4.1 Shallow and deep copy operations

Copying objects using shallow and deep operations

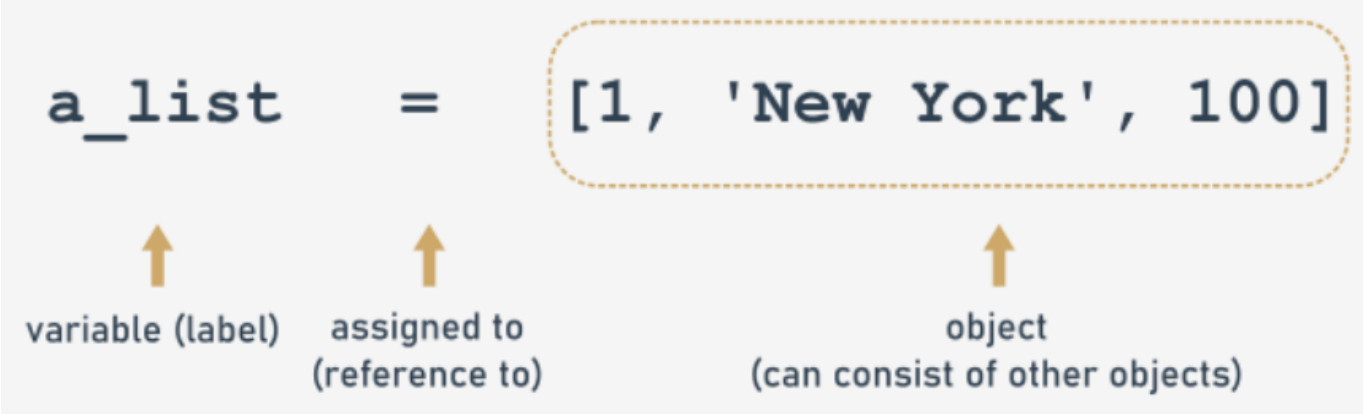
In this module, you'll learn how to copy Python objects. Specifically, you'll learn about:

- object: label vs. identity vs. value;
- the id() function and the is operand;
- shallow and deep copies of the objects.

It's hard to imagine writing a piece of Python code that performs any kind of data processing without making use of variables. As variables are fundamental elements that allow us to cope with objects, let's talk in detail about variables and objects, and possible ways of copying them.

When you spot the following clause:

```
a_list = [ 1, 'New York', 100]
```



(Note that an assignment statement is being used, so evaluation of the right side of the clause takes precedence over the left side.)

- At first, an object (a list in this example) is created in the computer's memory. Now the object has its identity;
- then the object is populated with other objects. Now our object has a value;
- finally a variable, which you should treat as a label or name binding, is created, and this label refers to a distinct place in the computer memory.

What is that object 'identity'? Why are the object value and label not enough?

The built-in `id()` function returns the 'identity' of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same `id()` value.

CPython implementation detail: This is the address of the object in the memory. Don't treat it as an absolute memory address.

Run the code presented in the right pane to see how the strings are located in the memory.

```
a_string = '10 days to departure'
b_string = '20 days to departure'
```

```
print('a_string identity:', id(a_string))
print('b_string identity:', id(b_string))
```

Remember that the memory addresses are different:

output

```
a_string identity: 8466656
b_string identity: 8466704
```

This function is rarely used in applications. More often you'll use it to debug the code or to experiment while copying objects. The side effect of this infrequent use is that some developers forget about its existence and create their own variables titled `id` to store some kind of identity or identifier.

As a result, a variable called `id` shadows the genuine function and makes it unreachable in the scope in which the variable has been defined. You should remember to avoid such situations!

When you have two variables referring to the same object, the return values of the id() function must be the same.

Run the code presented in the right pane to confirm our speculations:

```
a_string = '10 days to departure'
b_string = a_string

print('a_string identity:', id(a_string))
print('b_string identity:', id(b_string))
```

output

```
a_string identity: 8466704
b_string identity: 8466704
```

In this example, we haven't created a new list, but just created a new label that references the already created list.

This interesting behavior will be examined on the following pages.

What is the difference between the '==' and 'is' operators?

What should you do to compare two objects?

In order to compare two objects, you should start with the `'=='` operator as usual. This operator compares the values of both operands and checks for value equality. So here we witness a values comparison.

In fact, two distinct objects holding the same values could be compared, and the result would be 'True'. Moreover, when you compare two variables referencing the same object, the result would be also 'True'.

To check whether both operands refer to the same object or not, you should use the `'is'` operator. In other words, it responds to the question: "Are both variables referring to the same identity?"

Run the code presented in the editor.

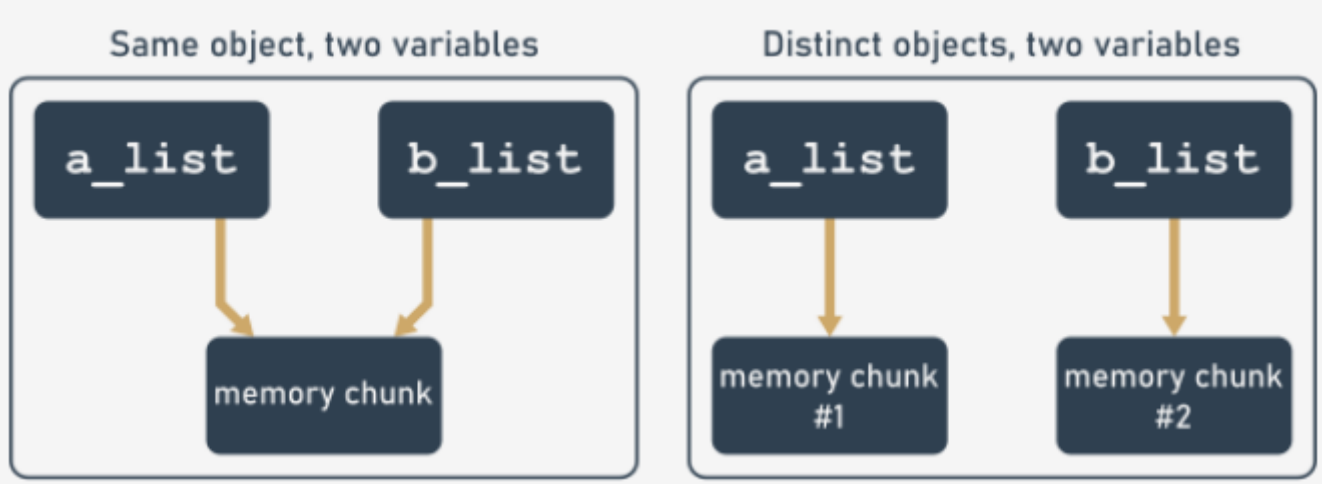
```
a_string = ['10', 'days', 'to', 'departure']
b_string = a_string

print('a_string identity:', id(a_string))
print('b_string identity:', id(b_string))
print('The result of the value comparison:', a_string == b_string)
print('The result of the identity comparison:', a_string is b_string)

print()

a_string = ['10', 'days', 'to', 'departure']
b_string = ['10', 'days', 'to', 'departure']

print('a_string identity:', id(a_string))
print('b_string identity:', id(b_string))
print('The result of the value comparison:', a_string == b_string)
print('The result of the identity comparison:', a_string is b_string)
```

The output is:

output

```
a_string identity: 3687888
b_string identity: 3687888
The result of the value comparison: True
The result of the identity comparison: True

a_string identity: 3689048
b_string identity: 9418632
The result of the value comparison: True
The result of the identity comparison: False
```

This could be depicted as follows:



When you process the data, you'll come to the point where you may want to have distinct copies of objects that you can modify without automatically modifying the original at the same time.

Let's have a look at the following code. Its intention is to:

- make a real, independent copy of `a_list`, (not just a copy reference). Using `[:]`, which is an array slice

syntax, we get a fresh copy of the `a_list` object;
- modify the original object;
- see the contents of both objects.

Pay attention to the code presented in the right pane, of which `a_list` is a compound object (an object that contains other objects, like lists, dictionaries, or class instances).

```
print("Part 1")
print("Let's make a copy")
a_list = [10, "banana", [997, 123]]
b_list = a_list[:]
print("a_list contents:", a_list)
print("b_list contents:", b_list)
print("Is it the same object?", a_list is b_list)

print()
print("Part 2")
print("Let's modify b_list[2]")
b_list[2][0] = 112
print("a_list contents:", a_list)
print("b_list contents:", b_list)
print("Is it the same object?", a_list is b_list)
```

When you run the code, you get the following output:

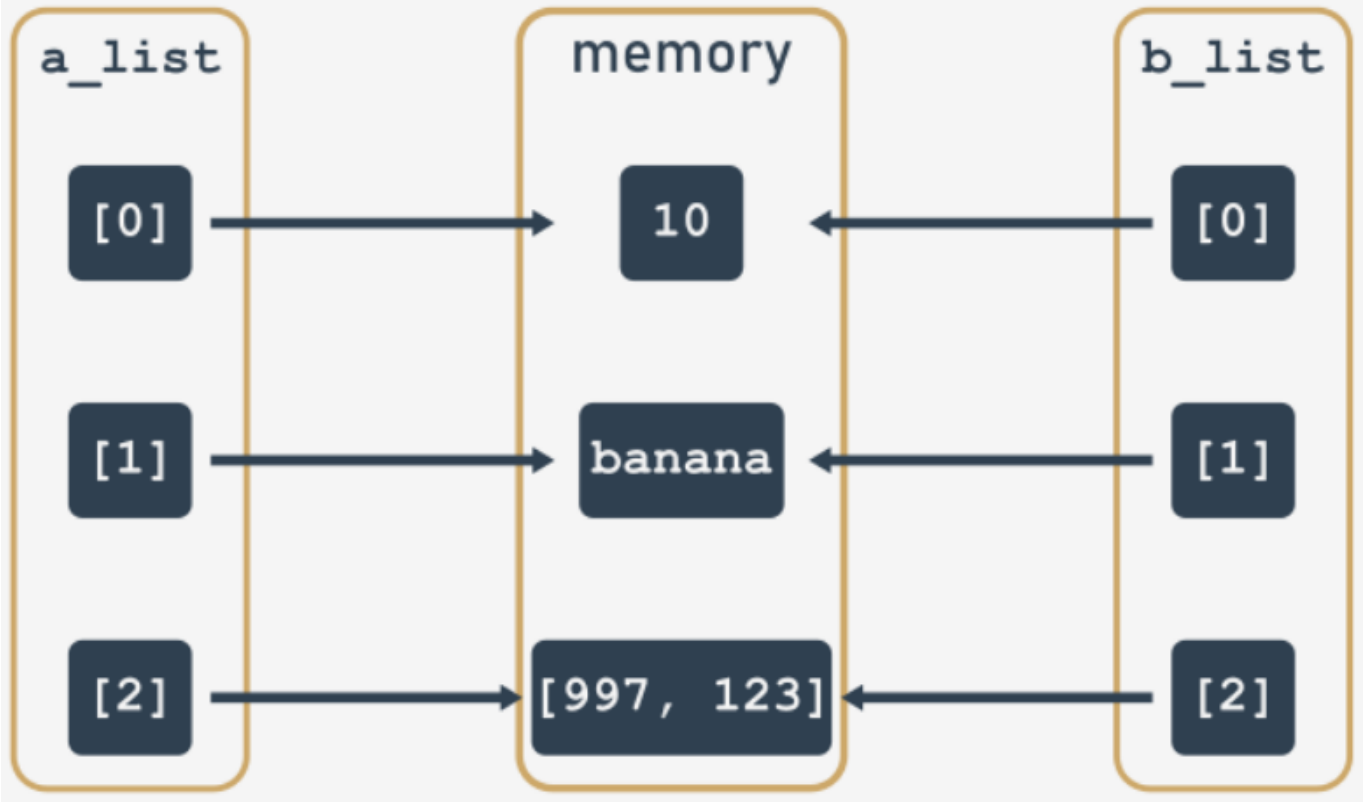
output

```
Part 1
Let's make a copy
a_list contents: [10, 'banana', [997, 123]]
b_list contents: [10, 'banana', [997, 123]]
Is it the same object? False

Part 2
Let's modify b_list[2]
a_list contents: [10, 'banana', [112, 123]]
b_list contents: [10, 'banana', [112, 123]]
Is it the same object? False
```

So, despite the fact that `b_list` is a copy of `a_list`, modifying `b_list` results in a modification of the `a_list` object.

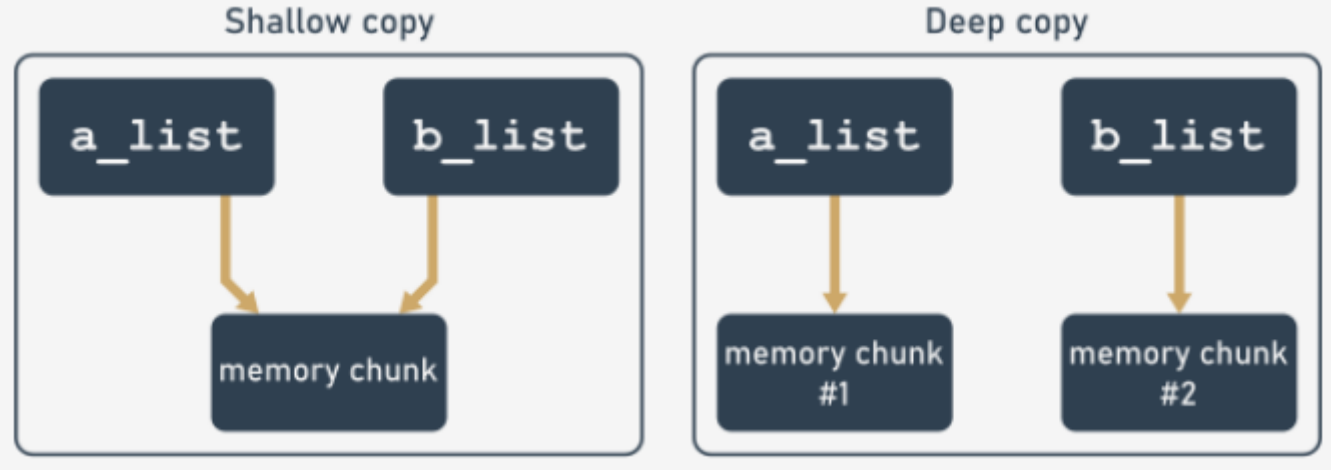
The explanation of the behavior presented on the previous page is:

- the `'a_list'` object is a compound object;
- we've run a **shallow copy** that constructs a new compound object, `b_list` in our example, and then populated it with references to the objects found in the original;
- as you can see, a shallow copy is only one level deep. The copying process does not recurse and therefore does not create copies of the child objects, but instead populates `b_list` with references to the already existing objects.



If you want to make an independent copy of a compound object (list, dictionary, custom class instance) you should make use of deep copy, which:

- constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original;
- takes more time to complete, as there are many more operations to be performed;
- is implemented by the `deepcopy()` function, delivered by the python 'copy' module



A code creating an independent copy of the `a_list` object should look like the code presented in the right pane.

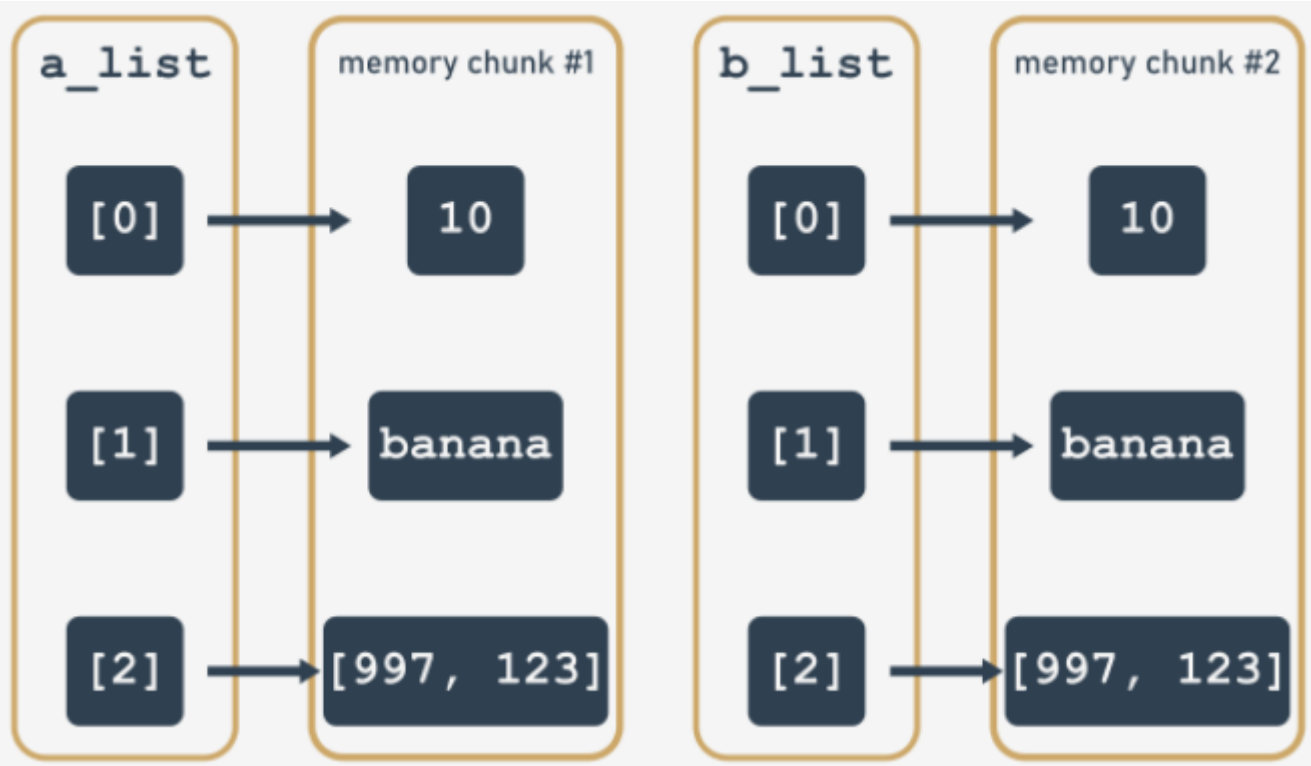
```
import copy

print("Let's make a deep copy")
a_list = [10, "banana", [997, 123]]
b_list = copy.deepcopy(a_list)
print("a_list contents:", a_list)
print("b_list contents:", b_list)
```

```
print("Is it the same object?", a_list is b_list)

print()
print("Let's modify b_list[2]")
b_list[2][0] = 112
print("a_list contents:", a_list)
print("b_list contents:", b_list)
print("Is it the same object?", a_list is b_list)
```

output

```
Let's make a deep copy
a_list contents: [10, 'banana', [997, 123]]
b_list contents: [10, 'banana', [997, 123]]
Is it the same object? False

Let's modify b_list[2]
a_list contents: [10, 'banana', [997, 123]]
b_list contents: [10, 'banana', [112, 123]]
Is it the same object? False
```

The graphical representation should look like the following:



The 'copy' module contains a function for shallow copying: `copy()`. Of course, you could say that for copying lists there is already the `[:]` notation, or `a_list=list(b_list)`, and for dictionaries you could use `a_dict = dict(b_dict)`.

But think about making use of polymorphism when you need a universal function to copy any type object, so that in that case using a `copy()` function is the smart way to accomplish the task.

In the following example, we'll compare the performance of three ways of copying a large compound object (a million three-element tuples).

```
import copy
import time

a_list = [(1,2,3) for x in range(1_000_000)]

print('Single reference copy')
time_start = time.time()
b_list = a_list
print('Execution time:', round(time.time() - time_start, 3))
print('Memory chunks:', id(a_list), id(b_list))
print('Same memory chunk?', a_list is b_list)

print()

print('Shallow copy')
time_start = time.time()
b_list = a_list[:]
print('Execution time:', round(time.time() - time_start, 3))
print('Memory chunks:', id(a_list), id(b_list))
print('Same memory chunk?', a_list is b_list)

print()

print('Deep copy')
time_start = time.time()
b_list = copy.deepcopy(a_list)
print('Execution time:', round(time.time() - time_start, 3))
print('Memory chunks:', id(a_list), id(b_list))
print('Same memory chunk?', a_list is b_list)
```

The first approach is a simple reference copy. This is done very quickly, as there's nearly nothing to be done by the CPU – just a copy of a reference to 'a_list'.

The second approach is a shallow copy. This is slower than the previous code, as there are 1,000,000 references (not objects) created.

The third approach is a deep copy. This is the most comprehensive operation, as there are 3,000,000 objects created.

Test it locally on your computer.

/

output

```
Single reference copy
Execution time: 0.0
Memory chunks: 140558259798656 140558259798656
Same memory chunk? True

Shallow copy
Execution time: 0.005
Memory chunks: 140558259798656 140558259799232
Same memory chunk? False

Deep copy
```

```
Execution time: 1.994
Memory chunks: 140558259798656 140558259802752
Same memory chunk? False
```

The same `deepcopy()` method could be utilized when you want to copy dictionaries or custom class objects.

The code on the right presents the code that safely copies the dictionary.

```
import copy

a_dict = {
    'first name': 'James',
    'last name': 'Bond',
    'movies': ['Goldfinger (1964)', 'You Only Live Twice']
    }
b_dict = copy.deepcopy(a_dict)
print('Memory chunks:', id(a_dict), id(b_dict))
print('Same memory chunk?', a_dict is b_dict)
print("Let's modify the movies list")
a_dict['movies'].append('Diamonds Are Forever (1971)')
print('a_dict movies:', a_dict['movies'])
print('b_dict movies:', b_dict['movies'])
```

Run the code to get the following output:

output

```
Memory chunks: 140113809094048 140113808387040
Same memory chunk? False
Let's modify the movies list
a_dict movies: ['Goldfinger (1964)', 'You Only Live Twice', 'Diamonds Are
Forever (1971)']
b_dict movies: ['Goldfinger (1964)', 'You Only Live Twice']
```

The code in the editor copies the dictionary in a safe manner.

```
import copy

class Example:
    def __init__(self):
        self.properties = ["112", "997"]
        print("Hello from __init__()")

a_example = Example()
b_example = copy.deepcopy(a_example)
print("Memory chunks:", id(a_example), id(b_example))
print("Same memory chunk?", a_example is b_example)
print()
print("Let's modify the movies list")
b_example.properties.append("911")
print('a_example.properties:', a_example.properties)
print('b_example.properties:', b_example.properties)
```

Run it to get the following output:

output

```
Hello from __init__()
Memory chunks: 140319166493840 140319166494096
Same memory chunk? False

Let's modify the movies list
a_example.properties: ['112', '997']
b_example.properties: ['112', '997', '911']
```

Pay attention to the fact that the `__init__()` method is executed only once, despite the fact we own two instances of the example class.

This method is not executed for the `b_example` object as the deepcopy function copies an already initialized object.

LAB

Scenario

Introduction Imagine you have been hired to help run a candy warehouse. **The task**

- Your task is to write a code that will prepare a proposal of reduced prices for the candies whose total weight exceeds 300 units of weight (we don't care whether those are kilograms or pounds)
- Your input is a list of dictionaries; each dictionary represents one type of candy. Each type of candy contains a key entitled 'weight', which should lead you to the total weight details of the given delicacy. The input is presented in the editor;
- Prepare a copy of the source list (this should be done with a one-liner) and then iterate over it to reduce the price of each delicacy by 20% if its weight exceeds the value of 300;
- Present an original list of candies and a list that contains the proposals;
- Check if your code works correctly when copying and modifying the candy item details.

Expected output

output

```
Source list of candies
{'name': 'Lolly Pop', 'price': 0.4, 'weight': 133}
{'name': 'Licorice', 'price': 0.1, 'weight': 251}
{'name': 'Chocolate', 'price': 1, 'weight': 601}
{'name': 'Sours', 'price': 0.01, 'weight': 513}
{'name': 'Hard candies', 'price': 0.3, 'weight': 433}
******************
Price proposal
{'name': 'Lolly Pop', 'price': 0.4, 'weight': 133}
{'name': 'Licorice', 'price': 0.1, 'weight': 251}
{'name': 'Chocolate', 'price': 0.8, 'weight': 601}
{'name': 'Sours', 'price': 0.008, 'weight': 513}
{'name': 'Hard candies', 'price': 0.24, 'weight': 433}
```

```
warehouse = list()
warehouse.append({'name': 'Lolly Pop', 'price': 0.4, 'weight': 133})
```

```
warehouse.append({'name': 'Licorice', 'price': 0.1, 'weight': 251})
warehouse.append({'name': 'Chocolate', 'price': 1, 'weight': 601})
warehouse.append({'name': 'Sours', 'price': 0.01, 'weight': 513})
warehouse.append({'name': 'Hard candies', 'price': 0.3, 'weight': 433})

print('Source list of candies')
for item in warehouse:
    print(item)
```

resultat

```
import copy

warehouse = list()
warehouse.append({'name': 'Lolly Pop', 'price': 0.4, 'weight': 133})
warehouse.append({'name': 'Licorice', 'price': 0.1, 'weight': 251})
warehouse.append({'name': 'Chocolate', 'price': 1, 'weight': 601})
warehouse.append({'name': 'Sours', 'price': 0.01, 'weight': 513})
warehouse.append({'name': 'Hard candies', 'price': 0.3, 'weight': 433})

print('Source list of candies')
for item in warehouse:
    print(item)

warehouse_b = copy.deepcopy(warehouse)

print('Source list of candies discount')
for item in warehouse_b:
    if item['weight'] > 300:
        item['price'] *= 0.8
for item in warehouse:
    print(item)

warehouse[0]['name'] = 'Lolly Pop Bis'

print('Source list of candies')
for item in warehouse:
    print(item)

print('Source list of candies discount')
for item in warehouse_b:
    print(item)
```

output

```
Source list of candies
{'name': 'Lolly Pop', 'price': 0.4, 'weight': 133}
{'name': 'Licorice', 'price': 0.1, 'weight': 251}
{'name': 'Chocolate', 'price': 1, 'weight': 601}
{'name': 'Sours', 'price': 0.01, 'weight': 513}
{'name': 'Hard candies', 'price': 0.3, 'weight': 433}
Source list of candies discount
{'name': 'Lolly Pop', 'price': 0.4, 'weight': 133}
{'name': 'Licorice', 'price': 0.1, 'weight': 251}
```

```
{'name': 'Chocolate', 'price': 1, 'weight': 601}
{'name': 'Sours', 'price': 0.01, 'weight': 513}
{'name': 'Hard candies', 'price': 0.3, 'weight': 433}
Source list of candies
{'name': 'Lolly Pop Bis', 'price': 0.4, 'weight': 133}
{'name': 'Licorice', 'price': 0.1, 'weight': 251}
{'name': 'Chocolate', 'price': 1, 'weight': 601}
{'name': 'Sours', 'price': 0.01, 'weight': 513}
{'name': 'Hard candies', 'price': 0.3, 'weight': 433}
Source list of candies discount
{'name': 'Lolly Pop', 'price': 0.4, 'weight': 133}
{'name': 'Licorice', 'price': 0.1, 'weight': 251}
{'name': 'Chocolate', 'price': 0.8, 'weight': 601}
{'name': 'Sours', 'price': 0.008, 'weight': 513}
{'name': 'Hard candies', 'price': 0.24, 'weight': 433}
```

LAB

Scenario

The previous task was a very easy one. Now let's rework the code a bit:

- introduce the Delicacy class to represent a generic delicacy. The objects of this class will replace the old school dictionaries. Suggested attribute names: name, price, weight;
- your class should implement the `__str__()` method to represent each object state;
- experiment with the `copy.copy()` and `deepcopy.copy()` methods to see the difference in how each method copies objects.

resultat

```
import copy

class Delicacy():
    def __init__(self,name, price, weight):
        self.name = name
        self.price = price
        self.weight = weight

    def __str__(self):
        return f"{self.name} - {self.price} - {self.weight}"

a = list()

d1 = Delicacy("aaa","10","100")
a.append(d1)
a.append(Delicacy("bbb","20","200"))
d2 = copy.copy(d1)
a.append(d2)
d3 = copy.deepcopy(d1)
d3.name = "ccc"
a.append(d3)
```

```
for i in a:
    print(i)

d2.name="dddd"
a.append(d2)
print()

for i in a:
    print(i)
```

```
  <code ; output>
```

aaa - 10 - 100 bbb - 20 - 200 aaa - 10 - 100 ccc - 10 - 100

aaa - 10 - 100 bbb - 20 - 200 dddd - 10 - 100 ccc - 10 - 100 dddd - 10 - 100 </code>

Section summary

Important things to remember:

- the `deepcopy()` method creates and persists new instances of source objects, whereas any shallow copy operation only stores references to the original memory address;
- a deep copy operation takes significantly more time than any shallow copy operation;
- the `deepcopy()` method copies the whole object, including all nested objects; it's an example of practical recursion taking place;
- deep copy might cause problems when there are cyclic references in the structure to be copied.