**PYTHON TUTORIAL**

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.

## 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 3.3.2 (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 the Enter:

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

If you are running older version of Python (Python 2.x), use of parenthesis as in**print** 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 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 −

**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 /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!

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 |
| 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 |  |  |

## 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/python3

# First comment

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

This produces the following result −

Hello, Python!

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

name = "Madisetti" # This is again comment

## 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

## [Python Strings](http://www.tutorialspoint.com/python_strings.htm)

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 −

#!/usr/bin/python3

str = 'Hello World!'

print (str) # Prints complete string

print (str[0]) # Prints first character of the string

print (str[2:5]) # Prints characters starting from 3rd to 5th

print (str[2:]) # Prints string starting from 3rd character

print (str \* 2) # Prints string two times

print (str + "TEST") # Prints concatenated string

This will produce the following result −

Hello World!

H

llo

llo World!

Hello World!Hello World!

Hello World!TEST

## [Python Lists](http://www.tutorialspoint.com/python_lists.htm)

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 −

#!/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 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']

## [Python Tuples](http://www.tutorialspoint.com/python_tuples.htm)

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

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

#!/usr/bin/python3

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

tinytuple = (123, 'john')

print (tuple) # Prints complete tuple

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

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

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

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

print (tuple + tinytuple) # Prints concatenated tuple

This 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/python3

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

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

tuple[2] = 1000 # Invalid syntax with tuple

list[2] = 1000 # Valid syntax with list

## [Python Dictionary](http://www.tutorialspoint.com/python_dictionary.htm)

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 produce the following result −

This is one

This is two

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

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

['sales', 6734, 'john']

## 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

# Arithmetic Operators Example

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

## Example

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

#!/usr/bin/python3

a = 21

b = 10

c = 0

c = a + b

print ("Line 1 - Value of c is ", c)

c = a - b

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

c = a \* b

print ("Line 3 - Value of c is ", c)

c = a / b

print ("Line 4 - Value of c is ", c )

c = a % b

print ("Line 5 - Value of c is ", c)

a = 2

b = 3

c = a\*\*b

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

a = 10

b = 5

c = a//b

print ("Line 7 - Value of c is ", c)

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

Line 1 - Value of c is 31

Line 2 - Value of c is 11

Line 3 - Value of c is 210

Line 4 - Value of c is 2.1

Line 5 - Value of c is 1

Line 6 - Value of c is 8

Line 7 - Value of c is 2

# Comparison Operator

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

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

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | If the values of two operands are equal, then the condition becomes true. | (a == b) is not true. |
| != | If values of two operands are not equal, then condition becomes true. |  |
| <> | If 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. |

Example

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

#!/usr/bin/python3

a = 21

b = 10

if ( a == b ):

print ("Line 1 - a is equal to b")

else:

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

if ( a != b ):

print ("Line 2 - a is not equal to b")

else:

print ("Line 2 - a is equal to b")

if ( a < b ):

print ("Line 3 - a is less than b" )

else:

print ("Line 3 - a is not less than b")

if ( a > b ):

print ("Line 4 - a is greater than b")

else:

print ("Line 4 - a is not greater than b")

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

if ( a <= b ):

print ("Line 5 - a is either less than or equal to b")

else:

print ("Line 5 - a is neither less than nor equal to b")

if ( b >= a ):

print ("Line 6 - b is either greater than or equal to b")

else:

print ("Line 6 - b is neither greater than nor equal to b")

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

Line 1 - a is not equal to b

Line 2 - a is not equal to b

Line 3 - a is not less than b

Line 4 - a is greater than b

Line 5 - a is either less than or equal to b

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

# Assignment Operators

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

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

### Example

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

#!/usr/bin/python3

a = 21

b = 10

c = 0

c = a + b

print ("Line 1 - Value of c is ", c)

c += a

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

c \*= a

print ("Line 3 - Value of c is ", c )

c /= a

print ("Line 4 - Value of c is ", c )

c = 2

c %= a

print ("Line 5 - Value of c is ", c)

c \*\*= a

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

c //= a

print ("Line 7 - Value of c is ", c)

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

Line 1 - Value of c is 31

Line 2 - Value of c is 52

Line 3 - Value of c is 1092

Line 4 - Value of c is 52.0

Line 5 - Value of c is 2

Line 6 - Value of c is 2097152

Line 7 - Value of c is 99864

BITWISE OPERATOR

|  |  |  |
| --- | --- | --- |
| **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 << = 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 >> = 15 (means 0000 1111) |

### Example

#!/usr/bin/python3

a = 60 # 60 = 0011 1100

b = 13 # 13 = 0000 1101

print ('a=',a,':',bin(a),'b=',b,':',bin(b))

c = 0

c = a & b; # 12 = 0000 1100

print ("result of AND is ", c,':',bin(c))

c = a | b; # 61 = 0011 1101

print ("result of OR is ", c,':',bin(c))

c = a ^ b; # 49 = 0011 0001

print ("result of EXOR is ", c,':',bin(c))

c = ~a; # -61 = 1100 0011

print ("result of COMPLEMENT is ", c,':',bin(c))

c = a << 2; # 240 = 1111 0000

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

c = a >> 2; # 15 = 0000 1111

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

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

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

result of AND is 12 : 0b1100

result of OR is 61 : 0b111101

result of EXOR is 49 : 0b110001

result of COMPLEMENT is -61 : -0b111101

result of LEFT SHIFT is 240 : 0b11110000

result of RIGHT SHIFT is 15 : 0b111