Python Keywords and Identifiers

Keywords are the reserved words in Python.

We cannot use a keyword as a 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 3.7. This number can vary slightly in the 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 is given below.

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	

Keywords in Python

Python Identifiers

An identifier is a name given to entities like class, functions, variables, etc. It helps to differentiate 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.

```
>>> global = 1
  File "<interactive input>", line 1
    global = 1
           ^
SyntaxError: invalid syntax
```

4. We cannot use special symbols like **!**, **@**, #, **$**, **%** etc. in our identifier.

```
>>> a@ = 0
  File "<interactive input>", line 1
    a@ = 0
     ^
SyntaxError: invalid syntax
```

5. Identifier can be of any length.

Things to Remember

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.

Python Statement, Indentation and Comments

In this article, you will learn about Python statements, why indentation is important and use of comments in programming.

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.

- script.py
- IPython Shell
-

```
1
2
3
4
for i in range(1,11):
    print(i)
    if i == 5:
        break
```

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.

- script.py
- IPython Shell
-

```
1

2

3

4
```

```
for i in range(1,11):
    print(i)
    if i == 5:
        break
```

Run

Powered by DataCamp

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:

```
1.    if True:
2.        print('Hello')
3.        a = 5
```

and

```
1.    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.

```
1.    #This is a comment
2.    #print out Hello
3.    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:

```
1.    #This is a long comment
2.    #and it extends
3.    #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.

```
1.    """This is also a
2.    perfect example of
3.    multi-line comments"""
```

Docstring in Python

Docstring is short for documentation string.

It is a 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:

- script.py
- IPython Shell
-

```
1

2

3

def double(num):
    """Function to double the value"""
    return 2*num
```

Run

Powered by DataCamp

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
```

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.

- script.py
- IPython Shell

```
1

2

3

4

for i in range(1,11):
    print(i)
    if i == 5:
        break
```

Run

Powered by DataCamp

Python Variables, Constants and Literals

In this article, you will learn about Python variables, constants, literals and their use cases.

Python Variables

A variable is a named location used to store data in the memory. It is helpful to think of variables as a container that holds data which can be changed later throughout programming. For example,

```
number = 10
```

Here, we have created a named `number`. We have assigned value 10 to the variable.

You can think variable as a bag to store books in it and those books can be replaced at any time.

```
number = 10
number = 1.1
```

Initially, the value of `number` was 10. Later it's changed to 1.1.

Note: In Python, we don't assign values to the variables, whereas Python gives the reference of the object (value) to the variable.

Assigning a value to a Variable in Python

As you can see from the above example, you can use the assignment operator `=` to assign a value to a variable.

Example 1: Declaring and assigning a value to a variable

- script.py
- IPython Shell

```
1

2

website = "apple.com"

print(website)
```

Run

Powered by DataCamp

When you run the program, the output will be:

```
apple.com
```

In the above program, we assigned a value `apple.com` to the variable `website`. Then we print the value assigned to `website` i.e. `apple.com`
Note : Python is a type inferred language; it can automatically know `apple.com` is a string and declare `website` as a string.

Example 2: Changing the value of a variable

- script.py
- IPython Shell

```
1
2
3
4
5
6
7
website = "apple.com"
print(website)
# assigning a new variable to website
```

```
website = "programiz.com"

print(website)
```

Run

Powered by DataCamp

When you run the program, the output will be:

```
apple.com
programiz.com
```

In the above program, we have assigned `apple.com` to the `website` variable initially. Then, it's value is changed to `programiz.com`.

Example 3: Assigning multiple values to multiple variables

- script.py
- IPython Shell

```
1

2

3

4

5

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

print (a)
```

```
print (b)

print (c)
```

Run

Powered by DataCamp

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

- script.py
- IPython Shell

```
1

2

3

4

5

x = y = z = "same"

print (x)

print (y)

print (z)
```

Run

Powered by DataCamp

The second program assigns the `same` string to all the three variables `x`, `y` and `z`.

Constants

A constant is a type of variable whose value cannot be changed. It is helpful to think of constants as containers that hold information which cannot be changed later.

Non technically, you can think of constant as a bag to store some books and those books cannot be replaced once placed inside the bag.

Assigning value to a constant in Python

In Python, constants are usually declared and assigned on a module. Here, the module means a new file containing variables, functions etc which is imported to main file. Inside the module, constants are written in all capital letters and underscores separating the words.

Example 3: Declaring and assigning value to a constant

Create a constant.py

```
1.    PI = 3.14
2.    GRAVITY = 9.8
```

Create a main.py

```
1.    import constant
2.
3.    print(constant.PI)
4.    print(constant.GRAVITY)
```

When you run the program, the output will be:

```
3.14
9.8
```

Rules and Naming convention for variables and constants

1. Create a name that makes sense. Suppose, `vowel` makes more sense than `v`.
2. Use camelCase notation to declare a variable. It starts with lowercase letter. For example:

```
3.     myName

4.     myAge
```

```
myAddress
```

5. Use capital letters where possible to declare a constant. For example:

```
6.     PI

7.     G

8.     MASS
```

```
TEMP
```

9. Never use special symbols like !, @, #, $, %, etc.
10. Don't start name with a digit.
11. Constants are put into Python modules and meant not be changed.
12. Constant and variable names should have combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_). For example:

```
13.    snake_case

14.    MACRO_CASE

15.    camelCase
```

```
CapWords
```

Literals

Literal is a raw data given in a variable or constant. In Python, there are various types of literals they are as follows:

Numeric Literals

Numeric Literals are immutable (unchangeable). Numeric literals can belong to 3 different numerical types Integer, Float and Complex.

Example 4: How to use Numeric literals in Python?

- script.py
- IPython Shell

```
1

2

3

4
```

5

6

7

8

9

10

11

12

13

14

15

```
a = 0b1010 #Binary Literals
b = 100 #Decimal Literal
c = 0o310 #Octal Literal
d = 0x12c #Hexadecimal Literal
#Float Literal
float_1 = 10.5
float_2 = 1.5e2
#Complex Literal
x = 3.14j
print(a, b, c, d)
print(float_1, float_2)
print(x, x.imag, x.real)
```

Run

When you run the program, the output will be:

```
10 100 200 300
10.5 150.0
3.14j 3.14 0.0
```

In the above program,

- We assigned integer literals into different variables. Here, `a` is binary literal, `b` is a decimal literal, `c` is an octal literal and `d` is a hexadecimal literal.
- When we print the variables, all the literals are converted into decimal values.
- 10.5 and 1.5e2 are floating point literals. 1.5e2 is expressed with exponential and is equivalent to `1.5 * 10`^{2}.
- We assigned a complex literal i.e 3.14j in variable x. Then we use imaginary literal (x.imag) and real literal (x.real) to create imaginary and real part of complex number.
  To learn more about Numeric Literals, refer Python Numbers.

String literals

A string literal is a sequence of characters surrounded by quotes. We can use both single, double or triple quotes for a string. And, a character literal is a single character surrounded by single or double quotes.

Example 7: How to use string literals in Python?

- script.py
- IPython Shell
-

```
fruits = ["apple", "mango", "orange"] #list

numbers = (1, 2, 3) #tuple

alphabets = {'a':'apple', 'b':'ball', 'c':'cat'} #dictionary

vowels = {'a', 'e', 'i' , 'o', 'u'} #set


print(fruits)

print(numbers)

print(alphabets)

print(vowels)

strings = "This is Python"

char = "C"

multiline_str = """This is a multiline string with more than
one line code."""

unicode = u"\u00dcnic\u00f6de"

raw_str = r"raw \n string"

print(strings)

print(char)

print(multiline_str)

print(unicode)

print(raw_str)
```

Run

When you run the program, the output will be:

```
This is Python
C
This is a multiline string with more than one line
code.
Ünicöde
raw \n string
```

In the above program, `This is Python` is a string literal and `C` is a character literal. The value with triple-quote `"""` assigned in the `multiline_str` is multi-line string literal.
The `u"\u00dcnic\u00f6de"` is a unicode literal which supports characters other than English and `r"raw \n string"` is a raw string literal.

Boolean literals

A Boolean literal can have any of the two values: `True` or `False`.

Example 8: How to use boolean literals in Python?

- script.py
- IPython Shell

1

2

3

4

5

6

```
7

8

9

x = (1 == True)

y = (1 == False)

a = True + 4

b = False + 10

print("x is", x)

print("y is", y)

print("a:", a)

print("b:", b)
```

Run

Powered by DataCamp

When you run the program, the output will be:

```
x is True
y is False
a: 5
b: 10
```

In the above program, we use boolean literal `True` and `False`. In Python, `True` represents the value as `1` and `False` as `0`. The value of `x` is `True` because `1` is equal to `True`. And, the value of `y` is `False` because `1` is not equal to `False`.
Similarly, we can use the `True` and `False` in numeric expressions as the value. The value of `a` is `5` because we add `True` which has value of `1` with `4`. Similarly, `b` is `10` because we add the `False` having value of `0` with `10`.

When you run the program, the output will be:

```
Available
None
```

In the above program, we define a `menu` function. Inside `menu`, when we set parameter as `drink` then, it displays `Available`. And, when the parameter is `food`, it displays `None`.

Literal Collections

There are four different literal collections List literals, Tuple literals, Dict literals, and Set literals.

Example 10: How to use literals collections in Python?

- script.py
- IPython Shell

1

2

3

4

5

6

7

8

9

```
fruits = ["apple", "mango", "orange"] #list

numbers = (1, 2, 3) #tuple

alphabets = {'a':'apple', 'b':'ball', 'c':'cat'} #dictionary

vowels = {'a', 'e', 'i' , 'o', 'u'} #set

print(fruits)

print(numbers)

print(alphabets)

print(vowels)
```

Run

Powered by DataCamp

When you run the program, the output will be:

```
['apple', 'mango', 'orange']
(1, 2, 3)
{'a': 'apple', 'b': 'ball', 'c': 'cat'}
{'e', 'a', 'o', 'i', 'u'}
```

In the above program, we created a list of `fruits`, tuple of `numbers`, dictionary `dict` having values with keys desginated to each value and set of `vowels`.
To learn more about literal collections, refer Python Data Types.