Python Mutable and Immutable

Summary: in this tutorial, you'll learn about the mutable and immutable in Python.

Introduction to mutable and immuable in Python

In Python, everything is an object. An object has its own internal state. Some objects allow you to change their internal state and others don't.

An object whose internal state can be changed is called a mutable object, while an object whose internal state cannot be changed is called an immutable object.

The following are examples of immutable objects:

- Numbers (https://www.pythontutorial.net/python-basics/python-numbers/) (int (https://www.pythontutorial.net/advanced-python/python-integers/) , float (https://www.pythontutorial.net/advanced-python/python-float/) , bool (https://www.pythontutorial.net/advanced-python/python-bool/) ,...)
- Strings (https://www.pythontutorial.net/python-basics/python-string/)
- Tuples (https://www.pythontutorial.net/python-basics/python-tuples/)
- Frozen sets

And the following are examples of mutable objects:

- Lists (https://www.pythontutorial.net/python-basics/python-list/)
- Sets (https://www.pythontutorial.net/python-basics/python-set/)
- Dictionaries (https://www.pythontutorial.net/python-basics/python-dictionary/)

User-defined classes can be mutable or immutable, depending on whether their internal state can be changed or not.

Python immutable example

When you declare a variable (https://www.pythontutorial.net/python-basics/python-variables/) and assign its an integer, Python creates a new integer object and sets the variable to reference that object:

```
counter = 100
```

To get the memory address referenced by a variable, you use the `id()` function. The `id()` function returns a based-10 number:

```
print(id(counter))
```

Output:

```
140717671523072
```

And to convert the base-10 number to hexadecimal, you can use the `hex()` function:

```
print(hex(id(100)))
```

Output:

```
0x7ffb62d32300
```

In the memory, you have a variable called `counter` that references an integer object located at the `0x7ffb62d32300` address:

The following changes the `counter` to 200 and displays its value to the screen:

```
counter = 200
print(counter)
```

Output:

```
200
```

It seems that the value of the object referenced by the `counter` variable changes, but it doesn't.

In fact, Python creates a new integer object with the value 200 and reassigns the `counter` variable so that it references the new object like this:

The reassignment doesn't change the value of the first integer object. It just reassigns the reference.

The following shows the memory address of the new object referenced by the `counter` variable:

```
counter = 200
print(counter)
```

```
print(hex(id(counter)))
```

Output:

```
0x7ffb62d32f80
```

Python mutable example

The following defines a list (https://www.pythontutorial.net/python-basics/python-list/) of numbers and displays the memory address of the list:

```
ratings = [1, 2, 3]
print(hex(id(ratings)))  # 0x1840f97a340
```

Behind the scene, Python creates a new list object and sets the `ranks` variable to reference the list:

When you add a number to the list like this:

```
ratings.append(4)
```

Python directly changes the value of the list object:

And Python doesn't create a new object like the previous immutable example.

The following code shows the value and memory address of the list referenced by the `ratings` variable:

```
print(ratings)  # [1, 2, 3, 4]
print(hex(id(ratings)))  # 0x1840f97a340
```

As you can see clearly from the output, the memory address of the list is the same.

Python mutable and immutable example

It's important to understand that immutable objects are not something frozen or absolutely constant. Let's take a look at an example.

The following defines a tuple (https://www.pythontutorial.net/python-basics/python-tuples/) whose elements are the two lists:

```
low = [1, 2, 3]
high = [4, 5]

rankings = (low, high)
```

Since the `rankings` is a tuple, it's immutable. So you cannot add a new element to it or remove an element from it.

However, the `rankings` tuple contains two lists that are mutable objects. Therefore, you can add a new element to the `high` list without any issue:

```
high.append(6)
print(rankings)
```

And the `rankings` tuple changes to the following:

```
([1, 2, 3], [4, 5, 6])
```

Summary

- An object whose internal state **cannot be changed** is called immutable for example a number, a string, and a tuple.
- An object whose internal state **can be changed** is called mutable for example a list, a set, and a dictionary.