

## Python

---

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

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

### Why to Learn Python?

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

Python is a MUST for students and working professionals to become a great Software Engineer specially when they are working in Web Development Domain. I will list down some of the key advantages of learning Python:

- **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.

### Characteristics of Python

Following are important characteristics of Python Programming –

- 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.

### It is used for:

- web development (server-side),
- software development,
- mathematics,

- system scripting.

---

### What can Python do?

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.

### Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way.

### Python Syntax compared to other programming languages

- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

## Features of Python

- **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.

### History of Python

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

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

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

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

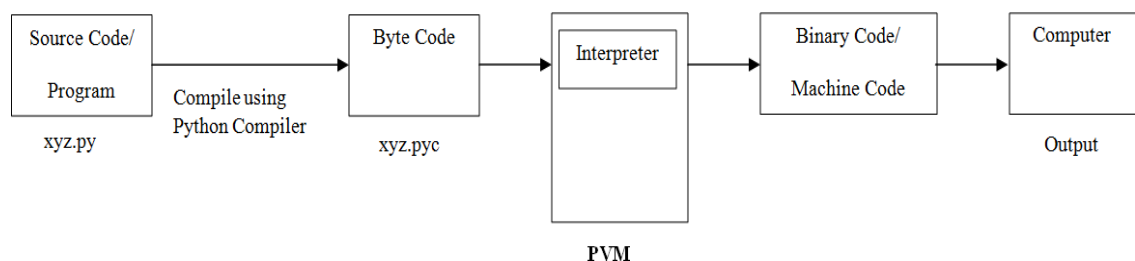
Python 1.0 was released in November 1994. In 2000, Python 2.0 was released. Python 2.7.11 is the latest edition of Python 2.

Meanwhile, Python 3.0 was released in 2008. Python 3 is not backward compatible with Python 2. The emphasis in Python 3 had been on the removal of duplicate programming constructs and modules so that "There should be one -- and preferably only one -- obvious way to do it." Python 3.5.1 is the latest version of Python 3.

## Python Virtual Machine

Python Virtual Machine (PVM) is a program which provides programming environment. The role of PVM is to convert the byte code instruction into machine code so the computer can execute those machine code instructions and display the output. Interpreter converts the byte code into machine code and sends that machine code to the computer processor for execution.

- Write Source Code/Program
- Compile the Program using Python Compiler
- Compiler Converts the Python Program into byte Code
- Computer/Machine can not understand Byte Code so we convert it into Machine Code using PVM
- PVM uses an interpreter which understand Byte Code so we convert it into machine code
- Machine Code instructions are then executed by the processor and results are displayed



## Memory Management in Python

Memory allocation is a process by which computer programs are assigned memory or space.

In python, all of this done on the backend by the python Memory Manager automatically.

In python, a variable is considered as tag that is tied to some value.

Python considers values as objects.

There are four different cases by which we can understand how memory management done in python :→

1. Different Variables with Different Values
2. Different Variable with same Values
3. Assign old variable to new variable
4. Overwriting Values

## Garbage Collection in Python

---

When a value in memory is no longer referenced by a variable, the python interpreter automatically removes it from memory. This process is known as garbage collection.

A module represents python code that performs a specific task. Garbage collector is a module in python that is useful to delete objects from memory which are not used in the program. The module that represents the garbage collector is named as gc. Garbage collector in the simplest way to maintain a count for each object regarding how many times that object is referenced (or used).

```
>>> amount=10 #garbage value
>>> amount=5
>>> print(amount)
5
>>> x=200
>>> print(x)
200
>>> x='Delhi'
>>> print(x)
Delhi
```

### **Comparisons between C-Java-Python**

Datatypes : union, structure	Supported	Supported	Not Supported	Supported
Pre-processor directives	Supported (#include, #define)	Supported (#include, #define)	Not Supported	Not Supported
Header files	Supported	Supported	Use Packages (import)	Use Packages (import)
Inheritance	No Inheritance	Supported	Multiple Inheritance not Supported	Supported
Overloading	No Overloading	Supported	Operator Overloading not Supported	Operator Overloading not Supported
Pointers	Supported	Supported	No Pointers	No Pointers
Code Translation	Compiled	Compiled	Interpreted	Interpreted

Aspects	C	C++	Java	Python
Developed Year	1972	1979	1991	1991
Developed By	Dennis Ritchie	Bjarne Stroustrup	James Gosling	Guido van Rossum
Successor of	BCPL	C	C(Syntax) & C++ (Structure)	ABC language
Paradigms	Procedural	Object Oriented	Object Oriented	Multi-Paradigm
Platform Dependency	Dependent	Dependent	Independent	Independent
Keywords	32	63	50 defined (goto, const unusable)	33

Storage Allocation	Uses malloc, calloc	Uses new, delete	uses garbage collector	uses garbage collector
Multi-threading and Interfaces	Not Supported	Not Supported	Supported	Supported
Exception Handling	No Exception handling	Supported	Supported	Supported
Templates	Not Supported	Supported	Not Supported	Supported
Storage class: auto, extern	Supported	Supported	Not Supported	Not Supported
Destructors	No Constructor or Destructor	Supported	Not Supported	Not Supported
Database Connectivity	Not Supported	Not Supported	Supported	Supported

## Writing first Python Program

---

Let us execute programs in different modes of programming.

### 1. Interactive Mode Programming

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

```
$ python
Python 2.4.3 (#1, Nov 11 2010, 13:34:43)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-48)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

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

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

#### OUTPUT :

```
Hello, Python!
```

### 2. 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!"
```

## Python Identifiers

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

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

Here are naming conventions for Python identifiers –



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

## **Reserved Words**

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

And	exec	Not
Assert	finally	Or
Break	for	Pass
Class	from	Print
Continue	global	Raise
Def	if	Return
Del	import	Try
Elif	in	While
Else	is	with
except	lambda	Yield

## **Lines and Indentation**

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

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

```
if True:
    print "True"
else:
    print "False"
```

However, the following block generates an error –

```
if True:
print "Answer"
print "True"
else:
print "Answer"
print "False"
```

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

### **Multi-Line Statements**

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

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

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

```
days = ['Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday']
```

### **Quotation in Python**

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

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

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

## **Comments in Python**

---

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

```
# First comment  
print "Hello, Python!" # second comment
```

This produces the following result –

```
Hello, Python!
```

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

```
name = "Madisetti" # This is again comment
```

You can comment multiple lines as follows –

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

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

```
"""  
This is a multiline  
comment.  
"""
```

## **Using Blank Lines**

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

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

## **Waiting for the User**

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

```
raw_input("\n\nPress the enter key to exit.")
```

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

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

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

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

### **Multiple Statement Groups as Suites**

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

Header lines begin the statement (with the keyword) and terminate with a colon ( : ) and are followed by one or more lines which make up the suite. For example –

```
if expression :  
    suite  
elif expression :  
    suite  
else :  
    suite
```

## **Python Variable**

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 –

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

print counter
print miles
print name
```

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

```
100
1000.0
John
```

### **Multiple Assignment**

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

```
a = b = c = 1
```

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

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

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

## **Data Types in Python**

A datatype represents the type of data stored into a variable or memory. The datatypes which are already available in python language are called Built-in datatypes. The datatypes which can be created by the programmers are called User-define datatypes.

### **Built-in datatypes**

The built-in datatypes are of five types :

- None Type
- Numeric Types
- Sequences
- Sets
- Mapping

## 1. None Type

---

None datatype represents as object that doesn't contain any value. In languages like Java, it is called 'null' object. But in python, it is called 'None' object.

In a Python program, maximum of only one 'None' object is provided. One of the uses of 'None' is that it is used inside a function as a default value of the arguments. When calling the function, if no value is passed, then the default value will be taken as 'None'. If some value is passed to the function, then that value is used by the function. In Boolean expressions, 'None' datatype represents 'False'.

## 2. Numeric Type/Number

The numeric types represent numbers. These are three sub types:

- ✚ int
- ✚ float
- ✚ complex

- **Int**

The int datatype represents an integer number. An integer number without any decimal point or fraction part. In Python, it is possible to store very large integer number as there is no limit for the size of an int datatype.

Ex:

20, 10, -50, -1002

pin\_code=564512

a=20

Here, 'a' is called int type variable since it is storing 20 which is an integer value.

```
y=10
```

```
print(y)
```

```
O/P : 10
```

```
type(y)
```

```
O/P : <class 'int'>
```

---

- **Float**

The float datatype represents floating point numbers. A floating point number is number that contain a decimal point.

Ex:

20.56, 10.89, -50.89, -0.8

```
price=10.70
```

```
print(price)
```

```
O/P : 10.70
```

```
type(price)
```

```
O/P : <class 'float'>
```

- **Complex**

A complex number is a number that is written in the form of  $a+bj$  or  $a+bJ$ ,

$a$ =Real Part of the number

$b$ =Imaginary part of the number

$j$  or  $J$ =Square root value of  $-1$

$a$  and  $b$  may contain integer or float number.

Ex:

$5+7j$ ,  $0.8+2j$

```
com=5+7j
```

```
print(com)
```

```
O/P : 5+7j
```

```
type(com)
```

```
O/P : <class 'complex'>
```

### 3. Sequences Type

Generally, a sequence represents a group of elements or items. For example, a group of integer numbers will form a sequence. There are six types of sequence

- ✚ str
- ✚ list
- ✚ tuple
- ✚ range

#### ➤ str datatype

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
```

This will produce the following result –

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

In python, str represents string datatype. A string represents group of characters. Strings are enclosed in double quotes or single quotes. Both are valid.



Ex:

```
str="hello"  
  
str1='hello'  
  
print(str)  
  
print(str1)
```

```
type(str)  
  
<class 'string'>
```

We can also write strings inside “””(triple double quotes) or ‘’’(triple single quotes) to span a group of lines include spaces.

```
Str="""hello  
Friends  
Good morning"""  
  
Str1="""hello  
Friends  
Good morning"""
```

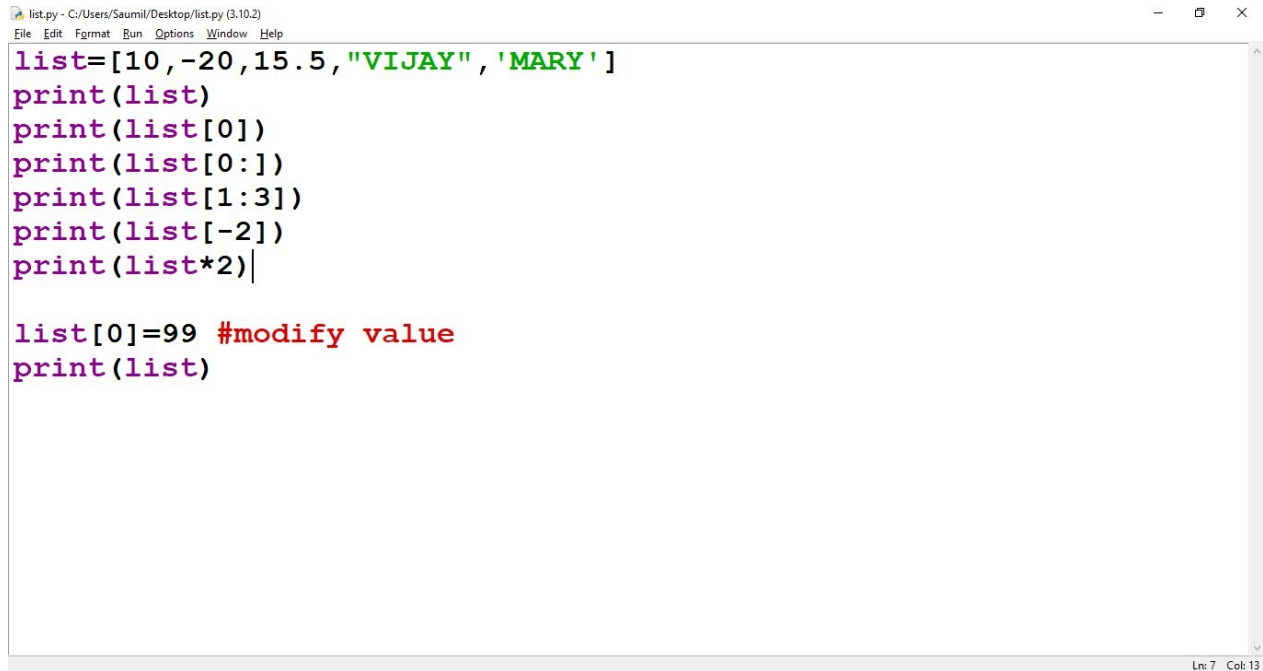
### ➤ **List datatype**

Lists in python are similar to arrays in C or Java. A list represents a group of elements. A list can store different types of elements which can be modified. The main difference between a list and an array is that a list can store different types of elements but an array can store only one type of elements. Also, list can grow dynamically in memory which means size is not fixed. But the size of array is fixed and they cannot grow at runtime. Lists are represented using square brackets[] and the elements are written in [], separated by commas.

Ex:

```
list=[10,-20,15.5,'vijay','mary']
```

will create a list with different types of elements. The slicing operation like[0:3] represents elements from 0<sup>th</sup> to 2<sup>nd</sup> positions.



```
list.py - C:/Users/Saumil/Desktop/list.py (3.10.2)
File Edit Format Run Options Window Help
list=[10,-20,15.5,"VIJAY",'MARY']
print(list)
print(list[0])
print(list[0:])
print(list[1:3])
print(list[-2])
print(list*2)

list[0]=99 #modify value
print(list)
```

### OUTPUT:

```
[10, -20, 15.5, 'VIJAY', 'MARY']
10
[10, -20, 15.5, 'VIJAY', 'MARY']
[-20, 15.5]
VIJAY
[10, -20, 15.5, 'VIJAY', 'MARY', 10, -20, 15.5, 'VIJAY', 'MARY']
```

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

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

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

print list      # Prints complete list
print list[0]   # Prints first element of the list
print list[1:3] # Prints elements starting from 2nd till 3rd
print list[2:]  # Prints elements starting from 3rd element
print tinylist * 2 # Prints list two times
```

```
print list + tinylist # Prints concatenated lists
```

This produce the following result –

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

### ➤ Tuple datatype

A tuple is similar to a list. A tuple contains a group of elements which can be of different types. The elements in the tuple are separated by commas and enclosed in parentheses (). Whereas the list elements can be modified, it is not possible to modify the tuple elements. That means a tuple can be treated as a read-only list.

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

Let's create a tuple as:

```
tpl=(10,-20,15.5,'VIJAY','MARY')

tpl[0]=99

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
tinytuple = (123, 'john')

print tuple           # Prints the complete tuple
print tuple[0]        # Prints first element of the tuple
print tuple[1:3]      # Prints elements of the tuple starting from 2nd till 3rd
print tuple[2:]        # Prints elements of the tuple starting from 3rd element
print tinytuple * 2    # Prints the contents of the tuple twice
print tuple + tinytuple # Prints concatenated tuples
```

This produce the following result –

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

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

```

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tuple[2] = 1000    # Invalid syntax with tuple
list[2] = 1000     # Valid syntax with list

```

### ➤ Range datatype

Range represents a sequence of numbers. The numbers in the range are not modifiable.

Ex :

```

Rg=range(5)          #0 1 2 3 4
Rg=range(10,20,2)    #10,12,14,16,18

```

rg

[0]	0
[1]	1
[2]	2
[3]	3
[4]	4

rg

[-5]	0
[-4]	1
[-3]	2
[-2]	3
[-1]	4

```

>>> rg=range(5)
>>> rg[0]
0
>>> rg[1]
1
>>> rgb=range(10,20,2)
>>> rgb[0]
10
>>> rgb[1]
12
>>> rgb[-1]
18

```

## 4. Sets datatype

A set is an unordered collection of elements much like a set in mathematics. The order of elements is not maintained in the sets. It means the elements may not appear in the same order as they are entered into the set.

A set does not accept duplicate elements.

Sets are unordered so we can not access its element using index.

There are three two types in sets:

✚ set datatype

✚ frozenset datatype

### ➤ set datatype

Sets are represented using curly brackets { }.

Ex :

```
data={10,20,30,"vijay","raj",40}
```

```
data={10,20,30,"vijay","raj",40,10,20}
```

```
>>> data={10,20,30,40,"vijay","raj",10,20}
>>> print(data)
set([40, 10, 'raj', 20, 'vijay', 30])
>>> print(data[0])

Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    print(data[0])
TypeError: 'set' object does not support indexing
```

The set 'data' is not maintaining the order of the elements. We entered the elements in the order 10,20,30,40,vijay,raj. But it is showing another order. Also, we repeated the elements 10 and 20 in the set, but it has stored only one 10 and 20.

We can use the set() function to create a set as:

```
>>> ch=set("hello")
>>> print(ch)
set(['h', 'e', 'l', 'o'])
```

Here, a set 'ch' is created with the characters h,e,l,o. Since a set does not store duplicate elements, it will not store the second 'l'. We can convert a list into a set using the set() function as:

```
>>> lst=[1,2,3,4,5]
>>> s=set(lst)
>>> print(s)
set([1, 2, 3, 4, 5])
```

The update() method is used to add elements to a set as:

```
>>> s.update([500,600])
>>> print(s)
set([1, 2, 3, 4, 5, 500, 600])
```

On the other hand, the remove() method is used to remove any particular element from a set as:

```
>>> s.remove(500)
>>> print(s)
set([1, 2, 3, 4, 5, 600])
```

### ➤ frozenset datatype

The frozenset datatype is same as the set datatype. The main difference is that the elements in the set datatypes can be modified; whereas, the elements of frozenset cannot be modified. We can create a frozenset by passing a set to frozenset() function as:

```
>>> s={50,60,70,80,90}
>>> print(s)
{80, 50, 70, 90, 60}

>>> fs=frozenset(s)
>>> print(fs)
frozenset({80, 50, 70, 90, 60})
```

Another way of creating a frozenset is by passing a string (a group of characters) to the frozenset() function as:

```
>>> fs=frozenset("abcdefg")
>>> print(fs)
frozenset({'d', 'a', 'f', 'g', 'b', 'c', 'e'})
```

However, update() and remove() methods will not work on frozensets since they cannot be modified or updated.

## 5.Mapping types

A map represents a group of elements in the form of key value pairs so that when the key is given, we can retrieve the value associated with it. The dict datatype is an example for a map. The 'dict' represents a 'dictionary' that contains pairs of elements such that the first element represents the key and the next one becomes its value. The key and its value should be separated by a colon(:) and every pair should be separated by a comma. All the elements should be enclosed inside curly brackets{ }.

We can create a dictionary by typing the roll numbers and names of students. Here, roll numbers are keys and names will become value. We write these key value pairs inside curly braces as:

```
d={10:'rahul',11:'rohan',12:'soham'}
```

Here, d is the name of dictionary. 10 is the key and its associated value is 'rahul'. 11 is the key and its associated value is 'rohan'.

We can create an empty dictionary without any elements as:

```
d={}
```

Later, we can store the key and values into d as:

```
d[10]='rahul'
```

```
d[11]='rohan'
```

In the preceding statements, 10 represents the key and 'rahul' its value. 11 represents the key and 'rohan' its value.

```
print(d)
```

It will display the dictionary as:

```
{10:'rahul',11:'rohan'}
```

We can perform various operations on dictionaries. To retrieve value upon giving the key, we can simply mention d[key]. To retrieve only keys from the dictionary, we can use the method keys() and to get only values, we can use the method values().

We can update the value of a key, as : d[key]=newvalue. We can delete a key and corresponding value, using del module, For example del d[11] will delete a key with 11 and its value.

```
>>> d={10: 'RAHUL', 11: 'ROHAN'}
>>> print(d)
{10: 'RAHUL', 11: 'ROHAN'}
>>> print(d[11])
ROHAN
>>> print(d.keys())
[10, 11]
>>> print(d.values())
['RAHUL', 'ROHAN']
>>> d[10]="mit"
>>> print(d)
{10: 'mit', 11: 'ROHAN'}
>>> del d[11]
>>> print(d)
{10: 'mit'}
```

**Boolean datatype**

The bool datatype represents Boolean value True or False, Python internally represents True as 1 and False as 0.

Ex:

True,False

True+True=2

True-False=1

a=10>5

print(a)

O/P : True

a=5>10

print(a)

O/P : False

**Determining the datatype of a variable**

To know the datatype of a variable or object, we can use the type() function .For example, type(a) displays the datatype of the variable

```
>>> a=15
>>> print(type(a))
<class 'int'>
```

Since we are storing an integer number 15 into variable 'a', the type of 'a' is assumed by the python interpreter as 'int'. Observe the last line . It shows class 'int'. It means the variable 'a' is an object of the class 'int'.



```

>>> a=15
>>> print(type(a))
<class 'int'>
>>> a=15.5
>>> print(type(a))
<class 'float'>
>>> s='hello'
>>> print(type(s))
<class 'str'>

```

## Data Type Conversion

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

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

Sr.No.	Function & Description
1	<b>int(x [,base])</b> Converts x to an integer. base specifies the base if x is a string.
2	<b>long(x [,base] )</b> Converts x to a long integer. base specifies the base if x is a string.
3	<b>float(x)</b> Converts x to a floating-point number.
4	<b>complex(real [,imag])</b> Creates a complex number.
5	<b>str(x)</b> Converts object x to a string representation.
6	<b>repr(x)</b> Converts object x to an expression string.
7	<b>eval(str)</b> Evaluates a string and returns an object.
8	<b>tuple(s)</b>

	Converts s to a tuple.
9	<b>list(s)</b> Converts s to a list.
10	<b>set(s)</b> Converts s to a set.
11	<b>dict(d)</b> Creates a dictionary. d must be a sequence of (key,value) tuples.
12	<b>frozenset(s)</b> Converts s to a frozen set.
13	<b>chr(x)</b> Converts an integer to a character.
14	<b>unichr(x)</b> Converts an integer to a Unicode character.
15	<b>ord(x)</b> Converts a single character to its integer value.
16	<b>hex(x)</b> Converts an integer to a hexadecimal string.
17	<b>oct(x)</b> Converts an integer to an octal string.

## Operator

Operators are the constructs which can manipulate the value of operands. Consider the expression  $4 + 5 = 9$ . Here, 4 and 5 are called operands and + is called operator.

### Types of Operator

Python language supports the following types of operators.

- Arithmetic Operators

- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

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

### Arithmetic Operators

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

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	$a + b = 30$
- Subtraction	Subtracts right hand operand from left hand operand.	$a - b = -10$
* Multiplication	Multiplies values on either side of the operator	$a * b = 200$
/ Division	Divides left hand operand by right hand operand	$b / a = 2$
% Modulus	Divides left hand operand by right hand operand and returns remainder	$b \% a = 0$
** Exponent	Performs exponential (power) calculation on operators	$a ** b = 10 \text{ to the power } 20$
//	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 $9.0 // 2.0 = 4.0$ , $-11 // 3 = -4$ , $-11.0 // 3 = -4.0$

### Comparison Operators

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

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

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

### Assignment Operators

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

Operator	Description	Example
----------	-------------	---------

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

### Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. Assume if  $a = 60$ ; and  $b = 13$ ; Now in the binary format their values will be 0011 1100 and 0000 1101 respectively. Following table lists out the bitwise operators supported by Python language with an example each in those, we use the above two variables (a and b) as operands –

$a = 0011\ 1100$

$b = 0000\ 1101$

-----

$a \& b = 0000\ 1100$

$a | b = 0011\ 1101$

$a^b = 0011\ 0001$

$\sim a = 1100\ 0011$

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

### 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	Used to reverse the logical state of its operand.	Not(a and b) is false.

NOT		
-----	--	--

### 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.

### Identity Operators

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

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here <b>is</b> 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 <b>is not</b> results in 1 if id(x) is not equal to id(y).

### Operators Precedence

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

Sr.No.	Operator & Description
--------	------------------------

1	<b>**</b>
	Exponentiation (raise to the power)
2	<b>~ + -</b>
	Complement, unary plus and minus (method names for the last two are <b>+(@)</b> and <b>-(@)</b> )
3	<b>* / % //</b>
	Multiply, divide, modulo and floor division
4	<b>+ -</b>
	Addition and subtraction
5	<b>&gt;&gt; &lt;&lt;</b>
	Right and left bitwise shift
6	<b>&amp;</b>
	Bitwise 'AND'
7	<b>^  </b>
	Bitwise exclusive 'OR' and regular 'OR'
8	<b>&lt;= &lt; &gt; &gt;=</b>
	Comparison operators
9	<b>&lt;&gt; == !=</b>
	Equality operators
10	<b>= %= /= //= -= += *= **=</b>
	Assignment operators
11	<b>is is not</b>
	Identity operators
12	<b>in not in</b>
	Membership operators
13	<b>not or and</b>
	Logical operators