Copying Python Data Structures

Introduction

When working with Python data structures, you will often need to make a copy.

For example:

- You want to keep the original data unchanged.
- You want to modify a separate version.
- You want to create a backup.
- You want to pass data to another part of your program.
- You are working with nested lists and dictionaries.

But copying data in Python is not always as simple as:

```
copy = original
```

That statement usually **does not create a real copy**.

It creates another reference to the same object.

To work correctly with data, you need to understand:

- Assignment
- Object references
- Shallow copies
- Deep copies
- `.copy()`
- Slicing
- `copy.copy()`
- `copy.deepcopy()`
- Nested data structures
- When each method should be used

---

1. Why Do We Need to Copy Data?

Suppose you have:

```
students = ["Hafsa", "Asim", "Ali"]
```

You want another version of the list:

```
backup = students
```

You might think:

> "Now I have two lists."

But you actually have **two variables referring to the same list**.

If you change `backup`:

```
backup.append("Sara")
```

the original also changes:

```
print(students)
```

Output:

```
['Hafsa', 'Asim', 'Ali', 'Sara']
```

This can cause unexpected bugs.

---

2. Assignment Is Not Copying

This is one of the most important concepts.

Consider:

```
numbers = [10, 20, 30]

other_numbers = numbers
```

This does not create a second list.

Conceptually:

```
numbers ──────┐
              ↓
        [10, 20, 30]
              ↑
              │
other_numbers ┘
```

Both variables refer to the same object.

---

3. Modifying One Reference

Now:

```
other_numbers.append(40)
```

The list becomes:

```
[10, 20, 30, 40]
```

Both variables see the change:

```
print(numbers)
print(other_numbers)
```

Output:

```
[10, 20, 30, 40]
[10, 20, 30, 40]
```

There is only one list.

---

4. How to Create a Real Copy

There are several ways to copy Python data structures.

For example, with a list:

`.copy()`

```
numbers = [10, 20, 30]

other_numbers = numbers.copy()
```

Slicing

```
other_numbers = numbers[:]
```

`list()`

```
other_numbers = list(numbers)
```

`copy.copy()`

```
import copy

other_numbers = copy.copy(numbers)
```

These create a **shallow copy**.

For simple, non-nested data, that is often enough.

---

5. Using `.copy()`

The easiest method for many Python data structures is `.copy()`.

Example:

```
students = ["Hafsa", "Asim", "Ali"]

backup = students.copy()
```

Now:

```
backup.append("Sara")
```

Check both:

```
print(students)
print(backup)
```

Output:

```
['Hafsa', 'Asim', 'Ali']
['Hafsa', 'Asim', 'Ali', 'Sara']
```

The lists are independent.

---

6. Copying Dictionaries

Dictionaries also provide `.copy()`.

```
student = {
    "name": "Hafsa",
    "age": 25
}

student_copy = student.copy()
```

Now change the copy:

```
student_copy["age"] = 26
```

The original remains unchanged:

```
print(student)
```

Output:

```
{'name': 'Hafsa', 'age': 25}
```

The copy contains:

```
{'name': 'Hafsa', 'age': 26}
```

---

7. Copying Sets

Sets also support `.copy()`.

```
skills = {"Python", "Git", "SQL"}

skills_copy = skills.copy()
```

Now:

```
skills_copy.add("JavaScript")
```

The original remains:

```
print(skills)
```

```
{'Python', 'Git', 'SQL'}
```

The copy becomes:

```
{'Python', 'Git', 'SQL', 'JavaScript'}
```

---

8. What About Tuples?

Tuples are immutable.

```
coordinates = (10, 20)
```

You normally do not need to copy a tuple just to prevent modifications because the tuple itself cannot be changed.

```
coordinates_copy = coordinates
```

is generally fine if the tuple contains only immutable values.

However, there is an important detail:

> A tuple can contain mutable objects.

For example:

```
data = ([1, 2], [3, 4])
```

The tuple itself cannot be modified, but the lists inside it can.

So nested mutability still matters.

---

9. Shallow Copy

A **shallow copy** creates a new outer object but does not recursively copy every object inside it.

This is the most important idea in this lesson.

Consider:

```
students = [
    ["Hafsa", "Python"],
    ["Asim", "JavaScript"]
]
```

Create a shallow copy:

```
students_copy = students.copy()
```

Now there are two different outer lists.

But the inner lists are still shared.

Conceptually:

```
students ————————→ Outer List A
                      │
                      ├——→ ["Hafsa", "Python"]
                      │
                      └——→ ["Asim", "JavaScript"]


students_copy ——→ Outer List B
                      │
                      ├——→ ["Hafsa", "Python"]
                      │
                      └——→ ["Asim", "JavaScript"]
```

The outer lists are different.

The inner lists are the same objects.

---

10. Shallow Copy Example

```
students = [
    ["Hafsa", "Python"],
    ["Asim", "JavaScript"]
]

students_copy = students.copy()
```

Now modify the outer list:

```
students_copy.append(["Ali", "SQL"])
```

The original does not change.

```
print(students)
```

Output:

```
[
    ["Hafsa", "Python"],
    ["Asim", "JavaScript"]
]
```

This works because the outer lists are separate.

---

11. But Nested Changes Are Different

Now try:

```
students_copy[0].append("Git")
```

Then:

```
print(students)
```

You may see:

```
[
    ["Hafsa", "Python", "Git"],
    ["Asim", "JavaScript"]
]
```

Why did the original change?

Because:

```
students_copy[0]
```

and:

```
students[0]
```

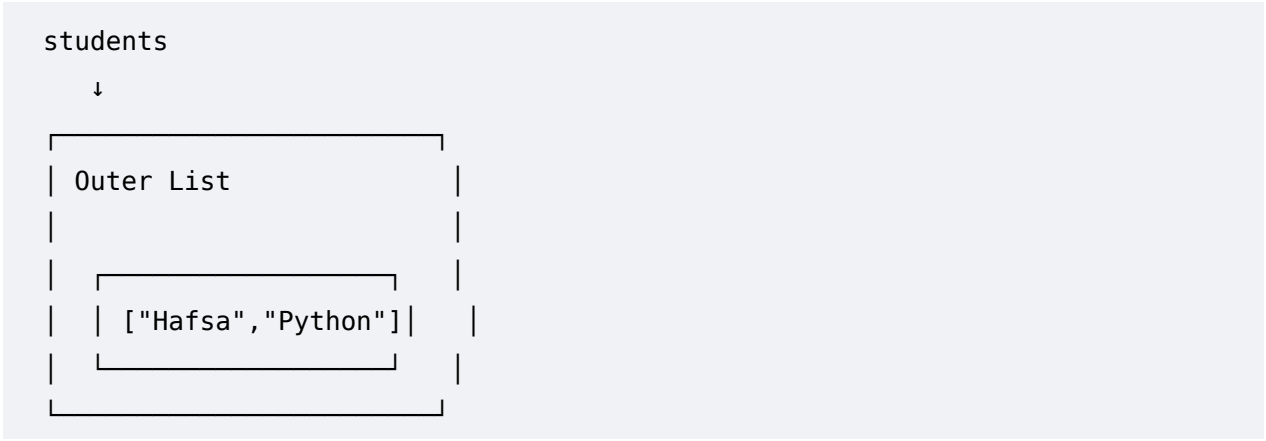
refer to the same inner list.

This is the main limitation of a shallow copy.

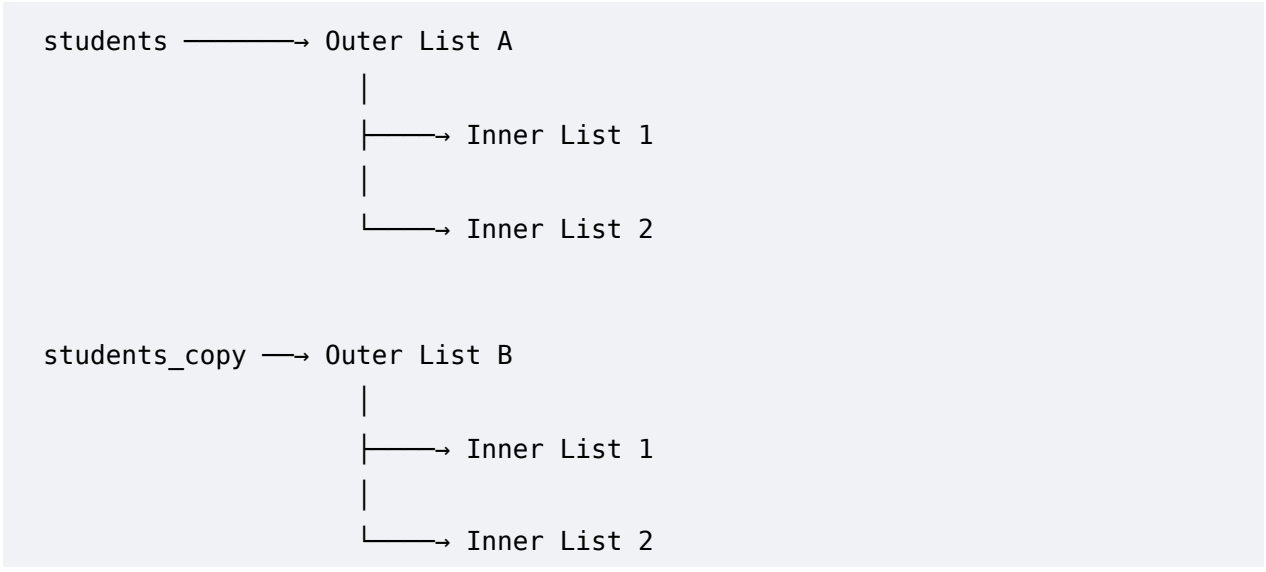
---

12. Visualizing Shallow Copy

Original:



After shallow copying:



Two outer lists.

Same inner lists.

---

13. Deep Copy

A **deep copy** creates a completely independent copy of the object and its nested objects.

Python provides:

```
from copy import deepcopy
```

Then:

```
students_copy = deepcopy(students)
```

Now nested lists are copied too.

---

14. Deep Copy Example

```
from copy import deepcopy

students = [
    ["Hafsa", "Python"],
    ["Asim", "JavaScript"]
]

students_copy = deepcopy(students)
```

Now:

```
students_copy[0].append("Git")
```

The original remains unchanged.

```
print(students)
```

Output:

```
[
    ["Hafsa", "Python"],
    ["Asim", "JavaScript"]
]
```

The copy contains:

```
[
    ["Hafsa", "Python", "Git"],
    ["Asim", "JavaScript"]
]
```

---

15. Shallow Copy vs Deep Copy

Feature	**Shallow Copy**	**Deep Copy**
New outer object	Yes	Yes

Copies nested objects	No	Yes
Nested objects shared	Usually yes	No
Faster	Usually	Usually slower
Good for simple structures	Yes	Yes
Good for deeply nested mutable data	Sometimes	Yes

The key difference:

> **Shallow copy copies the outer structure. Deep copy recursively copies nested objects.**

---

16. Using `copy.copy()`

Python's `copy` module provides:

```
import copy
```

Then:

```
new_data = copy.copy(old_data)
```

This creates a shallow copy.

For example:

```
import copy

numbers = [1, 2, 3]
```

```
numbers_copy = copy.copy(numbers)
```

This is similar to:

```
numbers_copy = numbers.copy()
```

for a list.

---

17. Using `copy.deepcopy()`

For a deep copy:

```
import copy

new_data = copy.deepcopy(old_data)
```

Example:

```
import copy

students = [
    ["Hafsa", "Python"],
    ["Asim", "JavaScript"]
]

students_copy = copy.deepcopy(students)
```

Now nested objects are independent.

---

18. Copying With List Slicing

For lists, slicing can create a shallow copy:

```
numbers = [1, 2, 3, 4]

numbers_copy = numbers[:]
```

This creates a new outer list.

You can verify:

```
numbers_copy.append(5)
```

The original remains:

```
print(numbers)
```

```
[1, 2, 3, 4]
```

---

19. Copying With `list()`

Another way:

```
numbers = [1, 2, 3]

numbers_copy = list(numbers)
```

This creates a new list.

Again, it is a shallow copy.

For simple lists:

```
numbers.copy()
```

```
numbers[:]
```

and:

```
list(numbers)
```

can all create independent outer lists.

---

20. Copying Dictionaries With `dict()`

You can also copy a dictionary using `dict()`:

```
student = {
    "name": "Hafsa",
    "age": 25
}

student_copy = dict(student)
```

This creates a shallow copy.

It works well for simple dictionaries.

---

21. Copying Nested Dictionaries

Consider:

```
student = {
    "name": "Hafsa",
```

```
    "skills": ["Python", "Git"]
}
```

Now:

```
student_copy = student.copy()
```

The outer dictionaries are separate.

But the nested skills list is shared.

So:

```
student_copy["skills"].append("SQL")
```

can also affect:

```
student["skills"]
```

This is because `.copy()` is shallow.

---

22. Deep Copying Nested Dictionaries

Use:

```
from copy import deepcopy

student_copy = deepcopy(student)
```

Now:

```
student_copy["skills"].append("SQL")
```

does not change the original:

```
print(student["skills"])
```

Output:

```
['Python', 'Git']
```

The copy contains:

```
['Python', 'Git', 'SQL']
```

---

23. Assignment vs Shallow Copy vs Deep Copy

Let's compare all three.

Assignment

```
b = a
```

```
Same object
```

Shallow copy

```
b = a.copy()
```

```
New outer object
Nested mutable objects may be shared
```

Deep copy

```
from copy import deepcopy

b = deepcopy(a)
```

```
New outer object
Nested objects are copied too
```

Remember:

```
Assignment
    ↓
Same object

Shallow copy
    ↓
New outer object
    ↓
Nested objects may be shared

Deep copy
    ↓
New outer object
    ↓
Nested objects copied recursively
```

---

24. Comparing Object Identity

Python's `is` operator can help you understand copying.

Example:

```
numbers = [1, 2, 3]

copy_numbers = numbers.copy()

print(numbers is copy_numbers)
```

Output:

```
False
```

They are different list objects.

But with assignment:

```
numbers = [1, 2, 3]

other = numbers

print(numbers is other)
```

Output:

```
True
```

They are the same object.

---

25. `==` vs `is`

These two operators answer different questions.

`==`

Checks whether values are equal.

```
a = [1, 2, 3]
b = [1, 2, 3]

print(a == b)
```

Output:

```
True
```

`is`

Checks whether they are the same object.

```
print(a is b)
```

Output:

```
False
```

They contain equal data but are separate objects.

---

26. Copying Strings

Strings are immutable:

```
name = "Hafsa"
```

You usually do not need to make a defensive copy of a string.

For example:

```
copy_name = name
```

Since strings cannot be modified in place, changing the variable:

```
copy_name = "Asim"
```

does not affect:

```
name
```

This is one advantage of immutability.

---

27. Copying Numbers

The same idea applies to numbers.

```
age = 25
other_age = age
```

Then:

```
other_age = 30
```

The original remains:

```
print(age)
```

Output:

```
25
```

There is no need for `deepcopy()` here.

---

28. Copying Tuples

A tuple is immutable:

```
coordinates = (10, 20)
```

You generally do not need to copy it to prevent direct modification.

But remember:

```
data = ([1, 2], [3, 4])
```

The tuple itself is immutable, but its nested lists are mutable.

So this is possible:

```
data[0].append(5)
```

The tuple structure did not change.

The nested list changed.

---

29. A Tuple Can Contain Mutable Objects

This is an important exception to memorize.

```
data = (
    [1, 2],
    [3, 4]
)
```

You cannot do:

```
data[0] = [10, 20]
```

But you can do:

```
data[0].append(3)
```

because the tuple contains a list, and the list is mutable.

So:

> **Immutability of a container does not automatically make everything inside it immutable.**

---

30. Copying Nested Data

Real applications often contain nested data.

For example:

```
users = [
    {
        "name": "Hafsa",
        "skills": ["Python", "Git"]
    },
    {
        "name": "Asim",
        "skills": ["JavaScript", "React"]
    }
]
```

A shallow copy:

```
users_copy = users.copy()
```

creates a new outer list.

But:

- User dictionaries are shared.
- Skills lists are shared.

If you need complete independence:

```
from copy import deepcopy

users_copy = deepcopy(users)
```

Now the nested objects are copied as well.

---

31. When Should You Use a Shallow Copy?

Use a shallow copy when:

- The data is flat.
- Nested objects are immutable.
- You intentionally want to share nested objects.
- You only need an independent outer collection.
- Copying everything would be unnecessary.

Example:

```
numbers = [1, 2, 3, 4]

numbers_copy = numbers.copy()
```

There are no nested mutable objects.

A shallow copy is enough.

---

32. When Should You Use a Deep Copy?

Deep copy can be useful when:

- Data is deeply nested.
- Nested objects are mutable.
- You need complete independence.
- You are creating a separate version of complex state.
- You need to modify the copy without affecting the original.

Example:

```
from copy import deepcopy

original = {
    "user": {
        "name": "Hafsa",
        "skills": ["Python", "Git"]
    }
}

backup = deepcopy(original)
```

Now the nested structures are independent.

---

33. Do Not Use Deep Copy Automatically

Deep copy is powerful, but you should not use it everywhere.

For a simple list:

```
numbers = [1, 2, 3]
```

this is unnecessary:

```
from copy import deepcopy

numbers_copy = deepcopy(numbers)
```

A simple:

```
numbers_copy = numbers.copy()
```

is enough.

Deep copying can require more memory and processing because Python has to recursively copy nested objects.

Choose the simplest method that satisfies your requirement.

---

34. Real World Example: Shopping Cart

Suppose:

```
cart = [
    {
        "name": "Laptop",
        "quantity": 1
    },
    {
        "name": "Mouse",
        "quantity": 2
    }
]
```

You want to create a separate cart for testing.

Using:

```
test_cart = cart.copy()
```

only copies the outer list.

If you change:

```
test_cart[0]["quantity"] = 5
```

the original cart may also be affected.

For a completely independent test cart:

```
from copy import deepcopy

test_cart = deepcopy(cart)
```

Now the nested dictionaries are independent too.

---

35. Real World Example: API Data

APIs often return nested dictionaries and lists.

Example:

```
response = {
    "user": {
        "name": "Hafsa",
        "skills": ["Python", "Git"]
    },
    "courses": [
        "Python",
        "JavaScript"
    ]
}
```

Suppose you want to transform the data without changing the original response.

A deep copy can be useful:

```
from copy import deepcopy

modified_response = deepcopy(response)
```

Now you can safely modify:

```
modified_response["user"]["skills"].append("SQL")
```

without changing the original `response`.

---

36. Real World Example: Application State

Imagine an application has:

```
state = {
    "theme": "dark",
    "user": {
        "name": "Hafsa",
        "notifications": True
    }
}
```

You may want to create a temporary version:

```
from copy import deepcopy

new_state = deepcopy(state)
```

Now you can test changes without modifying the original state.

This idea appears in:

- Web applications
- Mobile applications
- State management
- Testing
- Data processing
- Undo and redo systems

---

37. Copying and Functions

Suppose a function modifies a list:

```
def add_skill(skills):
    skills.append("Python")
```

If you do:

```
skills = ["Git"]

add_skill(skills)
```

the original list changes.

If you want the function to work on a separate list:

```
def add_skill(skills):
    skills = skills.copy()
    skills.append("Python")
    return skills
```

Now:

```
skills = ["Git"]

new_skills = add_skill(skills)

print(skills)
print(new_skills)
```

Output:

```
['Git']
['Git', 'Python']
```

The original list remains unchanged.

---

38. A Better Design: Return New Data

Instead of unexpectedly changing an input, a function can create and return a new object.

For example:

```
def add_skill(skills, skill):
    new_skills = skills.copy()
    new_skills.append(skill)
    return new_skills
```

Then:

```
skills = ["Python"]

updated_skills = add_skill(skills, "Git")
```

Now:

```
skills
→ ["Python"]

updated_skills
→ ["Python", "Git"]
```

This can make code easier to reason about.

---

39. Common Mistake: Using `=`

Wrong:

```
backup = original
```

when you need an independent copy.

Better:

```
backup = original.copy()
```

for a simple structure.

Or:

```
backup = deepcopy(original)
```

for nested mutable data.

---

40. Common Mistake: Thinking `.copy()` Is Always Deep

This:

```
copy_data = data.copy()
```

does **not** mean:

> "Copy everything inside."

It means:

> "Create a shallow copy of the outer object."

For nested structures, understand whether inner objects are shared.

---

41. Common Mistake: Deep Copying Everything

Do not automatically write:

```
deepcopy(data)
```

for every object.

For simple data, it may be unnecessary.

For example:

```
numbers = [1, 2, 3]
```

A shallow copy is enough:

```
numbers_copy = numbers.copy()
```

Use deep copy when nested mutable objects actually need to be independent.

---

42. Common Mistake: Forgetting Nested Data

Consider:

```
data = {
    "skills": ["Python", "Git"]
}

copy_data = data.copy()
```

Changing:

```
copy_data["skills"].append("SQL")
```

also affects:

```
data["skills"]
```

because the nested list is shared.

If this is not intended:

```
from copy import deepcopy

copy_data = deepcopy(data)
```

---

43. Problem Solving Framework

When you need to copy data, ask these questions.

Step 1: Is the object mutable?

If it is immutable, copying may not be necessary.

Examples:

```
str
int
float
tuple
```

Step 2: Is the structure nested?

If no:

```
copy = original.copy()
```

may be enough.

If yes, continue.

Step 3: Are nested objects mutable?

If no, a shallow copy may be enough.

If yes, consider deep copy.

Step 4: Do you need complete independence?

If yes:

```
from copy import deepcopy

copy = deepcopy(original)
```

Step 5: Do you intentionally want shared nested data?

If yes, a shallow copy may be exactly what you need.

---

44. Quick Decision Guide

```
Do I need a copy?
      |
      No
      ↓
Use the original

      Yes
      ↓

Is the object immutable?
      |
      Yes
      ↓
Copy usually unnecessary

      No
      ↓

Is the data nested?
      |
      No
      ↓
Shallow copy

      Yes
      ↓

Do nested mutable objects need independence?
      |
      Yes
      ↓
Deep copy

      No
      ↓
Shallow copy
```

---

45. Comparison Table

Method	Creates New Outer Object	Copies Nested Objects	Typical Use
`b = a`	No	No	Share same object
`a.copy()`	Yes	No	Simple/shallow copy
`a[:]`	Yes	No	Copy a list
`list(a)`	Yes	No	Copy a list
`dict(a)`	Yes	No	Copy a dictionary
`copy.copy(a)`	Yes	No	General shallow copy
`copy.deepcopy(a)`	Yes	Yes	Independent nested copy

---

46. Mini Project: Resource Editor

Imagine haas.dev has a resource:

```
resource = {
    "title": "Python Lists",
    "category": "Python",
    "tags": ["python", "lists", "programming"]
}
```

You want to create a draft version.

A simple assignment is unsafe:

```
draft = resource
```

Instead:

```
from copy import deepcopy

draft = deepcopy(resource)
```

Now modify the draft:

```
draft["title"] = "Python Lists Deep Dive"
draft["tags"].append("data-structures")
```

The original resource remains unchanged.

This is useful when building systems that need:

- Draft versions
- Preview versions
- Temporary edits
- Backup data
- Test data

---

47. 5 Minute Challenge

Predict the output:

```
data = {
    "skills": ["Python", "Git"]
}

copy_data = data.copy()

copy_data["skills"].append("SQL")

print(data)
print(copy_data)
```

Answer

Both contain:

```
{
    "skills": ["Python", "Git", "SQL"]
}
```

Why?

Because `.copy()` created a shallow copy.

The nested `skills` list is shared.

Challenge 2

Now predict:

```
from copy import deepcopy

data = {
    "skills": ["Python", "Git"]
}

copy_data = deepcopy(data)

copy_data["skills"].append("SQL")

print(data)
print(copy_data)
```

Answer

Original:

```
{
    "skills": ["Python", "Git"]
}
```

Copy:

```
{
    "skills": ["Python", "Git", "SQL"]
}
```

The nested list was copied.

---

48. Exercises

Exercise 1

Explain the difference between:

```
b = a
```

and:

```
b = a.copy()
```

Exercise 2

Create an independent copy of:

```
numbers = [10, 20, 30]
```

Then add `40` to the copy without changing the original.

Exercise 3

Create a shallow copy of:

```
student = {
    "name": "Hafsa",
    "age": 25
}
```

Change the age in the copy.

Exercise 4

Create a deep copy of:

```
student = {
    "name": "Hafsa",
    "skills": ["Python", "Git"]
}
```

Add `"SQL"` to the copy's skills without changing the original.

Exercise 5

Explain why this can change the original:

```
data_copy = data.copy()

data_copy["items"].append("New Item")
```

Exercise 6

Choose the correct approach:

Scenario A

A simple list of integers.

Scenario B

A nested dictionary containing lists and dictionaries.

Scenario C

A string.

Explain whether you need:

- Assignment
- Shallow copy
- Deep copy

---

49. Mini Quiz

Question 1

What does this do?

```
b = a
```

A. Creates a deep copy

B. Creates a shallow copy

C. Makes `b` refer to the same object as `a`

D. Converts `a` into a list

Answer: C

Question 2

What does `.copy()` usually create?

A. Deep copy

B. Shallow copy

C. No object

D. Immutable object

Answer: B

Question 3

Which function creates a deep copy?

A. `copy.copy()`

B. `copy.list()`

C. `copy.deepcopy()`

D. `deep.copy()`

Answer: C

Question 4

Why can `.copy()` still cause changes to nested data?

A. Python ignores nested objects

B. Nested objects may still be shared

C. `.copy()` converts lists into tuples

D. Dictionaries cannot be copied

Answer: B

---

50. Interview Questions

Beginner

1. What is the difference between assignment and copying?
2. What does `.copy()` do?
3. What is a shallow copy?
4. What is a deep copy?
5. Why doesn't `b = a` create an independent copy?
6. How can you copy a list?

Intermediate

7. What is the difference between shallow copy and deep copy?
8. Why can modifying a nested list affect the original after `.copy()`?
9. How does `copy.copy()` work?
10. How does `copy.deepcopy()` work?
11. What is the difference between `==` and `is` when testing copies?
12. When would you use a shallow copy instead of a deep copy?
13. Why can deep copying be more expensive?
14. How do mutable nested objects affect copying?

Practical

15. How would you safely copy an API response before modifying it?
16. How would you create an independent backup of a nested dictionary?
17. How would you copy a shopping cart for testing?
18. How would you prevent a function from modifying a caller's list?
19. How would you copy nested application state?

---

51. Cheat Sheet

```
ASSIGNMENT
b = a

Same object.
No copy.


SHALLOW COPY
b = a.copy()

New outer object.
Nested mutable objects may still be shared.


LIST SLICING
b = a[:]

Shallow copy of a list.


LIST CONSTRUCTOR
b = list(a)

Shallow copy of a list.


DICTIONARY CONSTRUCTOR
b = dict(a)

Shallow copy of a dictionary.


COPY MODULE
import copy

b = copy.copy(a)

Shallow copy.


DEEP COPY
```

```
import copy

b = copy.deepcopy(a)

Copies nested objects recursively.


IDENTITY
a is b

Checks whether both references point to the same object.


EQUALITY
a == b

Checks whether values are equal.
```

---

52. Key Takeaways

- `b = a` is **assignment**, not copying.
- Assignment makes both variables refer to the same object.
- `.copy()` creates a **shallow copy**.
- Slicing can create a shallow copy of a list.
- `list()` and `dict()` can create shallow copies of their respective structures.
- `copy.copy()` creates a shallow copy.
- `copy.deepcopy()` creates a deep copy.
- Shallow copies create a new outer object but may share nested objects.
- Deep copies recursively copy nested objects.
- Immutable objects usually do not need defensive copying.
- Use shallow copies for simple structures when appropriate.
- Use deep copies when nested mutable objects need to be independent.
- Do not use deep copy automatically; it can consume more resources than necessary.
- Understanding references, mutation, and copying helps prevent subtle bugs.

> **Remember: Assignment shares. Shallow copy separates the outer structure. Deep copy separates the nested structure too.**

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app

https://dev-roast-app.vercel.app/resources