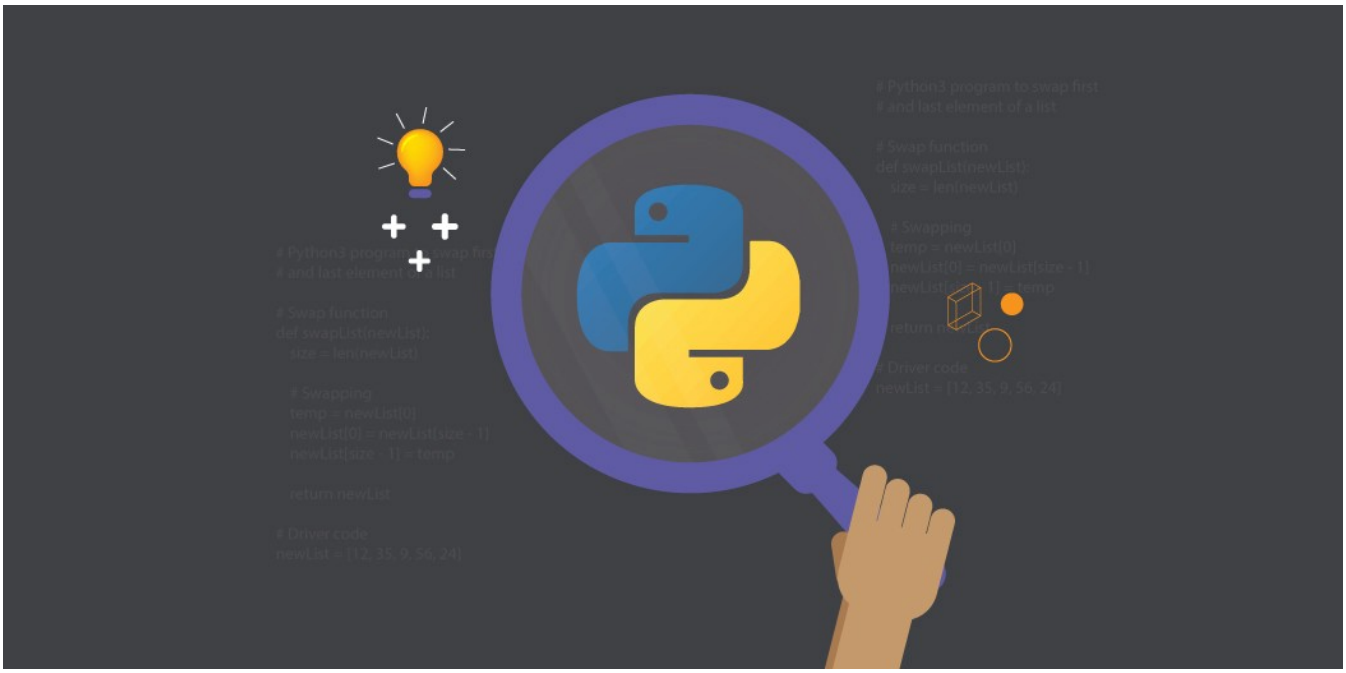
Python & Object-Oriented Programming

Luis Chaparro
Oct 14, 2019 · 9 min read



Python is a cross-platform language used in many applications that run on Windows and Unix-like systems — i.e. Linux or Mac OSX. Its standard library and its third-party libraries supply useful modules to perform many processes implemented in programming projects, like in Artificial Intelligence (AI), Web Development, Robotics, audio/video-based applications and so on. These features are some of the reasons why Python is one of the most commonly used programming languages, as shown in GitHub.

Python is a **multiparadigm language**, which is used to program in procedural and functional approach, but the heart Python is the **object-oriented programming** (OOP). This approach consists in that programs are organized around **objects,** which contain arbitrary amounts and kinds of data. These objects are created and can be modified during runtime. Each object is known as an instance of a **class,** which could be seen as a template composed of some variables and functions that are called

attributes references. Those attributes are divided by **data attributes** — or just attributes —, which maintains the state of every instance, and by **methods**, that modify their state.

There are four basic principles in the OOP:

- Inheritance: consists in that each class — subclass — has the ability to derive the attributes references from another class — base class — .
- Polymorphism: means that any object of a given class can be used as though it were an object of any of its class's base classes.
- Abstraction: refers to hide internal details and show only essential information of an object.
- Encapsulation: consists of restrictions of the access of the object's attribute references.

Those principles are convenient since some great features like reusability, data redundancy, code maintenance and security are achieved. The advantages are shown not only when the principles are implemented in Python but also in its source code — everything is an object. All data types, whether integers, floats or strings, and even composite data types — i.e. lists, tuples, sets and dictionaries — refer to objects.

This blog talks about the object's references — id, type, aliases — and object's mutability.

. . .

Id & Type

As explained above, every object is an instance of a class. Python contains some built-in classes for data types — integers, floats, strings and so on — . As every instance of a

class, those objects have attributes and methods defined by their classes — i.e. lists objects are modified by their methods like append or copy. The following image shows the class that belongs to each built-in type in Python.

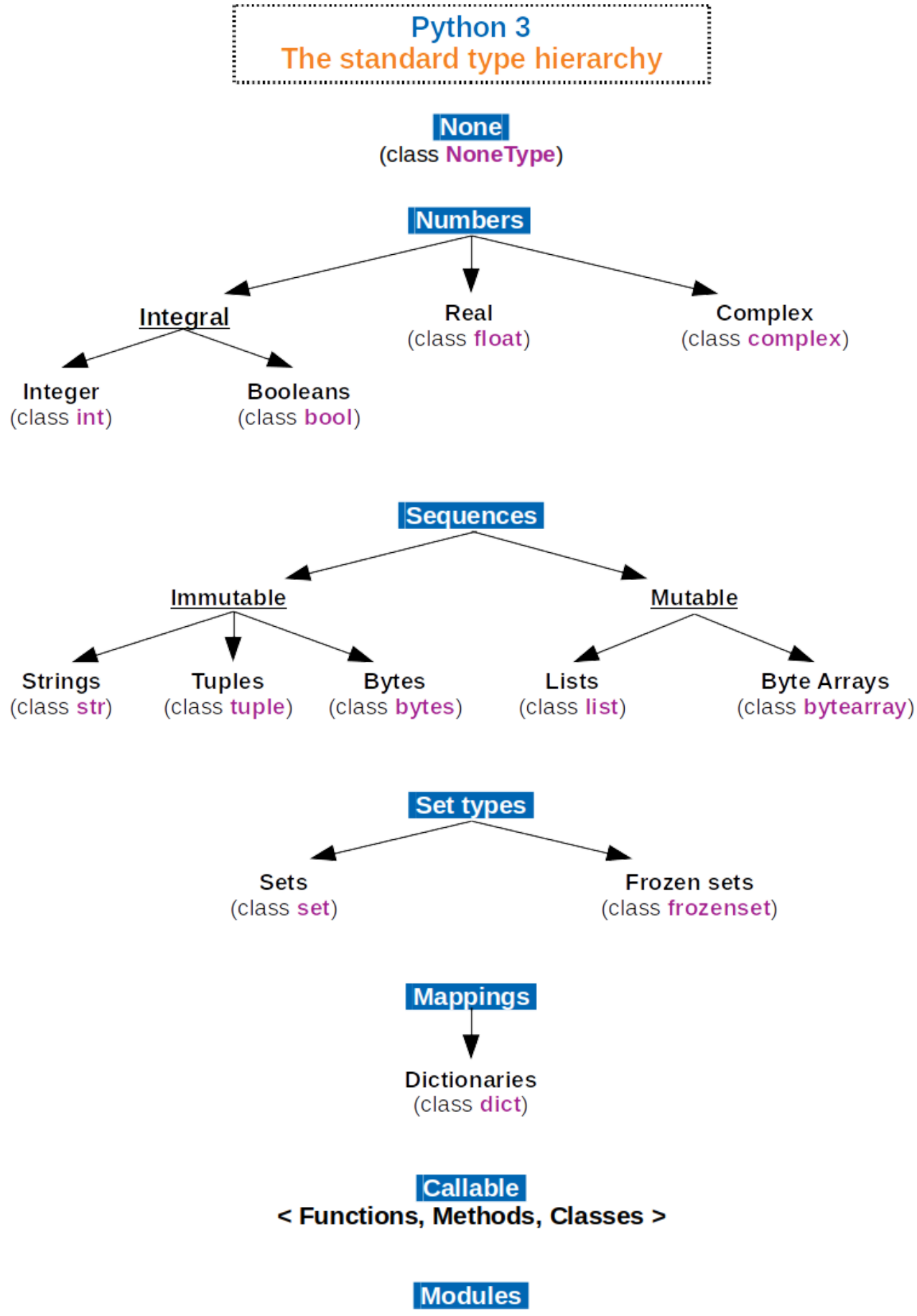


Fig 1. The standard type hierarchy in Python 3. Source: wikipedia.org

In order to know the class type of any object, `type` function must be used. This function returns the class which belongs every object created. Suppose there is the following Python script:

```
#!/usr/bin/python3

number_one = 1024
string_one = "Hello World"
list_one = [1, 2, 3]
tuple_one = (1, 2, 3)
print(type(number_one))
print(type(string_one))
print(type(list_one))
print(type(tuple_one))
```

The output of this script will be the class type:

```
<class 'int'>
<class 'str'>
<class 'list>
<class 'tuple'>
```

`Type` function is used for debugging process — i.e. to detemine the type of a text extracted from a web crawler.

Each object has an identity (id). This is represented by an integer and is guaranteed to be unique and constant for the object during its lifetime. This integer represents the address where the object resides in memory. Therefore, two objects with non-overlapping lifetimes may have the same id value.

Let's see what happens when two integer objects are created:

```
#!/usr/bin/python3

number_one = 1024
number_two = 1025
print(id(number_one)) #(id = 140225575063472)
print(id(number_two)) #(id = 140225575063344)
```

In this case, two integer objects with different value are created in a Python script. Python does not have variables as such, but instead has **object references**. So, `number_one` and `number_two` would be objects references of those two integers objects. Through id built-in function, it is possible to obtain the identity of each object in order to be printed.

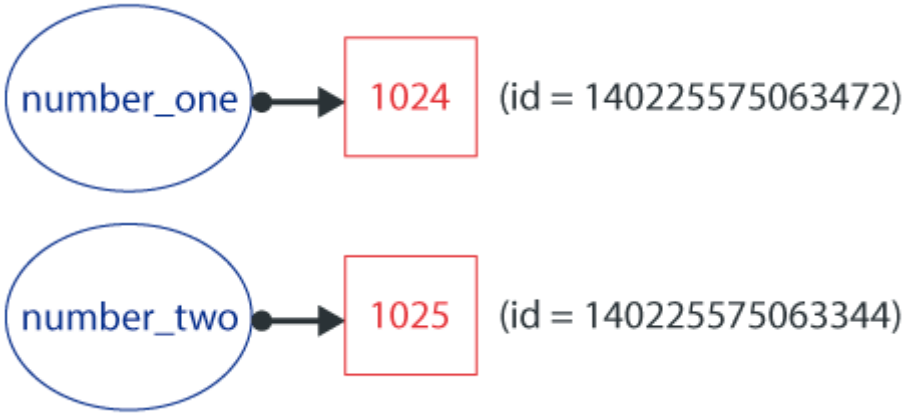


Fig 2. Object references, represented with circles, and integer objects, represented with rectangles.

Each variable refers to an object with a different id, as was explained above. The same behaviour happens when two string objects are defined:

```
#!/usr/bin/python3

string_one = "Hello"
string_two = "World"
print(id(string_one)) #(id = 140450207424840)
print(id(string_two)) #(id = 140450207425008)
```

Those objects —referred by `string_one` and `string_two` — are allocated in different memory spaces. But, what happens if those variables are equal:

```
#!/usr/bin/python3

string_one = "Hello"
string_two = string_one
print(id(string_one)) #(id = 140175261806920)
print(id(string_two)) #(id = 140175261806920)
```

In this case, the printed identities are the same. When the first line is executed, the string object and the object reference (`string_one`) are created . After the second line, an **alias** of `string_one` , which is called `string_two` is created too. An alias is an object reference that points to a same object referred by a object reference previously created. The following image graphically shows this behaviour.

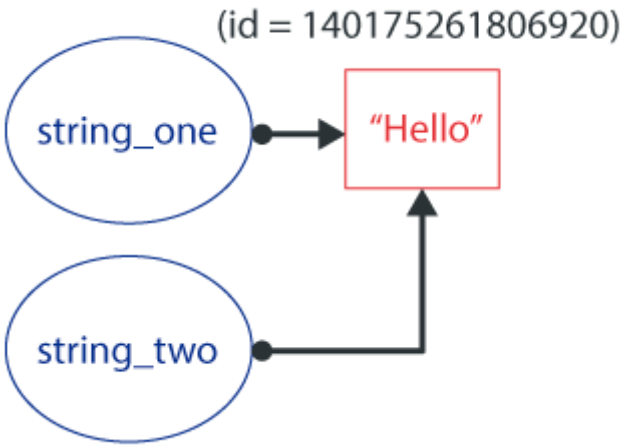


Fig 3. Object reference and alias that points to a same object in memory.

The equality of those object references can be checked through `==` and `is` conditions — the result is `True` when they refer to the same object:

```
#!/usr/bin/python3

string_one = "Hello"
string_two = string_one
print(string_one == string_two) #True
print(string_one is string_two) #True
```

In order to **optimize memory space**, Python automatically creates an **alias** of an object reference when they refer to an object with the same value, instead of creating several objects:

```
#!/usr/bin/python3

string_one = "Hello"
string_two = "Hello"
print(id(string_one)) #(id = 140175261806920)
print(id(string_two)) #(id = 140175261806920)
print(string_one == string_two) #True
print(string_one is string_two) #True
```

This behaviour happens with immutable objects, like integers, strings and tuples, but with mutable objects, like lists and dictionaries, Python creates different objects even though they have the same value:

```
#!/usr/bin/python3

list_one = ["Hello", "World"]
list_two = ["Hello", "World"]
print(id(list_one)) #(id = 140504185698440)
print(id(list_two)) #(id = 140504185698504)
print(list_one == list_two) #True
print(list_one is list_two) #False
```

In this example, the result of `==` condition is `True` because `list_one` and `list_two` have the same value, but the result of `is` condition is `False` due to each object reference points to a different object , which have different do not have the same `id`.

. . .

Object's mutability

Object's identity and object's type are unchangeable when these are created during runtime. However, depending of the object's type, their value can change. On the one hand, objects whose value can change are called **mutable objects.** On the other hand, objects with unchangeable values are known as **immutable objects.** The following image shows a list of objects with their mutability's type:

OBJECT'S TYPE	OBJECT'S MUTABILTY
int	Immutable object
bool	Immutable object
float	Immutable object
str	Immutable object
list	Mutable object
tuple	Immutable object
set	Mutable object
frozenset	Immutable object

frozenset	immutable object
dict	Mutable object

Fig 4. Object's mutability of standard types in Python

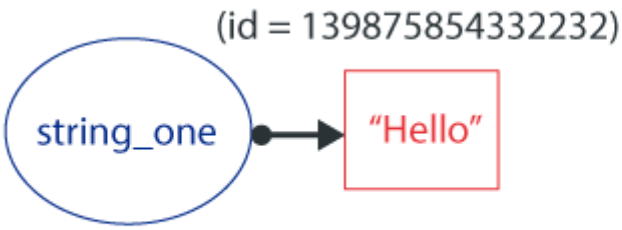
As shown above, the identies of two or more object references are the same when they points to an immutable object with the same value. But, every time the object reference's value change, Python creates another object:

```
#!/usr/bin/python3

number_one = 1024
print(id(number_one)) #(id = 140616357076912)
number_one = 1025
print(id(number_one)) #(id = 140616357076784)
```

In this case, an integer object with `140616357076912` id is created and `number_one` refers to that object. When `number_one` changes its value, it actually refers to a new integer object, with another id (`140616357076784`). If the test is done with a string object, this behavior happens:

```
#!/usr/bin/python3

string_one = "Hello"
print(id(number_one)) #(id = 139875854332232)
string_one = string_one + " World"
print(id(number_one)) #(id = 139875854332400)
```

Two objects are created and are referred by `string_one` . On the first line, `string_one` refers to an object with `Hello` as value. On the second line, `string_one` points to an new object, with a value `Hello World` . Python does not explicitly delete objects, making that unreachable objects may be garbage-collected. Once the scripts finishes, memory space are freed:



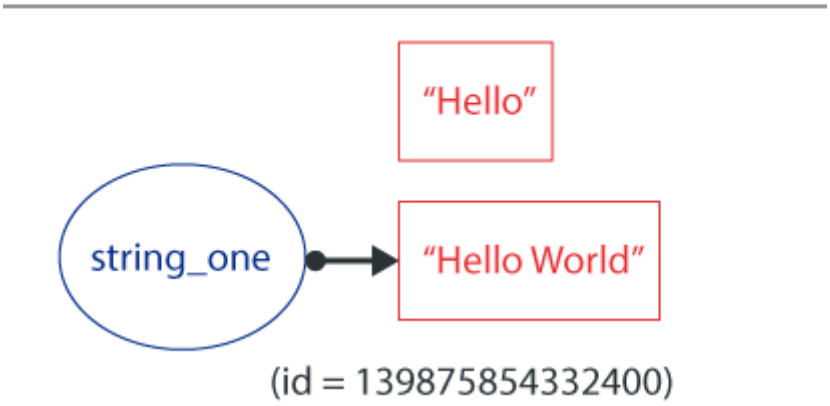


Fig 5. Graphic demostration of example when immutable objects created when a different value is assigned to an object reference (string_one).

In order to avoid garbage collection, immutable objects should be used when constant values need to be stored during runtime. Another alternative consists in deletes unused immutable objects with `del` keyword:

```
#!/usr/bin/python3

string_one = "Hello"
print(id(string_one)) #(id = 139875854332232)
del string_one #Removing unused object
string_one = "World"
print(id(string_one)) #(id = 139875854332400)
```

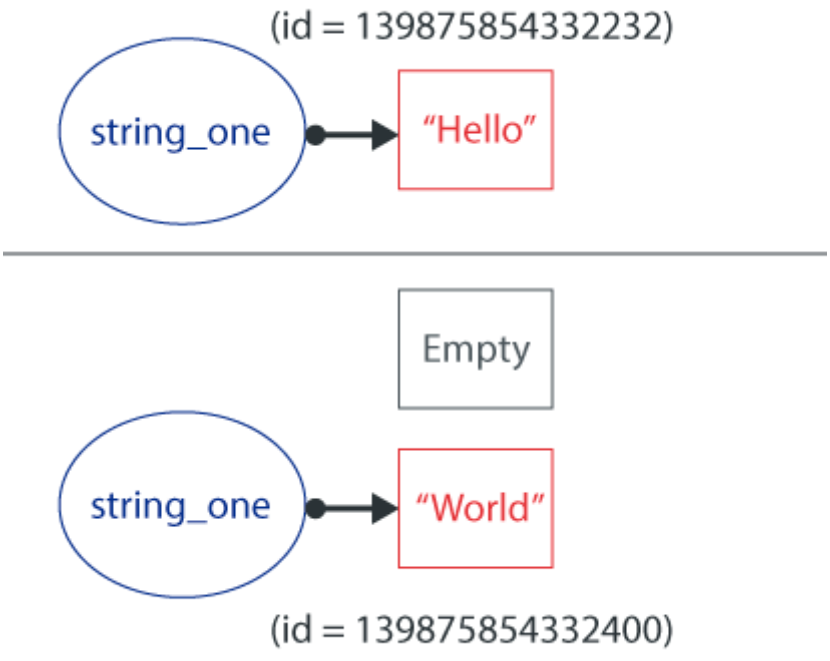


Fig 6. Graphic demostration of example when an immutable object is deleted when it is unused.

Depending in the way a mutable object's value is tried to change during its lifetime, Python creates a new object with the value value or modified the existing object. Suppose there is the following Python script where an element is added into a list:

```
#!/usr/bin/python3

list_one = ["Hello", "World"]
print(id(list_one)) #(id = 140704891648072)
list_one = ["Hello", "World", "Ok"]
print(id(list_one)) #(id = 139875854332232)
```

During the runtime, Python creates two objects and allocates them in the memory space. Similar than the example with string objects, `list_one` refers to two different objects. This happens with the following command lines:

```
#!/usr/bin/python3

list_one = ["Hello", "World"]
print(id(list_one)) #(id = 140704891648072)
list_one = list_one + ["Ok"]
print(id(list_one)) #(id = 139875854332232)
```

With the purpose to modify the existing list, avoiding a new object, the methods of the mutable objects must be used. As any class, these types counts with a couple of attributes references that changes the original object state. According to the last example, there are some ways to add an element in a list. The first one consists in using `.append` method:

```
#!/usr/bin/python3

list_one = ["Hello", "World"]
print(id(list_one)) #(id = 139800957999176)
list_one.append("Ok")
print(id(list_one)) #(id = 139800957999176)
```

The second one is to type `+=` to invoke the `__iadd__` special method of the class `list`:

```
#!/usr/bin/python3
```

```
list_one = ["Hello", "World"]
print(id(list_one)) #(id = 140183694727240)
list_one += ["Ok"]
print(id(list_one)) #(id = 140183694727240)
```

The mutability is shown on dictionaries too. Suppose there is added an item in a created `dict` object:

```
#!/usr/bin/python3

dict_one = {"Hello": 1, "World": 2}
print(id(dict_one)) #(id = 140163454592904)
dict_one["Ok"] = 3
print(id(dict_one)) #(id = 140163454592904)
```

The id of `dict_one` does not change. In the same way, it works when an item is deleted:

```
#!/usr/bin/python3

dict_one = {"Hello": 1, "World": 2}
print(id(dict_one)) #(id = 140524845111176)
del dict_one["World"]
print(id(dict_one)) #(id = 140524845111176)
```

Through `del` keyword, an element on a list can be removed too:

```
#!/usr/bin/python3

list_one = [1024, 1025, 1026]
print(list_one)
print(id(list_one))

del list_one[1]

print(list_one)
print(id(list_one))
```

The output of the last script is:

```
[1024, 1025, 1026]
139966595061832
[1024, 1026]
139966595061832
```

A mutable object is used when an object with a variable size is needed during runtime. Using correctly their methods, it is possible to avoid garbage-collection.

. . .

Functions & Arguments

Functions are compose by parameters — or arguments — and Python statements. Arguments can be of any data type — whether standard type or a created one. Actually, these parameters acts as alias of the given objects reference to the function. Suppose there is the following Python script:

```
#!/usr/bin/python3

def add_elem_list(list_func):
    list_one.append(3)

list_one = [1, 2]
print(list_one)
add_elem_list(list_one)
print(list_one)
```

The output of the execution is:

```
[1, 2]
[1, 2, 3]
```

Despite `add_elem_list` does not have a return value, this function changes the value of `list_one`. As shown on Fig 7, that happens since `list_func` refers to the same object allocated in memory space. Implictly, there is an equality between those two objects (`list_one == list_func`). As the argument is a mutable objects, when it changes through its methods, it is not created a new object.

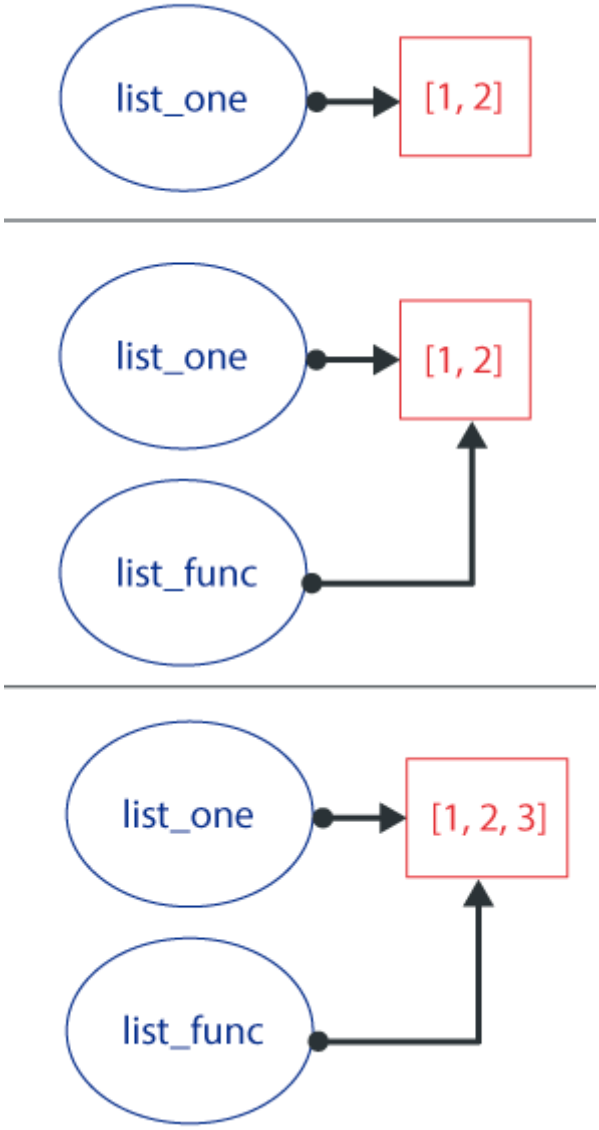


Fig 7. Graphic demostration of changing a mutable objects as an argument of a function.

An immutable object can not be modified in that way. Despite arguments are alias of that objects, with every change Python creates a new object:

```
#!/usr/bin/python3

def add_one_to_int(number_func):
    number_func += 1

number_one = 1024
print(number_one)
add_one_to_int(number_one)
print(number_one)
```

The output of the above script is:

```
1024
1024
```

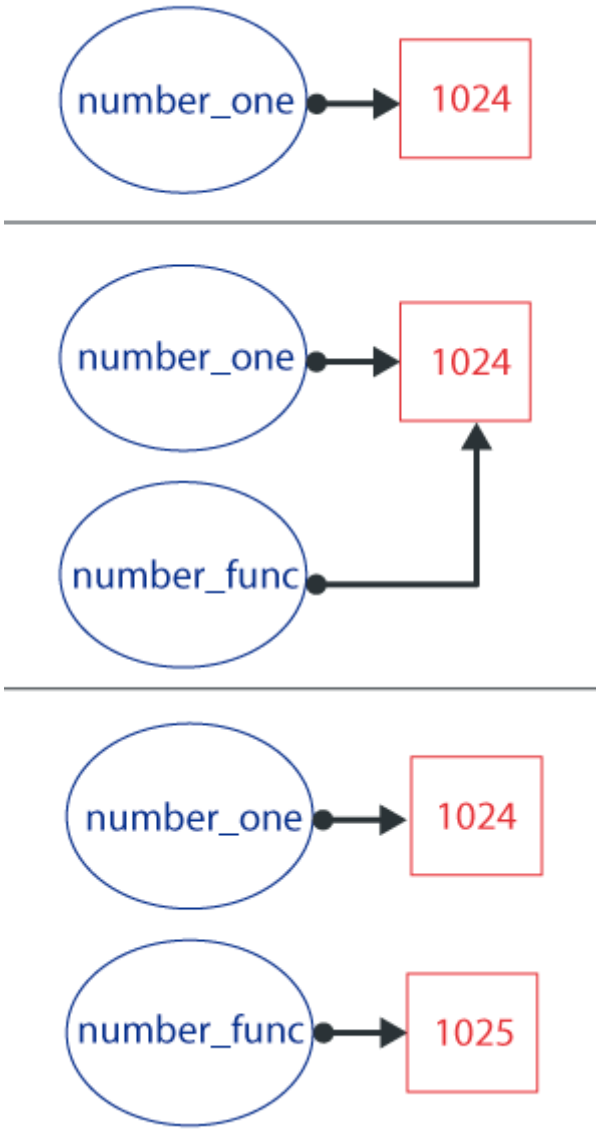


Fig 8. Graphic demostration of changing a mutable objects as an argument of a function.

In the example, `number_one` refers to an integer object that has `1024` as value. Then, `number_func` refers to the same object but, when it is added `1`, a new integer is created and `number_func` points to it — as shown in Fig 8. In order to get the result, a return value must be implemented in the function:

```
#!/usr/bin/python3

def add_one_to_int(number_func):
    return number_func + 1
```

Programming Object Oriented Python

```
number_one = 1024
print(number_one) #1024
number_one = add_one_to_int(number_one)
print(number_one) #1025
```

. . .

REFERENCES

- PYTHON. Python Documentation. Source: https://docs.python.org/3/
- SUMMERFIELD, Mark (2010). Programming in Python 3: A Complete Introduction to the Python Language. Addison-Wesley.