Image Source

What is the role of the **id** function in Python?

Definition

By definition, variables are a pictorial representation of boxes in which Python will store values identified by a label called a variable name.

Image Source

If computer memory were represented as a shelf with several numbered drawers, the value of a variable would be stored in one of the drawers.

The role of *id* in Python is to return the number of the memory cell where the value of the variable is stored.

To use **id** we use the following format: `id(variable_name)`. For example:

```
>>> # Declaring a variable
>>> a = 1
>>> id(a)
140735660221224
>>> b = "John"
>>> id(b)
2239249141168
```

In the previous example, we can see that the variables *a* and *b* have different memory location numbers, which means that the values of the variables are stored in different memory locations.

The id function does not always return the same memory cell number. The value obtained varies from one computer to another.

When writing a program in Python for primitive types such as **int**(integer), **float**(decimal point), **bool**(Boolean) and **str**(string), when two variables have the same value, both variables share the same memory slot as long as the value is identical. As in the examples below:

```
>>> # Case of integers
>>> a = 1
>>> b = 1
>>> id(a)
140735660221224
>>> id(b)
140735660221224
>>> id(a) == id(b)
True
>>>
>>> # If we change the value of a
>>> a = 2
>>> a
2
>>> b
1
>>> id(a)
140735660221256
>>> id(b)
140735660221224
>>> id(a) == id(b)
False
```

Shares the same memory location, so id is identical

Id of variable 'a' has changed because the value of variable 'a' is no longer the same and is therefore stored in another memory cell.

The same result is obtained with **strings** type **str**.

```
>>> # Case of string
>>> a = "hello"
>>> b = "hello"
>>> id(a)
2239249152752
>>> id(b)
2239249152752
>>> id(a) == id(b)
True
>>>
>>> # If we change the value of b
>>> b = "hi"
>>> a
'hello'
>>> b
'hi'
>>> id(a) == id(b)
False
```

This mechanism of sharing the same memory slot does not increase the memory occupied by a variable. 100 variables with the same value and 50 variables with the same value are linked to a single memory cell.

However, this is no longer the case for **Object** types such as **list**, **tuple** and **dict** (dictionaries). Any declaration of an object-type variable will be stored in a new memory cell, regardless of whether another variable exists or not.

```
>>> # Declaration of a list
>>> l1 = [1, 2]
>>> l2 = [1, 2]
>>> id(l1)
2239242078912
>>> id(l2)
2239248735616
>>> id(l1) == id(l2)
False
>>>
>>> # Declaration of a tuple
>>> t1 = 1, 2
>>> t2 = 1, 2
>>> id(t1) == id(t2)
False
>>>
>>> # Declaration of a dict
>>> d1 = {"nom": "toto"}
>>> d2 = {"nom": "toto"}
>>> id(d1) == id(d2)
False
```

Unlike primitive types, here we have an increase in the amount of memory occupied. 100 object-type variables will occupy 100 different memory slots.

When the **=** operation is used to assign one variable to another, the memory location is copied. For example, **a=b** means assigning the same memory location to the **a** and **b** variables, i.e. that of **b**. As the memory location is the same in the case of lists, this amounts to manipulating the same list.

```
>>> # Using the = operator on int
>>> a = 1
>>> id(a)
140735660221224
>>> b = a
>>> id(b)
140735660221224
>>> id(a) == id(b)
True
>>>
>>> # Using the = operator on lists
>>> l1 = [1, 2]
>>> id(l1)
2239249201920
>>> l2 = [1, 2]
>>> id(l2)
2239249202624
>>> id(l1) == id(l2)
False
>>> # The two lists are in two different memory locations: if you modify one list, the other is not affected.
>>> l1.append(3)
>>> l1
[1, 2, 3]
>>> # use of = changes the memory location
>>> l2 = l1
>>> id(l2)
2239249201920
>>> id(l1)
2239249201920
>>> id(l1) == id(l2)
True
>>> # if you modify one list, the other will be modified because the same memory cell is modified
>>> l1
[1, 2, 3]
>>> l2
[1, 2, 3]
>>> l1.append(4)
>>> l1
[1, 2, 3, 4]
>>> l2
[1, 2, 3, 4]
```

Ruler with input function

When the user enters the game, things change.

Previously, for primitive types, **id** returned the same value because the variables contained the same values. When the **input** function is used to allow the user to enter a value, the value entered by the user is stored in a new memory cell.

```
>>> # Create a variable with a value
>>> a = 1
>>> id(a)
140735660221224
>>> # Request a value from the user, which will be 1
>>> b = input()
1
>>> id(b)
140735660264576
>>>
>>> # If we convert the user's input to an integer, we'll get the same id
>>> b = int(b)
>>> b
1
>>> id(b)
140735660221224
```

When the user is asked to enter a number, the entry is stored in a new memory cell, as we don't know whether what the user is going to provide is a value already stored or not.

After user input, if we convert and have the same value as one already stored, the identifier becomes that of the currently stored value - this is called **mutation**.

This mutation mechanism does not work with strings or object types.

When it comes to the **str** type, the memory location where the user's value is stored will not change even if the same value already exists. This is why **str** strings are said to be **immutable**, as their **id** does not change when already allocated.

```
>>> # Defining a string variable
>>> name1 = "John"
>>> id(name1)
2239249198256
>>> # Ask the user to enter a string that will be the same value John
>>> name2 = input()
John
>>> name2
'John'
>>> id(name2)
2239203467440
>>> id(name1) == id(name2)
False
>>> # Even if we try to convert it doesn't change anything
>>> name1 = str(name1)
>>> name2 = str(name2)
>>> id(name1) == id(name2)
False
```

Comparisons rules with *is* and *is not*

In Python, the keywords **is not** and **is**, which are used by some programmers (including me, who fell into the trap, hence the purpose of this article), sometimes incorrectly, to check whether two variables have the same value.

However, this method is not at all the same when it comes to complex types such as **string**.

In reality, the **is** and **is not** keywords compare memory cells, i.e. the result of **id** values, not variable values.

These keywords verify that two variables are stored in the same memory cell. To compare values, we use **==** and **!=** when talking about strings.

```
>>> # 'is' and 'is not' without input on integers
>>> a = 1
>>> b = 1
>>> a is b
True
>>> id(a)
140735660221224
>>> id(b)
140735660221224
>>>
>>> # use of inputs
>>> a = 1
>>> b = input()
1
>>> a is b
False
>>> b = int(b)
>>> a is b
True
```

== and != on primitive types are identical to *is* and *is not*..

On the channels, it's not the same.

```
>>> # 'is' on channels without input
>>> a = "Hello"
>>> b = "Hello"
>>> a is b
True
>>> id(a) == id(b)
True
>>>
>>> # When you use 'input', it's not the same
>>> a = "Hi"
>>> b = input()
Hi
>>> a is b
False
>>> id(a) == id(b)
False
>>> # To compare values use ==
>>> a == b
True
```

To summarize:

- **is** and **is not** give the same result as **!=** and **==** for variables of simple types, since these values are mutable, and memory location identifiers change when the values are identical. Several variables with the same value will have the same memory location.
- **is** and **is not** are different from **!=** and **==** for complex variables such as strings. Two strings with the same values do not necessarily have the same memory cell identifiers. For strings, we use **==** to check the equality of string variable values and **!=** to check the difference between values.