🧠 Python: Mutable vs Immutable Explained 🚀

🔍 What is Mutability in Python?

🧩 In simple terms:

- **Mutable** = Can be **changed** after creation ☑️
- **Immutable** = **Cannot be changed** once created 🚫

🔐 Immutable Objects (🚫 Cannot be changed)

Type	Examples
`int`	`1`, `42`
`float`	`3.14`, `0.99`
`str`	`"hello"`, `'world'`
`tuple`	`(1, 2)`, `('a', 'b')`
`bool`	`True`, `False`
`frozenset`	`frozenset({1, 2, 3})`

🔒 Example:

```
>>> x = 10
>>> id(x)
140734294843888

>>> x += 1  # creates a new object!
>>> id(x)
140734294843920  # Different ID
```

🧠 `int` is immutable — modifying it actually creates a **new object**.

🔄 Mutable Objects (☑️ Can be changed)

Type	Examples
`list`	`[1, 2, 3]`
`dict`	`{"a": 1, "b": 2}`
`set`	`{1, 2, 3}`

Type	Examples
`bytearray`	`bytearray(b"abc")`
`Custom Class`	With modifiable attributes

🔧 Example:

```
>>> nums = [1, 2, 3]
>>> id(nums)
139800123

>>> nums.append(4)
>>> nums
[1, 2, 3, 4]

>>> id(nums)
139800123  # Same ID – changed in place!
```

🧪 Deeper: Why does this matter?

☑ Mutable:

You can change the object without changing its identity (memory address)

❌ Immutable:

Any "change" creates a **new object**

⚠ Common Pitfall with Mutable Default Args

```
def append_num(num, nums=[]):  # Bad! 😱
    nums.append(num)
    return nums

>>> append_num(1)
[1]
>>> append_num(2)
[1, 2]  # Unexpected! 😵
```

☑ **Fix it:**

```
def append_num(num, nums=None):
    if nums is None:
        nums = []
```

```
    nums.append(num)
    return nums
```

🧠 Immutables inside Mutables

```
a = [1, 2, (3, 4)]
a[2] = (5, 6)  # ☑ OK, can change the *element*
```

But the tuple `(3, 4)` itself can't be changed.

🔁 Copying and Mutability

`=`, `copy()`, and `deepcopy()`

```
import copy

a = [1, 2, [3, 4]]
b = copy.copy(a)       # Shallow copy
c = copy.deepcopy(a)   # Deep copy

a[2][0] = 99
print(b)  # [1, 2, [99, 4]]
print(c)  # [1, 2, [3, 4]]
```

☑ Use Cases

Scenario	Prefer
Configuration keys	Immutable (`tuple`)
Changing datasets	Mutable (`list`, `dict`)
Hashing in sets/dicts	Must be immutable (`str`, `int`, `tuple`)
Memory efficiency (fixed)	Immutable
Avoiding side-effects	Immutable

🧪 How to Check If Object is Mutable?

```
>>> a = [1, 2]
>>> id_before = id(a)
>>> a.append(3)
```

```
>>> id_before == id(a)
True  # ☑ Mutable

>>> s = "hello"
>>> id(s) == id(s + " world")
False  # 🚫 Immutable
```

💡 Pro Tips

🧠 Strings are immutable — every `+`, `replace()`, `strip()` returns a **new string** 🔁 Tuples are immutable, but **can contain mutables** 🧪 Use `id()` to track object identity ☑ Use `frozenset` when you want an immutable version of `set` 📌 Use `tuple` over `list` for fixed collections (e.g., coordinates)

📌 Summary Table

Type	Mutable?	Example	Notes
`int`	❌	`x = 1`	Any math creates new object
`str`	❌	`s = "abc"`	All ops return new string
`list`	☑	`l = [1, 2]`	Use for dynamic collections
`dict`	☑	`{"a": 1}`	Best for key-value pairs
`tuple`	❌	`(1, 2)`	Use when fixed and ordered
`set`	☑	`{1, 2}`	Unordered, no duplicates
`frozenset`	❌	`frozenset({})`	Immutable version of set

📊 Mutable vs Immutable Types in Python

🔢 Sr	📦 Data Type	🔄 Mutable?	🧪 Example	📝 Notes / Remarks
1️⃣	`int`	❌ No	`x = 10`	Immutable – any operation creates a new object
2️⃣	`float`	❌ No	`pi = 3.14`	Same as `int` – immutable
3️⃣	`bool`	❌ No	`is_on = True`	Internally treated as integers
4️⃣	`str`	❌ No	`s = "hello"`	Every change returns a new string
5️⃣	`tuple`	❌ No	`t = (1, 2, 3)`	Immutable, but can contain mutables
6️⃣	`frozenset`	❌ No	`fs = frozenset([1, 2])`	Immutable version of `set`

🔢 Sr	📦 Data Type	🔄 Mutable?	🧪 Example	📝 Notes / Remarks
7️⃣	`bytes`	❌ No	`b = b'abc'`	Immutable sequence of bytes
8️⃣	`list`	✅ Yes	`l = [1, 2, 3]`	Supports append, pop, etc.
9️⃣	`dict`	✅ Yes	`d = {"a": 1}`	Keys must be immutable
🔟	`set`	✅ Yes	`s = {1, 2, 3}`	Unordered, no duplicates, mutable
1️⃣1️⃣	`bytearray`	✅ Yes	`ba = bytearray(b"abc")`	Mutable version of bytes
1️⃣2️⃣	`custom class`	✅ Yes*	`class A: pass`	Mutable by default unless overridden

🧠 Key Points

- ❗ You **can't** use mutable types as keys in `dict` or elements in `set`.
- ✅ Immutable types are **hashable** and used in sets/dicts as keys.
- 🔁 Mutable objects allow **in-place** changes — great for performance but risky in shared environments.