Python Objects, References, Namespaces, and Mutability

Everything in Python is an object. Objects store data (which may include data structures) and methods — special functions that belong specifically to that object. Variable names don't hold the objects themselves; they hold references (pointers) to those objects in memory.

Scope in Python refers to the region of a program where a name (and therefore its referenced object) is accessible. Python defines four levels of scope, remembered by the acronym LEGB:

1. Local – Names defined inside a function.
2. Enclosing – Names in any outer (enclosing) function when functions are nested.
3. Global – Names defined at the top level of a Python file (outside any function or class).
4. Built-in – Names always available, such as len(), print(), and sum().

Each inner scope has access to the scopes above it. In other words, everything defined in a higher scope is visible to the scopes below it.

Whenever you assign an object to a variable in Python, you are actually assigning a reference to that object — not the object itself. Variables hold references to objects, and these relationships are stored in a namespace, a special dictionary where the keys are variable names and the values are the references. Each scope level has its own namespace.

Mutable Data Structures

A mutable data structure allows you to change its contents without changing its reference (memory address).

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

myList --> List Object (x0012)
              |
              +--> x0045 (int 1)
              +--> x0098 (int 2)
              +--> x0067 (int 3)
```

The variable myList holds a reference (x0012) to the list object. Each element inside the list has its own memory address. You can add, remove, or modify elements without changing the list's reference.

At a low level, the list object at x0012 contains:
• a reference count – how many variables refer to this list
• a length field – how many elements it currently holds
• a pointer to an internal array – an array of pointers to the actual element objects

Python lists don’t store the objects themselves — they store references (pointers) to those objects.

When you mutate myList, the reference (x0012) stays the same, but the list updates its internal array and length field.

Immutable Example — tuple

```
myTuple = (1, 2, 3)
myTuple = myTuple + (4,)

myTuple -->  x0025  -->  ( 1 , 2 , 3 )
                   ↓ (concatenate)
myTuple -->  x0079  -->  ( 1 , 2 , 3 , 4 )
```

The reference changes because tuples cannot be modified in place. Python creates a new object each time you change its contents.

Why Scalars Must Be Immutable

Python reuses scalar objects (like ints, floats, and strings) internally to save memory. If scalars were mutable, one variable could accidentally change another that shares the same reference.

```
a = 5
b = 5

a --> 5 (x1001)
b --> 5 (x1001)
```

Immutability ensures shared objects cannot be modified. If you 'change' a scalar, Python creates a new object instead.

Property	Scalars (int, float, bool, str)
Shared between variables	Yes
Mutable in place	No
Why not?	To prevent shared references from causing side effects
When changed	A new object is created and reference updated

Mutables vs Immutables

Scalars are shared automatically, so they must be immutable to stay safe. Lists and other containers are mutable but are only shared when explicitly assigned, which makes that sharing intentional and controllable.

Type	Shared by default?	Mutable?	Safe?	Why
Scalars (int, float, bool, str)	Yes (internally shared)	No	Safe	Immutability prevents side effects
Lists / dicts / sets	When explicitly assigned	Yes	Careful	You manage shared references

To avoid unwanted sharing, use copying:

```
b = a.copy()           # shallow copy
import copy
b = copy.deepcopy(a)  # deep copy
```

Now each variable points to a separate object with independent contents.