A Guide to Copying Objects in Python: Assignment, Shallow vs. Deep Copy

In Python, how you "copy" an object determines whether you get a new independent object or just another name for the original. This distinction is vital when working with mutable objects like lists and dictionaries, as it can lead to unintended side effects where modifying a "copy" also changes the original.

This guide covers the three fundamental ways to duplicate an object:

1. **Assignment (`=`)**: Not a copy at all.
2. **Shallow Copy**: Creates a new top-level object but shares references to nested objects.
3. **Deep Copy**: Creates a completely independent clone of the original and all its nested objects.

---

1. Assignment (`=`): Creating a New Label

When you use the assignment operator (`=`), you are **not creating a copy**. You are simply creating a new variable (a new label) that points to the **exact same object** in memory.

- **Behavior**: Any modification made through one variable will be visible through the other, because they both refer to the same object.
- **Check**: `a is b` will be `True`.

Example

Let's see this with a list, which is a mutable object.

```
# Create an original list
original_list = [1, 2, ['a', 'b']]

# Assign it to a new variable. This is NOT a copy.
assigned_list = original_list

print(f"original_list is assigned_list: {original_list is assigned_list}")
# Output: original_list is assigned_list: True

# Now, modify the "assigned" list
assigned_list[0] = 99
assigned_list[2].append('c')

# The original list is also changed!
print(f"Original list: {original_list}")
# Output: Original list: [99, 2, ['a', 'b', 'c']]

print(f"Assigned list: {assigned_list}")
# Output: Assigned list: [99, 2, ['a', 'b', 'c']]
```

When to use it: When you want multiple names for the same object, perhaps for clarity in different parts of your code, but you understand they are linked.

---

2. Shallow Copy: A One-Level-Deep Clone

A shallow copy creates a **new top-level object**, but then inserts **references** to the objects found in the original.

- **Behavior**: The copy is a new and independent object. However, its contents are references to the *same nested objects* as the original. If a nested object is mutable and you modify it, the change will appear in both the original and the copy.
- **Check**: `a is b` will be `False`, but for a nested object `a[i] is b[i]` might be `True`.

How to Create a Shallow Copy

There are several built-in ways to perform a shallow copy.

Object Type	Methods
Any	Use the `copy` module: `import copy; b = copy.copy(a)`
list	Slicing: `b = a[:]` Constructor: `b = list(a)` Method: `b = a.copy()`
dict	Constructor: `b = dict(a)` Method: `b = a.copy()`
set	Constructor: `b = set(a)` Method: `b = a.copy()`

Example

Let's use the same list as before.

```
import copy

original_list = [1, 2, ['a', 'b']]

# Create a shallow copy
shallow_copy = copy.copy(original_list)
# You would get the same result with:
# shallow_copy = original_list[:]
# shallow_copy = list(original_list)

print(f"original_list is shallow_copy: {original_list is shallow_copy}")
# Output: original_list is shallow_copy: False (They are different list objects)

print(f"original_list[2] is shallow_copy[2]: {original_list[2] is shallow_copy[2]}")
# Output: original_list[2] is shallow_copy[2]: True (The nested list is the SAME
object)

# --- Let's test the side effects ---

# 1. Modify a top-level, immutable element in the copy
```

```
shallow_copy[0] = 99
print(f"\nAfter changing top-level element:")
print(f"Original list:  {original_list}")   # Output: [1, 2, ['a', 'b']] (Unaffected)
print(f"Shallow copy: {shallow_copy}")    # Output: [99, 2, ['a', 'b']] (Changed)

# 2. Modify a nested, mutable element in the copy
shallow_copy[2].append('c')
print(f"\nAfter changing nested element:")
print(f"Original list:  {original_list}")   # Output: [1, 2, ['a', 'b', 'c']]
(AFFECTED!)
print(f"Shallow copy: {shallow_copy}")    # Output: [99, 2, ['a', 'b', 'c']] (Changed)
```

When to use it: This is the most common type of copy. It's fast and usually sufficient. Use it when your object contains only immutable data (like a list of numbers or strings) or when you are okay with sharing references to nested objects.

---

3. Deep Copy: A Complete, Independent Clone

A deep copy creates a **new top-level object** and then **recursively** creates copies of all the objects found inside it. The result is a completely independent clone with no shared references.

- **Behavior**: The copy and all of its nested objects are entirely separate from the original. Modifying the copy or any of its nested objects will have **no effect** on the original.
- **Check**: `a is b` is `False`, and for any nested object, `a[i] is b[i]` is also `False`.

How to Create a Deep Copy

You must use the `copy` module.

- `import copy; b = copy.deepcopy(a)`

Example

Again, using our nested list.

```
import copy

original_list = [1, 2, ['a', 'b']]

# Create a deep copy
deep_copy = copy.deepcopy(original_list)

print(f"original_list is deep_copy: {original_list is deep_copy}")
# Output: original_list is deep_copy: False

print(f"original_list[2] is deep_copy[2]: {original_list[2] is deep_copy[2]}")
# Output: original_list[2] is deep_copy[2]: False (The nested list is now a NEW
object)

# --- Let's test the side effects ---
```

```
# Modify a nested, mutable element in the copy
deep_copy[2].append('c')

print(f"\nAfter changing nested element:")
print(f"Original list: {original_list}")  # Output: [1, 2, ['a', 'b']] (UNAFFECTED!)
print(f"Deep copy:   {deep_copy}")      # Output: [1, 2, ['a', 'b', 'c']] (Changed)
```

When to use it: When your data structure contains other mutable objects (like a list of lists, or a dictionary of lists) and you need to ensure that the copy is completely independent of the original. Be aware that it is slower and consumes more memory than a shallow copy.

Summary Table: Copying Methods in Python

Method / Syntax	Type of Copy	Notes
`b = a`	**Assignment (No Copy)**	Creates a new label pointing to the *same* object. Fast but has side effects for mutable types.
`b = copy.copy(a)`	**Shallow Copy**	The standard way to create a shallow copy of any object type.
`b = copy.deepcopy(a)`	**Deep Copy**	The standard way to create a completely independent clone. Slower and more memory-intensive.
`b = a[:]`	**Shallow Copy**	A common and idiomatic way to shallow copy *lists* and other sequences.
`b = list(a)`	**Shallow Copy**	The `list` constructor creates a shallow copy from any iterable.
`b = a.copy()`	**Shallow Copy**	Available method for `list`, `dict`, and `set`.

Final Recommendations

- **Default to Shallow Copy**: For most day-to-day tasks, a shallow copy (`.copy()`, `[:]`, `list()`) is sufficient and efficient.
- **Be Wary of Nested Structures**: If you have lists inside lists, or dictionaries inside lists, be mindful. If you plan to modify these nested structures in your "copy," you almost certainly need a **deep copy**.
- **Use `is` for Debugging**: When you're unsure if two variables point to the same object, use `print(a is b)` to find out.
- **Immutable Types are Easy**: If your list or tuple contains only immutable types like numbers, strings, or booleans, a shallow copy behaves identically to a deep copy because there are no nested mutable objects to worry about.