List of Keywords in Python

Keywords are the reserved words in Python. We cannot use a keyword as variable name, function name or any other identifier.

Here's a list of all keywords in Python Programming

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Keywords in Python programming language | | | | |
| [False](https://www.programiz.com/python-programming/keyword-list#true_false) | [class](https://www.programiz.com/python-programming/keyword-list#class) | [finally](https://www.programiz.com/python-programming/keyword-list#finally) | [is](https://www.programiz.com/python-programming/keyword-list#is) | [return](https://www.programiz.com/python-programming/keyword-list#return) |
| [None](https://www.programiz.com/python-programming/keyword-list#none) | [continue](https://www.programiz.com/python-programming/keyword-list#break_continue) | [for](https://www.programiz.com/python-programming/keyword-list#for) | [lambda](https://www.programiz.com/python-programming/keyword-list#lambda) | [try](https://www.programiz.com/python-programming/keyword-list#except_raise_try) |
| [True](https://www.programiz.com/python-programming/keyword-list#true_false) | [def](https://www.programiz.com/python-programming/keyword-list#def) | [from](https://www.programiz.com/python-programming/keyword-list#from_import) | [nonlocal](https://www.programiz.com/python-programming/keyword-list#nonlocal) | [while](https://www.programiz.com/python-programming/keyword-list#while) |
| [and](https://www.programiz.com/python-programming/keyword-list#and_or_not) | [del](https://www.programiz.com/python-programming/keyword-list#del) | [global](https://www.programiz.com/python-programming/keyword-list#global) | [not](https://www.programiz.com/python-programming/keyword-list#and_or_not) | [with](https://www.programiz.com/python-programming/keyword-list#with) |
| [as](https://www.programiz.com/python-programming/keyword-list#as) | [elif](https://www.programiz.com/python-programming/keyword-list#if_else_elif) | [if](https://www.programiz.com/python-programming/keyword-list#if_else_elif) | [or](https://www.programiz.com/python-programming/keyword-list#and_or_not) | [yield](https://www.programiz.com/python-programming/keyword-list#yield) |
| [assert](https://www.programiz.com/python-programming/keyword-list#assert) | [else](https://www.programiz.com/python-programming/keyword-list#if_else_elif) | [import](https://www.programiz.com/python-programming/keyword-list#from_import) | [pass](https://www.programiz.com/python-programming/keyword-list#pass) |  |
| [break](https://www.programiz.com/python-programming/keyword-list#break_continue) | [except](https://www.programiz.com/python-programming/keyword-list#except_raise_try) | [in](https://www.programiz.com/python-programming/keyword-list#in) | [raise](https://www.programiz.com/python-programming/keyword-list#except_raise_try) |  |

The above keywords may get altered in different versions of Python. Some extra might get added or some might be removed. You can always get the list of keywords in your current version by typing the following in the prompt.

>>> import keyword

>>> print(keyword.kwlist)

['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']

**Description of Keywords in Python with examples**

**True, False**

True and False are truth values in Python. They are the results of comparison operations or logical (Boolean) operations in Python. For example:

>>> 1 == 1

True

>>> 5 > 3

True

>>> True or False

True

>>> 10 <= 1

False

>>> 3 > 7

False

>>> True and False

False

Here we can see that the first three statements are true so the interpreter returns True and returns False for the remaining three statements. True and False in python is same as 1and 0. This can be justified with the following example:

>>> True == 1

True

>>> False == 0

True

>>> True + True

2

**None**

None is a special constant in Python that represents the absence of a value or a null value.

It is an object of its own datatype, the NoneType. We cannot create multiple None objects but can assign it to variables. These variables will be equal to one another.

We must take special care that None does not imply False, 0 or any empty list, dictionary, string etc. For example:

>>> None == 0

False

>>> None == []

False

>>> None == False

False

>>> x = None

>>> y = None

>>> x == y

True

Void functions that do not return anything will return a None object automatically. None is also returned by functions in which the program flow does not encounter a return statement. For example:

def a\_void\_function():

a = 1

b = 2

c = a + b

x = a\_void\_function()

print(x)

**Output**

None

This program has a function that does not return a value, although it does some operations inside. So when we print x, we get None which is returned automatically (implicitly). Similarly, here is another example:

def improper\_return\_function(a):

if (a % 2) == 0:

return True

x = improper\_return\_function(3)

print(x)

**Output**

None

Although this function has a return statement, it is not reached in every case. The function will return True only when the input is even.

If we give the function an odd number, None is returned implicitly.

**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

**as**

as is used to create an alias while importing a module. It means giving a different name (user-defined) to a module while importing it.

As for example, Python has a standard module called math. Suppose we want to calculate what cosine pi is using an alias. We can do it as follows using as:

>>> import math as myAlias

>>>myAlias.cos(myAlias.pi)

-1.0

Here we imported the math module by giving it the name myAlias. Now we can refer to the math module with this name. Using this name we calculated cos(pi) and got -1.0 as the answer.

**assert**

assert is used for debugging purposes.

While programming, sometimes we wish to know the internal state or check if our assumptions are true. assert helps us do this and find bugs more conveniently. assert is followed by a condition.

If the condition is true, nothing happens. But if the condition is false, AssertionError is raised. For example:

>>> a = 4

>>> assert a < 5

>>> assert a > 5

Traceback (most recent call last):

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

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

AssertionError

For our better understanding, we can also provide a message to be printed with the AssertionError.

>>> a = 4

>>> assert a > 5, "The value of a is too small"

Traceback (most recent call last):

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

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

AssertionError: The value of a is too small

At this point we can note that,

assert condition, message

is equivalent to,

if not condition:

raise AssertionError(message)

**break, continue**

break and continue are used inside for and while loops to alter their normal behavior.

break will end the smallest loop it is in and control flows to the statement immediately below the loop. continue causes to end the current iteration of the loop, but not the whole loop.

This can be illustrated with the following two examples:

for i in range(1,11):

if i == 5:

break

print(i)

**Output**

1

2

3

4

Here, the for loop intends to print numbers from 1 to 10. But the if condition is met when i is equal to 5 and we break from the loop. Thus, only the range 1 to 4 is printed.

for i in range(1,11):

if i == 5:

continue

print(i)

**Output**

1

2

3

4

6

7

8

9

10

Here we use continue for the same program. So, when the condition is met, that iteration is skipped. But we do not exit the loop. Hence, all the values except 5 is printed out.

Learn more about [Python break and continue statement](https://www.programiz.com/python-programming/break-continue).

**class**

class is used to define a new user-defined class in Python.

Class is a collection of related attributes and methods that try to represent a real world situation. This idea of putting data and functions together in a class is central to the concept of object-oriented programming (OOP).

Classes can be defined anywhere in a program. But it is a good practice to define a single class in a module. Following is a sample usage:

class ExampleClass:

def function1(parameters):

…

def function2(parameters):

…

Learn more about [Python Objects and Class](https://www.programiz.com/python-programming/class).

**def**

def is used to define a user-defined function.

Function is a block of related statements, which together does some specific task. It helps us organize code into manageable chunks and also to do some repetitive task.

The usage of def is shown below:

def function\_name(parameters):

…

Learn more about [Python functions](https://www.programiz.com/python-programming/function).

**del**

del is used to delete the reference to an object. Everything is object in Python. We can delete a variable reference using del

>>> a = b = 5

>>> del a

>>> a

Traceback (most recent call last):

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

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

NameError: name 'a' is not defined

>>> b

5

Here we can see that the reference of the variable a was deleted. So, it is no longer defined. But b still exists.

del is also used to delete items from a list or a dictionary:

>>> a = ['x','y','z']

>>> del a[1]

>>> a

['x', 'z']

**if, else, elif**

if, else, elif are used for conditional branching or decision making.

When we want to test some condition and execute a block only if the condition is true, then we use if and elif. elif is short for else if. else is the block which is executed if the condition is false. This will be clear with the following example:

def if\_example(a):

if a == 1:

print('One')

elif a == 2:

print('Two')

else:

print('Something else')

if\_example(2)

if\_example(4)

if\_example(1)

**Output**

Two

Something else

One

Here, the function checks the input number and prints the result if it is 1 or 2. Any input other than this will cause the else part of the code to execute.

Learn more about [Python if and if...else Statement](https://www.programiz.com/python-programming/if-elif-else).

**except, raise, try**

except, raise, try are used with exceptions in Python.

Exceptions are basically errors that suggests something went wrong while executing our program. IOError, ValueError, ZeroDivisionError, ImportError, NameError, TypeError etc. are few examples of exception in Python. try...except blocks are used to catch exceptions in Python.

We can raise an exception explicitly with the raise keyword. Following is an example:

def reciprocal(num):

try:

r = 1/num

except:

print('Exception caught')

return

return r

print(reciprocal(10))

print(reciprocal(0))

**Output**

0.1

Exception caught

None

Here, the function reciprocal() returns the reciprocal of the input number.

When we enter 10, we get the normal output of 0.1. But when we input 0, a ZeroDivisionError is raised automatically.

This is caught by our try…except block and we return None. We could have also raised the ZeroDivisionError explicitly by checking the input and handled it elsewhere as follows:

if num == 0:

raise ZeroDivisionError('cannot divide')

**finally**

finally is used with try…except block to close up resources or file streams.

Using finally ensures that the block of code inside it gets executed even if there is an unhandled exception. For example:

try:

Try-block

except exception1:

Exception1-block

except exception2:

Exception2-block

else:

Else-block

finally:

Finally-block

Here if there is an exception in the Try-block, it is handled in the except or else block. But no matter in what order the execution flows, we can rest assured that the Finally-blockis executed even if there is an error. This is useful in cleaning up the resources.

Learn more about [exception handling in Python programming](https://www.programiz.com/python-programming/exception-handling).

**for**

for is used for looping. Generally we use for when we know the number of times we want to loop.

In Python we can use it with any type of sequence like a list or a string. Here is an example in which for is used to traverse through a list of names:

names = ['John','Monica','Steven','Robin']

for i in names:

print('Hello '+i)

**Output**

Hello John

Hello Monica

Hello Steven

Hello Robin

Learn more about [Python for loop](https://www.programiz.com/python-programming/for-loop).

**from, import**

import keyword is used to import modules into the current namespace. from…import is used to import specific attributes or functions into the current namespace. For example:

import math

will import the math module. Now we can use the cos() function inside it as math.cos(). But if we wanted to import just the cos() function, this can done using from as

from math import cos

now we can use the function simply as cos(), no need to write math.cos().

Learn more on [Python modules and import statement](https://www.programiz.com/python-programming/modules).

**global**

global is used to declare that a variable inside the function is global (outside the function).

If we need to read the value of a global variable, it is not necessary to define it as global. This is understood.

If we need to modify the value of a global variable inside a function, then we must declare it with global. Otherwise a local variable with that name is created.

Following example will help us clarify this.

globvar = 10

def read1():

print(globvar)

def write1():

global globvar

globvar = 5

def write2():

globvar = 15

read1()

write1()

read1()

write2()

read1()

**Output**

10

5

5

Here, the read1() function is just reading the value of globvar. So, we do not need to declare it as global. But the write1() function is modifying the value, so we need to declare the variable as global.

We can see in our output that the modification did take place (10 is changed to 5). The write2() also tries to modify this value. But we have not declared it as global.

Hence, a new local variable globvar is created which is not visible outside this function. Although we modify this local variable to 15, the global variable remains unchanged. This is clearly visible in our output.

**in**

in is used to test if a sequence (list, tuple, string etc.) contains a value. It returns True if the value is present, else it returns False. For example:

>>> a = [1, 2, 3, 4, 5]

>>> 5 in a

True

>>> 10 in a

False

The secondary use of in is to traverse through a sequence in a for loop.

for i in 'hello':

print(i)

**Output**

h

e

l

l

o

**is**

is is used in Python for testing object identity. While the == operator is used to test if two variables are equal or not, is is used to test if the two variables refer to the same object.

It returns True if the objects are identical and False if not.

>>> True is True

True

>>> False is False

True

>>> None is None

True

We know that there is only one instance of True, False and None in Python, so they are identical.

>>> [] == []

True

>>> [] is []

False

>>> {} == {}

True

>>> {} is {}

False

An empty list or dictionary is equal to another empty one. But they are not identical objects as they are located separately in memory. This is because list and dictionary are mutable (value can be changed).

>>> '' == ''

True

>>> '' is ''

True

>>> () == ()

True

>>> () is ()

True

Unlike list and dictionary, string and tuple are immutable (value cannot be altered once defined). Hence, two equal string or tuple are identical as well. They refer to the same memory location.

**lambda**

lambda is used to create an anonymous function (function with no name). It is an inline function that does not contain a return statement. It consists of an expression that is evaluated and returned. For example:

a = lambda x: x\*2

for i in range(1,6):

print(a(i))

**Output**

2

4

6

8

10

Here, we have created an inline function that doubles the value, using the lambda statement. We used this to double the values in a list containing 1 to 5.

Learn more about [Python lamda function](https://www.programiz.com/python-programming/anonymous-function).

**nonlocal**

The use of nonlocal keyword is very much similar to the global keyword. nonlocal is used to declare that a variable inside a nested function (function inside a function) is not local to it, meaning it lies in the outer inclosing function. If we need to modify the value of a non-local variable inside a nested function, then we must declare it with nonlocal. Otherwise a local variable with that name is created inside the nested function. Following example will help us clarify this.

def outer\_function():

a = 5

def inner\_function():

nonlocal a

a = 10

print("Inner function: ",a)

inner\_function()

print("Outer function: ",a)

outer\_function()

**Output**

Inner function: 10

Outer function: 10

Here, the inner\_function() is nested within the outer\_function.

The variable a is in the outer\_function(). So, if we want to modify it in the inner\_function(), we must declare it as nonlocal. Notice that a is not a global variable.

Hence, we see from the output that the variable was successfully modified inside the nested inner\_function(). The result of not using the nonlocal keyword is as follows:

def outer\_function():

a = 5

def inner\_function():

a = 10

print("Inner function: ",a)

inner\_function()

print("Outer function: ",a)

outer\_function()

**Output**

Inner function: 10

Outer function: 5

Here, we do not declare that the variable a inside the nested function is nonlocal. Hence, a new local variable with the same name is created, but the non-local a is not modified as seen in our output.

**pass**

pass is a null statement in Python. Nothing happens when it is executed. It is used as a placeholder.

Suppose we have a function that is not implemented yet, but we want to implement it in the future. Simply writing,

def function(args):

in the middle of a program will give us IndentationError. Instead of this, we construct a blank body with the pass statement.

def function(args):

pass

We can do the same thing in an empty class as well.

class example:

pass

**return**

return statement is used inside a function to exit it and return a value.

If we do not return a value explicitly, None is returned automatically. This is verified with the following example.

def func\_return():

a = 10

return a

def no\_return():

a = 10

print(func\_return())

print(no\_return())

**Output**

10

None

**while**

while is used for looping in Python.

The statements inside a while loop continue to execute until the condition for the whileloop evaluates to False or a break statement is encountered. Following program illustrates this.

i = 5

while(i):

print(i)

i = i – 1

**Output**

5

4

3

2

1

Note that 0 is equal to False.

Learn more about [Python while loop](https://www.programiz.com/python-programming/while-loop).

**with**

with statement is used to wrap the execution of a block of code within methods defined by the context manager.

Context manager is a class that implements \_\_enter\_\_ and \_\_exit\_\_ methods. Use of withstatement ensures that the \_\_exit\_\_ method is called at the end of the nested block. This concept is similar to the use of try…finally block. Here, is an example.

with open('example.txt', 'w') as my\_file:

my\_file.write('Hello world!')

This example writes the text Hello world! to the file example.txt. File objects have \_\_enter\_\_ and \_\_exit\_\_ method defined within them, so they act as their own context manager.

First the \_\_enter\_\_ method is called, then the code within with statement is executed and finally the \_\_exit\_\_ method is called. \_\_exit\_\_ method is called even if there is an error. It basically closes the file stream.

**yield**

yield is used inside a function like a return statement. But yield returns a generator.

Generator is an iterator that generates one item at a time. A large list of value will take up a lot of memory. Generators are useful in this situation as it generates only one value at a time instead of storing all the values in memory. For example,

>>> g = (2\*\*x for x in range(100))

will create a generator g which generates powers of 2 up to the number two raised to the power 99. We can generate the numbers using the next() function as shown below.

>>> next(g)

1

>>> next(g)

2

>>> next(g)

4

>>> next(g)

8

>>> next(g)

16

And so on… This type of generator is returned by the yield statement from a function. Here is an example.

def generator():

for i in range(6):

yield i\*i

g = generator()

for i in g:

print(i)

**Output**

0

1

4

9

16

25

Here, the function generator() returns a generator that generates square of numbers from 0 to 5. This is printed in the for loop.