Lesson 1B Reference: References vs Copies

📋 Quick Review: Lesson 1A Methods

- **append(item)** — Add to end
- **insert(index, item)** — Add at position
- **remove(value)** — Delete by value
- **pop()** — Remove and return last item

All these methods **MODIFY** the original list.

🧠 Two Types of Data in Python

Immutable (Cannot be changed after creation)

- Strings: `"hello"`
- Numbers: `42`
- Booleans: `True`

Immutable: Once created, the value is "locked" forever

Mutable (CAN be changed after creation)

- Lists: `[1, 2, 3]`
- Dictionaries: `{"key": "value"}`

Mutable: The value can be modified in place

🔍 Immutable Example: Strings

```
name = "hello"
name.upper()
print(name)
# Output: hello
```

Strings can't change! `.upper()` creates a NEW string.

To change a string:

```
name = "hello"
name = name.upper()   # Must reassign!
print(name)
# Output: HELLO
```

With immutable types, you must capture the new value!

📋 Mutable Example: Lists

```
nums = [1, 2, 3]
nums.append(4)
print(nums)
# Output: [1, 2, 3, 4]
```

Lists CAN change! `.append()` modifies the original. No reassignment needed!

🚙 Think of a Parking Garage

Imagine memory is a **parking garage**:

- **The List** = The actual car parked inside
- **The Variable** = A key card with the spot number

Multiple key cards can point to the SAME spot!

🔗 What is a Reference?

```
original = [1, 2, 3]
backup = original
```

This does NOT copy the list! It gives `backup` the same "spot number" as `original`.

Both variables point to the SAME list in memory!

Visualizing References

```
original = [1, 2, 3]
backup = original
```

In Memory:

```
original ──────┐
               ▼
          [1, 2, 3]    ← Only ONE list exists!
               ▲
backup ────────┘
```

😱 The Strange Behavior

```
original = [1, 2, 3]
backup = original

backup.append(99)

print(original)
# Output: [1, 2, 3, 99]
```

We changed `backup`... but `original` changed too!

Why? Both variables point to the SAME list.

That's Why Both Changed!

```
original = [1, 2, 3]
backup = original

backup.append(99)    # Modifies THE list
```

In Memory:

```
original ──────┐
               ▼
          [1, 2, 3, 99]    ← Same list, modified!
               ▲
backup ────────┘
```

✅ How Do We Check?

```
original = [1, 2, 3]
backup = original

print(original is backup)
# Output: True
```

The `is` keyword checks if two variables point to the SAME object

---

📋 Making Real Copies

Method 1: .copy()

```
original = [1, 2, 3]
backup = original.copy()

backup.append(99)

print(original)  # [1, 2, 3]
print(backup)    # [1, 2, 3, 99]
```

In Memory:

```
original ──▶ [1, 2, 3]

backup   ──▶ [1, 2, 3]    ← TWO separate lists!
```

Now they're independent!

Method 2: Slice [:]

```
original = [1, 2, 3]
backup = original[:]    # Empty slice = whole list

backup.append(99)
print(original)  # [1, 2, 3]
```

`[:]` creates a copy of the entire list

Method 3: list()

```
original = [1, 2, 3]
backup = list(original)

backup.append(99)
print(original)  # [1, 2, 3]
```

`list()` creates a new list from the original

📋 Quick Reference: 3 Ways to Copy

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

# All three create independent copies:
copy1 = original.copy()   # Most readable
copy2 = original[:]       # Most common
copy3 = list(original)    # Most explicit
```

Pick whichever you find most readable!

⚡ Functions & References

Passing a List to a Function

```
def add_item(items):
    items.append(99)

my_list = [1, 2, 3]
add_item(my_list)

print(my_list)
# Output: [1, 2, 3, 99]
```

The list gets modified! Functions receive a REFERENCE to the original list!

Why? Same Reference!

```
def add_item(items):    # items = reference to my_list
    items.append(99)    # modifies THE list
```

In Memory:

```
my_list ──────┐
              ▼
          [1, 2, 3, 99]
              ▲
items ────────┘ (inside function)
```

⬅ END Compare: Strings in Functions

```python
def shout(text):
    text = text.upper()

my_text = "hello"
shout(my_text)

print(my_text)
# Output: hello
```

String unchanged! Immutable types can't be modified.

⬅ END The Pattern

Lists (Mutable)

```python
def process(items):
    items.append(1)
    # No return needed!
```

- Function modifies original
- Changes persist after call

Strings (Immutable)

```
def process(text):
    text = text.upper()
    return text  # MUST return!
```

- Function can't modify original
- Must return new value

🛡️ Protecting Your Lists

What if you DON'T want the function to modify your list?

```
def safe_process(items):
    working_copy = items.copy()  # Make a copy first!
    working_copy.append(99)
    return working_copy
```

Copy inside the function to protect the original!

Safe Function Example

```
def safe_process(items):
    working_copy = items.copy()
    working_copy.append(99)
    return working_copy

my_list = [1, 2, 3]
result = safe_process(my_list)

print(my_list)   # [1, 2, 3] - Safe!
print(result)    # [1, 2, 3, 99]
```

Original protected! Function returns the modified copy.

⚠️ A Sneaky Gotcha

```
def add_to_list(items, value):
    items = items + [value]
    return items

my_list = [1, 2, 3]
add_to_list(my_list, 99)

print(my_list)
# Output: [1, 2, 3]
```

The + Operator Creates NEW Lists!

```
def add_to_list(items, value):
    items = items + [value]   # Creates NEW list!
    return items

my_list = [1, 2, 3]
add_to_list(my_list, 99)      # Return value ignored!

print(my_list)
# Output: [1, 2, 3]
```

The `+` operator ALWAYS creates a new list! So `items` gets reassigned to a new list, but `my_list` still points to the old one.

✅ Two Solutions

Modify in place

```
def add_to_list(items, val):
    items.append(val)
    # No return needed
```

Return new list

```
def add_to_list(items, val):
    return items + [val]

# Must capture!
my_list = add_to_list(my_list, 99)
```

Choose one approach and be consistent!

🎯 Quick Check Example

```
a = [1, 2, 3]
b = a           # Reference (same list as a)
c = a.copy()    # Copy (different list)

b.append(4)     # Modifies a AND b's list
c.append(5)     # Only modifies c's list

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

`a` and `b` share a list. `c` is independent!

📚 Summary: What You Learned

Core Concepts

1. 🧠 **Mutable types (lists) can be changed; immutable (strings) cannot**
2. 🔗 **Assignment (=) creates a reference, not a copy**
3. 📋 **Use .copy(), [:], or list() to make real copies**
4. ⚡ **Functions receive references to lists - they can modify originals!**
5. 🛡️ **Copy inside functions to protect original data**
6. ⚠️ **The + operator creates new lists, doesn't modify**

🎯 The Big Question

"Will this modify the original, or create something new?"

Ask yourself this every time you work with lists!

This prevents 90% of beginner bugs!

📖 Key Methods & Operators Summary

Methods that MODIFY the original list:

- `list.append(item)`
- `list.insert(index, item)`
- `list.remove(value)`
- `list.pop()`
- `list.extend(other_list)`
- `list.sort()`
- `list.reverse()`

Operations that CREATE NEW lists:

- `list1 + list2`
- `list * n`
- `list[:]` (slice)
- `list.copy()`
- `list(original)`
- `sorted(list)`

Checking identity vs equality:

- `list1 is list2` — Same object in memory?
- `list1 == list2` — Same contents?

💡 Common Patterns

Pattern 1: Modify Original

```
def process(items):
    items.append(value)
    # No return needed

my_list = [1, 2, 3]
process(my_list)  # my_list is now modified
```

Pattern 2: Return New List

```
def process(items):
    return items + [value]

my_list = [1, 2, 3]
my_list = process(my_list)  # Must reassign
```

Pattern 3: Protect Original

```
def process(items):
    copy = items[:]
    copy.append(value)
    return copy

my_list = [1, 2, 3]
result = process(my_list)  # my_list unchanged
```

🔍 Debugging Tips

When your list unexpectedly changes:

1. Check if you used `=` instead of `.copy()`
2. Check if a function modified it (functions get references!)

3. Check if multiple variables point to the same list

Use `is` to debug:

```
a = [1, 2, 3]
b = a
print(a is b)  # True means they share the same list!
```

Use `id()` to see memory addresses:

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

print(id(a))  # 140234567890
print(id(b))  # 140234567890 (same as a!)
print(id(c))  # 140234598765 (different!)
```