Shallow Copy vs Deep Copy in Python

Shallow Copy vs Deep Copy in Python: Comprehensive Guide

Introduction

When working with Python objects, especially mutable ones like lists and dictionaries, understanding how copying works is crucial. Python provides two types of copying mechanisms: **shallow copy** and **deep copy**. The distinction between them becomes critical when dealing with nested or complex data structures.

1. Understanding Object References in Python

Before diving into copying, let's understand how Python handles objects:

```python
# Example 1: Basic reference behavior
original_list = [1, 2, 3]
reference = original_list

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

Explanation: Here, `reference` is not a copy—it's just another name pointing to the same object in memory. When we modify `reference`, we're actually modifying `original_list` because they both point to the same memory location.

2. Shallow Copy

A **shallow copy** creates a new object, but it doesn't create copies of nested objects. Instead, it copies references to the nested objects.

2.1 Creating Shallow Copies

There are multiple ways to create shallow copies:

```python
import copy

# Method 1: Using the copy module
original = [1, 2, 3]
shallow = copy.copy(original)

# Method 2: Using list() constructor
shallow2 = list(original)

# Method 3: Using slice notation
shallow3 = original[:]

# Method 4: Using the .copy() method
shallow4 = original.copy()
```

2.2 Shallow Copy with Simple Objects

```python
# Example 2: Shallow copy with simple list
original = [1, 2, 3, 4]
shallow = original.copy()

shallow.append(5)

print("Original:", original)  # Output: [1, 2, 3, 4]
print("Shallow:", shallow)    # Output: [1, 2, 3, 4, 5]
```

Explanation: With simple, non-nested structures, a shallow copy behaves like a completely independent copy. Changes to one don't affect the other because integers are immutable objects.

2.3 The Problem: Shallow Copy with Nested Objects

```python
# Example 3: Shallow copy with nested list
original = [[1, 2, 3], [4, 5, 6]]
shallow = original.copy()

# Modify the nested list
shallow[0].append(4)

print("Original:", original)  # Output: [[1, 2, 3, 4], [4, 5, 6]]
print("Shallow:", shallow)    # Output: [[1, 2, 3, 4], [4, 5, 6]]
```

Explanation: This is where shallow copy shows its limitation! The outer list is copied, but the inner lists are not. Both `original[0]` and `shallow[0]` point to the same inner list in memory. When we

modify the inner list through `shallow`, the change appears in `original` too.

Visual Representation:

```
original → [reference_to_list1, reference_to_list2]
                    ↓                    ↓
shallow  → [reference_to_list1, reference_to_list2]
```

2.4 More Complex Examples

PYTHON

```python
# Example 4: Dictionary with nested lists
original_dict = {
    'name': 'Alice',
    'scores': [85, 90, 78],
    'metadata': {'age': 25}
}

shallow_dict = original_dict.copy()

# Modifying a simple value (string)
shallow_dict['name'] = 'Bob'
print("Original name:", original_dict['name'])  # Output: Alice
print("Shallow name:", shallow_dict['name'])    # Output: Bob

# Modifying nested list
shallow_dict['scores'].append(95)
print("Original scores:", original_dict['scores'])  # Output: [85, 90, 78, 95]
print("Shallow scores:", shallow_dict['scores'])    # Output: [85, 90, 78, 95]

# Modifying nested dictionary
shallow_dict['metadata']['age'] = 30
print("Original age:", original_dict['metadata']['age'])  # Output: 30
print("Shallow age:", shallow_dict['metadata']['age'])    # Output: 30
```

Explanation:

- Changing `'name'` (a string, which is immutable) doesn't affect the original because Python creates a new string object
- Modifying the nested `'scores'` list affects both because they share the same list reference
- Modifying the nested `'metadata'` dictionary affects both for the same reason

3. Deep Copy

A **deep copy** creates a new object and recursively creates copies of all nested objects. It creates a completely independent clone.

3.1 Creating Deep Copies

```python
import copy

original = [[1, 2, 3], [4, 5, 6]]
deep = copy.deepcopy(original)
```

3.2 Deep Copy in Action

```python
# Example 5: Deep copy with nested list
import copy

original = [[1, 2, 3], [4, 5, 6]]
deep = copy.deepcopy(original)

# Modify the nested list
deep[0].append(4)

print("Original:", original)  # Output: [[1, 2, 3], [4, 5, 6]]
print("Deep:", deep)          # Output: [[1, 2, 3, 4], [4, 5, 6]]
```

Explanation: Now the changes to the deep copy don't affect the original! The `deepcopy()` function created entirely new inner lists, so they're completely independent.

Visual Representation:

```
original → [reference_to_list1_original, reference_to_list2_original]

deep     → [reference_to_list1_copy, reference_to_list2_copy]
```

3.3 Complex Nested Structures

PYTHON

```python
# Example 6: Deeply nested structure
original = {
    'users': [
        {'name': 'Alice', 'friends': ['Bob', 'Charlie']},
        {'name': 'David', 'friends': ['Eve', 'Frank']}
    ],
    'settings': {
        'theme': 'dark',
        'notifications': {'email': True, 'push': False}
    }
}

# Shallow copy
shallow = original.copy()
# Deep copy
deep = copy.deepcopy(original)

# Modify deeply nested value
shallow['users'][0]['friends'].append('George')
deep['users'][0]['friends'].append('Hannah')

print("Original friends:", original['users'][0]['friends'])
# Output: ['Bob', 'Charlie', 'George']

print("Shallow friends:", shallow['users'][0]['friends'])
# Output: ['Bob', 'Charlie', 'George']

print("Deep friends:", deep['users'][0]['friends'])
# Output: ['Bob', 'Charlie', 'George', 'Hannah']
```

Explanation: The shallow copy shares the nested structures with the original, so modifications propagate. The deep copy is completely independent at all levels.

4. Special Cases and Edge Cases

4.1 Immutable Objects

```python
# Example 7: Shallow copy with tuples
original = (1, 2, [3, 4])
shallow = copy.copy(original)

# Tuples are immutable, but they can contain mutable objects
original[2].append(5)

print("Original:", original)  # Output: (1, 2, [3, 4, 5])
print("Shallow:", shallow)    # Output: (1, 2, [3, 4, 5])
```

Explanation: Even though tuples are immutable, they can contain mutable objects (like lists). A shallow copy of a tuple containing a list will share that list reference.

4.2 Custom Objects

```python
# Example 8: Copying custom objects
class Person:
    def __init__(self, name, friends):
        self.name = name
        self.friends = friends

original_person = Person("Alice", ["Bob", "Charlie"])

# Shallow copy
shallow_person = copy.copy(original_person)
shallow_person.friends.append("David")

print("Original friends:", original_person.friends)
# Output: ['Bob', 'Charlie', 'David']

# Deep copy
deep_person = copy.deepcopy(original_person)
deep_person.friends.append("Eve")

print("Original friends:", original_person.friends)
# Output: ['Bob', 'Charlie', 'David']
print("Deep friends:", deep_person.friends)
# Output: ['Bob', 'Charlie', 'David', 'Eve']
```

Explanation: Custom objects follow the same rules. Shallow copies share references to mutable attributes, while deep copies create independent clones.

4.3 Circular References

```python
# Example 9: Deep copy with circular references
original = [1, 2]
original.append(original)  # Circular reference!

deep = copy.deepcopy(original)
deep[0] = 999

print("Original[0]:", original[0])  # Output: 1
print("Deep[0]:", deep[0])          # Output: 999
```

Explanation: Python's `deepcopy()` is smart enough to handle circular references without infinite loops. It keeps track of objects it has already copied.

5. Performance Considerations

```python
# Example 10: Performance comparison
import copy
import time

# Create a large nested structure
large_structure = [[i for i in range(1000)] for j in range(1000)]

# Time shallow copy
start = time.time()
shallow = copy.copy(large_structure)
shallow_time = time.time() - start

# Time deep copy
start = time.time()
deep = copy.deepcopy(large_structure)
deep_time = time.time() - start

print(f"Shallow copy time: {shallow_time:.4f} seconds")
print(f"Deep copy time: {deep_time:.4f} seconds")
```

Explanation: Deep copy is significantly slower because it needs to recursively copy all nested objects. Use shallow copy when you know you won't modify nested objects, or when performance is critical.

7. Common Pitfalls and Best Practices

7.1 Pitfall: Assuming copy() is enough

PYTHON

```python
# Example 14: The most common mistake
data = {
    'users': [{'name': 'Alice'}, {'name': 'Bob'}]
}

# Wrong approach
backup = data.copy()  # Shallow copy
backup['users'][0]['name'] = 'Charlie'

print(data['users'][0]['name'])  # Charlie (oops!)

# Correct approach
backup = copy.deepcopy(data)
backup['users'][0]['name'] = 'Charlie'
print(data['users'][0]['name'])  # Alice (preserved!)
```

7.2 Best Practice: Know Your Data Structure

PYTHON

```python
# Example 15: Choose the right copy method
# For simple, flat structures
simple_list = [1, 2, 3, 4, 5]
copy1 = simple_list.copy()  # Shallow is fine and faster

# For nested structures you'll modify
nested_list = [[1, 2], [3, 4]]
copy2 = copy.deepcopy(nested_list)  # Deep copy needed

# For nested structures you won't modify (read-only)
read_only_nested = [[1, 2], [3, 4]]
copy3 = read_only_nested.copy()  # Shallow is fine and faster
```

8. Quick Reference Table

Scenario	Shallow Copy	Deep Copy
Simple list of numbers	✅ Works perfectly	✅ Works (but slower)
Nested lists	⚠️ Shares inner lists	✅ Independent copies
Dictionaries with nested objects	⚠️ Shares nested objects	✅ Independent copies
Read-only nested structures	✅ Fast and safe	✅ Safe (but slower)
Will modify nested objects	❌ Unsafe	✅ Required
Performance critical	✅ Faster	⚠️ Slower

Scenario	Shallow Copy	Deep Copy
Custom objects with mutable attributes	⚠ Shares attributes	✅ Independent copies

9. Memory Visualization

```python
# Example 16: Understanding memory references
import copy

original = [[1, 2], [3, 4]]
shallow = original.copy()
deep = copy.deepcopy(original)

print("Original inner list ID:", id(original[0]))
print("Shallow inner list ID:", id(shallow[0]))   # Same as original!
print("Deep inner list ID:", id(deep[0]))         # Different!
```

Explanation: The `id()` function shows that shallow copy shares the same inner list objects (same memory address), while deep copy creates new objects (different memory addresses).

Summary

Use Shallow Copy when:

- Working with flat (non-nested) data structures
- Nested objects won't be modified
- Performance is critical
- You want to share some data between copies intentionally

Use Deep Copy when:

- Working with nested data structures that will be modified
- You need complete independence between original and copy
- Creating backups or snapshots of complex state
- Preventing unexpected side effects in your code

The key insight is that **shallow copy creates a new container but shares the contents**, while **deep copy creates new containers all the way down**. Understanding this distinction will help you avoid subtle bugs and write more robust Python code.