🔍 Internal Working of Python: Copy, References & Memory Management

🧠 1. Understanding Python Variables – It's all about **References**

In Python, **everything is an object**, and **variables are just references (pointers)** to those objects.

```
a = 10
b = a
```

Here:

- `a` points to the object `10`.
- `b` **also points to** the same object `10`.

🧪 Both `a is b` and `a == b` are `True`.

👉 **Integers are immutable**, so any change results in a new object.

🔢 2. Reference Counting – Python's 🧹 Memory Manager

Python uses **reference counting** for memory management:

```
import sys
a = []
print(sys.getrefcount(a))  # usually prints 2 due to temp reference in function call
```

- The count increases when a new reference is made.
- Decreases when a reference is deleted (`del` or re-assignment).
- When count hits **zero**, 🧹 Python's **Garbage Collector** destroys the object.

📃 3. Mutable vs Immutable – The DNA 🧬 of Behavior

Type	Mutable	Example
int, str	❌ No	`a = 10`, `s = "hi"`
list, dict	☑️ Yes	`lst = [1, 2]`

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

```
b.append(3)
print(a)  # [1, 2, 3] ➡ both `a` and `b` point to the same object!
```

✂ 4. Slicing – Copy or Not? 🤔

☑ Immutable types (like `str`, `tuple`) always return a new object.

```
s1 = "hello"
s2 = s1[:]
print(s1 is s2)  # False
```

⚠ For mutable types (like `list`), slicing **creates a shallow copy**.

```
lst1 = [1, 2, 3]
lst2 = lst1[:]
print(lst1 is lst2)  # False (new list)
```

But beware! ⚠

```
nested = [[1, 2], [3, 4]]
shallow = nested[:]
shallow[0][0] = 999
print(nested)  # [[999, 2], [3, 4]]
```

👉 The inner lists are still **shared**!

🔄 5. Copying – Identity Crisis 😅

◇ Assignment (`=`)

Just creates another reference to the **same** object:

```
a = [1, 2]
b = a
b.append(3)
print(a)  # [1, 2, 3]
```

◇ Shallow Copy (`copy.copy()`, `[:]`)

- Creates a **new outer object**.

- **Inner objects are still shared**.

```
import copy
original = [[1, 2], [3, 4]]
shallow = copy.copy(original)
shallow[0][0] = 100
print(original)  # [[100, 2], [3, 4]]
```

◇ Deep Copy (`copy.deepcopy()`)

Creates a full copy — outer + inner objects:

```
deep = copy.deepcopy(original)
deep[0][0] = 999
print(original)  # [[1, 2], [3, 4]]
```

☑ Changes in `deep` don't affect `original`.

🧪 6. Edge Case Behavior

🎯 Int Behavior

```
a = 256
b = 256
print(a is b)  # True ☑ (cached small ints)

a = 257
b = 257
print(a is b)  # False ✖ (new objects beyond cache range)
```

🎯 List Behavior

```
lst = [1, 2, 3]
copied = lst[:]
copied.append(4)
print(lst)      # [1, 2, 3]
print(copied)   # [1, 2, 3, 4]
```

But...

```python
a = [[1], [2]]
b = a[:]
b[0].append(999)
print(a)  # [[1, 999], [2]] — inner list is shared!
```

🧪 7. Copy inside Functions

```python
def modify(lst):
    lst.append(99)

a = [1, 2]
modify(a)
print(a)  # [1, 2, 99] — passed by reference!
```

If you want to **avoid side-effects**, pass a **copy**:

```python
modify(a[:])  # or use copy.deepcopy(a)
```

🚀 Tips to Master Internals

- Use `id()` to track object identity.
- Use `sys.getrefcount()` to debug references.
- Use `is` to check **object identity**, not equality.
- Use `copy.deepcopy()` to fully isolate data.
- Slice carefully with nested structures!

🧠 Summary Table

Operation	Type of Copy	Outer Object	Inner Objects	Use Case
`a = b`	No copy	Same	Same	Share reference
`a[:]`	Shallow copy	New	Same	Quick copy
`copy.copy(a)`	Shallow copy	New	Same	Like slice
`copy.deepcopy(a)`	Deep copy	New	New	Full isolation

💡 Final Thoughts

- In Python, **knowing how objects and references work** is 🔑 to writing safe, efficient code.
- It's not about copying values — it's about understanding **who points to what** 🔁.
- Always test for `is` vs `==`, especially with mutable types!