Lesson 28. Aliasing and copying lists and dictionaries

After reading lesson 28, you'll be able to

- Make aliases for mutable objects (lists and dictionaries)
- Make copies of mutable objects (lists and dictionaries)
- Make sorted copies of lists
- Remove elements from mutable objects based on certain criteria

Mutable objects are great to use because they allow you to modify the object itself without making a copy. When your mutable objects are large, this behavior makes sense because otherwise, making a copy of a large item every time you make a change to it is expensive and wasteful. But using mutable objects introduces a side effect that you need to be aware of: you can have more than one variable bound to the same mutable object, and the object can be mutated via both names.

Consider this

Think of a famous person. What aliases do they have, or what other names or nicknames do they go by?

Answer: Bill Gates

Nicknames: Bill, William, William Gates, William Henry Gates III

Suppose you have data on the famous computer scientist Grace Hopper. Let's say Grace Hopper is an object, and her value is a list of labels: `["programmer", "admiral", "female"]`. To friends she might be known as *Grace*, to others as *Ms. Hopper*, and her nickname is *Amazing Grace*. All these names are aliases for the same person, the same object with the same string of labels. Now suppose that someone who knows her as Grace adds another value to her list of labels: `"deceased"`. To people who know her as Grace, her list of labels is now `["programmer", "admiral", "female", "deceased"]`. But because the labels refer to the same person, every other one of her aliases now also refers to the new list of labels.

28.1. USING OBJECT ALIASES

In Python, *variable names* are names that point to an object. The object resides at a specific location in computer memory. In lesson 24, you used the `id()` function to see a numerical representation of the memory location of an object.

28.1.1. Aliases of immutable objects

Before looking at mutable objects, let's look at what happens when you use the assignment operator (equal sign) between two variable names that point to immutable objects. Type the following commands in the console and use the `id()` function to see the memory locations of the variables `a` and `b`:

```
a = 1
id(a)
Out[2]: 1906901488

b = a
id(b)
Out[4]: 1906901488
```

The `Out` lines tell you the output of the `id()` function. Notice that both `a` and `b` are names that point to the same object (an integer with value `1`). What happens if you change the object that `a` points to? In the following code, you reassign variable name `a` to point to a different object:

```
Out[6]: 1906901520

id(b)
Out[7]: 1906901488

a
Out[8]: 2

b
Out[9]: 1
```

Notice that the variable named `a` now points to a completely different object with a different memory location. But this operation doesn't change the variable name to which `b` points, so the object that `b` points to is at the same memory location as before.

Quick check 28.1

You have a variable named `x` that points to an immutable object with `x = "me"`. Running `id(x)` gives `2899205431680`. For each of the following lines, determine whether the ID of that variable will be the same as `id(x)`. Assume that the lines are executed one after another:

1

```
y = x    # what is id(y)
```

2

```
z = y    # what is id(z)
```

3

```
a = "me" # what is id(a)
```

28.1.2. Aliases of mutable objects

You can do the same sequence of commands as in section 28.1.1 on a mutable object, such as a list. In the following code, you can see that using the assignment operator between a variable name that points to a list behaves in the same way as with an immutable object. Using the assignment operator on a mutable object doesn't make a copy; it makes an alias. An *alias* is another name for the same object:

```
genius = ["einstein", "galileo"]
id(genius)
Out[9]: 2899318203976

smart = genius
id(smart)
Out[11]: 2899318203976
```

The memory location that objects `genius` and `smart` point to are the same because they point to the same object. Figure 28.1 shows how the variables `smart` and `genius` point to the same object.

Figure 28.1. In the left panel, you create the variable `genius` pointing to the list `["einstein", "galileo"]`. In the right panel, the variable `smart` points to the same object as `genius`.

Quick check 28.2

You have a variable named `x` that points to a mutable object with `x = ["me", "I"]`. Running `id(x)` gives `2899318311816`. For each of the following lines, determine whether the ID of that variable will be the same as `id(x)`. Assume that the lines are executed one after another:

1

```
y = x # what is id(y)
```

```
z = y # what is id(z)
```

3

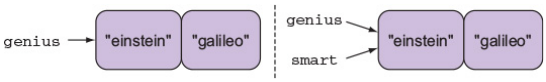
```
a = ["me", "I"] # what is id(a)
```

The difference between mutable and immutable objects is evident in the next set of commands, when you mutate the list:

```
genius.append("shakespeare")
id(genius)
Out[13]: 2899318203976

id(smart)
Out[14]: 2899318203976

genius
Out[16]: ["einstein", "galileo", "shakespeare"]

smart
Out[15]: ["einstein", "galileo", "shakespeare"]
```

When you modify a mutable object, the object itself is changed. When you append a value to the list pointed to by `genius`, the list object itself is changed. Variable names `genius` and `smart` still point to the same object at the same memory location. The object pointed to by variable name `smart` is also changed (because it points to the same thing as the variable `genius`). This is shown in figure 28.2.

Figure 28.2. When the list object `["einstein", "galileo"]` is modified through the variable `genius`, the variable `smart` also points to the modified list object.



Using the equal sign between mutable lists implies that if you modify the list through one variable name, all other variable names pointing to the same list will point to the mutated value.

Quick check 28.3

You have a variable named `x` that points to a mutable object with `x = ["me", "I"]`. For each of the following points, answer the question:

1

Does `x` change after the following lines are executed?

```
y = x
x.append("myself")
```

2

Does `x` change after the following lines are executed?

```
y = x
y.pop()
```

3

Does `x` change after the following lines are executed?

```
y = x
y.append("myself")
```

4

Does `x` change after the following lines are executed?

```
y = x
y.sort()
```

5

```
y = [x, x]
y.append(x)
```

28.1.3. Mutable objects as function parameters

In unit 5, you saw that variables inside a function are independent of variables outside the function. You could have a variable named `x` outside the function and a variable named `x` as a parameter to the function. They don't interfere with each other because of scoping rules. When working with mutable objects, passing mutable objects as actual parameters to a function implies that the actual parameter to the function will be an alias.

Listing 28.1 shows code that implements a function. The function is named `add_word()`. Its input parameters are a dictionary, a word, and a definition. The function mutates the dictionary so that even when accessed outside the function, the dictionary contains the newly added word. The code then calls the function with a dictionary named `words` as the actual parameter. In the function call, the dictionary named `d` is the formal parameter and is now an alias for the dictionary `words`. Any changes made to `d` inside the function reflect when you access the dictionary `words`.

Listing 28.1. Function that mutates a dictionary

```
def add_word(d, word, definition):
    """ d, dict that maps strings to lists of strings
        word, a string
        definition, a string
        Mutates d by adding the entry word:definition
        If word is already in d, append definition to
        Does not return anything
    """
    if word in d:
        d[word].append(definition)
    else:
        d[word] = [definition]

words = {}
add_word(words, 'box', 'fight')
print(words)
add_word(words, 'box', 'container')
print(words)
add_word(words, 'ox', 'animal')
print(words)
```

- ***1*** **A function that takes in a dictionary, a string (word), and another string (definition)**
- ***2*** **Word in dictionary**
- ***3*** **Adds definition to end of key's value list**
- ***4*** **Word not in dictionary**
- ***5*** **Makes new list with one word as the key's value**
- ***6*** **Outside function; creates an empty dictionary**
- ***7*** **Calls the function with the dictionary named "words" as the actual parameter**
- ***8*** **Prints {'box': ['fight']}**
- ***9*** **Calls the function again to append to the value for the key "box"**
- ***10*** **Prints {'box': ['fight', 'container']}**
- ***11*** **Prints {'ox': ['animal'], 'box': ['fight', 'container']}**
- ***12*** **Calls the function again to add another entry**

28.2. MAKING COPIES OF MUTABLE OBJECTS

When you want to make a copy of a mutable object, you need to use a function that makes it clear to Python that you want to make a copy. There are two ways to do this: make a new list with the same elements as another list, or use a function.

28.2.1. Commands to copy mutable objects

One way to make a copy is to make a new list object with the same values as the other one. Given a list `artists` with some elements, the following command creates a new list object and binds the variable name `painters` to it:

```
painters = list(artists)
```

The new list object has the same elements as `artists`. For example, the following code shows that the objects that lists `painters` and `artists` point to are different because modifying one doesn't modify the other:

```
artists = ["monet", "picasso"]
painters = list(artists)
painters.append("van gogh")

painters
Out[24]: ["monet", "picasso", "van gogh"]

artists
Out[25]: ["monet", "picasso"]
```

The other way is to use the `copy()` function. If `artists` is a list, the following command creates a new object that has the same elements as `artists`, but copied into the new object:

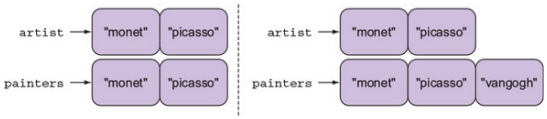
```
painters = artists.copy()
```

The following code shows how to use the `copy` command:

```
artists = ["monet", "picasso"]
painters = artists.copy()
painters.append("van gogh")

painters
Out[24]: ["monet", "picasso", "van gogh"]

artists
Out[25]: ["monet", "picasso"]
```

From the console output, you can see that the list objects pointed to by `painters` and `artists` are separate because making changes to one doesn't affect the other. Figure 28.3 shows what it means to make a copy.

Figure 28.3. In the panel on the left, making a copy of the object `["monet", "picasso"]` makes a new object with the same elements. In the panel on the right, you can mutate one object without interfering with the other object.



28.2.2. Getting copies of sorted lists

You saw that you can sort a list so that the list itself is modified directly. For a list `L`, the command is `L.sort()`. In some situations, you'd like to keep your original list and get a sorted copy of the list, while keeping the original unchanged.

Instead of making a copy of the list and then sorting the copy, Python has a function that allows you to do it in one line. The following command shows a function that returns a sorted version of a list and stores it in another list:

```
kid_ages = [2,1,4]
sorted_ages = sorted(kid_ages)

sorted_ages
Out[61]: [1, 2, 4]

kid_ages
Out[62]: [2, 1, 4]
```

You can see that the variable `sorted_ages` points to a sorted list, but the original list `kid_ages` remains unchanged. Previously, when you wrote the command `kid_ages.sort()`, `kid_ages` would be changed so that it got sorted without a copy being made.

Quick check 28.4

Write a line of code that achieves each of the following:

1

Creates a variable named `order` that's a sorted copy of a list named `chaos`

2

Sorts a list named `colors`

3

28.2.3. A word of caution when iterating over mutable objects

Often you want to have code that removes items from a mutable object, provided they meet some sort of criteria. For example, suppose you have a dictionary of songs and their ratings. You want to remove all songs from the dictionary that have a rating of 1.

Listing 28.2 tries (but fails) to accomplish this. The code iterates through every key in the dictionary. It checks whether the value associated with that key is a 1. If so, it removes the pair from the dictionary. The code fails to run and shows the error message `RuntimeError: dictionary changed size during iteration`. Python doesn't allow you to change the size of a dictionary as you're iterating over it.

Listing 28.2. Attempt to remove elements from a dictionary while iterating over it

```
songs = {"Wannabe": 1, "Roar": 1, "Let It Be": 5, "Re

for s in songs.keys():
    if songs[s] == 1:
        songs.pop(s)
```

- ***1* songs dictionary**
- ***2* Iterates through every pair**
- ***3* If the rating value is 1...**
- ***4* ...removes the song with that value**

Suppose you try to do the same thing, except that instead of a dictionary, you have a list. The following listing shows how you might accomplish this, but also fails to do the right thing. The code doesn't fail this time. But it has a behavior different from what you expected; it gives the wrong value for `songs`: `[1, 5, 4]` instead of `[5, 4]`.

Listing 28.3. Attempt to remove elements from a list while iterating over it

```
songs = [1, 1, 5, 4]                 1

for s in songs:                      2
    if s == 1:                       3
        songs.pop(s)                 4
print(songs)                         5
```

- ***1* Song rating list**
- ***2* Iterates through every rating**
- ***3* If the rating value is 1...**
- ***4* ...removes the song with that value**
- ***5* Prints [1,5,4]**

You can see the buggy effect of removing items from a list while iterating over it. The loop gets to the element at index 0, sees that it's a 1, and removes it from the list. The list is now `[1, 5, 4]`. Next, the loop looks at the element at index 1. This element is now from the mutated list `[1, 5, 4]`, so it looks at the number 5. This number isn't equal to 1, so it doesn't remove it. Then it finally looks at the element at index 2 from the list `[1, 5, 4]`, the number 4. It's also not equal to 1, so it keeps it. The issue here is that when you popped the first 1 you saw, you decreased the length of the list. Now the index count is one off from the original list. Effectively, the second 1 in the original list `[1, 1, 5, 4]` was skipped.

If you ever need to remove (or add) items to a list, you'll first make a copy. You can iterate over the list copy and then start fresh on the original list by adding items you want to keep, as you iterate over the copied list. The following listing shows how to modify the code in listing 28.3 to do the right thing. This code doesn't cause an error, and the correct value for `songs` is now `[5, 4]`.

Listing 28.4. A correct way to remove elements from a list while iterating over it

```
songs = [1, 1, 5, 4]                 1
songs_copy = songs.copy()            2
songs = []                           3
for s in songs_copy:                 4
    if s != 1:                       5
        songs.append(s)              6
print(songs)                         7
```

- ***1* Original ratings list**
- ***2* Makes a copy of the object**

- *4* **For every rating in the list**
- *5* **If the rating is one to keep...**
- *6* **...adds the rating to the original list**
- *7* **Prints [5,4]**

28.2.4. Why does aliasing exist?

If aliasing an object introduces the problem of inadvertently mutating an object you didn't intend to change, why use aliasing in the first place? Why not just make copies all the time? All Python objects are stored in computer memory. Lists and dictionaries are "heavy" objects, unlike an integer or a Boolean. If you make copies, for example, every time you make a function call, this can severely cripple the program with many function calls. If you have a list of the names of all people in the United States, copying that list every time you want to add someone new can be slow.

SUMMARY

In this lesson, my objective was to teach you about subtleties of dealing with mutable objects. Mutable objects are useful because they can store a lot of data that can easily be modified in place. Because you're dealing with mutable objects that contain many elements, making copies with every operation becomes inefficient in terms of computer time and space. By default, Python aliases objects so that using the assignment operator makes a new variable that points to the same object; this is called an *alias*. Python recognizes that in some situations you want to make a copy of a mutable object, and it allows you to explicitly tell it that you want to do so. Here are the major takeaways:

- Python aliases all object types.
- Aliasing a mutable object may lead to unexpected side effects.
- Modifying a mutable object through one alias leads to seeing the change through all other aliases of that object.
- You can make a copy of a mutable object by making a new object and copying over all elements of the original one.

Let's see if you got this...

Q28.1

Write a function named `invert_dict` that takes as input a dictionary. The function returns a new dictionary; the values are now the original keys, and the keys are now the original values. Assume that the values of the input dictionary are immutable and unique.

Q28.2

Write a function named `invert_dict_inplace` that takes as input a dictionary. The function doesn't return anything. It mutates the dictionary passed in so that the values are now the original keys, and the keys are now the original values. Assume that the values of the input dictionary are immutable and unique.