What are mutable and immutable objects in Python3?

python3 | Mutable object | Immutable object

Hey guys!! Interested to know something about **Mutability** and **Immutability**. Let's it check out!

Mutable Object

A **mutable object** is a changeable object that can be modified after it is created. Let's get it straight using an example.

Code

```
# List
lst = [1,2,3,4]

# printing list
print("The list is: ", lst)
# printing the id of list
print("The id of list is: ", id(lst))

# mutating second element of list
lst[2] = 4

# printing list
print("The list after change is: ", lst)
# printing the id of list
print("After changing the id of list is: ", id(lst))
```

What have you observed?

Two things, right? The `list` and its `id` before and after changing the list. The `list` in these two cases is *different*, but the list `id` is the *same*.

Now, the question is, what is this `id`?

It corresponds to the *object's location* in the **memory** and is specific to Python implementation.

Let's check out practically:

```
# Integer
a = 6
print("The id of 'a' is: ", id(a))

# String
str = "teacher"
print("The id of 'str' is: ", id(str))

# List
lst = [1,2,3,4,5,6]
print("The id of 'l' is: ", id(lst))
```

From this, you can observe that `ids` exist lists and all objects.

Immutable Object:

An immutable object is an object whose state can never be modified after it is created.

Code

```
# Integer
a = 6

# printing before mutating
print("Initial id: ",id(a))

# mutating interger `a`
a = 7

# printing after mutating
print("Id after changing: ",id(a))
```

The same variable `a` has multiple `id's`, right? Here the `id` represents an integer that corresponds to the memory location where a value is stored.

Remember that 6 is stored at `10919488` and, after it is changed to 7, the memory location is no longer the same.

Hence, 7 is created in another memory location because variable `a` is **immutable**.

Discussion:

Let's check out some examples that will enable us to discuss various scenarios.

```
# Integer
a = 6

# printing
print("Id of 'a' variable: ",id(a))

# no mutation
b = a

# printing
print("Id of 'b' variable:",id(b))
```

In the above example, what are the ids of `a` and `b`? Both are same!!

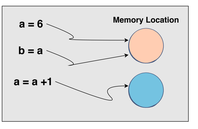
The reason is that both `a=6` and `b=6` point to the same memory location.

Let's check out another example:

```
# Integer
a = 6

# printing id of 'a'
print("a:", id(a))

# printing id of '6'
print("6:", id(6))

# no mutation
b = a
print("b:",id(b))

# mutating
a = a + 1
print("a:",id(a))


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

In the first two lines, the id of `a` and `b` are the same, but in the last two lines, the id of `a` is different from the id of `b`.

What could be the reason?



List of Mutable and Immutable objects

Mutable objects: list, dict, set, byte array

Immutable objects: int, float, complex, string, tuple, frozen set, bytes

Immutability of Tuple

– Until now, based on our discussion, we thought that immutable objects will never change their state, size, or the stored value.

– This is true for int, float, string, etc.

– A Tuple is considered immutable since, unlike its other counterparts, its state never changes.

– But a tuple is a mixture of both immutable and mutable objects.

– In a tuple, we might not change an immutable object(e.g., string), but we can change a mutable object (e.g., list).

See the example below :

```
t =("name", [1,2,3,4])
# here we cannot change the "name" and add additional string to it
# but we can change the list in the tuple
print(t) # before change of list
print(id(t))

t[1][3] = 5

print(t) #after change of list
print(id(t))

t[1].append(6) # we can increase the internal size of list

print(t)
print(id(t))
```

From the above code, we can observe that the overall size of the tuple isn't changed since it's immutable and there is no additional element being added to the tuple.

But we are changing the internal element list by changing the existing values or elements and adding new ones.

> Externally a tuple looks immutable.

Ok, we are done with it!!

So what have we learned...

– Python handles mutable and immutable objects in different ways.

– Mutable objects are of great use in scenarios where there is a need to change the object size (e.g., list, set, and so on...).

– Immutable objects's are used when you want an object you created to always stay the same as it was when created.

Truth

> *"Immutable objects are accessible in a quicker way than mutable objects."*

> *"Changing Immutable objects is fundamentally expensive as it involves creating a copy; but, changing mutable objects is considered cheap."*