CMPUT 175

UNIVERSITY OF ALBERTA

Mutable Object VS Immutable Object

Immutable Object: Whenever an object is instantiated, it is assigned a unique object id. The type of the object is defined at the runtime and it **can't be changed** afterward.

Mutable Object: Its state **can be changed** if it is a mutable object.

In simple words, a **mutable** object **can be changed** after it is created, and an **immutable** object **can't.**

https://www.geeksforgeeks.org/mutable-vs-immutable-objects-in-python/

Python Mutable and Immutable Object

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)	

https://medium.com/@meghamohan/mutable-and-immutable-side-of-python-c2145cf72747

Example of immutable object

In this example, we will take a **tuple** and try to modify its value at a particular index and print it. As a tuple is an **immutable** object, it will throw an error when we try to modify it.

```
# Python code to test that
# tuples are immutable

tuple1 = (0, 1, 2, 3)
tuple1[0] = 4
print(tuple1)
```

```
Traceback (most recent call last):
  File
"e0eaddff843a8695575daec34506f126.py"
, line 3, in
    tuple1[0]=4
  TypeError: 'tuple' object does not
  support item assignment
```

https://www.geeksforgeeks.org/mutable-vs-immutable-objects-in-python/

Example of immutable object

In this example, we will take a **string** and try to modify its value. Similar to the tuple, strings are **immutable** and will throw an error.

```
# Python code to test that
# strings are immutable

message = "Welcome to GeeksforGeeks"
message[0] = 'p'
print(message)
```

```
Traceback (most recent call last):

  File
"/home/ff856d3c5411909530c4d328eec
a165b.py", line 3, in

    message[0] = 'p'

TypeError: 'str' object does not
support item assignment
```

https://www.geeksforgeeks.org/mutable-vs-immutable-objects-in-python/

Example of mutable object

A **list** in Python is **mutable**, that is, it allows us to change its value once it is created

```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)

my_list.insert(1, 5)
print(my_list)

my_list.remove(2)
print(my_list)

popped_element = my_list.pop(0)
print(my_list)
print(popped_element)
```

```
Output:
[1, 2, 3, 4]
[1, 5, 2, 3, 4]
[1, 5, 3, 4]
[5, 3, 4]
  1
```

https://www.geeksforgeeks.org/mutable-vs-immutable-objects-in-python/

Example of mutable object (advanced)

Update the original list in a function will also apply the update to the original list.

```
initial_list = [1, 2, 3]

def duplicate_last(a_list):
    last_element = a_list[-1]
    a_list.append(last_element)
    return a_list

new_list = duplicate_last(a_list = initial_list)
print(new_list)
print(initial_list)
```

```
Output:
[1, 2, 3, 3]
[1, 2, 3, 3]
```

Example of mutable object (advanced)

If we do not want the original list to be changed, we can use a **copy()** method.

```
initial_list = [1, 2, 3]

def duplicate_last(a_list):
    last_element = a_list[-1]
    a_list.append(last_element)
    return a_list

new_list = duplicate_last(a_list = initial_list.copy()) # copy list
print(new_list)
print(initial_list)
```

```
Output:
[1, 2, 3, 3]
[1, 2, 3]
```

why: using `.copy()` creates a separate copy of the list, so that instead of pointing to `initial_list` itself, `a_list` points to a new list that starts as a copy of `initial_list`. Any changes that are made to `a_list` after that point are made to that separate list, not `initial_list` itself, thus the global value of `initial_list` is unchanged.