Python Variables and Memory: When is `a` the Same as `b`?

A common question in Python is: if we assign the same value to two different variables, do they point to the same object in memory?

```
a = 500
b = 500

# Do 'a' and 'b' share the same memory location?
```

The short answer is: **it depends on the object's type (immutable vs. mutable) and Python's internal optimizations.**

Let's break this down.

1. The Fundamental Concept: Variables are Labels for Objects

First, it's crucial to understand that variables in Python are not boxes that contain data. Instead, they are **labels** or **names** that **point to objects** in memory.

An object is a region of memory that has a value, a type, and a unique ID.

We can inspect the unique ID of any object using the built-in `id()` function. This ID is essentially its memory address.

```
x = "hello"
# The variable 'x' is a name that points to a string object whose value is "hello".
print(id(x)) # e.g., 140163319807344
```

2. The `is` Operator vs. the `==` Operator

To understand this topic, you must know the difference between `is` and `==`:

- `==` (Equality): Checks if the **values** of two objects are the same.
- `is` (Identity): Checks if two variables point to the **exact same object** in memory (i.e., `id(a) == id(b)`).

```
list_a = [1, 2, 3]
list_b = [1, 2, 3]

print(list_a == list_b) # True, because their contents are equal.
print(list_a is list_b) # False, because they are two separate list objects in memory.
print(f"ID of list_a: {id(list_a)}") # e.g., 2324138753088
print(f"ID of list_b: {id(list_b)}") # e.g., 2324138754112 (a different ID)
```

Now, let's see how different data types behave.

3. Case 1: Immutable Types (The "Often Yes" Case)

Immutable objects are those that cannot be changed after creation. This includes `int`, `float`, `str`, `tuple`, `bool`, and `frozenset`.

Because they can't be changed, Python can perform an optimization called **interning**: if you create a new immutable object with the same value as an existing one, Python can just point the new variable to the existing object to save memory.

Integers

Python pre-allocates and caches all integers from **-5 to 256**. Any variable assigned a value in this range will point to the same object.

```
# Integers in the range [-5, 256] are interned
a = 100
b = 100
print(f"a = {a}, b = {b}")
print(f"a is b: {a is b}") # True! They point to the same object.
print(f"id(a): {id(a)}, id(b): {id(b)}") # Same ID

# Integers outside this range are usually NOT interned
x = 500
y = 500
print(f"\nx = {x}, y = {y}")
print(f"x is y: {x is y}") # False! Two different objects are created.
print(f"id(x): {id(x)}, id(y): {id(y)}") # Different IDs
```

Strings

Python also interns some strings. This is an implementation detail and not a strict guarantee, but it typically applies to:

- Short strings.
- Strings that look like identifiers (e.g., variable names).
- Strings created at compile time.

```
# Short, simple strings are often interned
s1 = "hello"
s2 = "hello"
print(f"s1 is s2: {s1 is s2}") # True

# Strings created dynamically or that are more complex might not be
s3 = "a-very-long-and-complex-string-that-is-unlikely-to-be-reused"
s4 = "a-very-long-and-complex-string-that-is-unlikely-to-be-reused"
print(f"s3 is s4: {s3 is s4}") # Often False

# Strings constructed at runtime are usually not interned
s5 = "".join(['h', 'e', 'l', 'l', 'o'])
s6 = "hello"
print(f"s5 is s6: {s5 is s6}") # False
print(f"s5 == s6: {s5 == s6}") # True
```

`None`, `True`, and `False` (Singletons)

There is only **one** `None` object, one `True` object, and one `False` object in a Python program. They are singletons. Any variable assigned to them will always point to the

same object.

```
a = None
b = None
print(f"a is b: {a is b}") # Always True

x = True
y = True
print(f"x is y: {x is y}") # Always True
```

4. Case 2: Mutable Types (The "Almost Always No" Case)

Mutable objects are those that can be changed after creation. This includes `list`, `dict`, and `set`.

For mutable types, Python will **almost always create a new object in memory** each time you define one, even if it has the same content as another.

This is essential for correctness. If `a = [1, 2]` and `b = [1, 2]` pointed to the same object, modifying `b` (e.g., `b.append(3)`) would also modify `a`, which would be confusing and lead to bugs.

```
# Lists
list1 = [1, 2, 3]
list2 = [1, 2, 3]

print(f"list1 == list2: {list1 == list2}") # True (values are the same)
print(f"list1 is list2: {list1 is list2}") # False (different objects in memory)
print(f"id(list1): {id(list1)}, id(list2): {id(list2)}") # Different IDs

# Dictionaries
dict1 = {'a': 1}
dict2 = {'a': 1}

print(f"\ndict1 == dict2: {dict1 == dict2}") # True
print(f"dict1 is dict2: {dict1 is dict2}") # False
```

The one exception is when you explicitly assign one variable to another. In this case, you are just creating a new label pointing to the *same* object.

```
list_x = [10, 20]
list_y = list_x # list_y is now another name for the same list object

print(f"list_x is list_y: {list_x is list_y}") # True!

# If you modify one, the other changes too
list_y.append(30)
print(f"list_x: {list_x}") # Output: [10, 20, 30]
print(f"list_y: {list_y}") # Output: [10, 20, 30]
```

Summary Table

Data Type	Example	a is	Reason

		b?	
Small `int`	`a=10, b=10`	**True**	**Integer Interning** (for -5 to 256) for performance.
Large `int`	`a=500, b=500`	**False**	Outside the cached range; new objects are created.
Short `str`	`a="hi", b="hi"`	**True**	**String Interning** for performance.
Long/Complex `str`	`a="...", b="..."`	**False**	Interning is not guaranteed and often skipped for complex strings.
`bool` / `None`	`a=True, b=True`	**True**	They are **singletons**; only one instance exists.
`list`	`a=[], b=[]`	**False**	Mutable; a new object is needed to prevent side effects.
`dict`	`a={}, b={}`	**False**	Mutable; a new object is needed.
`set`	`a={1}, b={1}`	**False**	Mutable; a new object is needed.
`tuple`	`a=(), b=()`	**True**	An empty tuple is a singleton. For non-empty, it's an optimization and may be `True`.

Conclusion: Why Does It Matter?

1. **Performance & Memory:** Interning immutable types saves memory and makes comparisons faster.
2. **Correctness:** Understanding the `is` vs. `==` distinction is vital for avoiding bugs, especially when working with mutable data or singletons like `None`.
3. **Rule of Thumb:**
   - Use `==` to compare the **values** of objects. This is what you want 99% of the time.
   - Use `is` only when you specifically need to check if two variables refer to the **exact same object in memory**. The most common and safe use case for `is` is checking for singletons: `if my_var is None:`.