Object References, Mutability, and Recycling:

In python, variables are not boxes containing the object itself, but labels sticked to the object. This concept is very important when it comes to referencing an object because an object can have one variable, label, name, or alias, or more than one. But one variable, can only be bound to one object. There are three main concepts that will be discussed, the object’s identity, the object’s value, and the object’s alias, or aliases. Here we see why a tuple can be immutable, but the values that it contains can change, what results in the immutable object, to change.

From this particular situation comes a problem that sometimes can be invisible, or maybe you have never encounter it. This problem is that arguments passed into a function are contained inside of a tuple, and that is for good reason. So, is there going to be a problem if we change the tuple’s value? That doesn’t make much sense now so let’s take a look, to the **Function Parameter Referencing Problem**.

Variables Are Not Boxes:

The figure shown next is a simple interaction that the “variable as boxes” idea can’t explain. The figure follows a code demonstrating in a console session that the boxes idea is wrong, and the labels are more accurate.

Correct and Incorrect way of seeing the assignment in python:

>>> a = [1, 2, 3]

>>> b = a

>>> a.append(4)

>>> b

[1, 2, 3, 4]

Here we can see how the change made in a , also affects b. This is because a and b are not different objects itself, a and b are just a way of referencing the list object [1, 2, 3] in memory, so when the change is done using a to reference the object, when you call it using b, the change will also be visible since what it changed was the object itself. This can be better illustrated in the next figure.

Graphical user interface, text

Description automatically generated

If you imagine variables are like boxes, you can’t make sense of assignment in Python; instead, think of variables as sticky notes

Objects are called before the assignment:

>>> class Gizmo:

... def \_\_init\_\_(self):

... print('Gizmo id: %d' % id(self))

...

>>> x = Gizmo()

Gizmo id: 4301489152

>>> y = Gizmo() \* 10

Gizmo id: 4301489432

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: unsupported operand type(s) for \*: 'Gizmo' and 'int'

>>>

>>> dir()

['Gizmo', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_',

'\_\_package\_\_', '\_\_spec\_\_', 'x']

Here we have created a class named Gizmo(), this class’s \_\_init\_\_ method has a print inside of it. This means that whenever the class is initialized, whenever a new object from this class is created, it will print that line. This is very important, and we made is to see clearly step by step what happens when. Let’s focus our attention on the line in blue. We assign the Gizmo()\*10 to the variable y. Obviously, since we didn’t overwrite the \_\_mult\_\_ method from the main object, it will try to do it and it will fail, raising an operational error between a Gizmo object and the int. Now, here is the fun part, we can see a line being printed after the blue line which means that the object was created, and the \_\_init\_\_ function did run, but right after, the Error was raised and even when the object was created, the variable y was never bound to the left side, because an Error was raised, and the variable, was never created, but the object was. We can see that the variable was never created because when we run the dir() command, the variable doesn’t show anywhere in the global variables (Second blue part).

Identity, Equality, and Aliases:

Let’s talk now about when is an object equal to another and when is an object the very object itself (What? Yes, keep reading). We will use a dictionary pointing to someone’s name, a doctor with a very long name.

Lewis Carroll is the pen name of Prof. Charles Lutwidge Dodgson. Mr. Carroll is not only equal to Prof. Dodgson: they are one and the same.

>> charles = {'name': 'Charles L. Dodgson', 'born': 1832}

>>> lewis = charles

>>> lewis is charles

True

>>> id(charles), id(lewis)

(4300473992, 4300473992)

>>> lewis['balance'] = 950

>>> charles

{'name': 'Charles L. Dodgson', 'balance': 950, 'born': 1832}

We can see here that both lewis and charles have the same id, this means that they point to the same position in memory, to the same object. This is why, when we add a new field to the dictionary (the lines in blue), and we do so use the variable lewis, this also affects the variable charles, because ethe variable is just a name, what is actually changed is the object which they reference.

Let’s see another case where we have another dictionary, of another person, who’s data are the same, but the id is different, meaning that it points somewhere else.

>>> alex = {'name': 'Charles L. Dodgson', 'born': 1832, 'balance': 950}

>>> alex == charles

True

>>> alex is not charles

True

Here we have an object that is a replica of the first object referenced with the variables lewis and charles, when we compare them with the == (this is the same that a.\_\_eq\_\_(b) ) they are compared to be equal because that is how the dictionaries are compared to each other. Then, when we use the is not syntaxis we get True. This is because the is uses the same \_\_eq\_\_ method as the original object, it directly compares the object’s ids.

The real meaning of an object’s ID is Implementation-Dependent. In CPython, id() returns the memory address of the object, but it may be something else in another Python interpreter. The key point is that the ID is guaranteed to be a unique numeric label, and it will never change during the life of the object.

Choosing Between == and is:

The == operator compares the values of objects (the data they hold), while is compares their identities. We often care about values and not identities, so == appears more frequently than is in Python code. However, if you are comparing a variable to a singleton, then it makes sense to use is. By far, the most common case is checking whether a variable is bound to None. This is the recommended way to do it:

**x is None**

And the proper way to write its negation is:

**x is not None**

The is operator is faster than == because it cannot be overloaded, so Python does not have to find and invoke special methods to evaluate it, and computing is as simple as comparing two integer IDs. In contrast, a == b is syntactic sugar for a.\_\_eq\_\_(b). The \_\_eq\_\_ method inherited from object compares object IDs, so it produces the same result as is. But most built-in type override \_\_eq\_\_ with more meaningful implementations that actually take into account the values of the object attributes. Equality may involve a lot of processing—for example, when comparing large collections or deeply nested structures.

The Relative Immutability of Tuples:

Tuples, like most Python collections—lists, dicts, sets, etc.—hold **references** to **objects**. If the referenced items are mutable, they may change even if the tuple itself does not. In other words, the immutability of tuples really refers to the physical contents of the tuple data structure (i.e., the references it holds), and does not extend to the referenced objects.

>>> t1 = (1, 2, [30, 40])

>>> t2 = (1, 2, [30, 40])

>>> t1 == t2

True

>>> id(t1[-1])

4302515784

>>> t1[-1].append(99)

>>> t1

(1, 2, [30, 40, 99])

>>> id(t1[-1])

4302515784

>>> t1 == t2

False

Here we can see that the only thing that makes a tuple immutable, is the fact that the ids of the objects it contains must remain the same. This means that you can change any mutable type contained in a tuple because even if you change the object, the objects id won’t change, unless you try and assign it other object or change a flat sequence like str, bytes, and array.array which contain the actual object instead of containing a reference to the object.

Copies Are Shallow by Default:

The easiest way to copy a list (or most built-in mutable collections) is to use the built-in constructor for the type itself. For example:

>>> l1 = [3, [55, 44], (7, 8, 9)]

>>> l2 = list(l1)

>>> l2

[3, [55, 44], (7, 8, 9)]

>>> l2 == l1

True

>>> l2 is l1

False

For lists and other mutable sequences, the shortcut l2 = l1[:] also makes a copy. However, using the constructor or [:] produces a shallow copy (i.e., the outermost container is duplicated, but the copy is filled with references to the same items held by the original container). This saves memory and causes no problems if all the items are immutable. But if there are mutable items, this may lead to unpleasant surprises.

To visualize this better we can use the Online Python Tutor **(https://pythontutor.com/)** which is an amazing tool that shows us step by step how is it that our code is running, and which object is being referenced where.

Graphical user interface

Description automatically generated

View generated using the python online tutor where we can see that l1 and l2 refer to distinct lists, but the lists share references to the same inner list object [66, 55, 44] and tuple (7, 8, 9).

Deep and Shallow Copies of Arbitrary Objects:

Working with shallow copies is not always a problem, but sometimes you need to make deep copies The copy module provides the deepcopy and copy functions that return deep and shallow copies of arbitrary objects.

Bus picks up and drops off passengers:

class Bus:

    def \_\_init\_\_(self, passengers=None):

        if passengers is None:

            self.passengers = []

        else:

            self.passengers = list(passengers)

    def pick(self, name):

        self.passengers.append(name)

    def drop(self, name):

        self.passengers.remove(name)

Using the copy module and its functions copy.copy and copy.deepcopy we can get very different results when making copies of the same object.

>>> import copy

>>> bus1 = Bus(['Alice', 'Bill', 'Claire', 'David'])

>>> bus2 = copy.copy(bus1)

>>> bus3 = copy.deepcopy(bus1)

>>> id(bus1), id(bus2), id(bus3)

(4301498296, 4301499416, 4301499752)

>>> bus1.drop('Bill')

>>> bus2.passengers

['Alice', 'Claire', 'David']

>>> id(bus1.passengers), id(bus2.passengers), id(bus3.passengers)

(4302658568, 4302658568, 4302657800)

>>> bus3.passengers

['Alice', 'Bill', 'Claire', 'David']

*Here we can see that we have effectively created a copy from the original object bus1 when creating the objects bus2 and bus3, we can know that by checking the object’s id, which is different in each case. But the objects inside bus1 and bus2 are pointing to the same internal objects because a shallow copy was used, in this case the copy.copy function. The bus3 on the other hand was created using the copy.deepcopy function, which creates new internal objects inside the new bus. This difference between bus2 and bus3 results in two main features. By creating an objects which internal values point at the same position in memory we save space, but we take the risk of undesired events. This doesn’t happen in the bus1-bus3 comparison because internal objects are different which results in more space used in memory, but complete freedom to manipulate one without affecting the other.*

Cyclic References:

A cyclic reference is when one object references another object, and this object references the first one at the same time. Both objects reference each other infinitively, as two mirrors one in front of the other. Note that making deep copies is not a simple matter in the general case. Objects may have cyclic references that would cause a naïve algorithm to enter an infinite loop. The deepcopy function remembers the objects already copied to handle cyclic references gracefully.

Cyclic references: b refers to a, b is appended to a; deepcopy still manages to copy a:

>>> a = [10, 20]

>>> b = [a, 30]

>>> a.append(b)

>>> a

[10, 20, [[...], 30]]

>>> from copy import deepcopy

>>> c = deepcopy(a)

>>> c

[10, 20, [[...], 30]]

Function Parameters as References:

The only mode of parameter passing in Python is call by sharing. This simply means that whenever passing an argument to the function, this argument name we gave to it, becomes one more alias to the objects we are referencing. This means that any mutable object can be modifiable when passed to a function. Non mu6table objects cannot be modified because the function cannot change the identity of the objects itself but it can change the mutable values while keeping the objects identity.

A function may change any mutable object it receives:

>>> def f(a, b):

... a += b

... return a

-----------------------------------Integer----------------------------------

>>> x = 1

>>> y = 2

>>> f(x, y)

3>>> x, y

(1, 2)

-------------------------------------List-----------------------------------

>>> s = [1, 2]

>>> r = [3, 4]

>>> f(s, r)

[1, 2, 3, 4]

>>> s, r

([1, 2, 3, 4], [3, 4])

-------------------------------------Tuple----------------------------------

>>> t = (10, 20)

>>> u = (30, 40)

>>> f(t, u)

(10, 20, 30, 40)

>>> t, u

((10, 20), (30, 40))

IMPORTANT

*This function will take two arguments and try to extend the first argument to the second using the += operator. This operator works differently across every builtin. The use of this operator in a flat sequence like the string will return a completely new object unbounding the name from one object and placing it to the new resulting object. This also happens with the integers. In the first case, “x” points to position in memory of the number 1, and also the variable “a”; then the variable “y” points to the number 2, as well as the variable name “b”. When the function runs and we use the operator += we detach the variable name “a” from the number 1, but not the variable name “x”, which means that when returning the variable a, we will obtain the integer 3, but when we check the variable “x” is still referencing the number 1 because ”x” was never unbound and bound. The same happens in the case oof the tuple.*

*In the list’s case, we also have to variables referencing the list, the variables “a” and “s” reference the list [1,2] and the variables “b” and “r” reference the list [3,4]. The difference here is that the operator += does not return a new object when is used in lists, instead it will extend the list to the second element, changing its value, but not the object itself. This means that in the end, the variables “a” and “s” reference the extended list [1,2,3,4] with the same initial ID and the variables “b” and “r” reference the list [3,4].*

Mutable Types as Parameter Defaults: Bad Idea:

Whenever passing a default parameter to a function I python we must becarefull not to pass a mutable object as a list because we might be referencing the same objects through all the calls of the same function object. Here we tried to be clever and instead of having a default value of passengers=None, we have passengers=[], thus avoiding the if in the previous \_\_init\_\_. This “cleverness” gets us into trouble.

A simple class to illustrate the danger of a mutable default:

class HauntedBus:

    """A bus model haunted by ghost passengers"""

    def \_\_init\_\_(self, passengers=[]):

        self.passengers = passengers

    def pick(self, name):

        self.passengers.append(name)

    def drop(self, name):

        self.passengers.remove(name)

>>> bus1 = HauntedBus(['Alice', 'Bill'])

>>> bus1.passengers

['Alice', 'Bill']

>>> bus1.pick('Charlie')

>>> bus1.drop('Alice')

>>> bus1.passengers

['Bill', 'Charlie']

>>> bus2 = HauntedBus()

>>> bus2.pick('Carrie')

>>> bus2.passengers

['Carrie']

>>> bus3 = HauntedBus()

>>> bus3.passengers

['Carrie']

>>> bus3.pick('Dave')

>>> bus2.passengers

['Carrie', 'Dave']

>>> bus2.passengers is bus3.passengers

True

>>> bus1.passengers

['Bill', 'Charlie']

Here we can see that estrange things happen to the bus, and the reason why, is because the variable passengers inside the bus2 and the bus3 is pointing to the same object. This doesn’t happen with the bus1 because he was initialized with another objects different than the default “[]”.

>>> HauntedBus.\_\_init\_\_.\_\_defaults\_\_[0] is bus2.passengers

True

Defensive Programming with Mutable Parameters:

When we are passing an argument to a function we must also think if we expect the original argument to be modified or not:

Passengers disappear when dropped by a TwilightBus:

>>> basketball\_team = ['Sue', 'Tina', 'Maya', 'Diana', 'Pat']

>>> bus = TwilightBus(basketball\_team)

>>> bus.drop('Tina')

>>> bus.drop('Pat')

>>> basketball\_team

['Sue', 'Maya', 'Diana']

This is bad code, whenever a student is dropped from the buss, is also dropped from the basketball team list. This is because the object used in the function to add or remove names is the same object as passed. This can be solved by calling the builtin list class initializer and create a copy of the object.

def \_\_init\_\_(self, passengers=None):

if passengers is None:

self.passengers = []

else:

self.passengers = list(passengers)

del and Garbage Collection:

The del statement deletes names, not objects. An object may be garbage collected as result of a del command, but only if the variable deleted holds the last reference to the object, or if the object becomes unreachable. If two objects refer to each other, they may be destroyed if the garbage collector determines that they are otherwise unreachable because their only references are their mutual references. Rebinding a variable may also cause the number of references to an object to reach zero, causing its destruction. In CPython, the primary algorithm for garbage collection is reference counting. Essentially, each object keeps count of how many references point to it. As soon as that refcount reaches zero, the object is immediately destroyed: CPython calls the \_\_del\_\_ method on the object (if defined) and then frees the memory allocated to the object. In CPython 2.0, a generational garbage collection algorithm was added to detect groups of objects involved in reference cycles—which may be unreachable even with outstand‐ ing references to them, when all the mutual references are contained within the group. Other implementations of Python have more sophisticated garbage collectors that do not rely on reference counting, which means the \_\_del\_\_ method may not be called immediately when there are no more references to the object.