**Python**

Python is a cross-platform programming language, meaning, it runs on multiple platforms like Windows, Mac OS X, Linux, Unix and has even been ported to the Java and .NET virtual machines. It is free and open source.

**Starting The Interpreter**

After installation, the python interpreter lives in the installed directory.

By default it is /usr/local/bin/pythonX.X in Linux/Unix and C:\PythonXX in Windows, where the 'X' denotes the version number. To invoke it from the shell or the command prompt we need to add this location in the search path.

Search path is a list of directories (locations) where the operating system searches for executables. For example, in Windows command prompt, we can type set path=%path%;c:\python33 (python33 means version 3.3, it might be different in your case) to add the location to path for that particular session.

In Mac OS we need not worry about this as the installer takes care about the search path.

Now there are various ways to start Python.

**1. Immediate mode**

Typing python in the command line will invoke the interpreter in immediate mode. We can directly type in Python expressions and press enter to get the output.

>>>

is the Python prompt. It tells us that the interpreter is ready for our input. Try typing in 1 + 1 and press enter. We get 2 as the output. This prompt can be used as a calculator. To exit this mode type exit() or quit() and press enter.

**2. Script mode**

This mode is used to execute Python program written in a file. Such a file is called a **script**. Scripts can be saved to disk for future use. Python scripts have the extension .py, meaning that the filename ends with .py.

For example: helloWorld.py

To execute this file in script mode we simply write python helloWorld.py at the command prompt.

**Integrated Development Environment (IDE)**

We can use any text editing software to write a Python script file.

We just need to save it with the .py extension. But using an IDE can make our life a lot easier. IDE is a piece of software that provides useful features like code hinting, syntax highlighting and checking, file explorers etc. to the programmer for application development.

Using an IDE can get rid of redundant tasks and significantly decrease the time required for application development.

IDEL is a graphical user interface (GUI) that can be installed along with the Python programming language and is available from the official website.

We can also use other commercial or free IDE according to our preference. We have used the PyScripter IDE for our testing and we recommend the same. It is free and open source.

**Python Keywords**

Keywords are the reserved words in Python.

We cannot use a keyword as variable name, function name or any other identifier. They are used to define the syntax and structure of the Python language.

In Python, keywords are case sensitive.

There are 33 keywords in Python 3.3. This number can vary slightly in course of time.

All the keywords except True, False and None are in lowercase and they must be written as it is. The list of all the keywords are given below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Keywords in Python programming language | | | | |
| False | Class | Finally | is | return |
| None | continue | For | lambda | Try |
| True | Def | From | nonlocal | while |
| And | Del | Global | not | With |
| As | Elif | If | or | yield |
| Assert | Else | Import | pass |  |
| Break | Except | In | raise |  |

Looking at all the keywords at once and trying to figure out what they mean might be overwhelming.

If you want to have an overview, here is the complete list of all the keywords with examples.

**Python Identifiers**

Identifier is the name given to entities like class, functions, variables etc. in Python. It helps differentiating one entity from another.

**Rules for writing identifiers**

1. Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (\_). Names like myClass, var\_1 and print\_this\_to\_screen, all are valid example.
2. An identifier cannot start with a digit. 1variable is invalid, but variable1 is perfectly fine.
3. Keywords cannot be used as identifiers.
4. >>> global = 1
5. File "<interactive input>", line 1
6. global = 1
7. ^

SyntaxError: invalid syntax

1. We cannot use special symbols like !, @, #, $, % etc. in our identifier.
2. >>> a@ = 0
3. File "<interactive input>", line 1
4. a@ = 0
5. ^

SyntaxError: invalid syntax

1. Identifier can be of any length.

**Things to care about**

Python is a case-sensitive language. This means, Variable and variable are not the same. Always name identifiers that make sense.

While, c = 10 is valid. Writing count = 10 would make more sense and it would be easier to figure out what it does even when you look at your code after a long gap.

Multiple words can be separated using an underscore, this\_is\_a\_long\_variable.

We can also use camel-case style of writing, i.e., capitalize every first letter of the word except the initial word without any spaces. For example: camelCaseExample

**Python Statement**

Instructions that a Python interpreter can execute are called statements. For example, a = 1is an assignment statement. if statement, for statement, while statement etc. are other kinds of statements which will be discussed later.

**Multi-line statement**

In Python, end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line continuation character (\). For example:

a = 1 + 2 + 3 + \

4 + 5 + 6 + \

7 + 8 + 9

This is explicit line continuation. In Python, line continuation is implied inside parentheses ( ), brackets [ ] and braces { }. For instance, we can implement the above multi-line statement as

a = (1 + 2 + 3 +

4 + 5 + 6 +

7 + 8 + 9)

Here, the surrounding parentheses ( ) do the line continuation implicitly. Same is the case with [ ] and { }. For example:

colors = ['red',

'blue',

'green']

We could also put multiple statements in a single line using semicolons, as follows

a = 1; b = 2; c = 3

**Python Indentation**

Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation.

A code block (body of a function, loop etc.) starts with indentation and ends with the first unindented line. The amount of indentation is up to you, but it must be consistent throughout that block.

Generally four whitespaces are used for indentation and is preferred over tabs. Here is an example.

for i in range(1,11):

print(i)

if i == 5:

break

The enforcement of indentation in Python makes the code look neat and clean. This results into Python programs that look similar and consistent.

Indentation can be ignored in line continuation. But it's a good idea to always indent. It makes the code more readable. For example:

if True:

print('Hello')

a = 5

and

if True: print('Hello'); a = 5

both are valid and do the same thing. But the former style is clearer.

Incorrect indentation will result into IndentationError.

**Python Comments**

Comments are very important while writing a program. It describes what's going on inside a program so that a person looking at the source code does not have a hard time figuring it out. You might forget the key details of the program you just wrote in a month's time. So taking time to explain these concepts in form of comments is always fruitful.

In Python, we use the hash (#) symbol to start writing a comment.

It extends up to the newline character. Comments are for programmers for better understanding of a program. Python Interpreter ignores comment.

#This is a comment

#print out Hello

print('Hello')

**Multi-line comments**

If we have comments that extend multiple lines, one way of doing it is to use hash (#) in the beginning of each line. For example:

#This is a long comment

#and it extends

#to multiple lines

Another way of doing this is to use triple quotes, either ''' or """.

These triple quotes are generally used for multi-line strings. But they can be used as multi-line comment as well. Unless they are not docstrings, they do not generate any extra code.

"""This is also a

perfect example of

multi-line comments"""

**Docstring in Python**

Docstring is short for documentation string.

It is a [string](https://www.programiz.com/python-programming/string) that occurs as the first statement in a module, function, class, or method definition. We must write what a function/class does in the docstring.

Triple quotes are used while writing docstrings. For example:

def double(num):

"""Function to double the value"""

return 2\*num

Docstring is available to us as the attribute \_\_doc\_\_ of the function. Issue the following code in shell once you run the above program.

>>> print(double.\_\_doc\_\_)

Function to double the value

**Python Variables**

A variable is a location in memory used to store some data (value).

They are given unique names to differentiate between different memory locations. The rules for writing a variable name is same as the rules for writing identifiers in Python.

We don't need to declare a variable before using it. In Python, we simply assign a value to a variable and it will exist. We don't even have to declare the type of the variable. This is handled internally according to the type of value we assign to the variable.

**Variable assignment**

We use the assignment operator (=) to assign values to a variable. Any type of value can be assigned to any valid variable.

a = 5

b = 3.2

c = "Hello"

Here, we have three assignment statements. 5 is an integer assigned to the variable a.

Similarly, 3.2 is a floating point number and "Hello" is a string (sequence of characters) assigned to the variables b and c respectively.

**Multiple assignments**

In Python, multiple assignments can be made in a single statement as follows:

a, b, c = 5, 3.2, "Hello"

If we want to assign the same value to multiple variables at once, we can do this as

x = y = z = "same"

This assigns the "same" string to all the three variables.

**Data types in Python**

Every value in Python has a datatype. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes.

There are various data types in Python. Some of the important types are listed below.

**Python Numbers**

Integers, floating point numbers and complex numbers falls under Python numbers category. They are defined as int, float and complex class in Python.

We can use the type() function to know which class a variable or a value belongs to and the isinstance() function to check if an object belongs to a particular class.

a = 5

print(a, "is of type", type(a))

a = 2.0

print(a, "is of type", type(a))

a = 1+2j

print(a, "is complex number?", isinstance(1+2j,complex))

Integers can be of any length, it is only limited by the memory available.

A floating point number is accurate up to 15 decimal places. Integer and floating points are separated by decimal points. 1 is integer, 1.0 is floating point number.

Complex numbers are written in the form, x + yj, where x is the real part and y is the imaginary part. Here are some examples.

>>> a = 1234567890123456789

>>> a

1234567890123456789

>>> b = 0.1234567890123456789

>>> b

0.12345678901234568

>>> c = 1+2j

>>> c

(1+2j)

Notice that the float variable b got truncated.

**Python List**

List is an ordered sequence of items. It is one of the most used datatype in Python and is very flexible. All the items in a list do not need to be of the same type.

Declaring a list is pretty straight forward. Items separated by commas are enclosed within brackets [ ].

>>> a = [1, 2.2, 'python']

We can use the slicing operator [ ] to extract an item or a range of items from a list. Index starts form 0 in Python.

a = [5,10,15,20,25,30,35,40]

# a[2] = 15

print("a[2] = ", a[2])

# a[0:3] = [5, 10, 15]

print("a[0:3] = ", a[0:3])

# a[5:] = [30, 35, 40]

print("a[5:] = ", a[5:])

Lists are mutable, meaning, value of elements of a list can be altered.

>>> a = [1,2,3]

>>> a[2]=4

>>> a

[1, 2, 4]

**Python Tuple**

Tuple is an ordered sequence of items same as list.The only difference is that tuples are immutable. Tuples once created cannot be modified.

Tuples are used to write-protect data and are usually faster than list as it cannot change dynamically.

It is defined within parentheses () where items are separated by commas.

>>> t = (5,'program', 1+3j)

We can use the slicing operator [] to extract items but we cannot change its value.

0t = (5,'program', 1+3j)

# t[1] = 'program'

print("t[1] = ", t[1])

# t[0:3] = (5, 'program', (1+3j))

print("t[0:3] = ", t[0:3])

# Generates error

# Tuples are immutable

t[0] = 10

**Python Strings**

String is sequence of Unicode characters. We can use single quotes or double quotes to represent strings. Multi-line strings can be denoted using triple quotes, ''' or """.

>>> s = "This is a string"

>>> s = '''a multiline

Like list and tuple, slicing operator [ ] can be used with string. Strings are immutable.

s = 'Hello world!'

# s[4] = 'o'

print("s[4] = ", s[4])

# s[6:11] = 'world'

print("s[6:11] = ", s[6:11])

# Generates error

# Strings are immutable in Python

s[5] ='d'

**Python Set**

Set is an unordered collection of unique items. Set is defined by values separated by comma inside braces { }. Items in a set are not ordered.

a = {5,2,3,1,4}

# printing set variable

print("a = ", a)

# data type of variable a

print(type(a))

We can perform set operations like union, intersection on two sets. Set have unique values. They eliminate duplicates.

>>> a = {1,2,2,3,3,3}

>>> a

{1, 2, 3}

Since, set are unordered collection, indexing has no meaning. Hence the slicing operator [] does not work.

>>> a = {1,2,3}

>>> a[1]

Traceback (most recent call last):

File "<string>", line 301, in runcode

File "<interactive input>", line 1, in <module>

TypeError: 'set' object does not support indexing

**Python Dictionary**

Dictionary is an unordered collection of key-value pairs.

It is generally used when we have a huge amount of data. Dictionaries are optimized for retrieving data. We must know the key to retrieve the value.

In Python, dictionaries are defined within braces {} with each item being a pair in the form key:value. Key and value can be of any type.

>>> d = {1:'value','key':2}

>>> type(d)

<class 'dict'>

We use key to retrieve the respective value. But not the other way around.

d = {1:'value','key':2}

print(type(d))

print("d[1] = ", d[1]);

print("d['key'] = ", d['key']);

# Generates error

print("d[2] = ", d[2]);

### Conversion between data types

We can convert between different data types by using different type conversion functions like int(), float(), str() etc.

>>> float(5)

5.0

Conversion from float to int will truncate the value (make it closer to zero).

>>> int(10.6)

10

>>> int(-10.6)

-10

Conversion to and from string must contain compatible values.

>>> float('2.5')

2.5

>>> str(25)

'25'

>>> int('1p')

Traceback (most recent call last):

File "<string>", line 301, in runcode

File "<interactive input>", line 1, in <module>

ValueError: invalid literal for int() with base 10: '1p'

We can even convert one sequence to another.

>>> set([1,2,3])

{1, 2, 3}

>>> tuple({5,6,7})

(5, 6, 7)

>>> list('hello')

['h', 'e', 'l', 'l', 'o']

To convert to dictionary, each element must be a pair

>>> dict([[1,2],[3,4]])

{1: 2, 3: 4}

>>> dict([(3,26),(4,44)])

{3: 26, 4: 44}

Python provides numerous built-in functions that are readily available to us at the Python prompt.

Some of the functions like input() and print() are widely used for standard input and output operations respectively. Let us see the output section first.

## Python Output Using print() function

We use the print() function to output data to the standard output device (screen).

We can also output data to a file, but this will be discussed later. An example use is given below.

print('This sentence is output to the screen')

# Output: This sentence is output to the screen

a = 5

print('The value of a is', a)

# Output: The value of a is 5

In the second print() statement, we can notice that a space was added between the [string](https://www.programiz.com/python-programming/string)and the value of variable a.This is by default, but we can change it.

The actual syntax of the print() function is

print(\*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

Here, objects is the value(s) to be printed.

The sep separator is used between the values. It defaults into a space character.

After all values are printed, end is printed. It defaults into a new line.

The file is the object where the values are printed and its default value is sys.stdout(screen). Here are an example to illustrate this.

print(1,2,3,4)

# Output: 1 2 3 4

print(1,2,3,4,sep='\*')

# Output: 1\*2\*3\*4

print(1,2,3,4,sep='#',end='&')

# Output: 1#2#3#4&

### Output formatting

Sometimes we would like to format our output to make it look attractive. This can be done by using the str.format() method. This method is visible to any string object.

>>> x = 5; y = 10

>>> print('The value of x is {} and y is {}'.format(x,y))

The value of x is 5 and y is 10

Here the curly braces {} are used as placeholders. We can specify the order in which it is printed by using numbers (tuple index).

print('I love {0} and {1}'.format('bread','butter'))

# Output: I love bread and butter

print('I love {1} and {0}'.format('bread','butter'))

# Output: I love butter and bread

We can even use keyword arguments to format the string.

>>> print('Hello {name}, {greeting}'.format(greeting = 'Goodmorning', name = 'Joy'))

Hello Joy, Goodmorning

We can even format strings like the old sprintf() style used in C programming language. We use the % operator to accomplish this.

>>> x = 12.3456789

>>> print('The value of x is %3.2f' %x)

The value of x is 12.35

>>> print('The value of x is %3.4f' %x)

The value of x is 12.3457

## Python Input

Up till now, our programs were static. The value of variables were defined or hard coded into the source code.

To allow flexibility we might want to take the input from the user. In Python, we have the input() function to allow this. The syntax for input() is

input([prompt])

where prompt is the string we wish to display on the screen. It is optional.

>>> num = input('Enter a number: ')

Enter a number: 10

>>> num

'10'

Here, we can see that the entered value 10 is a string, not a number. To convert this into a number we can use int() or float() functions.

>>> int('10')

10

>>> float('10')

10.0

This same operation can be performed using the eval() function. But it takes it further. It can evaluate even expressions, provided the input is a string

>>> int('2+3')

Traceback (most recent call last):

File "<string>", line 301, in runcode

File "<interactive input>", line 1, in <module>

ValueError: invalid literal for int() with base 10: '2+3'

>>> eval('2+3')

5

## Python Import

When our program grows bigger, it is a good idea to break it into different modules.

A module is a file containing Python definitions and statements. Python modules have a filename and end with the extension .py.

Definitions inside a module can be imported to another module or the interactive interpreter in Python. We use the import keyword to do this.

For example, we can import the math module by typing in import math.

import math

print(math.pi)

Now all the definitions inside math module are available in our scope. We can also import some specific attributes and functions only, using the from keyword. For example:

>>> from math import pi

>>> pi

3.141592653589793

While importing a module, Python looks at several places defined in sys.path. It is a list of directory locations.

>>> import sys

>>> sys.path

['',

'C:\\Python33\\Lib\\idlelib',

'C:\\Windows\\system32\\python33.zip',

'C:\\Python33\\DLLs',

'C:\\Python33\\lib',

'C:\\Python33',

'C:\\Python33\\lib\\site-packages']

**Arithmetic operators**

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication etc.

|  |  |  |
| --- | --- | --- |
| Arithmetic operators in Python | | |
| Operator | Meaning | Example |
| + | Add two operands or unary plus | x + y +2 |
| - | Subtract right operand from the left or unary minus | x - y -2 |
| \* | Multiply two operands | x \* y |
| / | Divide left operand by the right one (always results into float) | x / y |
| % | Modulus - remainder of the division of left operand by the right | x % y (remainder of x/y) |
| // | Floor division - division that results into whole number adjusted to the left in the number line | x // y |
| \*\* | Exponent - left operand raised to the power of right | x\*\*y (x to the power y) |

**Example #1: Arithmetic operators in Python**

x = 15

y = 4

# Output: x + y = 19

print('x + y =',x+y)

# Output: x - y = 11

print('x - y =',x-y)

# Output: x \* y = 60

print('x \* y =',x\*y)

# Output: x / y = 3.75

print('x / y =',x/y)

# Output: x // y = 3

print('x // y =',x//y)

# Output: x \*\* y = 50625

print('x \*\* y =',x\*\*y)

When you run the program, the output will be:

x + y = 19

x - y = 11

x \* y = 60

x / y = 3.75

x // y = 3

x \*\* y = 50625

**Comparison operators**

Comparison operators are used to compare values. It either returns True or False according to the condition.

|  |  |  |
| --- | --- | --- |
| Comparision operators in Python | | |
| Operator | Meaning | Example |
| > | Greater that - True if left operand is greater than the right | x > y |
| < | Less that - True if left operand is less than the right | x < y |
| == | Equal to - True if both operands are equal | x == y |
| != | Not equal to - True if operands are not equal | x != y |
| >= | Greater than or equal to - True if left operand is greater than or equal to the right | x >= y |
| <= | Less than or equal to - True if left operand is less than or equal to the right | x <= y |

**Example #2: Comparison operators in Python**

x = 10

y = 12

# Output: x > y is False

print('x > y is',x>y)

# Output: x < y is True

print('x < y is',x<y)

# Output: x == y is False

print('x == y is',x==y)

# Output: x != y is True

print('x != y is',x!=y)

# Output: x >= y is False

print('x >= y is',x>=y)

# Output: x <= y is True

print('x <= y is',x<=y)

**Logical operators**

Logical operators are the and, or, not operators.

|  |  |  |
| --- | --- | --- |
| Logical operators in Python | | |
| Operator | Meaning | Example |
| And | True if both the operands are true | x and y |
| Or | True if either of the operands is true | x or y |
| Not | True if operand is false (complements the operand) | not x |
| **and, or , not**  and, or, not are the logical operators in Python. and will result into True only if both the operands are True. The truth table for and is given below:   |  |  |  | | --- | --- | --- | | Truth table for and | | | | A | B | A and B | | True | True | True | | True | False | False | | False | True | False | | False | False | False |   or will result into True if any of the operands is True. The truth table for or is given below:   |  |  |  | | --- | --- | --- | | Truth table for or | | | | A | B | A or B | | True | True | True | | True | False | True | | False | True | True | | False | False | False |   not operator is used to invert the truth value. The truth table for not is given below:   |  |  | | --- | --- | | Truth tabel for not | | | A | not A | | True | False | | False | True |   some example of their usage are given below  >>> True and False  False  >>> True or False  True  >>> not False  True |  |  |

**Example #3: Logical Operators in Python**

x = True

y = False

# Output: x and y is False

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

# Output: x or y is True

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

# Output: not x is False

print('not x is',not x)

**Bitwise operators**

Bitwise operators act on operands as if they were string of binary digits. It operates bit by bit, hence the name.

For example, 2 is 10 in binary and 7 is 111.

**In the table below:** Let x = 10 (0000 1010 in binary) and y = 4 (0000 0100 in binary)

|  |  |  |
| --- | --- | --- |
| Bitwise operators in Python | | |
| Operator | Meaning | Example |
| & | Bitwise AND | x& y = 0 (0000 0000) |
| | | Bitwise OR | x | y = 14 (0000 1110) |
| ~ | Bitwise NOT | ~x = -11 (1111 0101) |
| ^ | Bitwise XOR | x ^ y = 14 (0000 1110) |
| >> | Bitwise right shift | x>> 2 = 2 (0000 0010) |
| << | Bitwise left shift | x<< 2 = 40 (0010 1000) |

**Assignment operators**

Assignment operators are used in Python to assign values to variables.

a = 5 is a simple assignment operator that assigns the value 5 on the right to the variable a on the left.

There are various compound operators in Python like a += 5 that adds to the variable and later assigns the same. It is equivalent to a = a + 5.

|  |  |  |
| --- | --- | --- |
| Assignment operators in Python | | |
| Operator | Example | Equivatent to |
| = | x = 5 | x = 5 |
| += | x += 5 | x = x + 5 |
| -= | x -= 5 | x = x – 5 |
| \*= | x \*= 5 | x = x \* 5 |
| /= | x /= 5 | x = x / 5 |
| %= | x %= 5 | x = x % 5 |
| //= | x //= 5 | x = x // 5 |
| \*\*= | x \*\*= 5 | x = x \*\* 5 |
| &= | x &= 5 | x = x & 5 |
| |= | x |= 5 | x = x | 5 |
| ^= | x ^= 5 | x = x ^ 5 |
| >>= | x >>= 5 | x = x >> 5 |
| <<= | x <<= 5 | x = x << 5 |

**Special operators**

Python language offers some special type of operators like the identity operator or the membership operator. They are described below with examples.

**Identity operators**

is and is not are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

|  |  |  |
| --- | --- | --- |
| Identity operators in Python | | |
| Operator | Meaning | Example |
| Is | True if the operands are identical (refer to the same object) | x is True |
| is not | True if the operands are not identical (do not refer to the same object) | x is not True |

**Example #4: Identity operators in Python**

x1 = 5

y1 = 5

x2 = 'Hello'

y2 = 'Hello'

x3 = [1,2,3]

y3 = [1,2,3]

# Output: False

print(x1 is not y1)

# Output: True

print(x2 is y2)

# Output: False

print(x3 is y3)

Here, we see that x1 and y1 are integers of same values, so they are equal as well as identical. Same is the case with x2 and y2 (strings).

But x3 and y3 are list. They are equal but not identical. Since list are mutable (can be changed), interpreter locates them separately in memory although they are equal.

**Membership operators**

in and not in are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, [set](https://www.programiz.com/python-programming/set) and dictionary).

In a dictionary we can only test for presence of key, not the value.

|  |  |  |
| --- | --- | --- |
| Operator | Meaning | Example |
| In | True if value/variable is found in the sequence | 5 in x |
| not in | True if value/variable is not found in the sequence | 5 not in x |

**Example #5: Membership operators in Python**

x = 'Hello world'

y = {1:'a',2:'b'}

# Output: True

print('H' in x)

# Output: True

print('hello' not in x)

# Output: True

print(1 in y)

# Output: False

print('a' in y)

Here, 'H' is in x but 'hello' is not present in x (remember, Python is case sensitive). Similary, 1 is key and 'a' is the value in dictionary y. Hence, 'a' in y returns False.

Decision making is required when we want to execute a code only if a certain condition is satisfied.

The if…elif…else statement is used in Python for decision making.

## Python if Statement Syntax

if test expression:

statement(s)

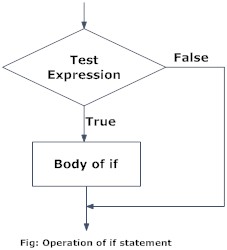
Here, the program evaluates the test expression and will execute statement(s) only if the text expression is True.

If the text expression is False, the statement(s) is not executed.

In Python, the body of the if statement is indicated by the indentation. Body starts with an indentation and the first unindented line marks the end.

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

### Python if Statement Flowchart



### Example: Python if Statement

# If the number is positive, we print an appropriate message

num = 3

if num > 0:

print(num, "is a positive number.")

print("This is always printed.")

num = -1

if num > 0:

print(num, "is a positive number.")

print("This is also always printed.")

When you run the program, the output will be:

3 is a positive number

This is always printed

This is also always printed.

In the above example, num > 0 is the test expression.

The body of if is executed only if this evaluates to True.

When variable num is equal to 3, test expression is true and body inside body of if is executed.

If variable num is equal to -1, test expression is false and body inside body of if is skipped.

The print() statement falls outside of the if block (unindented). Hence, it is executed regardless of the test expression.

## Python if...else Statement

### Syntax of if...else

if test expression:

Body of if

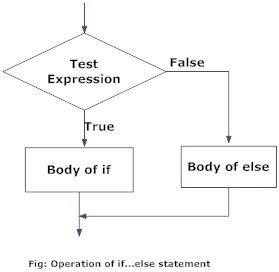
else:

Body of else

The if..else statement evaluates test expression and will execute body of if only when test condition is True.

If the condition is False, body of else is executed. Indentation is used to separate the blocks.

### Python if..else Flowchart



### Example of if...else

# Program checks if the number is positive or negative

# And displays an appropriate message

num = 3

# Try these two variations as well.

# num = -5

# num = 0

if num >= 0:

print("Positive or Zero")

else:

print("Negative number")

In the above example, when num is equal to 3, the test expression is true and body of if is executed and body of else is skipped.

If num is equal to -5, the test expression is false and body of else is executed and body of if is skipped.

If num is equal to 0, the test expression is true and body of if is executed and body of else is skipped.

## Python if...elif...else

### Syntax of if...elif...else

if test expression:

Body of if

elif test expression:

Body of elif

else:

Body of else

The elif is short for else if. It allows us to check for multiple expressions.

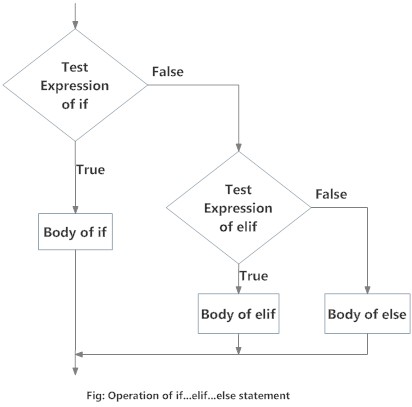
If the condition for if is False, it checks the condition of the next elif block and so on.

If all the conditions are False, body of else is executed.

Only one block among the several if...elif...else blocks is executed according to the condition.

The if block can have only one else block. But it can have multiple elif blocks.

### Flowchart of if...elif...else



### Example of if...elif...else

# In this program,

# we check if the number is positive or

# negative or zero and

# display an appropriate message

num = 3.4

# Try these two variations as well:

# num = 0

# num = -4.5

if num > 0:

print("Positive number")

elif num == 0:

print("Zero")

else:

print("Negative number")

When variable num is positive, Positive number is printed.

If num is equal to 0, Zero is printed.

If num is negative, Negative number is printed

**Python Nested if statements**

We can have a if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming.

Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. This can get confusing, so must be avoided if we can.

**Python Nested if Example**

# In this program, we input a number

# check if the number is positive or

# negative or zero and display

# an appropriate message

# This time we use nested if

num = float(input("Enter a number: "))

if num >= 0:

if num == 0:

print("Zero")

else:

print("Positive number")

else:

print("Negative number")

**Output 1**

Enter a number: 5

Positive number

**Output 2**

Enter a number: -1

Negative number

**Output 3**

Enter a number: 0

Zero

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

## Syntax of for Loop

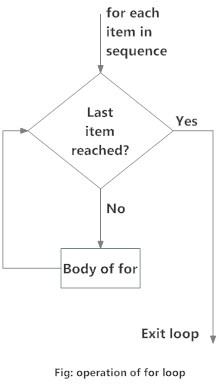
for val in sequence:

Body of for

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

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

## Flowchart of for Loop



## Example: Python for Loop

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

# List of numbers

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

# variable to store the sum

sum = 0

# iterate over the list

for val in numbers:

sum = sum+val

# Output: The sum is 48

print("The sum is", sum)

when you run the program, the output will be:

The sum is 48

**The range() function**

We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers).

We can also define the start, stop and step size as range(start,stop,step size). step size defaults to 1 if not provided.

This function does not store all the values in memory, it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.

To force this function to output all the items, we can use the function list().

The following example will clarify this.

# Output: range(0, 10)

print(range(10))

# Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(list(range(10)))

# Output: [2, 3, 4, 5, 6, 7]

print(list(range(2, 8)))

# Output: [2, 5, 8, 11, 14, 17]

print(list(range(2, 20, 3)))

We can use the range() function in for loops to iterate through a sequence of numbers. It can be combined with the len() function to iterate though a sequence using indexing. Here is an example.

# Program to iterate through a list using indexing

genre = ['pop', 'rock', 'jazz']

# iterate over the list using index

for i in range(len(genre)):

print("I like", genre[i])

When you run the program, the output will be:

I like pop

I like rock

​I like jazz

## for loop with else

A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts.

break statement can be used to stop a for loop. In such case, the else part is ignored.

Hence, a for loop's else part runs if no break occurs.

Here is an example to illustrate this.

digits = [0, 1, 5]

for i in digits:

print(i)

else:

print("No items left.")

When you run the program, the output will be:

0

1

5

No items left.

**Syntax of while Loop in Python**

while test\_expression:

Body of while

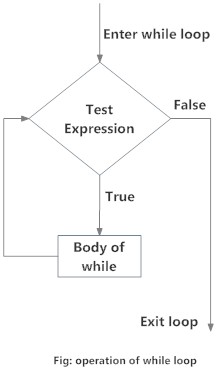
In while loop, test expression is checked first. The body of the loop is entered only if the test\_expression evaluates to True. After one iteration, the test expression is checked again. This process continues until the test\_expression evaluates to False.

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

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

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

**Flowchart of while Loop**



**Example: Python while Loop**

n = 10

# initialize sum and counter

sum = 0

i = 1

while i <= n:

sum = sum + i

i = i+1 # update counter

# print the sum

print("The sum is", sum)

When you run the program, the output will be:

Enter n: 10

The sum is 55

In the above program, the test expression will be True as long as our counter variable i is less than or equal to n (10 in our program).

We need to increase the value of counter variable in the body of the loop. This is very important (and mostly forgotten). Failing to do so will result in an infinite loop (never ending loop).

Finally the result is displayed.

## while loop with else

Same as that of [for loop](https://www.programiz.com/python-programming/for-loop), we can have an optional else block with while loop as well.

The else part is executed if the condition in the while loop evaluates to False. The while loop can be terminated with a [break statement](https://www.programiz.com/python-programming/break-continue).

In such case, the else part is ignored. Hence, a while loop's else part runs if no break occurs and the condition is false.

Here is an example to illustrate this.

counter = 0

while counter < 3:

print("Inside loop")

counter = counter + 1

else:

print("Inside else")

**Output**

Inside loop

Inside loop

Inside loop

Inside else

Here, we use a counter variable to print the string Inside loop three times.

On the forth iteration, the condition in while becomes False. Hence, the else part is executed.

## Python break statement

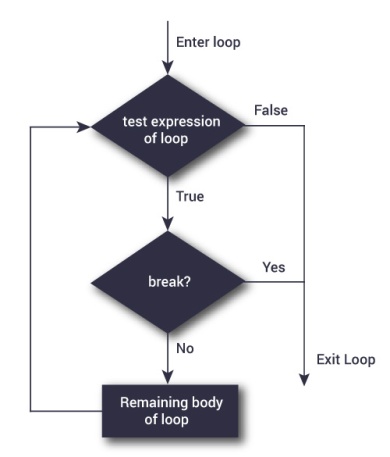
The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

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

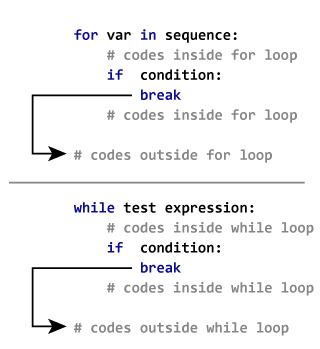
### Syntax of break

break

### Flowchart of break



The working of break statement in [for loop](https://www.programiz.com/python-programming/for-loop) and [while loop](https://www.programiz.com/python-programming/while-loop) is shown below.



### Example: Python break

# Use of break statement inside loop

for val in "string":

if val == "i":

break

print(val)

print("The end")

**Output**

s

t

r

The end

In this program, we iterate through the "string" sequence. We check if the letter is "i", upon which we break from the loop. Hence, we see in our output that all the letters up till "i" gets printed. After that, the loop terminates.

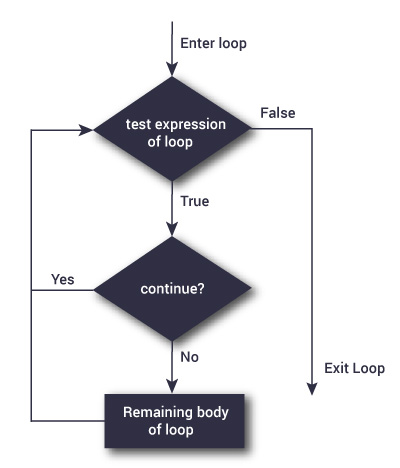
## Python continue statement

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

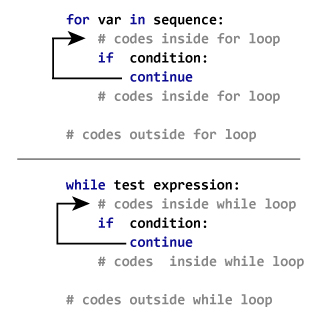
### Syntax of Continue

continue

### Flowchart of continue



The working of continue statement in for and while loop is shown below.



### Example: Python continue

for val in "string":

if val == "i":

continue

print(val)

print("The end")

## Syntax of pass

pass

We generally use it as a placeholder.

Suppose we have a [loop](https://www.programiz.com/python-programming/for-loop)or a [function](https://www.programiz.com/python-programming/function) that is not implemented yet, but we want to implement it in the future. They cannot have an empty body. The interpreter would complain. So, we use the pass statement to construct a body that does nothing.

## Example: pass Statement

# pass is just a placeholder for

# functionality to be added later.

sequence = {'p', 'a', 's', 's'}

for val in sequence:

pass

**The infinite loop**

We can create an infinite loop using while statement. If the condition of while loop is always True, we get an infinite loop.

**Example #1: Infinite loop using while**

# An example of infinite loop

# press Ctrl + c to exit from the loop

while True:

num = int(input("Enter an integer: "))

print("The double of",num,"is",2 \* num)

**Output**

Enter an integer: 3

The double of 3 is 6

Enter an integer: 5

The double of 5 is 10

Enter an integer: 6

The double of 6 is 12

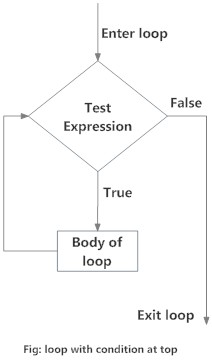
Enter an integer:

Traceback (most recent call last):

**Loop with condition at the top**

This is a normal while loop without break statements. The condition of the while loop is at the top and the loop terminates when this condition is False.

**Flowchart of Loop With Condition at Top**



**Example #2: Loop with condition at the top**

# Try different numbers

# initialize sum and counter

sum = 0

i = 1

while i <= n:

sum = sum + i

i = i+1 # update counter

# print the sumn = 10

# Uncomment to get user input

#n = int(input("Enter n: "))

print("The sum is",sum)

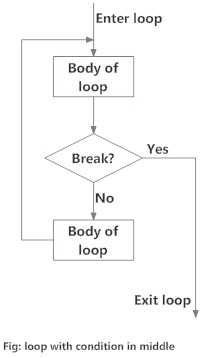
When you run the program, the output will be:

The sum is 55

**Loop with condition in the middle**

This kind of loop can be implemented using an infinite loop along with a conditional break in between the body of the loop.

**Flowchart of Loop with Condition in Middle**



**Example #3: Loop with condition in the middle**

# Program to illustrate a loop with condition in the middle.

# Take input from the user untill a vowel is entered

vowels = "aeiouAEIOU"

# infinite loop

while True:

v = input("Enter a vowel: ")

# condition in the middle

if v in vowels:

break

print("That is not a vowel. Try again!")

print("Thank you!")

**Output**

Enter a vowel: r

That is not a vowel. Try again!

Enter a vowel: 6

That is not a vowel. Try again!

Enter a vowel: ,

That is not a vowel. Try again!

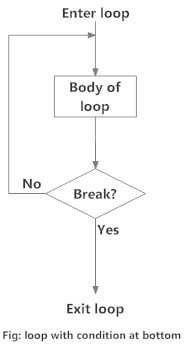
Enter a vowel: u

Thank you!

**Loop with condition at the bottom**

This kind of loop ensures that the body of the loop is executed at least once. It can be implemented using an infinite loop along with a conditional break at the end. This is similar to the do...while loop in C.

**Flowchart of Loop with Condition at Bottom**



**Example #4: Loop with condition at the bottom**

# Python program to illustrate a loop with condition at the bottom

# Roll a dice untill user chooses to exit

# import random module

import random

while True:

input("Press enter to roll the dice")

# get a number between 1 to 6

num = random.randint(1,6)

print("You got",num)

option = input("Roll again?(y/n) ")

# condition

if option == 'n':

break

**Output**

Press enter to roll the dice

You got 1

Roll again?(y/n) y

Press enter to roll the dice

You got 5

Roll again?(y/n) n