PYTHON

UNIT - I

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

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

History of Python

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

- Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol- 68, SmallTalk, and Unix shell and other scripting languages.
- Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).
- Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.
- Python 1.0 was released in November 1994. In 2000, Python 2.0 was released. Python 2.7.11 is the latest edition of Python 2.
- Meanwhile, Python 3.0 was released in 2008. Python 3 is not backward compatible with Python 2. The emphasis in Python 3 had been on the removal of duplicate programming constructs and modules so that "There should be one -- and preferably only one -- obvious way to do it." Python 3.5.1 is the latest version of Python 3.

Python Features

Python's features include-

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

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

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

There are three different ways to start Python-

(1) Interactive Interpreter

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

Enter **python** the command line.

Start coding right away in the interactive interpreter.

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

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

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

(2) Script from the Command-line

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

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

Note: Be sure the file permission mode allows execution.

(3) Integrated Development Environment

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

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

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

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

First Python Program

Let us execute the programs in different modes of programming.

Interactive Mode Programming

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

```
$ python

Python 3.3.2 (default, Dec 10 2013, 11:35:01) [GCC 4.6.3] on Linux
  Type "help", "copyright", "credits", or "license" for more
  information.

>>>

On Windows:

Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on
win32

Type "copyright", "credits" or "license()" for more information.
```

```
>>>
```

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

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

If you are running the older version of Python (Python 2.x), use of parenthesis as **inprint** function is optional. This produces the following result-

```
Hello, Python!
```

Script Mode Programming

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

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

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

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

On Linux

```
$ python test.py
```

This produces the following result-

```
Hello, Python!
```

On Windows

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

This produces the following result-

```
Hello, Python!
```

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

```
#!/usr/bin/python3

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

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

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

This produces the following result-

```
Hello, Python!
```

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

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

Here are naming conventions for Python identifiers-

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

Reserved Words

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

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

Python does not use braces({}) to indicate blocks of code for class and function dPefyitnhiotinon3s or

flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

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

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

However, the following block generates an error-

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

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

```
#!/usr/bin/python3 import sys                                                    Python 3

try:

# open file stream

file = open(file_name, "w") except IOError:
    print ("There was an error writing to", file_name)
  sys.exit() print ("Enter '", file_finish,) print "' When
  finished"
  while file_text != file_finish: file_text = raw_input("Enter text: ") if file_text ==
      file_finish: # close the file file.close
      break file.write(file_text)
  file.write("\n") file.close()

file_name = input("Enter filename: ") if len(file_name) ==

  0: print ("Next time please enter something")

sys.exit() try:
file = open(file_name, "r") except IOError:
    print ("There was an error reading file")
  sys.exit() file_text = file.read() file.close()
print (file_text)
```

Multi-Line Statements

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

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

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

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

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

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

```
word = 'word'

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

Comments in Python

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

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

This produces the following result-

```
Hello, Python!
```

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

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

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

```
# This is a comment.

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

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

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

Waiting for the User

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

```
#!/usr/bin/python3

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

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

Multiple Statements on a Single Line

The semicolon (;) allows multiple statements on a single line given that no statement starts a new code block. Here is a sample snip using the semicolon-

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

Multiple Statement Groups as Suites

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

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

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

Command Line Arguments

Many programs can be run to provide you with some basic information about how they should be run. Python enables you to do this with **-h**:

```
$ python -h

usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ... Options and arguments
(and corresponding environment variables):
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser (also PYTHONDEBUG=x)
-E     : ignore environment variables (such as PYTHONPATH)
-h     : print this help message and exit [ etc. ]
```

You can also program your script in such a way that it should accept various options. Command Line Arguments is an advance topic. Let us understand it.

Command Line Arguments

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

```
$ python test.py arg1 arg2 arg3
```
Python 3

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

- **sys.argv** is the list of command-line arguments.
    - **len(sys.argv)** is the number of command-line arguments. Here sys.argv[0] is the program i.e. the script name.

Example

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

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

Now run the above script as follows −

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

This produces the following result-

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

Variables

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

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

Assigning Values to Variables

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

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

```
#!/usr/bin/python3

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

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

```
100
1000.0
John
```

Multiple Assignment

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

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

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

For example-

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

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

Standard Data Types

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

Python has five standard data types-

- Numbers
- String
- List
- Tuple
- Dictionary

Python Numbers

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

```
var1 = 1

var2 = 10
```

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

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

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

```
del var

del var_a, var_b
```

Python supports three different numerical types −

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

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

Examples

Here are some examples of numbers-

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

A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x and y are real numbers and j is the imaginary unit.

Python Strings

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

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

```
#!/usr/bin/python3 str = 'Hello World!'

print (str)        # Prints complete string

print (str[0])     # Prints first character of the string
print (str[2:5])   # Prints characters starting from 3rd to 5th print (str[2:])      # string starting from 3rd character print (str * 2)        # Prints string two times
print (str + "TEST") # Prints concatenated string
```

This will produce the following result-

```
Hello World!

H

llo

llo World!

Hello World!Hello World! Hello World!TEST
```

Python Lists

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

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

```
#!/usr/bin/python3

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

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

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

This produces the following result-

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

abcd

[786, 2.23]

[2.23, 'john', 70.200000000000003]

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

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

Python Tuples

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

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

```
#!/usr/bin/python3

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 ) tinytuple = (123,
'john') print (tuple)      # Prints complete tuple

print (tuple[0])   # Prints first element of the tuple
print (tuple[1:3]) # Prints elements starting from 2nd till 3rd print (tuple[2:]) # Prints elements starting from 3rd element print (tinytuple * 2)       # Prints tuple two times

print (tuple + tinytuple) # Prints concatenated tuple
```

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

abcd

(786, 2.23)

(2.23, 'john', 70.200000000000003)

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

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

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

```
#!/usr/bin/python3

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ] tuple[2] =
1000 # Valid syntax with list
                                        # Invalid syntax with tuple
                                        list[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/python3 dict = {}

dict['one'] = "This is one" dict[2]      = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'} print
(dict['one'])                                   # Prints value for
'one' key print (dict[2])          # Prints value for 2 key

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

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

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

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

Data Type Conversion

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

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

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

Operators

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

Types of Operator

Python language supports the following types of operators-

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

Let us have a look at all the operators one by one.

Python Arithmetic Operators

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

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

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

```
#!/usr/bin/python3 a = 21
b = 10
c = 0
c = a + b
print ("Line 1 - Value of c is ", c) c = a -
b
print ("Line 2 - Value of c is ", c ) c = a
* b
print ("Line 3 - Value of c is ", c) c = a /
b
print ("Line 4 - Value of c is ", c ) c = a
% b
print ("Line 5 - Value of c is ", c) a = 2
b = 3
c = a**b
print ("Line 6 - Value of c is ", c) a = 10
b = 5
c = a//b
print ("Line 7 - Value of c is ", c)
```

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

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

Python Comparison Operators

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

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

Example

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

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

```
print ("Line 1 - a is not equal to b")                                    Python 3


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


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


if ( a > b ):
print ("Line 4 - a is greater than b") else:
print ("Line 4 - a is not greater than b")
  a,b=b,a #values of a and b swapped. a becomes 10, b becomes 21 if ( a <=
      b ): print ("Line 5 - a is either less than or equal to b") else:
print ("Line 5 - a is neither less than nor equal to b")


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

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

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

  Line 6 - b is either greater than or equal to b
```

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

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

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

```
#!/usr/bin/python
  3 a = 21
b = 10
c = 0
c = a + b
print ("Line 1 - Value of c is ",
  c) c += a
print ("Line 2 - Value of c is ", c
  ) c *= a
print ("Line 3 - Value of c is ", c
  ) c /= a
print ("Line 4 - Value of c is ", c
  ) c = 2 c %= a
print ("Line 5 - Value of c is ",
  c) c **= a
print ("Line 6 - Value of c is ",
  c) c //= a
print ("Line 7 - Value of c is ", c)
```

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

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

Python Bitwise Operators

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

a = 0011 1100

b = 0000 1101

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

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

The following Bitwise operators are supported by Python language-

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

```
#!/usr/bin/python3

a = 60  # 60 = 0011 1100

b =                     # 13 = 0000
13                      1101
print ('a=',a,':',bin(a),'b=',b,':',bin(b)) c
= 0
c = a                   # 12 = 0000
& b;                    1100
print ("result of AND is ",

c,':',bin(c)) c = a | b;    # 61 = 0011

1101

print ("result of OR is ",
c = a                   # 49 = 0011
c,':',bin(c))           0001
print ("result of EXOR is ",

c,':',bin(c)) c = ~a;       # -61 = 1100

0011

print ("result of COMPLEMENT is ",

c,':',bin(c)) c = a << 2;# 240 = 1111 0000

print ("result of LEFT SHIFT is ",

c,':',bin(c)) c = a >> 2;# 15 = 0000 1111

print ("result of RIGHT SHIFT is ",
c,':',bin(c))
```

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

```
a= 60 : 0b111100 b= 13 : 0b1101

result of AND is 12 : 0b1100 result of OR is 61 : 0b111101 result of EXOR is 49 :

 0b110001

result of COMPLEMENT is -61 : -0b111101 result of LEFT SHIFT is 240 : 0b11110000

 result of RIGHT SHIFT is 15 : 0b111
```

Python Logical Operators

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

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

Python Membership Operators

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

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

```
#!/usr/bin/python3 a =
10

b = 20

list = [1, 2, 3, 4, 5 ]
if ( a in list ):

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

   print ("Line 2 - b is not available in the given list") else: print
   ("Line 2 - b is available in the given list")
c=b/a

if ( c in list ):

   print ("Line 3 - a is available in the given list") else: print
("Line 3 - a is not available in the given list")
```

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

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

Python Identity Operators

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

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

Example

```
#!/usr/bin/python3 a =
20

b = 20

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

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

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

b = 30

print ('Line 4','a=',a,':',id(a), 'b=',b,':',id(b)) if ( a is
not b ):

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

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

```
Line 1 a= 20 : 1594701888 b= 20 : 1594701888

Line 2 - a and b have same identity Line 3 - a and b have same identity
  Line 4 a= 20 : 1594701888 b= 30 : 1594702048

Line 5 - a and b do not have same identity
```

Python Operators Precedence

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

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

Operator precedence affects the evaluation of an an expression.

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

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

```
#!/usr/bin/pyth
on3 a = 20

b = 10

c = 15

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


e = ((a + b) * c)              # (30 * 15 ) / 5 print ("Value of ((a + b) * c) / d is ",
/ d                            e)
e = (a + b) * (c /             # (30) * (15/5) print ("Value of (a + b) * (c / d) is ",
d)                             e)
e = a + (b * c) / d # 20 + (150/5) print ("Value of a + (b * c) / d is ",
e)
```

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

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

Value of (a + b) * (c / d) is 90.0 Value of a + (b * c) / d is 50.0
```

Expression

It is a combination of operators and operands that is interpreted to produce some other value. In any programming language, an expression is evaluated as per the precedence of its operators. So that if there is more than one operator in an expression, their precedence decides which operation will be performed first. We have many different types of expressions in Python. Let's discuss all types along with some exemplar codes :

```
# Constant Expressions

x = 15 + 1.3

print(x)
```

Output
16.3

2. Arithmetic Expressions: An arithmetic expression is a combination of numeric values, operators, and sometimes parenthesis. The result of this type of expression is also a numeric value. The operators used in these expressions are arithmetic operators like addition, subtraction, etc. Here are some arithmetic operators in Python:

Operators	**Syntax**	**Functioning**
+	x + y	Addition
–	x – y	Subtraction
*	x * y	Multiplication
/	x / y	Division
//	x // y	Quotient
%	x % y	Remainder
**	x ** y	Exponentiation

Example:
Let's see an exemplar code of arithmetic expressions in Python :

- Python3

```
# Arithmetic Expressions

x = 40

y = 12



add = x + y

sub = x - y
```

```
pro = x * y

div = x / y



print(add)

print(sub)

print(pro)

print(div)
```

Output
52

28

480

3.3333333333333335

3. Integral Expressions: These are the kind of expressions that produce only integer results after all computations and type conversions.
Example:

- Python3

```
# Integral Expressions

a = 13

b = 12.0



c = a + int(b)

print(c)
```

Output
25

4. Floating Expressions: These are the kind of expressions which produce floating point numbers as result after all computations and type conversions.
Example:

- Python3

```
# Floating Expressions

a = 13

b = 5



c = a / b
```

```
print(c)
```

Output
2.6

5. Relational Expressions: In these types of expressions, arithmetic expressions are written on both sides of relational operator (> , < , >= , <=). Those arithmetic expressions are evaluated first, and then compared as per relational operator and produce a boolean output in the end. These expressions are also called Boolean expressions.
Example:

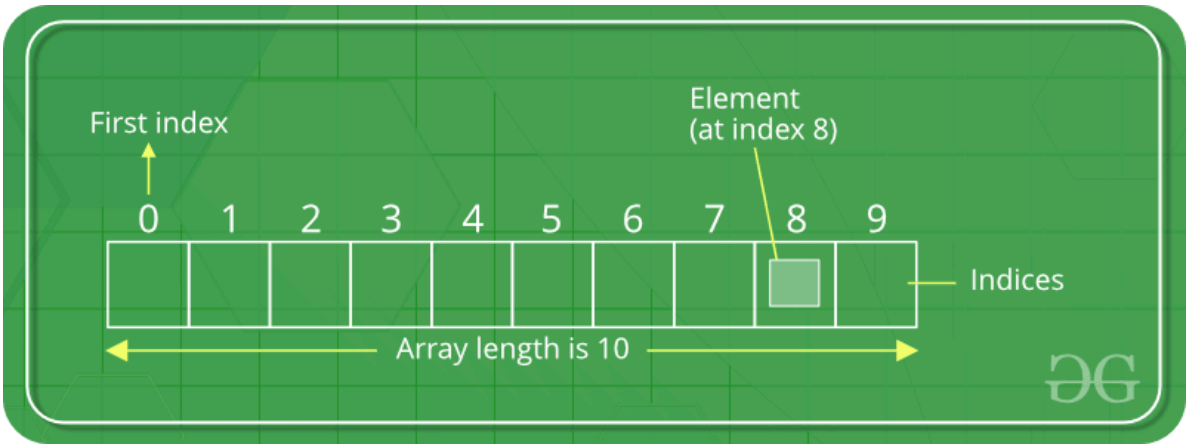
- Python3

```
# Relational Expressions

a = 21

b = 13

c = 40

d = 37



p = (a + b) >= (c - d)

print(p)
```

Output
True

Python Arrays

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).



Access the Elements of an Array

You refer to an array element by referring to the *index number*.

Example

Get the value of the first array item:

```
x = cars[0]
```

The Length of an Array

Use the len() method to return the length of an array (the number of elements in an array).

Example

Return the number of elements in the cars array:

```
x = len(cars)
```

Try it Yourself »

Adding Array Elements

You can use the append() method to add an element to an array.

Example

Add one more element to the cars array:

```
cars.append("Honda")
```

Removing Array Elements

You can use the pop() method to remove an element from the array.

Example

Delete the second element of the cars array:

```
cars.pop(1)
```

Array Methods

Python has a set of built-in methods that you can use on lists/arrays.

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position

pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list