Objects in Memeory and Aliasing, Mutation ,Cloning

In python everything is an object

- integer, strings, floats
- tuple
- lists
- dictionary
- boolean
- functions
- Exceptions

To check

```
isinstance("hello", object)
```

Built-in Objects in Python and their Methods

1️⃣ Lists

These are some built-in **list methods** in Python:

- .extend()
- .append()
- .insert()
- .count()
- .index()
- .sort()
- .pop()
- .remove()

```
>>> a =[1,5,6,2,8]
>>> a.extend([7,2])
>>> a
[1,5,6,2,8,7,2]
>>> a.append(8)
>>> a
[1,5,6,2,8,7,2,8]
>>> a.insert(1,3)
>>> a
[1,3,5,6,2,8,7,2,8]
>>> a.count(2)
2
>>> a.index(5)
2
>>> a.sort()
>>> a
```

```
[1,2,2,3,5,6,7,8,8]
>>> a.pop()
8
>>> a
[1,2,2,3,5,6,7,8]
>>> a.remove(1)
>>> a
[2,2,3,5,6,7,8]
```

2️⃣ Tuples

For tuples we have:

- .count()
- .index()

```
>>> b =(1,6,2,8)
>>> b.count(6)
1
>>> b.index(2)
2
```

3️⃣ Strings

For strings we have:

- .capitalize()
- .casefold()
- .count()
- .isdigit()
- .islower()
- .isspace()
- .replace()
- .strip()

```
>>> c ="OOP"
>>> c.capitalize()
'Oop'
>>> c.casefold()
'oop'
>>> c.count("O")
2
>>> c.isdigit()
False
>>> c.islower()
False
>>> c.isspace()
False
>>> c.replace("O","A")
'AAP'
```

```
>>> c.strip()
'OOP'
```

For more info on string methods, I suggest reading this article: Python String Methods.

4️⃣ Dictionary

And for dictionaries we have:

- .get()
- .keys()
- .values()

```
>>> d ={"a":5,"b":10}
>>> d.get("b")
10
>>> d.items()
dict_items([('a',5),('b',10)])
>>> d.keys()
dict_keys(['a','b'])
>>> d.popitem()
('b',10)
>>> d.values()
dict_values([5])
```

id() function

It returns the identity of an object.This acts as the address of the object in memory
Two objects may have the same id if one of them deleted during the runtime

```
class Circle:

    def __init__(self, radius, color):
        self.radius = radius
        self.color = color

circle1 = Circle(54,"red")
circle2 = Circle(23,"yellow")
circle3 = circle2
print(circle1 is circle2)
print(circle3 is circle2)
circle3.color = "blue"
print(circle2.color)
```

output

```
False
True
blue
```

```
print(id(circle1))
print(circle1) # is the hexadecimal version of the id which shows the memory address of the
object
print(hex(id(circle1)))
```

output

```
140095708904248
<__main__.Circle object at 0x7f6a92f6bb38>
0x7f6a92f6bb38
```

<__main__.Circle object at 0x7f6a92f6bb38> gives the hexadeximal version of the object id

is operator

- used to check if two variables reference the same object in memory.True if they refer to the same object and False if they refer to different objects

'==' --- to check if the values of the objects of the two variables are the same

'is' --- to check if the variables have the same referecne to the object

```
a = [5,1,3,7,3]
b = [5,1,3,7,3]
print(id(a))
print(id(b))
print(a is b) #the is operator is used to see if they refer to the same object in memory
print(a==b)
```

output

```
140095708881800
140095708881864
False
True
```

Comparing Objects of User-Defined Classes with ==

the == operator compares the values, not if they are the same object. But take a look at this example:

```
>>> class Dog:
        def __init__(self, age):
                self.age = age
>>> a = Dog(5)
>>> b = Dog(5)
>>> a == b
False
```

They comparison operator doesn't return True, even if their instance attributes have the same value. Why is this?

Objects created from user-defined classes have to meet two conditions for the expression obj1 == obj2 to evaluate to True.

- They have to refer to the same object (x is y has to evaluate to True)
- The expression hash(x) == hash(y) has to evaluate to True.

The hash() function maps the object to a unique integer. For more information on the hash() built-in function, please refer to this article.

According to the Python Documentation:

".... **all objects compare unequal (except with themselves)** ... **x == y implies both that x is y and hash(x) == hash(y).**"- source
To find more information on this specific quote of the documentation, please refer to the article.

This is why comparing an object from a user-defined class with itself using the == operator returns True:

```
>>> class Dog:
        def __init__(self, age):
                self.age = age
>>> a = Dog(5)
>>> b = Dog(5)
>>> a == b
False
>>> a == a
True
>>> b == b
True
>>> b == a
False
```

As you can see, the values returned by hash() are not equal for the two objects:

```
>>> hash(a)
-2143773140
>>> hash(b)
3710489
```

So when the hash values are different, this expression hash(a) == hash(b) will be False and the expression a is b and hash(a) == hash(b) returns False, so a == b returns False.

Exceptions

small integers in the range of -5 to 256 give a reference back the exisiting object

```
a = 5
b = 5
print(a is b)
```

OUTPUT

```
True
```

```
a = "Hello"
b = "Hello"
print(a is b)
```

OUTPUT

```
True
```

as strings are immutable objects.The interpretor makes the same object for the same string for memory optimization

```
c = "abcd"
d = "".join(["a","b","c","d"])
print(c is d)
```

OUTPUT

```
False
```

Rules of string interning

- ~~All strings of lenght 1 and 0 are interned~~
- ~~Strings that are composed of ASCII leterrs(a-z,A-Z) ,digits(0,9) or sepecial symbols are not interned~~

more info medium

To confirm that **objects are passed by reference in Python**, we can see what happens in this example using the id() function:

```
# List object
>>> a = [5, 6, 2]
# Function that takes a list object as argument
>>> def f(x):
        print(id(x))
        return x
# Get the ids
>>> id(a)
63319880
>>> f(a)
63319880
# Value returned
[5, 6, 2]
```

Notice how the id of the formal parameter is the same id of the object that was passed as the argument, which confirms that we are working with the original object in memory that was passed as argument. A reference to that object was passed as argument, not a copy of the object.

Aliasing

Aliasing when the same memory address can be accessed using different names

ex :

```
a = {"b":5,"a":24,"c":56}
print(id(a))
z = a
print(id(z)) #same id
```

OUTPUT

```
139905938554952
139905938554952
```

some of the risks are you can change the attributes of varibles due to the same reference,hence the original name would also be affected

Mutation

It is ablity to change after being created.In python there is mutable and immutable objects

mutable objects

- lists
- sets
- dictionaries

immutable objects

- Booleans
- Integers
- Floats
- Strings
- Tuples

example

```
a = [1,2,3,4,5]
a[2] = -5
print(a)
b = (1,2,3,4,5)
b[2] = -5
print(b)
```

OUPUT

```
[1, 2, -5, 4, 5]
Traceback (most recent call last):
  File "memory.py", line 56, in <module>
    b[2] = -5
TypeError: 'tuple' object does not support item assignment
```

tuple is immutable while list is mutable

advantages

- more memory effiecent as we can reuse

disadvantages

- can lead to bugs as we might change the value by mistake

can lead to bugs as we might change the value by mistake

```
e = {[1,2,3]:"a",[4,5,6]:"b"}
```

OUTPUT

```
Traceback (most recent call last):
  File "memory.py", line 59, in <module>
    e = {[1,2,3]:"a",[4,5,6]:"b"}
TypeError: unhashable type: 'list'
```

as the keys are first configured to a hashtable

But this works as tuple is immutable

```
f = {(1,2,3):"a",(4,5,6):"b"}
```

You have to be very careful with built-in methods because some of them mutate the original object.

For example:

```
>>> a = [6, 2, 7, 1]
>>> a.sort()
>>> a
[1, 2, 6, 7]
```

Here you can see (above) that the .sort() method mutated the original list.

To achieve this same functionality without mutating the original object, you should use the sorted() function.

```
>>> a = [6, 2, 7, 1]
>>> sorted(a)
[1, 2, 6, 7]
>>> a
[6, 2, 7, 1]
```

As you can see, this function returns a sorted "version" of the list (a copy) without modifying the original list.

avoid using mutable data types such as lists as default arguments:

Default arguments are initialized when the methods are initially processed, so there is only one copy of each default argument. They are not created when you call the method, they are created when the program starts to run.

1️⃣ Example

If you use a list as a default argument, the same list (reference) will be reused as the default argument for

every method call.

Below, you can see how we use an empty list as the default argument for the clients parameter in __init__().

You would expect this to work normally, creating an empty list (a new object) for the default argument every time that you create an instance. **But this is not what happens...**

When we start adding elements to the list, you can see that the two instances were modified (please see the example below).

```
>>> class WaitingList:
        def __init__(self, clients=[]): # The default argument is an empty list
                self.clients = clients
        def add_client(self, client):
                self.clients.append(client)
# Create the instances
>>> waiting_list1 = WaitingList()
>>> waiting_list2 = WaitingList()
# Add a client to the first waiting list
>>> waiting_list1.add_client("Jake")
# Both of them were modified!
>>> waiting_list1.clients
['Jake']
>>> waiting_list2.clients
['Jake']
```

What truly happens behind the scenes is that when this line is executed: self.clients.append(client), the new client is added to the same list! Not to a separate list that corresponds to each instance.

You can check that self.clients references the same list in the two instances with the id() function. Notice how the ids are equal.

```
>>> class WaitingList:
        def __init__(self, clients=[]):
                self.clients = clients
                print("List id:", id(self.clients))
        def add_client(self, client):
                self.clients.append(client)
>>> waiting_list1 = WaitingList()
List id: 48967144
>>> waiting_list2 = WaitingList()
List id: 48967144
```

2️⃣ Solution

The solution to this problem is to avoid using the list directly as a default argument, and use this instead:

```
class WaitingList:
        def __init__(self, clients=None):
                if clients == None:
                        self.clients = []
```

```
                else:
                        self.clients = clients
        def add_client(self, client):
                self.clients.append(client)
```

None is used as the default argument so you can omit the argument when you create the instance. If the value of clients is None, the attribute is initialized as an empty list. Otherwise, the value passed as argument is assigned.

And this will generate the behavior that you expect:

```
>>> waiting_list1 = WaitingList()
>>> waiting_list2 = WaitingList()
>>> waiting_list1.add_client("Jake")
>>> waiting_list1.clients
['Jake']
>>> waiting_list2.clients
[]
```

Now the instances reference two separate lists and modifying one doesn't modify the other.

A very important distinction for immutable objects is that they can contain mutable objects, and these mutable objects can be modified even if the container cannot be modified.

Here is an example that illustrates this. We have this initial list:

```
>>> a = ([1, 2, 3], "abc", 56)
```

If we try to change an element of the tuple, we get an error because tuples are immutable objects:

```
>>> a[0] = 12
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    a[0] = 12
TypeError: 'tuple' object does not support item assignment
```

⚠️ But... if we try to modify an element of the tuple that **is** mutable, we won't get any errors.

Here we change the element at index 1 of the element at index 0 of the tuple. We are changing the number 2 of this list [1, 2, 3] for the number 4 ([1, 4, 3]).

```
>>> a[0][1] = 4
>>> a
([1, 4, 3], 'abc', 56)
```

We modified it! No errors were thrown. So choosing an immutable "container" object (e.g a tuple) doesn't "protect" the elements themselves from change if they are mutable (e.g lists).

Cloning

to create a copy of the object

```
<clone_var> = <list>[:]
```

Example

```
a = [1,2,3,4,5]
b = a[:]
print(id(a))
print(id(b))
```

OUTPUT

```
140016487683016
140016487683656
```

the id is different as a clone is created

Shallow vs. Deep Copy of an Object

Now let's see the two different methods that you can use to create copies of the objects that you are working with (e.g. lists, tuples, dictionaries, custom objects, and others).

1️⃣ Shallow Copy

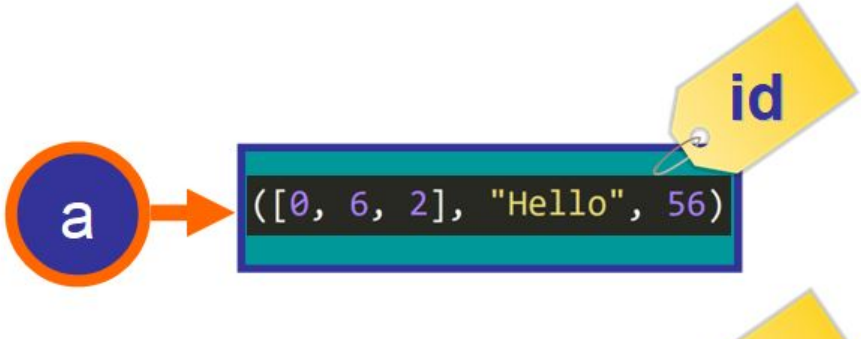
When you make a **shallow copy** of an object, you are making a new object in memory, a new reference, but the content of the object will still point to the same objects.

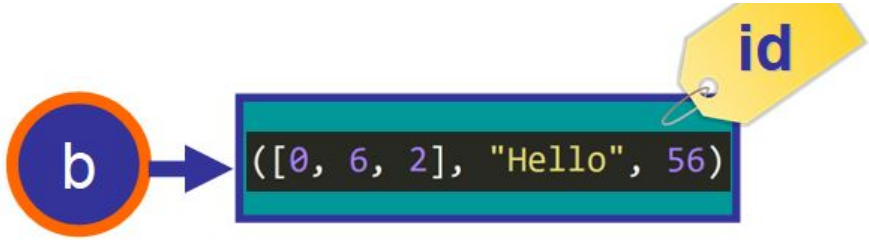
Let me illustrate this:

If we have a tuple that contains a list (which is a mutable object) the tuple and we create a clone of the tuple, like this:

```
>>> a = ([0, 6, 2], "Hello", 56)
>>> b = a[:]
```

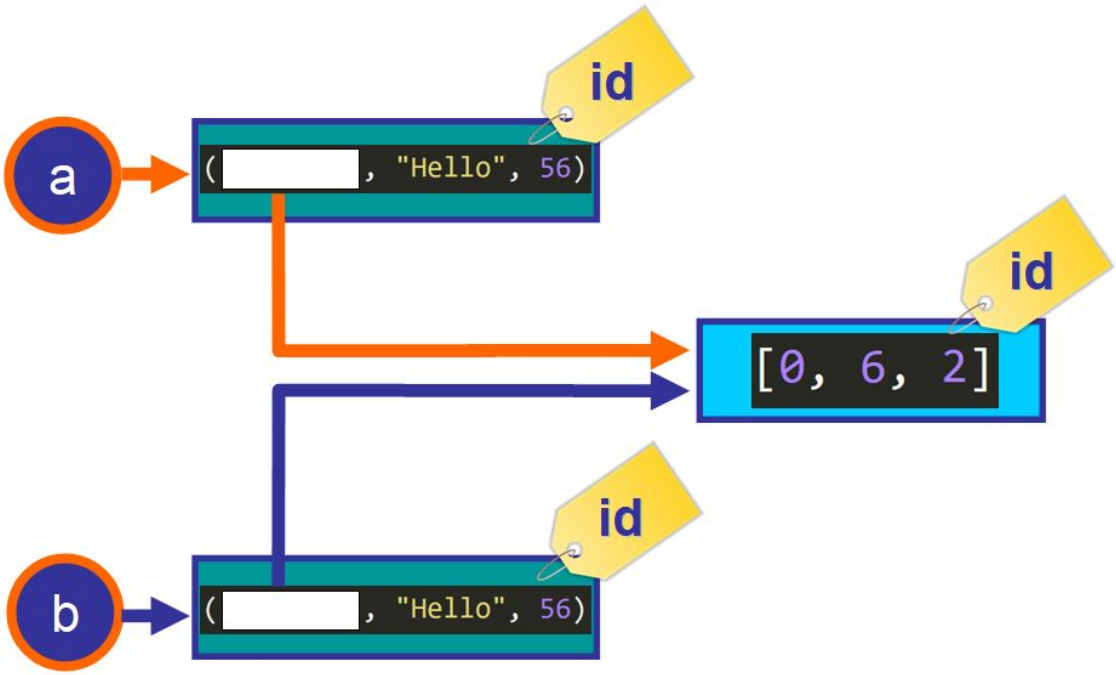
We could think that this is the result (below). Two independent objects with a different list object in each one them. But...





This is not the case. The tuple only contains a reference to the list object, not the list object itself.

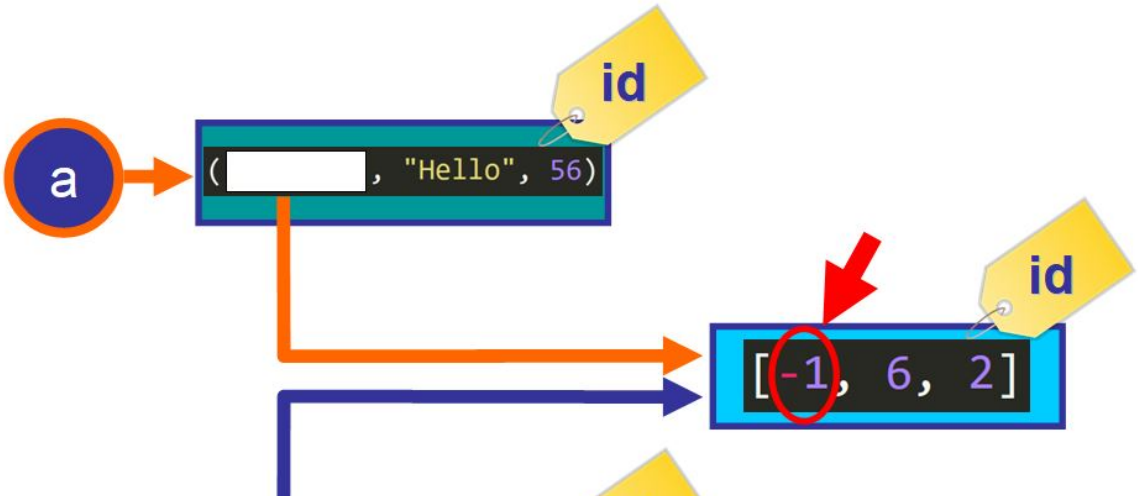
This would be a more accurate diagram of the result (only taking the list as an example. The other objects are stored as references as well):

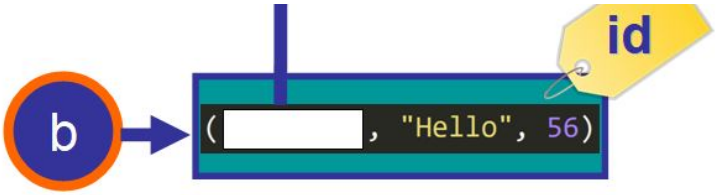


So if you modify the list contained in the tuple referenced by b, like this:

```
>>> b[0][0] = -1
```

You are actually modifying the list in memory, so both tuples are affected since they only contain a **reference** to the list:





This is why the list was modified "in both tuples", even if that wasn't our initial intention:

```
>>> a
([-1, 6, 2], 'Hello', 56)
>>> b
([-1, 6, 2], 'Hello', 56)
```

⚠️ **This is something that you need to be really careful of.**

This is another example with the .copy() method that you can call on dictionaries:

```
>>> a = {"Nora": ["055-452-322", "Washington Ave."], "Gino": ["006-545", "5th Avenue"]}
>>> b = a.copy()
>>> a
{'Nora': ['055-452-322', 'Washington Ave.'], 'Gino': ['006-545', '5th Avenue']}
>>> b
{'Nora': ['055-452-322', 'Washington Ave.'], 'Gino': ['006-545', '5th Avenue']}
# If you modify an element of the list
>>> b["Nora"][0] = "56"
# They are both affected. The original and the copy.
>>> b
{'Nora': ['56', 'Washington Ave.'], 'Gino': ['006-545', '5th Avenue']}
>>> a
{'Nora': ['56', 'Washington Ave.'], 'Gino': ['006-545', '5th Avenue']}
```

Shallow copies of dictionaries can be made using dict.copy(), and of lists by assigning a slice of the entire list, for example, copied_list = original_list[:]. - source

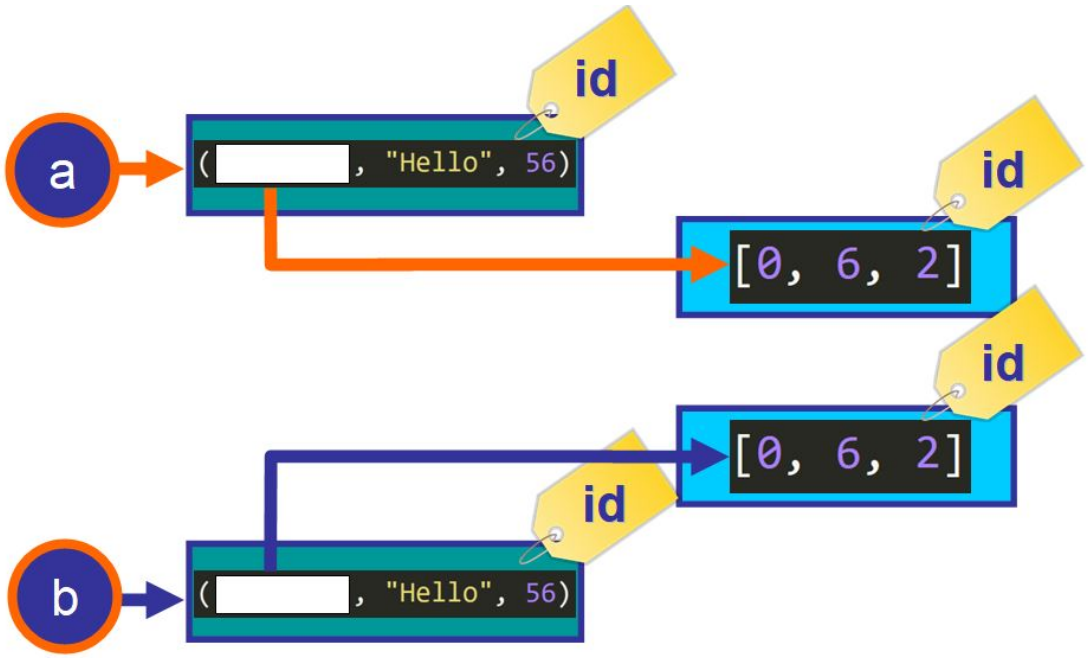
💡 **Tips:** You can also use the **copy module** to create a shallow copy.

To do this, you would have to add this line: import copy at the very top of your script, like this:

```
>>> import copy
>>> a = ([5, 2, 6, 2], "Welcome", 67)
>>> b = copy.copy(a)
>>> b[0][0] = -1
# They are both modified
>>> a
([-1, 2, 6, 2], 'Welcome', 67)
>>> b
([-1, 2, 6, 2], 'Welcome', 67)
```

2️⃣ Deep Copy

With a deep copy, in addition to creating a copy of the "container" object, you also create a copy of the elements contained in the object.



So if you modify the list, **you will not affect other names or objects that reference them**: