Day 1

<https://fts.capgemini.com/private/35637904535082/AutomationDriveAcademy-Python-V...-WednesdayApril1820182.16.56PM.mp4>   
<https://fts.capgemini.com/private/35637904535082/AutomationDriveAcademy-Python-V...-WednesdayApril1820189.22.03AM.mp4>

Day 2

<https://fts.capgemini.com/private/4489512087870/PythonDay2.docx>   
<https://fts.capgemini.com/private/49249925815831/AutomationDriveAcademy-Python-V...-ThursdayApril1920182.03.45PM.mp4>   
<https://fts.capgemini.com/private/49249925815831/AutomationDriveAcademy-Python-V...-ThursdayApril1920189.23.03AM.mp4>

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.

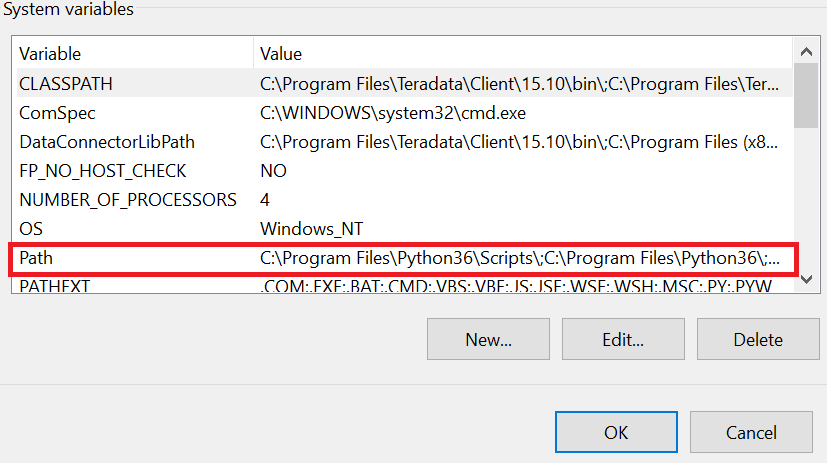
Welcome to Python!

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')

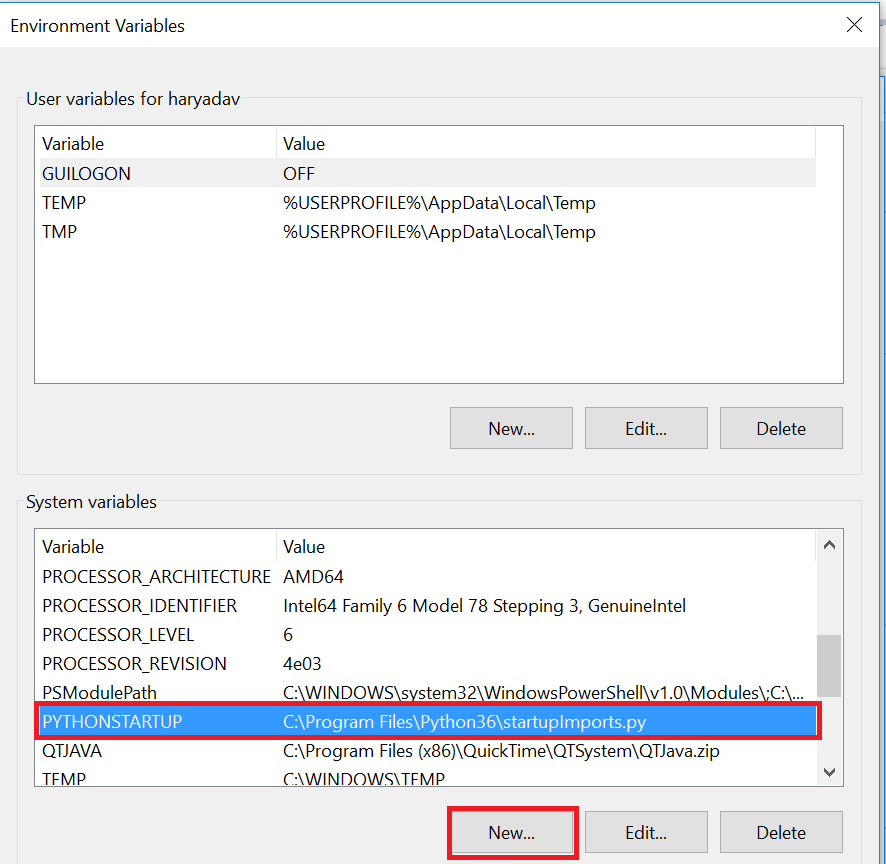
print('Welcome to Python')

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 3, 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

For window

C:\Program Files\Python36>chdir D:/Users/haryadav/Desktop/Python

C:\Program Files\Python36>D:

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

Hello, Python!

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.

## **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"

print(word)

print(sentence)

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

29 states and seven union territories."""

print(paragraph)

paragraph

'India is our country. It has**\n**29 states and seven union territories.'

print(paragraph)

India is our country. It has

29 states and seven union territories.

## **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.

# This is a comment line.

if False:

""" This is to

show you

multiline comments."""

if False:

print("True")

print ("Hello, Python!")

print ("This is our first program!")

else:

print("False")

print ("Hello, Python!")

print ("This is our first program!")

## **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\nPlease enter file name.")

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 equally.

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 (1) or False (0).
* **Numbers** can be integers (1 and 2), floats (1.1 and 1.2), fractions (1/2 and 2/3), or even complex numbers (5+2j). 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 stucture 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

v\_rec = [1111, 'Dev', 'Singh', 'Sr.DBA', 70000]

v\_rec

[1111, 'Dev', 'Singh', 'Sr.DBA', 70000]

v\_rec.insert(3,'Rajpoot')

v\_rec

[1111, 'Dev', 'Singh', 'Rajpoot', 'Sr.DBA', 70000]

del v\_rec[2]

v\_rec

[1111, 'Dev', 'Rajpoot', 'Sr.DBA', 70000]

**# 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 composite 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'}

dict['CG Bangalore'] = "Whitefield","ATP","DTP"

dict['CG Bangalore']

('Whitefield', 'ATP', 'DTP')

dict['CG Bangalore'][0]

'Whitefield'

dict['CG Bangalore'][1]

'ATP'

dict['CG Bangalore'][2]

'DTP'

**# 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

emp\_rec = {7369 : ("SMITH","CLERK",7902,"17-DEC-80",80000,20), 7499 : ("ALLEN","SALESMAN",7698, "20-FEB-81",16000,300,30)}

print(emp\_rec)

{7369: ('SMITH', 'CLERK', 7902, '17-DEC-80', 80000, 20), 7499: ('ALLEN', 'SALESMAN', 7698, '20-FEB-81', 16000, 300, 30)}

Display data in vertical format like tablular record.

Insert in between.