Immutable Vs Hashable Concepts

Created By @Mohamed Hamed

Artificial Intelligence student at KFS university, you can find me down:

- LinkedIn
- GitHub
- YouTube
- ✔ To get a folder about Python (Questions, Explaining and Notes)

Immutable and Hashable

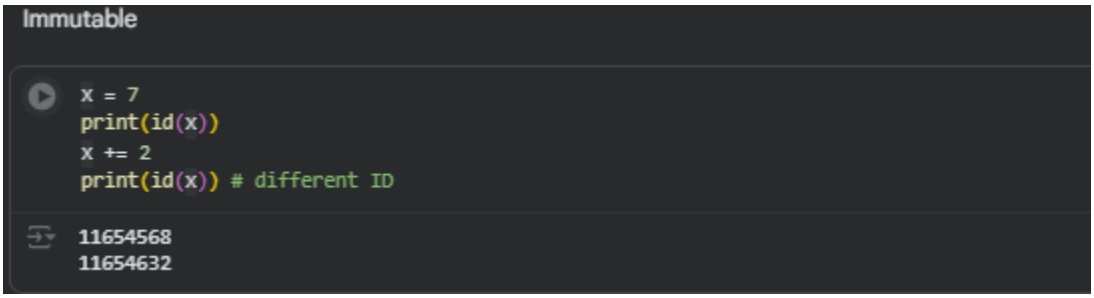
These concepts are important in **Python** So, let's define both concepts (immutability & hashability) clearly and simply as i can.

Immutable (Can't be changed)

An object is **immutable** if its value cannot be changed after it's created. So once it's made, its contents stay frozen forever. If you "change" it, **Python** actually creates a new object in memory and opposite it is the **mutable** object (changed directly in memory).

Examples of Immutable Types:

- **All numeric types**
- **String**
- **Tuble (if it contains only immutable items)**



x+= 2:

*** Rebinds the name x to point to the new object.**
*** Different ID means a new object was created.**
*** The original 7 object still exists but you have no variable pointing to it anymore.**

But what about the oldest value?

The old object (7) stays in memory **temporarily** until **Python's garbage collector** (GC) sees that no variables are referencing it anymore, then it frees **(deallocates)** that memory**.**

```
x ⟶ [5]        # first
x ⟶ [6]        # now points to a new object
    [5] ❌ no names → GC will clean it
```

if you want to keep the old value:

You must store it before reassigning:

```
x = 5
old_x = x       # save the reference
x += 1
print(old_x, x)
```

```
5 6
```

But in the mutable type:

```
y = [1, 3, 5]
print(id(y))
y.append(7)
print(id(y)) # same ID
```

```
137261364723328
137261364723328
```

*** The same list object was modified in place.**
*** So y still points to the same memory address.**

if you want to keep the old value:

You must store or copy it before modification.

```
lst = [1, 2, 3]
old_lst = lst.copy()  # or list(lst) or deep copy
lst.append(4)

print(old_lst)  # [1, 2, 3]
print(lst)      # [1, 2, 3, 4]
```

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

But what about it? 'Tuble (if it contains only immutable items)'

- A tuple itself is immutable:

So the tuple structure (the number of items and their positions) is always immutable. Then you can't change, add, or remove elements inside a tuple directly.

```
t = (1, 2, 3)
t[0] = 99
# TypeError: 'tuple' object does not support item assignment
```

But immutability inside matters

Even though the tuple can't change,
the objects stored inside the tuple might be mutable.

That's why i said:
A tuple is truly immutable only if all its elements are immutable.

```
t = ([5, 7], 1, 2, 3)
t[0][1]= 5
print(t)
```

```
([5, 5], 1, 2, 3)
```

Hashability (Has a unique fixed fingerprint)

An object is hashable if it has a hash value that **never** changes during its lifetime and can be **compared** to other objects.

*** the hash value is an int returned by the built-in hash() function.**
*** python uses this fingerprint (hash value) to quickly locate objects inside hash-based data structures, like:**

- **dict → keys are hashed**
- **set → elements are hashed**

when doing it:

```
d = {"apple": 1, "banana": 2}
```

Python computes:

```
hash("apple") → hash value (integer)
hash("banana") → another hash value
```

*** It then stores those hash values in a hash table, which lets Python find "banana" much faster than the traditional method (not our point now).**

But why hash values must be constant?

Imagine if you could change a key after inserting it in a dictionary:

```
key = [1, 2, 3]      # list (mutable)
d = {key: "value"}   # Not allowed
```

*** Python forbids this cuz if you modify the list later (e.g., append 4), the hash would change, and the dictionary would lose track of the key's position. So hashability guarantees stable lookup.**

*** A hashable object = immutable in behavior, with a fixed identity, a stable hash, and equality logic.**

Rule	Description	Why It Matters
1. Must have __hash__()	The object must implement a __hash__() method that returns an integer (its hash value).	Used by Python to find the object's bucket in a dictionary or set quickly (O(1) lookup).
2. Must have __eq__()	The object must be comparable — it defines equality via __eq__().	If two objects have the same hash, Python uses __eq__() to confirm whether they are truly the same key.
3. Hash value must stay constant	The object's hash must not change during its lifetime.	Ensures dictionary/set can always find the object correctly; if hash changes, lookups would fail.

Note:

A tuple is:

- **Immutable → always true (you can't reassign its elements)**
- **Hashable → only true if all elements inside are hashable (and thus immutable)**

All hashable built-in types are immutable, but not all immutable types are automatically hashable

*** A hashable object has a *stable identity* in terms of value comparison and hashing. That's why dictionaries and sets can rely on them to stay consistent (even if other objects around them change).**

*** Passing an unhashable or mutable object where a hashable one is expected ⇒ TypeError.**

*** So the rule is:**
If Python expects a hashable object (for dict/set key or hash()), and you give it a mutable/unhashable one → TypeError: unhashable type: '<type_name>'

* But why it's TypeError, not ValueError?

Cuz the type of object (list, dict, set, etc.) is inherently incapable of being hashed it's not about the value, it's about the type's behavior (Errors in py).