *file* object provides functions to manipulate files.

A **file** object is created using *open* function and here is a list of functions which can be called on this object:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods with Description** |
| 1 | [**file.close()**](https://www.tutorialspoint.com/python/file_close.htm)  Close the file. A closed file cannot be read or written any more. |
| 2 | [**file.flush()**](https://www.tutorialspoint.com/python/file_flush.htm)  Flush the internal buffer, like stdio's fflush. This may be a no-op on some file-like objects. |
| 3 | [**file.fileno()**](https://www.tutorialspoint.com/python/file_fileno.htm)  Returns the integer file descriptor that is used by the underlying implementation to request I/O operations from the operating system. |
| 4 | [**file.isatty()**](https://www.tutorialspoint.com/python/file_isatty.htm)  Returns True if the file is connected to a tty(-like) device, else False. |
| 5 | [**file.next()**](https://www.tutorialspoint.com/python/file_next.htm)  Returns the next line from the file each time it is being called. |
| 6 | [**file.read([size])**](https://www.tutorialspoint.com/python/file_read.htm)  Reads at most size bytes from the file (less if the read hits EOF before obtaining size bytes). |
| 7 | [**file.readline([size])**](https://www.tutorialspoint.com/python/file_readline.htm)  Reads one entire line from the file. A trailing newline character is kept in the string. |
| 8 | [**file.readlines([sizehint])**](https://www.tutorialspoint.com/python/file_readlines.htm)  Reads until EOF using readline() and return a list containing the lines. If the optional sizehint argument is present, instead of reading up to EOF, whole lines totalling approximately sizehint bytes (possibly after rounding up to an internal buffer size) are read. |
| 9 | [**file.seek(offset[, whence])**](https://www.tutorialspoint.com/python/file_seek.htm)  Sets the file's current position |
| 10 | [**file.tell()**](https://www.tutorialspoint.com/python/file_tell.htm)  Returns the file's current position |
| 11 | [**file.truncate([size])**](https://www.tutorialspoint.com/python/file_truncate.htm)  Truncates the file's size. If the optional size argument is present, the file is truncated to (at most) that size. |
| 12 | [**file.write(str)**](https://www.tutorialspoint.com/python/file_write.htm)  Writes a string to the file. There is no return value. |
| 13 | [**file.writelines(sequence)**](https://www.tutorialspoint.com/python/file_writelines.htm)  Writes a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. |

[OS Object Methods](https://www.tutorialspoint.com/python/os_file_methods.htm): This provides methods to process files as well as directories.

The **os** module provides a big range of useful methods to manipulate files and directories.

|  |  |
| --- | --- |
| **Sr.No.** | **Methods with Description** |
| 1 | [**os.access(path, mode)**](https://www.tutorialspoint.com/python/os_access.htm)  Use the real uid/gid to test for access to path. |
| 2 | [**os.chdir(path)**](https://www.tutorialspoint.com/python/os_chdir.htm)  Change the current working directory to path |
| 3 | [**os.chflags(path, flags)**](https://www.tutorialspoint.com/python/os_chflags.htm)  Set the flags of path to the numeric flags. |
| 4 | [**os.chmod(path, mode)**](https://www.tutorialspoint.com/python/os_chmod.htm)  Change the mode of path to the numeric mode. |
| 5 | [**os.chown(path, uid, gid)**](https://www.tutorialspoint.com/python/os_chown.htm)  Change the owner and group id of path to the numeric uid and gid. |
| 6 | [**os.chroot(path)**](https://www.tutorialspoint.com/python/os_chroot.htm)  Change the root directory of the current process to path. |
| 7 | [**os.close(fd)**](https://www.tutorialspoint.com/python/os_close.htm)  Close file descriptor fd. |
| 8 | [**os.closerange(fd\_low, fd\_high)**](https://www.tutorialspoint.com/python/os_closerange.htm)  Close all file descriptors from fd\_low (inclusive) to fd\_high (exclusive), ignoring errors. |
| 9 | [**os.dup(fd)**](https://www.tutorialspoint.com/python/os_dup.htm)  Return a duplicate of file descriptor fd. |
| 10 | [**os.dup2(fd, fd2)**](https://www.tutorialspoint.com/python/os_dup2.htm)  Duplicate file descriptor fd to fd2, closing the latter first if necessary. |
| 11 | [**os.fchdir(fd)**](https://www.tutorialspoint.com/python/os_fchdir.htm)  Change the current working directory to the directory represented by the file descriptor fd. |
| 12 | [**os.fchmod(fd, mode)**](https://www.tutorialspoint.com/python/os_fchmod.htm)  Change the mode of the file given by fd to the numeric mode. |
| 13 | [**os.fchown(fd, uid, gid)**](https://www.tutorialspoint.com/python/os_fchown.htm)  Change the owner and group id of the file given by fd to the numeric uid and gid. |
| 14 | [**os.fdatasync(fd)**](https://www.tutorialspoint.com/python/os_fdatasync.htm)  Force write of file with filedescriptor fd to disk. |
| 15 | [**os.fdopen(fd[, mode[, bufsize]])**](https://www.tutorialspoint.com/python/os_fdopen.htm)  Return an open file object connected to the file descriptor fd. |
| 16 | [**os.fpathconf(fd, name)**](https://www.tutorialspoint.com/python/os_fpathconf.htm)  Return system configuration information relevant to an open file. name specifies the configuration value to retrieve. |
| 17 | [**os.fstat(fd)**](https://www.tutorialspoint.com/python/os_fstat.htm)  Return status for file descriptor fd, like stat(). |
| 18 | [**os.fstatvfs(fd)**](https://www.tutorialspoint.com/python/os_fstatvfs.htm)  Return information about the filesystem containing the file associated with file descriptor fd, like statvfs(). |
| 19 | [**os.fsync(fd)**](https://www.tutorialspoint.com/python/os_fsync.htm)  Force write of file with filedescriptor fd to disk. |
| 20 | [**os.ftruncate(fd, length)**](https://www.tutorialspoint.com/python/os_ftruncate.htm)  Truncate the file corresponding to file descriptor fd, so that it is at most length bytes in size. |
| 21 | [**os.getcwd()**](https://www.tutorialspoint.com/python/os_getcwd.htm)  Return a string representing the current working directory. |
| 22 | [**os.getcwdu()**](https://www.tutorialspoint.com/python/os_getcwdu.htm)  Return a Unicode object representing the current working directory. |
| 23 | [**os.isatty(fd)**](https://www.tutorialspoint.com/python/os_isatty.htm)  Return True if the file descriptor fd is open and connected to a tty(-like) device, else False. |
| 24 | [**os.lchflags(path, flags)**](https://www.tutorialspoint.com/python/os_lchflags.htm)  Set the flags of path to the numeric flags, like chflags(), but do not follow symbolic links. |
| 25 | [**os.lchmod(path, mode)**](https://www.tutorialspoint.com/python/os_lchmod.htm)  Change the mode of path to the numeric mode. |
| 26 | [**os.lchown(path, uid, gid)**](https://www.tutorialspoint.com/python/os_lchown.htm)  Change the owner and group id of path to the numeric uid and gid. This function will not follow symbolic links. |
| 27 | [**os.link(src, dst)**](https://www.tutorialspoint.com/python/os_link.htm)  Create a hard link pointing to src named dst. |
| 28 | [**os.listdir(path)**](https://www.tutorialspoint.com/python/os_listdir.htm)  Return a list containing the names of the entries in the directory given by path. |
| 29 | [**os.lseek(fd, pos, how)**](https://www.tutorialspoint.com/python/os_lseek.htm)  Set the current position of file descriptor fd to position pos, modified by how. |
| 30 | [**os.lstat(path)**](https://www.tutorialspoint.com/python/os_lstat.htm)  Like stat(), but do not follow symbolic links. |
| 31 | [**os.major(device)**](https://www.tutorialspoint.com/python/os_major.htm)  Extract the device major number from a raw device number. |
| 32 | [**os.makedev(major, minor)**](https://www.tutorialspoint.com/python/os_makedev.htm)  Compose a raw device number from the major and minor device numbers. |
| 33 | [**os.makedirs(path[, mode])**](https://www.tutorialspoint.com/python/os_makedirs.htm)  Recursive directory creation function. |
| 34 | [**os.minor(device)**](https://www.tutorialspoint.com/python/os_minor.htm)  Extract the device minor number from a raw device number. |
| 35 | [**os.mkdir(path[, mode])**](https://www.tutorialspoint.com/python/os_mkdir.htm)  Create a directory named path with numeric mode mode. |
| 36 | [**os.mkfifo(path[, mode])**](https://www.tutorialspoint.com/python/os_mkfifo.htm)  Create a FIFO (a named pipe) named path with numeric mode mode. The default mode is 0666 (octal). |
| 37 | [**os.mknod(filename[, mode=0600, device])**](https://www.tutorialspoint.com/python/os_mknod.htm)  Create a filesystem node (file, device special file or named pipe) named filename. |
| 38 | [**os.open(file, flags[, mode])**](https://www.tutorialspoint.com/python/os_open.htm)  Open the file file and set various flags according to flags and possibly its mode according to mode. |
| 39 | [**os.openpty()**](https://www.tutorialspoint.com/python/os_openpty.htm)  Open a new pseudo-terminal pair. Return a pair of file descriptors (master, slave) for the pty and the tty, respectively. |
| 40 | [**os.pathconf(path, name)**](https://www.tutorialspoint.com/python/os_pathconf.htm)  Return system configuration information relevant to a named file. |
| 41 | [**os.pipe()**](https://www.tutorialspoint.com/python/os_pipe.htm)  Create a pipe. Return a pair of file descriptors (r, w) usable for reading and writing, respectively. |
| 42 | [**os.popen(command[, mode[, bufsize]])**](https://www.tutorialspoint.com/python/os_popen.htm)  Open a pipe to or from command. |
| 43 | [**os.read(fd, n)**](https://www.tutorialspoint.com/python/os_read.htm)  Read at most n bytes from file descriptor fd. Return a string containing the bytes read. If the end of the file referred to by fd has been reached, an empty string is returned. |
| 44 | [**os.readlink(path)**](https://www.tutorialspoint.com/python/os_readlink.htm)  Return a string representing the path to which the symbolic link points. |
| 45 | [**os.remove(path)**](https://www.tutorialspoint.com/python/os_remove.htm)  Remove the file path. |
| 46 | [**os.removedirs(path)**](https://www.tutorialspoint.com/python/os_removedirs.htm)  Remove directories recursively. |
| 47 | [**os.rename(src, dst)**](https://www.tutorialspoint.com/python/os_rename.htm)  Rename the file or directory src to dst. |
| 48 | [**os.renames(old, new)**](https://www.tutorialspoint.com/python/os_renames.htm)  Recursive directory or file renaming function. |
| 49 | [**os.rmdir(path)**](https://www.tutorialspoint.com/python/os_rmdir.htm)  Remove the directory path |
| 50 | [**os.stat(path)**](https://www.tutorialspoint.com/python/os_stat.htm)  Perform a stat system call on the given path. |
| 51 | [**os.stat\_float\_times([newvalue])**](https://www.tutorialspoint.com/python/os_stat_float_times.htm)  Determine whether stat\_result represents time stamps as float objects. |
| 52 | [**os.statvfs(path)**](https://www.tutorialspoint.com/python/os_statvfs.htm)  Perform a statvfs system call on the given path. |
| 53 | [**os.symlink(src, dst)**](https://www.tutorialspoint.com/python/os_symlink.htm)  Create a symbolic link pointing to src named dst. |
| 54 | [**os.tcgetpgrp(fd)**](https://www.tutorialspoint.com/python/os_tcgetpgrp.htm)  Return the process group associated with the terminal given by fd (an open file descriptor as returned by open()). |
| 55 | [**os.tcsetpgrp(fd, pg)**](https://www.tutorialspoint.com/python/os_tcsetpgrp.htm)  Set the process group associated with the terminal given by fd (an open file descriptor as returned by open()) to pg. |
| 56 | [**os.tempnam([dir[, prefix]])**](https://www.tutorialspoint.com/python/os_tempnam.htm)  Return a unique path name that is reasonable for creating a temporary file. |
| 57 | [**os.tmpfile()**](https://www.tutorialspoint.com/python/os_tmpfile.htm)  Return a new file object opened in update mode (w+b). |
| 58 | [**os.tmpnam()**](https://www.tutorialspoint.com/python/os_tmpnam.htm)  Return a unique path name that is reasonable for creating a temporary file. |
| 59 | [**os.ttyname(fd)**](https://www.tutorialspoint.com/python/os_ttyname.htm)  Return a string which specifies the terminal device associated with file descriptor fd. If fd is not associated with a terminal device, an exception is raised. |
| 60 | [**os.unlink(path)**](https://www.tutorialspoint.com/python/os_unlink.htm)  Remove the file path. |
| 61 | [**os.utime(path, times)**](https://www.tutorialspoint.com/python/os_utime.htm)  Set the access and modified times of the file specified by path. |
| 62 | [**os.walk(top[, topdown=True[, onerror=None[, followlinks=False]]])**](https://www.tutorialspoint.com/python/os_walk.htm)  Generate the file names in a directory tree by walking the tree either top-down or bottom-up. |
| 63 | [**os.write(fd, str)**](https://www.tutorialspoint.com/python/os_write.htm)  Write the string str to file descriptor fd. Return the number of bytes actually written. |

# Python Exceptions Handling

Python provides two very important features to handle any unexpected error in your Python programs and to add debugging capabilities in them −

* **Exception Handling:**
* **Assertions:**

List of Standard Exceptions:

|  |  |
| --- | --- |
| **EXCEPTION NAME** | **DESCRIPTION** |
| Exception | Base class for all exceptions |
| StopIteration | Raised when the next() method of an iterator does not point to any object. |
| SystemExit | Raised by the sys.exit() function. |
| StandardError | Base class for all built-in exceptions ***except StopIteration and SystemExit.*** |
| ArithmeticError | Base class for all errors that occur for numeric calculation. |
| OverflowError | Raised when a calculation exceeds maximum limit for a numeric type. |
| FloatingPointError | Raised when a floating point calculation fails. |
| ZeroDivisionError | Raised when division or modulo by zero takes place for all numeric types. |
| AssertionError | Raised in case of failure of the Assert statement. |
| AttributeError | Raised in case of failure of attribute reference or assignment. |
| EOFError | Raised when there is no input from either the raw\_input() or input() function and the end of file is reached. |
| ImportError | Raised when an import statement fails. |
| KeyboardInterrupt | Raised when the user interrupts program execution, usually by pressing Ctrl+c. |
| LookupError | Base class for all lookup errors. |
| IndexError  KeyError | Raised when an index is not found in a sequence.  Raised when the specified key is not found in the dictionary. |
| NameError | Raised when an identifier is not found in the local or global namespace. |
| UnboundLocalError  EnvironmentError | Raised when trying to access a local variable in a function or method but no value has been assigned to it.  Base class for all exceptions that occur outside the Python environment. |
| IOError  IOError | Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.  Raised for operating system-related errors. |
| SyntaxError  IndentationError | Raised when there is an error in Python syntax.  Raised when indentation is not specified properly. |
| SystemError | Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit. |
| SystemExit | Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit. |
| TypeError | Raised when an operation or function is attempted that is invalid for the specified data type. |
| ValueError | Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified. |
| RuntimeError | Raised when a generated error does not fall into any category. |
| NotImplementedError | Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented. |

### **Assertions in Python**

An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.

The easiest way to think of an assertion is like to a **raise-if** statement (or to be more accurate, a **raise-if-not** statement). An expression is tested, and if the result comes up false, an exception is raised.

Assertions are carried out by the assert statement, the newest keyword to Python, introduced in version 1.5.

Programmers often place assertions at the start of a function to check for ***valid input***, and after a function call to check for ***valid output.***

### **The *assert* Statement**

When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an *AssertionError* exception.

The syntax for assert is −

assert Expression[, Arguments]

If the assertion fails, Python uses ArgumentExpression as the argument for the AssertionError. AssertionError exceptions can be caught and handled like any other exception using the ***try-except*** statement, but if not handled, they will terminate the program and produce a traceback.

### **Example 1.**

Here is a function that converts a temperature from degrees Celsius to Fahrenheit. Since zero degrees celsius is as cold as it gets, the function bails out if it sees a negative temperature

#!/usr/bin/python

def CelsiusToFahrenheit(Temperature):

assert (Temperature >= 0),"Colder than absolute zero!"

return (Temperature \* 1.8)+32

print(CelsiusToFahrenheit(37))

print(int(CelsiusToFahrenheit(37)))

print(CelsiusToFahrenheit (-5))

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

print(CelsiusToFahrenheit(37))

98.60000000000001

print(int(CelsiusToFahrenheit(37)))

98

print(CelsiusToFahrenheit (-5))

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 2, in CelsiusToFahrenheit

AssertionError: Colder than absolute zero!

### **Example 2.**

def value\_comparison\_f():

return 3

def test\_value\_comparison\_f():

# argument not specified

assert value\_comparison\_f() == 4

test\_value\_comparison\_f()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 3, in test\_value\_comparison\_f

**AssertionError**

def test\_set\_comparison():

set1 = set("12345")

set2 = set("678910")

assert (set1 == set2), "Set elements should match."

test\_set\_comparison()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 4, in test\_set\_comparison

AssertionError: Set elements should match.

## **What is Exception?**

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

## **Handling an exception**

If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.

### **Syntax**

Here is simple syntax of ***try....except...else*** blocks −

try:

You do your operations here;

......................

except *ExceptionI*:

If there is ExceptionI, then execute this block.

except *ExceptionII*:

If there is ExceptionII, then execute this block.

......................

else:

If there is no exception then execute this block.

Here are few important points about the above-mentioned syntax −

* A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
* You can also provide a generic except clause, which handles any exception.
* After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
* The else-block is a good place for code that does not need the try: block's protection.

### **Example**

This example opens a file, writes content in the, file and comes out gracefully because there is no problem at all:

#!/usr/bin/python

NOTE: In my case without admin priviliges, I got this error.

C:\Users\haryadav>chdir C:\Program Files\Python36\fileIO\

C:\Program Files\Python36\fileIO>python

Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32

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

Hari Yadav

Welcome to python

try:

fh = open("testfile.txt", "w")

fh.write("This is my test file for exception handling!!")

except IOError:

print("Error: can\'t find file or read data")

else:

print("Written content in the file successfully")

fh.close()

This produces the following result −

Error: can't find file or read data

NOTE: With admin priviliges.

C:\Users\haryadav>chdir C:\Program Files\Python36\fileIO\

C:\Program Files\Python36\fileIO>python

Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32

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

Hari Yadav

Welcome to python

try:

fh = open("testfile.txt", "w")

fh.write("This is my test file for exception handling!!")

except IOError:

print("Error: can\'t find file or read data")

else:

print("Written content in the file successfully")

fh.close()

This produces the following result −

Written content in the file successfully

## **The *except* Clause with No Exceptions**

You can also use the except statement with no exceptions defined as follows −

try:

You do your operations here;

......................

except:

If there is any exception, then execute this block.

......................

else:

If there is no exception then execute this block.

Microsoft Windows [Version 10.0.15063]

(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\haryadav>chdir C:\Program Files\Python36\fileIO\

C:\Program Files\Python36\fileIO>python

Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32

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

Hari Yadav

Welcome to python

try:

fh = open("testfile.txt", "w")

fh.write("This is my test file for exception handling!!")

except IOError:

print("Error: can\'t find file or read data")

else:

print("Written content in the file successfully")

fh.close()

Error: can't find file or read data

try:

#fh = open("testfile.txt", "w")

v\_num =100/0

except:

print("Unknown error. Please check your programm carefully.")

Unknown error. Please check your programm carefully.

try:

fh = open("testfile.txt", "w")

# v\_num =100/0

except:

print("Unknown error. Please check your programm carefully.")

Unknown error. Please check your programm carefully.

This kind of a **try-except** statement catches all the exceptions that occur. Using this kind of **try-except** statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

## **The *except* Clause with Multiple Exceptions**

You can also use the same *except* statement to handle multiple exceptions as follows −

try:

You do your operations here;

......................

except(Exception1[, Exception2[,...ExceptionN]]]):

If there is any exception from the given exception list,

then execute this block.

......................

else:

If there is no exception then execute this block.

NOTE: dev\_resume.txt file is not available.

try:

f = open("dev\_resume.txt")

# name = input("Enter your name : ")

except (KeyboardInterrupt, FileNotFoundError, PermissionError):

print("\nPlease try again.")

NOTE: After run, Press control + c to interupt the input from keyboard.

try:

# f = open("dev\_resume.txt")

name = input("Enter your name : ")

except (KeyboardInterrupt, FileNotFoundError, PermissionError):

print("\nPlease try again.")

## **The try-finally Clause**

You can use a **finally:** block along with a **try:** block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this −

try:

You do your operations here;

......................

Due to any exception, this may be skipped.

finally:

This would always be executed.

......................

You cannot use *else* clause as well along with a finally clause.

### **Example**

#!/usr/bin/python

try:

fh = open("testfile", "w")

fh.write("This is my test file for exception handling!!")

finally:

print("Error: can\'t find file or read data")

If you do not have permission to open the file in writing mode, then this will produce the following result:

Error: can't find file or read data  
Traceback (most recent call last):

File "<stdin>", line 2, in <module>

PermissionError: [Errno 13] Permission denied: 'testfile'

Same example can be written more cleanly as follows −

#!/usr/bin/python

try:

fh = open("testfile.txt", "w")

try:

fh.write("This is my test file for exception handling!!")

finally:

print("Going to close the file")

fh.close()

except IOError:

print("Error: can\'t find file or read data")

When an exception is thrown in the *try* block, the execution immediately passes to the *finally* block. After all the statements in the *finally* block are executed, the exception is raised again and is handled in the *except* statements if present in the next higher layer of the *try-except* statement.

## **Argument of an Exception**

An exception can have an *argument*, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows −

try:

You do your operations here;

......................

except *ExceptionType, Argument*:

You can print value of Argument here...

If you write the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.

This variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.

### **Example**

Following is an example for a single exception −

#!/usr/bin/python

# Define a function here.

def temp\_convert(var):

try:

return int(var)

except ValueError, Argument:

print "The argument does not contain numbers\n", Argument

# Call above function here.

temp\_convert("xyz");

This produces the following result −

The argument does not contain numbers

invalid literal for int() with base 10: 'xyz'

## **Raising an Exceptions**

You can raise exceptions in several ways by using the raise statement. The general syntax for the **raise** statement is as follows.

### **Syntax**

raise [Exception [, args [, traceback]]]

Here, *Exception* is the type of exception (for example, NameError) and *argument* is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None.

The final argument, traceback, is also optional (and rarely used in practice), and if present, is the traceback object used for the exception.

### **Example**

An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows −

def functionName( level ):

if level < 1:

raise "Invalid level!", level

# The code below to this would not be executed

# if we raise the exception

**Note:** In order to catch an exception, an "except" clause must refer to the same exception thrown either class object or simple string. For example, to capture above exception, we must write the except clause as follows −

try:

Business Logic here...

except "Invalid level!":

Exception handling here...

else:

Rest of the code here...

## **User-Defined Exceptions**

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example related to *RuntimeError*. Here, a class is created that is subclassed from *RuntimeError*. This is useful when you need to display more specific information when an exception is caught.

In the try block, the user-defined exception is raised and caught in the except block. The variable e is used to create an instance of the class *Networkerror*.

class Networkerror(RuntimeError):

def \_\_init\_\_(self, arg):

self.args = arg

So once you defined above class, you can raise the exception as follows −

try:

raise Networkerror("Bad hostname")

except Networkerror,e:

print e.args

# Python 3 - Object Oriented

Python has been an object-oriented language since the time it existed. Due to this, creating and using classes and objects are downright easy.

## **Overview of OOP Terminology**

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

## **Creating Classes**

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

class ClassName:

'Optional class documentation string'

class\_suite

* The class has a documentation string, which can be accessed via ***ClassName.\_\_doc\_\_***.
* The ***class\_suite*** consists of all the component statements defining class members, data attributes and functions.

### **Example**

Following is an example of a simple Python class –

del(Employee)

class Employee:

'Common base class for all employees'

empCount = 0

def \_\_init\_\_(self, name, salary):

self.name = name

self.salary = salary

Employee.empCount += 1

def displayCount(self):

print ("Total Employee %d" % Employee.empCount)

def displayEmployee(self):

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

* The variable ***empCount*** is a class variable whose value is shared among all the instances in this class. This can be accessed as ***Employee.empCount*** from inside the class or outside the class.
* The first method ***\_\_init\_\_()*** is a special method, which is called ***class constructor or initialization method*** that Python calls when you create a new instance of this class.
* You declare other class methods like normal functions with the exception that the first argument to each method is *self*. Python adds the *self* argument to the list for you; you do not need to include it when you call the methods.

## **Creating Instance Objects**

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

# This would create first object of Employee class

emp1 = Employee("Zara", 2000)

# This would create second object of Employee class

emp2 = Employee("Manni", 5000)

## **Accessing Attributes**

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

emp1.displayEmployee()

emp2.displayEmployee()

print ("Total Employee %d" % Employee.empCount)

Now, putting all the concepts together −

#!/usr/bin/python3

class Employee:

'Common base class for all employees'

empCount = 0

def \_\_init\_\_(self, name, salary):

self.name = name

self.salary = salary

Employee.empCount += 1

def displayCount(self):

print ("Total Employee %d" % Employee.empCount)

def displayEmployee(self):

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

#This would create first object of Employee class"

emp1 = Employee("Zara", 2000)

#This would create second object of Employee class"

emp2 = Employee("Manni", 5000)

emp1.displayEmployee()

emp2.displayEmployee()

print ("Total Employee %d" % Employee.empCount)

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

Name : Zara ,Salary: 2000

Name : Manni ,Salary: 5000

Total Employee 2

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

emp1.displayEmployee()

Name : Zara , Salary: 2000

emp1.salary = 7000 # Update 'salary' attribute.

emp1.name = 'Mark Tully' # Update 'name' attribute.

emp1.displayEmployee()

Name : Mark Tully , Salary: 7000

emp1.age = 50 # Add 'age' attribute.

emp1.displayEmployee()

Name : Mark Tully , Salary: 7000

del emp1.salary # Delete 'salary' attribute.

emp1.displayEmployee()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 11, in displayEmployee

AttributeError: 'Employee' object has no attribute 'salary'

emp2.displayEmployee()

Name : Manni , Salary: 5000

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

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

hasattr(emp1, 'salary') # Returns true if 'salary' attribute exists

False

hasattr(emp1, 'name') # Returns true if 'name' attribute exists

True

hasattr(emp1, 'age') # Returns true if 'age' attribute exists

True

getattr(emp1, 'name') # Returns value of 'name' attribute

'Mark Tully'

getattr(emp1, 'age') # Returns value of 'age' attribute

50

getattr(emp1, 'salary') # Returns value of 'salary' attribute

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

**AttributeError: 'Employee' object has no attribute 'salary'**

setattr(emp1, 'salary', 7000) # Set attribute 'salary' at 7000

getattr(emp1, 'salary') # Returns value of 'salary' attribute

7000

delattr(emp1, 'salary') # Delete attribute 'salary'

getattr(emp1, 'salary') # Returns value of 'salary' attribute

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'Employee' object has no attribute 'salary'

## **Built-In Class Attributes**

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

* **\_\_dict\_\_** − Dictionary containing the class's namespace.
* **\_\_doc\_\_** − Class documentation string or none, if undefined.
* **\_\_name\_\_** − Class name.
* **\_\_module\_\_** − Module name in which the class is defined. This attribute is "\_\_main\_\_" in interactive mode.
* **\_\_bases\_\_** − A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

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

#!/usr/bin/python3

class Employee:

'Common base class for all employees'

empCount = 0

def \_\_init\_\_(self, name, salary):

self.name = name

self.salary = salary

Employee.empCount += 1

def displayCount(self):

print ("Total Employee %d" % Employee.empCount)

def displayEmployee(self):

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

emp1 = Employee("Zara", 2000)

emp2 = Employee("Manni", 5000)

print ("Employee.\_\_doc\_\_:", Employee.\_\_doc\_\_)

print ("Employee.\_\_name\_\_:", Employee.\_\_name\_\_)

print ("Employee.\_\_module\_\_:", Employee.\_\_module\_\_)

print ("Employee.\_\_bases\_\_:", Employee.\_\_bases\_\_)

print ("Employee.\_\_dict\_\_:", Employee.\_\_dict\_\_ )

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

Employee.\_\_doc\_\_: Common base class for all employees

Employee.\_\_name\_\_: Employee

Employee.\_\_module\_\_: \_\_main\_\_

Employee.\_\_bases\_\_: (<class 'object'>,)

Employee.\_\_dict\_\_: {

'displayCount': <function Employee.displayCount at 0x0160D2B8>,

'\_\_module\_\_': '\_\_main\_\_', '\_\_doc\_\_': 'Common base class for all employees',

'empCount': 2, '\_\_init\_\_':

<function Employee.\_\_init\_\_ at 0x0124F810>, 'displayEmployee':

<function Employee.displayEmployee at 0x0160D300>,

'\_\_weakref\_\_':

<attribute '\_\_weakref\_\_' of 'Employee' objects>, '\_\_dict\_\_':

<attribute '\_\_dict\_\_' of 'Employee' objects>

}

## **Destroying Objects (Garbage Collection)**

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

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

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

a = 40 # Create object <40>

b = a # Increase ref. count of <40>

c = [b] # Increase ref. count of <40>

del a # Decrease ref. count of <40>

b = 100 # Decrease ref. count of <40>

c[0] = -1 # Decrease ref. count of <40>

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

### **Example**

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

#!/usr/bin/python3

class Point:

def \_\_init\_\_( self, x=0, y=0):

self.x = x

self.y = y

def \_\_del\_\_(self):

class\_name = self.\_\_class\_\_.\_\_name\_\_

print (class\_name, "destroyed")

pt1 = Point()

pt2 = pt1

pt3 = pt1

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

del pt1

del pt2

del pt3

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

3083401324 3083401324 3083401324

Point destroyed

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

In the above example, assuming definition of a Point class is contained in *point.py* and there is no other executable code in it.

#!/usr/bin/python3

import point

p1 = point.Point()

## **Class Inheritance**

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

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

### **Syntax**

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

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

'Optional class documentation string'

class\_suite

### **Example**

#!/usr/bin/python3

class Parent: # define parent class

parentAttr = 100

def \_\_init\_\_(self):

print ("Calling parent constructor")

def parentMethod(self):

print ('Calling parent method')

def setAttr(self, attr):

Parent.parentAttr = attr

def getAttr(self):

print ("Parent attribute :", Parent.parentAttr)

class Child(Parent): # define child class

def \_\_init\_\_(self):

print ("Calling child constructor")

def childMethod(self):

print ('Calling child method')

c = Child() # instance of child

c.childMethod() # child calls its method

c.parentMethod() # calls parent's method

c.setAttr(200) # again call parent's method

c.getAttr() # again call parent's method

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

Calling child constructor

Calling child method

Calling parent method

Parent attribute : 200

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

class A: # define your class A

.....

class B: # define your calss B

.....

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

.....

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

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

## **Overriding Methods**

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

### **Example**

#!/usr/bin/python3

class Parent: # define parent class

def myMethod(self):

print ('Calling parent method')

class Child(Parent): # define child class

def myMethod(self):

print ('Calling child method')

c = Child() # instance of child

c.myMethod() # child calls overridden method

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

Calling child method

## **Base Overloading Methods**

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

|  |  |
| --- | --- |
| **S.No.** | **Method, Description & Sample Call** |
| 1 | **\_\_init\_\_ ( self [,args...] )**  Constructor (with any optional arguments)  Sample Call : *obj = className(args)* |
| 2 | **\_\_del\_\_( self )**  Destructor, deletes an object  Sample Call : *del obj* |
| 3 | **\_\_repr\_\_( self )**  Evaluatable string representation  Sample Call : *repr(obj)* |
| 4 | **\_\_str\_\_( self )**  Printable string representation  Sample Call : *str(obj)* |
| 5 | **\_\_cmp\_\_ ( self, x )**  Object comparison  Sample Call : *cmp(obj, x)* |

## **Overloading Operators**

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

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

### **Example**

#!/usr/bin/python3

class Vector:

def \_\_init\_\_(self, a, b):

self.a = a

self.b = b

def \_\_str\_\_(self):

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

def \_\_add\_\_(self,other):

return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)

v2 = Vector(5,-2)

print (v1 + v2)

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

Vector(7,8)

## **Data Hiding**

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

### **Example**

#!/usr/bin/python3

class JustCounter:

\_\_secretCount = 0

def count(self):

self.\_\_secretCount += 1

print (self.\_\_secretCount)

counter = JustCounter()

counter.count()

counter.count()

print (counter.\_\_secretCount)

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

1

2

Traceback (most recent call last):

File "test.py", line 12, in <module>

print counter.\_\_secretCount

AttributeError: JustCounter instance has no attribute '\_\_secretCount'

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

.........................

print (counter.\_JustCounter\_\_secretCount)

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

1

2

2

# Python 3 - Regular Expressions

A *regular expression* is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. Regular expressions are widely used in UNIX world.

The module **re** provides full support for Perl-like regular expressions in Python. The **re** module raises the exception **re.error** if an error occurs while compiling or using a regular expression.

We would cover two important functions, which would be used to handle regular expressions. Nevertheless, a small thing first: There are various characters, which would have special meaning when they are used in regular expression. To avoid any confusion while dealing with regular expressions, we would use Raw Strings as **r'expression'**.

### **Basic patterns that match single chars**

|  |  |
| --- | --- |
| **S.No.** | **Expression & Matches** |
| 1 | **a, X, 9, <**  ordinary characters just match themselves exactly. |
| 2 | **. (a period)**  matches any single character except newline '\n' |
| 3 | **\w**  matches a "word" character: a letter or digit or underbar [a-zA-Z0-9\_]. |
| 4 | **\W**  matches any non-word character. |
| 5 | **\b**  boundary between word and non-word |
| 6 | **\s**  matches a single whitespace character -- space, newline, return, tab |
| 7 | **\S**  matches any non-whitespace character. |
| 8 | **\t, \n, \r**  tab, newline, return |
| 9 | **\d**  decimal digit [0-9] |
| 10 | **^**  matches start of the string |
| 11 | **$**  match the end of the string |
| 12 | **\**  inhibit the "specialness" of a character. |

### **Compilation flags**

Compilation flags let you modify some aspects of how regular expressions work. Flags are available in the re module under two names, a long name such as **IGNORECASE** and a short, one-letter form such as I.

|  |  |
| --- | --- |
| **S.No.** | **Flag & Meaning** |
| 1 | **ASCII, A**  Makes several escapes like \w, \b, \s and \d match only on ASCII characters with the respective property. |
| 2 | **DOTALL, S**  Make, match any character, including newlines |
| 3 | **IGNORECASE, I**  Do case-insensitive matches |
| 4 | **LOCALE, L**  Do a locale-aware match |
| 5 | **MULTILINE, M**  Multi-line matching, affecting ^ and $ |
| 6 | **VERBOSE, X (for ‘extended’)**  Enable verbose REs, which can be organized more cleanly and understandably |

## **The match Function**

This function attempts to match RE *pattern* to *string* with optional *flags*.

Here is the syntax for this function −

re.match(pattern, string, flags = 0)

Here is the description of the parameters −

|  |  |
| --- | --- |
| **S.No.** | **Parameter & Description** |
| 1 | **pattern**  This is the regular expression to be matched. |
| 2 | **string**  This is the string, which would be searched to match the pattern at the beginning of string. |
| 3 | **flags**  You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below. |

The *re.match* function returns a **match** object on success, **None** on failure. We use *group(num)* or *groups()* function of **match** object to get matched expression.

|  |  |
| --- | --- |
| **S.No.** | **Match Object Method & Description** |
| 1 | **group(num = 0)**  This method returns entire match (or specific subgroup num) |
| 2 | **groups()**  This method returns all matching subgroups in a tuple (empty if there weren't any) |

### **Example 1.**

#!/usr/bin/python3

import re

line = "Cats are smarter than dogs"

matchObj = re.match( r'(.\*) are (.\*?) .\*', line, re.M|re.I)

if matchObj:

print ("matchObj.group() : ", matchObj.group())

print ("matchObj.group(1) : ", matchObj.group(1))

print ("matchObj.group(2) : ", matchObj.group(2))

else:

print ("No match!!")

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

matchObj.group() : Cats are smarter than dogs

matchObj.group(1) : Cats

matchObj.group(2) : smarter

### **Example 2.**

#!/usr/bin/python3

import re

# Sample strings.

list = ["dog dot", "do don't", "dumb-dumb", "no match"]

# Loop.

for element in list:

# Match if two words starting with letter d.

m = re.match("(d\w+)\W(d\w+)", element)

# See if success.

if m:

print(m.groups())

NOTE: Pattern details

Pattern: (d\w+)\W(d\w+)

d Lowercase letter d.

\w+ One or more word characters.

\W A non-word character.

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

('dog', 'dot')

('do', 'don')

('dumb', 'dumb')

### **Example 3.**

# Sample strings.

list = ["dog dot", "do don't", "dumb-dumb", "no match"]

# Loop.

for element in list:

# Match if two words starting with letter d.

m = re.match("(n\w+)\W(m\w+)", element)

# See if success.

if m:

print(m.groups())

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

('no', 'match')

## **The search Function**

**Search**. This method is different from match. Both apply a pattern. But search attempts this at all possible starting points in the string. Match just tries the first starting point.

So:

**Search** scans through the input string and tries to match at any location.

Here is the syntax for this function −

re.search(pattern, string, flags = 0)

Here is the description of the parameters −

|  |  |
| --- | --- |
| **S.No.** | **Parameter & Description** |
| 1 | **pattern**  This is the regular expression to be matched. |
| 2 | **string**  This is the string, which would be searched to match the pattern anywhere in the string. |
| 3 | **flags**  You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below. |

The *re.search* function returns a **match** object on success, **none** on failure. We use *group(num)*or *groups()*function of **match** object to get the matched expression.

|  |  |
| --- | --- |
| **S.No.** | **Match Object Method & Description** |
| 1 | **group(num = 0)**  This method returns entire match (or specific subgroup num) |
| 2 | **groups()**  This method returns all matching subgroups in a tuple (empty if there weren't any) |

### **Example 1.**

#!/usr/bin/python3

import re

line = "Cats are smarter than dogs";

searchObj = re.search( r'(.\*) are (.\*?) .\*', line, re.M|re.I)

if searchObj:

print ("searchObj.group() : ", searchObj.group())

print ("searchObj.group(1) : ", searchObj.group(1))

print ("searchObj.group(2) : ", searchObj.group(2))

else:

print ("Nothing found!!")

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

matchObj.group() : Cats are smarter than dogs

matchObj.group(1) : Cats

matchObj.group(2) : smarter

### **Example 2.**

#!/usr/bin/python3

import re

# Input.

value = "voorheesville"

m = re.search("(vi.\*)", value)

if m:

# This is reached.

print("search:", m.group(1))

m = re.match("(vi.\*)", value)

if m:

# This is not reached.

print("match:", m.group(1))

NOTE: Pattern details

Pattern: (vi.\*)

vi The lowercase letters v and i together.

.\* Zero or more characters of any type.

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

Output

search: ville

## **Matching Versus Searching**

Python offers two different primitive operations based on regular expressions: **match** checks for a match only at the beginning of the string, while **search** checks for a match anywhere in the string.

### **Example**

#!/usr/bin/python3

import re

line = "Cats are smarter than dogs";

matchObj = re.match( r'dogs', line, re.M|re.I)

if matchObj:

print ("match --> matchObj.group() : ", matchObj.group())

else:

print ("No match!!")

searchObj = re.search( r'dogs', line, re.M|re.I)

if searchObj:

print ("search --> searchObj.group() : ", searchObj.group())

else:

print ("Nothing found!!")

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

No match!!

search --> matchObj.group() : dogs

## **The Search and Replace function (sub)**

One of the most important **re** methods that use regular expressions is **sub**.

### **Syntax**

re.sub(pattern, repl, string, max=0)

This method replaces all occurrences of the RE *pattern* in *string* with *repl*, substituting all occurrences unless *max* is provided. This method returns modified string.

### **Example**

#!/usr/bin/python3

import re

phone = "2004-959-559 # This is Phone Number"

# Delete Python-style comments

num = re.sub(r'#.\*$', "", phone)

print ("Phone Num : ", num)

# Remove anything other than digits

num = re.sub(r'\D', "", phone)

print ("Phone Num : ", num)

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

Phone Num : 2004-959-559

Phone Num : 2004959559

## **The split Function**

**Split**. The **re.split()** method accepts a pattern argument. This pattern specifies the delimiter. With it, we can use any text that matches a pattern as the delimiter to separate text data.

A split() method is also available directly on a string. This method handles no regular expressions. It is simpler.

### **Syntax**

re.split(pattern, string)

We split the string on one or more non-digit characters.

**Example 1.**

#!/usr/bin/python3

import re

# Input string.

value = "one 1 two 2 three 3"

# Separate on one or more non-digit characters.

result = re.split("\D+", value)

# Print results.

for element in result:

print(element)

NOTE: Pattern details

Pattern: \D+

\D+ One or more non-digit characters.

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

1

2

3

**Example 2.**

**Split the string where ever there is capital letter.**

import re

# Input string.

value = "DogCatElephant"

# Separate on one or more non-digit characters.

result = re.split("([A-Z][^A-Z]\*)", value)

# Print results.

for element in result:

print(element)

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

Dog

Cat

Elephant

[s for s in re.split("([A-Z][^A-Z]\*)", "DogCatElephant") if s]

['Dog', 'Cat', 'Elephant']

## **The Findall Function**

**Findall**. This is similar to **split()**. **Findall** accepts a pattern that indicates which strings to return in a list. It is like **split()** but we specify matching parts, not delimiters.

### **Syntax**

re.findall(pattern, string)

We scan a string for all words starting with the letter d or p, and with one or more following word characters.

**Example 1.**

import re

# Input.

value = "abc 123 def 456 dot map pat"

# Find all words starting with d or p.

list = re.findall("[dp]\w+", value)

# Print result.

print(list)

**NOTE: Pattern details**

Pattern: [dp]\w+

[dp] lowercase d, or a lowercase p.

\w+ One or more word characters.

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

['def', 'dot', 'pat']

## **Regular Expression Modifiers: Option Flags**

Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. You can provide multiple modifiers using exclusive OR (|), as shown previously and may be represented by one of these −

|  |  |
| --- | --- |
| **S.No.** | **Modifier & Description** |
| 1 | **re.I**  Performs case-insensitive matching. |
| 2 | **re.L**  Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior (\b and \B). |
| 3 | **re.M**  Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string). |
| 4 | **re.S**  Makes a period (dot) match any character, including a newline. |
| 5 | **re.U**  Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B. |
| 6 | **re.X**  Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker. |

## **Regular Expression Patterns**

Except for the control characters, **(+ ? . \* ^ $ ( ) [ ] { } | \)**, all characters match themselves. You can escape a control character by preceding it with a backslash.

The following table lists the regular expression syntax that is available in Python −

Here is the list of regular expression syntax in Python.

## **Regular Expression Examples**

### **Literal characters**

|  |  |
| --- | --- |
| **S.No.** | **Example & Description** |
| 1 | **python**  Match "python". |

## **Character classes**

|  |  |
| --- | --- |
| **S.No.** | **Example & Description** |
| 1 | **[Pp]ython**  Match "Python" or "python" |
| 2 | **rub[ye]**  Match "ruby" or "rube" |
| 3 | **[aeiou]**  Match any one lowercase vowel |
| 4 | **[0-9]**  Match any digit; same as [0123456789] |
| 5 | **[a-z]**  Match any lowercase ASCII letter |
| 6 | **[A-Z]**  Match any uppercase ASCII letter |
| 7 | **[a-zA-Z0-9]**  Match any of the above |
| 8 | **[^aeiou]**  Match anything other than a lowercase vowel |
| 9 | **[^0-9]**  Match anything other than a digit |

## **Special Character Classes**

|  |  |
| --- | --- |
| **S.No.** | **Example & Description** |
| 1 | **.**  Match any character except newline |
| 2 | **\d**  Match a digit: [0-9] |
| 3 | **\D**  Match a nondigit: [^0-9] |
| 4 | **\s**  Match a whitespace character: [ \t\r\n\f] |
| 5 | **\S**  Match nonwhitespace: [^ \t\r\n\f] |
| 6 | **\w**  Match a single word character: [A-Za-z0-9\_] |
| 7 | **\W**  Match a nonword character: [^A-Za-z0-9\_] |

## **Repetition Cases**

|  |  |
| --- | --- |
| **S.No.** | **Example & Description** |
| 1 | **ruby?**  Match "rub" or "ruby": the y is optional |
| 2 | **ruby\***  Match "rub" plus 0 or more ys |
| 3 | **ruby+**  Match "rub" plus 1 or more ys |
| 4 | **\d{3}**  Match exactly 3 digits |
| 5 | **\d{3,}**  Match 3 or more digits |
| 6 | **\d{3,5}**  Match 3, 4, or 5 digits |

## **Nongreedy repetition**

This matches the smallest number of repetitions −

|  |  |
| --- | --- |
| **S.No.** | **Example & Description** |
| 1 | **<.\*>**  Greedy repetition: matches "<python>perl>" |
| 2 | **<.\*?>**  Nongreedy: matches "<python>" in "<python>perl>" |

## **Grouping with Parentheses**

|  |  |
| --- | --- |
| **S.No.** | **Example & Description** |
| 1 | **\D\d+**  No group: + repeats \d |
| 2 | **(\D\d)+**  Grouped: + repeats \D\d pair |
| 3 | **([Pp]ython(,)?)+**  Match "Python", "Python, python, python", etc. |

## **Backreferences**

This matches a previously matched group again −

|  |  |
| --- | --- |
| **S.No.** | **Example & Description** |
| 1 | **([Pp])ython&\1ails**  Match python&pails or Python&Pails |
| 2 | **(['"])[^\1]\*\1**  Single or double-quoted string. \1 matches whatever the 1st group matched. \2 matches whatever the 2nd group matched, etc. |

## **Alternatives**

|  |  |
| --- | --- |
| **S.No.** | **Example & Description** |
| 1 | **python|perl**  Match "python" or "perl" |
| 2 | **rub(y|le)**  Match "ruby" or "ruble" |
| 3 | **Python(!+|\?)**  "Python" followed by one or more ! or one ? |

## **Anchors**

This needs to specify match position.

|  |  |
| --- | --- |
| **S.No.** | **Example & Description** |
| 1 | **^Python**  Match "Python" at the start of a string or internal line |
| 2 | **Python$**  Match "Python" at the end of a string or line |
| 3 | **\APython**  Match "Python" at the start of a string |
| 4 | **Python\Z**  Match "Python" at the end of a string |
| 5 | **\bPython\b**  Match "Python" at a word boundary |
| 6 | **\brub\B**  \B is nonword boundary: match "rub" in "rube" and "ruby" but not alone |
| 7 | **Python(?=!)**  Match "Python", if followed by an exclamation point. |
| 8 | **Python(?!!)**  Match "Python", if not followed by an exclamation point. |

## **Special Syntax with Parentheses**

|  |  |
| --- | --- |
| **S.No.** | **Example & Description** |
| 1 | **R(?#comment)**  Matches "R". All the rest is a comment |
| 2 | **R(?i)uby**  Case-insensitive while matching "uby" |
| 3 | **R(?i:uby)**  Same as above |
| 4 | **rub(?:y|le))**  Group only without creating \1 backreference |