# Python Mutability vs Immutability

## Introduction

When working with Python data, you will often change values.

But not every Python object can be changed after it is created.

This leads to an important concept:

> **Mutability vs Immutability**

Understanding this concept helps you understand:

- Lists and tuples
- Sets and dictionaries
- Strings
- Variables and objects
- Functions
- Memory references
- Why some operations create new objects
- Why some operations modify existing objects
- Why certain objects can be used as dictionary keys

---

# 1. What Does Mutable Mean?

**Mutable** means:

> An object can be changed after it has been created.

For example, a list is mutable.

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

numbers[0] = 100

print(numbers)
```

Output:

```
[100, 20, 30]
```

We changed the existing list.

We did not need to create a completely new list.

---

# 2. What Does Immutable Mean?

**Immutable** means:

> An object cannot be changed after it has been created.

For example, strings are immutable.

```
name = "Hafsa"
```

You cannot directly change one character:

```
name[0] = "X"
```

Python gives an error because strings cannot be modified in place.

Instead, you create a new string:

```
name = "Xafsa"
```

The variable `name` now refers to a new string.

---

# 3. The Main Difference

Think of it this way:

### Mutable

```
Object created
      ↓
Object can change
      ↓
Same object can be modified
```

### Immutable

```
Object created
      ↓
Object cannot change
      ↓
Create another object instead
```

This difference is extremely important in Python.

---

# 4. Common Mutable Types

The main mutable built-in collection types are:

### List

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

### Dictionary

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

### Set

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

These objects can be modified after creation.

---

# 5. Common Immutable Types

Common immutable Python types include:

- `int`
- `float`
- `bool`
- `str`
- `tuple`
- `frozenset`
- `NoneType`

Examples:

```
age = 25
```

```
price = 99.99
```

```
name = "Hafsa"
```

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

These objects cannot be changed in place.

---

# 6. Mutable Example: List

Consider:

```
skills = ["Python", "Git"]
```

We can modify the list:

```
skills.append("SQL")
```

Now:

```
print(skills)
```

Output:

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

The list itself was modified.

---

# 7. Immutable Example: String

Consider:

```
language = "Python"
```

You cannot do:

```
language[0] = "J"
```

Instead, create a new string:

```
language = "Jython"
```

The original string was not modified.

The variable now refers to another string object.

---

# 8. Important: Variables Are Not the Same as Objects

This is one of the most important ideas.

Consider:

```
name = "Hafsa"
```

It is tempting to think:

> "`name` contains the string."

A better mental model is:

```
name ————→ "Hafsa"
```

The variable is a **reference to an object**.

The object is the actual string.

This becomes especially important when discussing mutability.

---

# 9. Assignment Does Not Automatically Create a Copy

Consider:

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

other_numbers = numbers
```

Now:

```
numbers ──────────┐
                  ↓
              [1, 2, 3]
                  ↑
                  │
other_numbers ────┘
```

Both variables refer to the same list.

Now change the list:

```
other_numbers.append(4)
```

What happens?

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

Output:

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

Why?

Because there is only one list.

Both variables refer to it.

---

# 10. Mutable Objects Can Produce Unexpected Changes

Consider:

```
student = ["Hafsa", 25]

student_copy = student

student_copy.append("Python")
```

You might expect only `student_copy` to change.

But:

```
print(student)
```

also produces:

```
['Hafsa', 25, 'Python']
```

Because:

```
student_copy = student
```

does not create an independent copy.

It creates another reference to the same object.

---

# 11. How to Create a Copy

If you want a separate list, you can create a copy.

### Using `.copy()`

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

other_students = students.copy()

other_students.append("Ali")

print(students)
print(other_students)
```

Output:

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

Now they are separate list objects.

---

# 12. Copying With Slicing

For lists, you can also use slicing:

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

copy_numbers = numbers[:]
```

Now:

```
copy_numbers.append(4)
```

The original remains unchanged:

```
print(numbers)
```

Output:

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

---

# 13. Mutable vs Immutable With Assignment

Let's compare.

## Mutable Object

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

b.append(4)
```

Both references see the change:

```
a → [1, 2, 3, 4]
b → [1, 2, 3, 4]
```

---

## Immutable Object

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

Both initially refer to the same integer object.

Now:

```
b = 20
```

This does not modify the integer `10`.

Instead, `b` is assigned to another integer object.

```
a → 10

b → 20
```

The original `10` was not changed.

---

# 14. Why Can't Integers Be Changed?

Consider:

```
x = 10
```

You cannot modify the integer itself:

```
x.some_operation_that_changes_10()
```

Instead, operations create or reference another integer value.

For example:

```
x = x + 5
```

Conceptually:

```
Before:

x → 10

After:

x → 15
```

The integer `10` was not changed into `15`.

The variable `x` now refers to `15`.

---

# 15. Strings Are Immutable

Strings are one of the most important examples of immutable objects.

```
name = "Hafsa"
```

You cannot change one character:

```
name[0] = "S"
```

Instead:

```
name = "Safsa"
```

Or create a modified string:

```
name = name.replace("H", "S")
```

The `replace()` method returns a new string.

It does not modify the original string.

---

# 16. Lists Are Mutable

Lists can be changed directly.

```
languages = ["Python", "JavaScript"]
```

You can:

```
languages.append("Java")
```

You can:

```
languages[0] = "C++"
```

You can:

```
languages.remove("JavaScript")
```

The list itself changes.

---

# 17. Tuples Are Immutable

Consider:

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

You cannot do:

```
coordinates[0] = 50
```

Python raises an error.

If you need different coordinates, create a new tuple:

```
coordinates = (50, 20)
```

Again, the old tuple was not modified.

---

# 18. Sets Are Mutable

A set can be modified.

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

skills.add("SQL")
```

Now:

```
print(skills)
```

contains:

```
Python
Git
SQL
```

You can also remove values:

```
skills.remove("Git")
```

So sets are mutable.

---

# 19. Dictionaries Are Mutable

Dictionaries can also be changed.

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

Update a value:

```
user["age"] = 26
```

Add a new key:

```
user["role"] = "Developer"
```

Remove a key:

```
del user["age"]
```

The dictionary itself changes.

---

## 20. A Quick Comparison

| Type | Mutable? |
|---|---|
| `int` | No |
| `float` | No |
| `bool` | No |
| `str` | No |
| `tuple` | No |
| `frozenset` | No |
| `list` | Yes |
| `set` | Yes |
| `dict` | Yes |

A simple memory trick:

> **Collections such as lists, sets, and dictionaries are generally mutable, while basic values such as strings, numbers, and tuples are immutable.**

---

## 21. Why Does Mutability Matter?

Mutability matters because it affects how your program behaves when multiple variables refer to the same object.

For example:

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

backup = data
```

Changing `data` also affects what `backup` sees:

```
data.append(4)
```

Now:

```
print(backup)
```

Output:

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

This can be useful.

But it can also cause bugs if you did not expect it.

---

# 22. Function Example With Mutable Data

Consider:

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

Now:

```
my_skills = ["Git"]

add_skill(my_skills)

print(my_skills)
```

Output:

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

Why?

Because the function received a reference to the same mutable list.

The function modified that list.

---

# 23. Function Example With Immutable Data

Now consider an integer:

```
def increase_age(age):
    age = age + 1
```

Call it:

```
my_age = 25

increase_age(my_age)

print(my_age)
```

Output:

```
25
```

Why did the original value not change?

Because integers are immutable.

Inside the function:

```
age = age + 1
```

creates/references another integer.

It does not modify the original integer object.

---

# 24. Important: Python Is Not Simply "Pass by Reference"

You may hear people say:

> "Python passes variables by reference."

This is an oversimplification.

A better explanation is:

> **Python passes object references by assignment.**

The function receives a reference to the object.

What happens next depends on whether the object is mutable and whether the function mutates it or simply reassigns its local variable.

For beginners, remember:

```
Mutable object
→ Can be modified inside a function.

Immutable object
→ Cannot be modified in place.
```

---

# 25. Mutation vs Reassignment

These are not the same thing.

## Mutation

Changing the existing object.

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

numbers.append(4)
```

The list was mutated.

---

### Reassignment

Making a variable refer to another object.

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

numbers = [10, 20]
```

The variable now refers to a different list.

This distinction is extremely important.

---

## 26. Example of Mutation

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

numbers.append(4)
```

Conceptually:

```
numbers ──→ [1, 2, 3, 4]
```

The same list was changed.

---

## 27. Example of Reassignment

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

numbers = [10, 20]
```

Conceptually:

```
Before:

numbers ──→ [1, 2, 3]


After:

numbers ──→ [10, 20]
```

The original list was not modified.

The variable simply points to another object.

---

# 28. Why Strings Can Look Like They Change

Consider:

```
name = "Hafsa"

name = name.upper()
```

Output:

```
HAFSA
```

It may look like Python modified the original string.

But it did not.

`upper()` returns a new string.

Conceptually:

```
Before:

name → "Hafsa"

After:

name → "HAFSA"
```

The original string `"Hafsa"` remained unchanged.

---

# 29. Nested Mutable Objects

Mutability becomes more interesting with nested data.

Consider:

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

The outer list is mutable.

The dictionary inside it is mutable.

The skills list is also mutable.

You can do:

```
students[0]["skills"].append("SQL")
```

Now:

```
print(students)
```

Output:

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

Several mutable objects exist inside one another.

---

# 30. Shallow Copy

Consider:

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

copy_students = students.copy()
```

The outer list is copied.

But the nested dictionary and nested list are still shared.

So:

```
copy_students[0]["skills"].append("SQL")
```

can also affect the nested data visible through `students`.

This is called a **shallow copy**.

---

# 31. Deep Copy

When you need completely independent nested data, Python provides `deepcopy()`.

```
from copy import deepcopy

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

copy_students = deepcopy(students)
```

Now nested objects are copied as well.

Changing:

```
copy_students[0]["skills"].append("SQL")
```

does not modify the original nested list.

```
print(students)
```

Still contains:

```
["Python", "Git"]
```

---

# 32. Why Immutable Objects Can Be Useful

Immutability provides some useful properties.

## 1. More predictable data

An immutable object cannot unexpectedly change.

For example:

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

Other parts of the program cannot modify the tuple itself.

---

## 2. Safer sharing

If multiple parts of your program use the same immutable object, one part cannot mutate it.

---

## 3. Can be used as dictionary keys

Immutable hashable objects can be dictionary keys.

For example:

```
locations = {
    (10, 20): "Point A",
    (30, 40): "Point B"
}
```

A tuple can be used as a key when its contents are hashable.

A list cannot:

```
locations = {
    [10, 20]: "Point A"
}
```

This produces an error because lists are mutable and unhashable.

---

# 33. Hashability

You will encounter the term **hashable** when working with dictionaries and sets.

A simplified definition:

> An object is hashable if Python can calculate a stable hash value for it during its lifetime.

Common hashable objects include:

```
int
float
str
bool
tuple containing hashable values
```

Common unhashable objects include:

```
list
dict
set
```

This is why you can do:

```
data = {
    "name": "Hafsa"
}
```

But not:

```
data = {
    ["name"]: "Hafsa"
```

```
}
```

The list cannot be used as a dictionary key.

---

# 34. Mutable Default Arguments: A Common Trap

Consider this function:

```
def add_item(item, items=[]):
    items.append(item)
    return items
```

Now:

```
print(add_item("Python"))
print(add_item("Git"))
```

You might expect:

```
["Python"]
["Git"]
```

But the result is:

```
["Python"]
["Python", "Git"]
```

Why?

Because the default list is created once and reused between calls.

This is a classic example of why understanding mutability matters.

---

# 35. The Better Approach

Use `None` as the default:

```
def add_item(item, items=None):
    if items is None:
        items = []

    items.append(item)
    return items
```

Now:

```
print(add_item("Python"))
print(add_item("Git"))
```

Output:

```
["Python"]
["Git"]
```

A fresh list is created for each call where no list is provided.

---

# 36. Mutable Objects in Real Applications

Imagine a shopping cart:

```
cart = []
```

Adding products should change the cart:

```
cart.append("Laptop")
cart.append("Mouse")
```

A mutable list is useful here because the cart naturally changes over time.

Another example is application settings:

```
settings = {
    "theme": "light",
    "notifications": True
}
```

You may need to update settings:

```
settings["theme"] = "dark"
```

A mutable dictionary makes sense.

---

# 37. Immutable Data in Real Applications

Imagine coordinates:

```
point = (10, 20)
```

If the point represents a fixed location, immutability can communicate that the values should not be modified.

Another example is configuration constants:

```
SUPPORTED_FORMATS = ("pdf", "png", "jpg")
```

A tuple can communicate that this collection is intended to remain fixed.

---

# 38. Practical Example: haas.dev Resources

Imagine a resource represented as:

```
resource = {
    "title": "Python Lists",
    "category": "Python",
    "level": "Beginner"
}
```

This is mutable.

If the resource category changes:

```
resource["category"] = "Programming"
```

the dictionary changes.

But suppose you have fixed website dimensions:

```
website_size = (1200, 800)
```

A tuple can represent those fixed values.

The choice depends on whether the data is expected to change.

---

# 39. Common Mistakes

## Mistake 1: Thinking `=` Copies a List

Wrong assumption:

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

`b` is not an independent copy.

Both variables refer to the same list.

Use:

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

when you need a separate shallow copy.

---

## Mistake 2: Thinking `replace()` Changes a String

```
name = "Hafsa"
```

```
name.replace("H", "S")
```

The result is returned, but `name` does not automatically change.

You need:

```
name = name.replace("H", "S")
```

---

### Mistake 3: Confusing Mutation With Reassignment

```
numbers.append(4)
```

is mutation.

```
numbers = [4, 5]
```

is reassignment.

They are different operations.

---

### Mistake 4: Assuming Every Collection Is Mutable

This is wrong:

```
List → mutable
Tuple → mutable
Set → mutable
Dictionary → mutable
String → mutable
```

Strings and tuples are immutable.

---

## 40. How to Check Whether Two Variables Refer to the Same Object

Python provides the `is` operator.

Example:

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

print(a is b)
```

Output:

```
True
```

They refer to the same object.

Now:

```
a = [1, 2, 3]
b = a.copy()

print(a is b)
```

Output:

```
False
```

They are separate list objects.

---

## 41. `is` vs `==`

Do not confuse:

```
is
```

with:

```
==
```

### `==`

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
```

The lists contain the same values, but they are different objects.

---

# 42. A Useful Mental Model

Remember:

```
Mutable
    ↓
Existing object can change

Immutable
    ↓
Existing object cannot change
    ↓
Create/use another object instead
```

Then remember:

```
=
assignment

==
value comparison

is
object identity
```

These three concepts should not be confused.

---

# 43. Mini Project: Safe Student Data

Suppose you have:

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

You want to create a backup that can be modified independently.

A simple assignment is not enough:

```
backup = student
```

Instead, because the dictionary contains a nested list, use:

```
from copy import deepcopy
```

```
backup = deepcopy(student)
```

Now:

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

does not change the original student's skills.

This is useful when working with:

- User profiles
- API data
- Configuration
- Application state
- Temporary backups
- Form data

---

# 44. 5 Minute Challenge

Predict the output before running this:

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

other = numbers

other.append(4)

print(numbers)
print(other)
```

### Answer

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

Why?

Both variables refer to the same mutable list.

---

## Challenge 2

Predict:

```
name = "Hafsa"

other = name
```

```
other = "Asim"

print(name)
print(other)
```

### Answer

```
Hafsa
Asim
```

Why?

Strings are immutable.

Reassigning `other` does not modify `name`.

---

# 45. Exercises

## Exercise 1

Identify whether each type is mutable or immutable:

```
list
tuple
string
dictionary
set
integer
```

---

## Exercise 2

Explain why this changes both variables:

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

b.append(3)
```

---

## Exercise 3

Fix this code so that changing `b` does not change `a`:

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

---

## Exercise 4

Explain why this does not change the original string:

```
name = "Hafsa"
name.upper()
```

---

### Exercise 5

What is the difference between:

```
a == b
```

and:

```
a is b
```

---

## 46. Mini Quiz

### Question 1

Which of these is mutable?

A. String

B. Tuple

C. List

D. Integer

**Answer:** C. List

---

### Question 2

What happens when two variables reference the same mutable list?

A. They always become independent

B. Changes through one reference can be visible through the other

C. Python creates a copy automatically

D. The list becomes immutable

**Answer:** B.

---

### Question 3

Which operator checks object identity?

A. `==`

B. `=`

C. `is`

D. `!=`

**Answer:** C. `is`

---

# 47. Interview Questions

### Beginner

1. What does mutable mean?
2. What does immutable mean?
3. Which Python data structures are mutable?
4. Which common Python types are immutable?
5. Are strings mutable?
6. Are lists mutable?
7. Are tuples mutable?

### Intermediate

8. What is the difference between mutation and reassignment?
9. Why does `b = a` not create a copy of a list?
10. What is the difference between `is` and `==`?
11. What is a shallow copy?
12. What is a deep copy?
13. Why are mutable default arguments dangerous?
14. Why can tuples sometimes be used as dictionary keys while lists cannot?

### Practical

15. How would you safely copy nested data?
16. What happens when a mutable object is passed to a function?
17. Why does modifying a list inside a function affect the original list?
18. Why does changing an integer inside a function not change the caller's integer?
19. How does immutability help prevent unexpected changes?

---

# 48. Cheat Sheet

```
MUTABLE
An object that can be changed after creation.

Examples:
list
dict
set
```

IMMUTABLE
An object that cannot be changed after creation.

Examples:
int
float
bool
str
tuple
frozenset


MUTATION
Changing an existing object.

numbers.append(4)


REASSIGNMENT
Making a variable refer to another object.

numbers = [10, 20]


COPY
Create a separate object.

new_list = old_list.copy()


DEEP COPY
Create independent copies of nested objects.

from copy import deepcopy
new_data = deepcopy(old_data)


==
Checks value equality.

```
is
Checks object identity.
```

---

# 49. Key Takeaways

- **Mutable** objects can be changed after creation.
- **Immutable** objects cannot be changed after creation.
- Lists, sets, and dictionaries are mutable.
- Strings, numbers, and tuples are immutable.
- Variables hold references to objects.
- `b = a` does not automatically copy a mutable object.
- Mutation changes an existing object.
- Reassignment makes a variable refer to another object.
- `.copy()` creates a shallow copy.
- `deepcopy()` creates independent nested copies.
- `==` compares values.
- `is` checks whether two variables refer to the same object.
- Understanding mutability is especially important when working with functions and nested data.
- Mutable default arguments can cause unexpected behavior.
- Immutable objects can be useful when data should remain stable.

> **The key question is not "Can I change this variable?" but "Can the object this variable refers to be changed?"**

Visit haas.dev for more resources and guides.

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

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