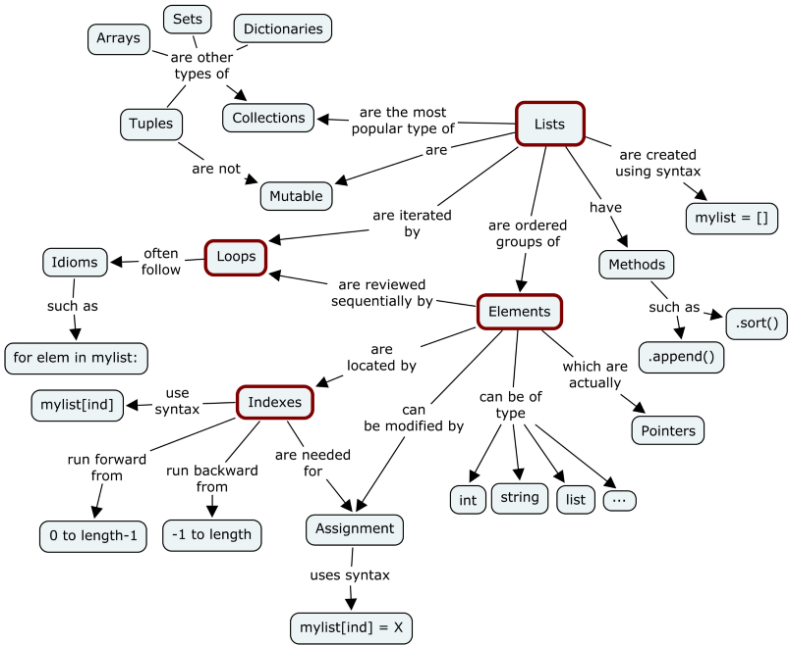
megha mohan Follow
May 25, 2017 · 5 min read

Mutable vs Immutable Objects in Python

Everything in Python is an object. And what every newcomer to Python should quickly learn is that all objects in Python can be either **mutable** or **immutable**.



Lets dive deeper into the details of it… Since everything in Python is an Object, every variable holds an object instance. When an object is initiated, it is assigned a unique object id. Its type is defined at runtime and once set can never change, however its state can be changed if it is mutable. Simple put, a **mutable** object can be changed after it is created, and an **immutable** object can't.

> Objects of built-in types like (int, float, bool, str, tuple, unicode) are immutable. Objects of built-in types like (list, set, dict) are mutable. Custom classes are generally mutable. To simulate immutability in a class, one should override attribute setting and deletion to raise exceptions.

Class	Description	Immutable?
bool	Boolean value	✓
int	integer (arbitrary magnitude)	✓
float	floating-point number	✓
list	mutable sequence of objects	
tuple	immutable sequence of objects	✓
str	character string	✓
set	unordered set of distinct objects	
frozenset	immutable form of set class	✓
dict	associative mapping (aka dictionary)	

Now comes the question, how do we find out if our variable is a mutable or immutable object. For this we should understand what 'ID' and 'TYPE' functions are for.

ID and TYPE

The built-in function **id**() returns the identity of an object as an integer. This integer usually corresponds to the object's location in memory, although this is specific to the Python implementation and the platform being used. The **is** operator compares the identity of two objects.

The built-in function type() returns the type of an object. Lets look at a simple example

```
''' Example 1 '''
>>> x = "Holberton"
>>> y = "Holberton"
>>> id(x)
140135852055856
>>> id(y)
140135852055856
>>> print(x is y) '''comparing the types'''
True

''' Example 2 '''
>>> a = 50
>>> type(a)
<class: 'int'>
>>> b = "Holberton"
>>> type(b)
<class: 'string'>
```

We have now seen how to compare two simple string variables to find out the types and id's .So using these two functions, we can check to see how different types of objects are associated with variables and how objects can be changed .

Mutable and Immutable Objects

So as we discussed earlier, a mutable object can change its state or contents and immutable objects cannot.

> ***Mutable objects**:*

> *list, dict, set, byte array*

> ***Immutable objects:***

> *int, float, complex, string, tuple, frozen set [note: immutable version of set], bytes*

A practical example to find out the mutability of object types

```
x = 10
```

```
x = y
```

We are creating an object of type int. identifiers x and y points to the same object.

```
id(x) == id(y)
```

```
id(y) == id(10)
```

if we do a simple operation.

```
x = x + 1
```

Now

```
id(x) != id(y)
```

```
id(x) != id(10)
```

The object in which x was tagged is changed. object 10 was never modified. **Immutable objects doesn't allow modification after creation**

In the case of **mutable objects**

```
m = list([1, 2, 3])
```

```
n = m
```

We are creating an object of type list. identifiers m and m tagged to the same list object, which is a collection of 3 immutable int objects.

```
id(m) == id(n)
```

Now poping an item from list object does change the object,

```
m.pop()
```

object id will not be changed

```
id(m) == id(n)
```

m and n will be pointing to the same list object after the modification. The list object will now contain [1, 2].

So what have we seen so far from the above examples?

- Python handles mutable and immutable objects differently.
- Immutable are quicker to access than mutable objects.
- Mutable objects are great to use when you need to change the size of the object, example list, dict etc.. Immutables are used when you need to ensure that the object you made will always stay the same.
- Immutable objects are fundamentally expensive to “change”, because doing so involves creating a copy. Changing mutable objects is cheap.

Exceptions in immutability..

Not all of the immutable objects are actually immutable. Confused? Let me explain.

As discussed earlier, Python containers liked tuples are immutable. That means value of a `tuple` can't be changed after it is created. But the "value" of a tuple is infact a sequence of names with unchangeable bindings to objects. The key thing to note is that the *bindings* are unchangeable, not the objects they are bound to.

Let us consider a tuple **t = ('holberton', [1, 2, 3])**

The above tuple **t** contains elements of different data types, the first one is an immutable string and the second one is a mutable list.The tuple itself isn't mutable . i.e. it doesn't have any methods for changing its contents. Likewise, the string is immutable because strings don't have any mutating methods. But the list object does have mutating methods, so it can be changed. This is a subtle point, but nonetheless important: the "value" of an immutable object *can't* change, but it's constituent objects *can*.

How objects are passed to Functions

Its important for us to know difference between mutable and immutable types and how they are treated when passed onto functions .Memory efficiency is highly affected when the proper objects are used.

For example if a mutable object is called by reference in a function, it can change the original variable itself. Hence to avoid this, the original variable needs to be copied to another variable. Immutable objects can be called by reference because its value cannot be changed anyways.

```
def updateList(list1):
    list1 += [10]

n = [5, 6]
print(id(n))                  # 140312184155336

updateList(n)
print(n)                      # [5, 6, 10]
print(id(n))                  # 140312184155336
```

As we can see from the above example, we have called the list via **call by reference**, so the changes are made to the original list itself.

Lets take a look at another example:

```
def updateNumber(n):
    print(id(n))
    n += 10
```

```
b = 5
print(id(b))                    # 10055680
updateNumber(b)                 # 10055680
print(b)                        # 5
```

In the above example the same object is passed to the function, but the variables value doesn't change even though the object is identical. This is called **pass by value**. So what is exactly happening here? When the value is called by the function, only the value of the variable is passed, not the object itself. So the variable referencing the object is not changed, but the object itself is being changed but within the function scope only. Hence the change is not reflected.