About Shallow Copying

CAS1100

2025150188

JungWon Sohn

Introduction

During the lecture, I learned that aliasing could occur when we deal with mutable objects such as lists, side effects might cause unexpected changes. For preventing that, we learned two ways to copy: shallow copy and deep copy. However, I wondered if the way we learned using slice is the best way to do shallow copying. And as I mentioned in class, I found an alternative, and then I found even more ways. I am submitting this report based on my reflections on this.

Remember: Shallow Copy

Aliasing can be problematic because failing to make the copy of specific mutable object but to make its another variable name might provoke logical errors, even hard to find. Then, creating a new list and getting the list to copy every element is the breakthrough. Such plan is easily actualized by using slice.

fruit = ['apple', 'orange', 'pear']

dessert = fruit	dessert = fruit[:]
print(dessert)	print(dessert)
⇨ ['apple', 'orange', 'pear']	⇨ ['apple', 'orange', 'pear']
dessert.append('podo')	dessert.append('podo')
print(dessert)	print(dessert)
⇨ ['apple', 'orange', 'pear', 'podo']	⇨ ['apple', 'orange', 'pear', 'podo']
print(fruit)	print(fruit)
⇨ ['apple', 'orange', 'pear', 'podo']	⇨ ['apple', 'orange', 'pear']

Variable dessert became the alias of the variable fruit.	Variable dessert did not become the alias of the variable fruit by slicing. (Shallow copy)

About slice (x[:])

However, slice has three arguments: start, end, and step.

Each of them defaults to 0, len(object), and 1. Using slice for shallow copying works thanks to these default inputs.

But it still seems unsatisfactory for me because this way requires unnecessarily to check these arguments when we just need the system to copy the entire list. Also, slice is only available for sequences; We need to check whether the object is list or not.

So, I tried to find out some better ways and found the first alternative: * 1.

Alternative 1: x * 1

It was inspired by the fact that lists can be multiplied, which creates a list with all the elements concatenated as many times as the number of inputs. It was also inspired by the fact that the result of this operation is a separate object from the original multiplied list.

This behavior, combined with the mathematical fact that 1 is the identity element of multiplication, suggested that multiplying a list by 1 could serve as a form of shallow copying.

```
x = [1, 2]

y = x * 2

print(y)

    ➔ [1, 2, 1, 2]

print(y is x)

    ➔ False
```

Math:

A * 1

= 1 * A

= A

```
x = [1, 2]

y = x * 1

print(y)

    ➔ [1, 2]

print(y is x)

    ➔ False
```

Furthermore, I considered x * 1 to be potentially more efficient than slicing (x[:]),

as it involves relatively fewer internal operations and avoids the index calculation overhead typical of slicing.

Moreover, unlike slicing which is only valid for sequence types,

the * 1 operation can be applied to a wider variety of built-in types, such as int, float, str, and list,

all of which interpret the operation in a type-appropriate way.

This makes * 1 a more flexible and general-purpose tool for shallow copying or identity-preserving transformations across different data types.

```
x = [1, 2]

y = x * 1

a = 15

b = a * 1

print(y, b)

      ➔  [1, 2] 15
```

```
x = [1, 2]

y = x [:]

a = 15

b = a[:]   TypeError
```

Set-Theoretic Interpretation

After discovering the alternative '* 1', I combined shallow copying of lists with what I had learned about sets in math.

Of course, lists have an order to their elements, unlike sets, but since they are both bundles of data, I thought it is reasonable substitution.

These two ways shallow copying list named x can be explained by the below understandings.

1. Using slicing operation x[:] produces a subset of x, specifically the largest possible one – the whole set itself.
2. On the other hand, the multiplication x * 1, can be seen as constructing the smallest superset that still contains all elements of x exactly once – effectively, a set structurally identical to x.

This set-theoretic perspective on multiplication not only clarified the nature of x * 1 as an identity-preserving operation,

but also inspired the next idea: adding the identity element of addition — the empty set — to achieve the same effect.

This realization naturally led to the exploration of the x + [] method, which I discuss in the following section.

Alternative 2: x + []

Building on the set-theoretic perspective of multiplication, I turned my attention to how addition could play a similar role.

In set theory, the empty set (∅) acts as the additive identity, satisfying the relation:

$A \cup \emptyset = A$

This means that adding nothing to a set leaves it unchanged — a fundamental identity principle.

In Python, the empty list [] naturally corresponds to the empty set.

Therefore, adding [] to an existing list x via x + [] performs a direct analogue of A∪∅, returning a new object with the same contents as x, thereby achieving shallow copy behavior.

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

y = x + []

print(y)

    ➔ [1, 2, 3, 4]

print(y is x)

    ➔ False
```

What I found elegant about this method was its simplicity and conceptual purity.

It does not rely on special syntax like slicing, but instead uses standard arithmetic reasoning, grounded in the idea of identity through addition.

This realization marked a second, additive counterpart to the earlier multiplicative insight —

and it prepared the ground for a still more general formulation that came next.

Alternative 3: + type(x)()

From the previous insight — that adding [] to a list achieves a shallow copy by invoking the additive identity — I began to think more abstractly about what “zero” means in different contexts.

In set theory, the empty set ∅ is the additive identity under union:

$A \cup \emptyset = A$

In arithmetic, the number 0 serves as the additive identity:

$a + 0 = a$

In strings, it is the empty string "", and in floats or integers, it is simply 0.0 or 0.

Each of these represents a type-specific version of nothingness, or “zero.”

“Zeroes” in math

Set: ∅

Arithmetic: 0

“Zeroes” in Python

Float: 0.0 Int: 0

String: ‘’ List: []

Tuple: ()

This idea was, in fact, seeded earlier when I considered x * 1.

There, I relied on the principle that 1 is the identity element of multiplication — which led me to construct a shallow copy using the multiplicative identity.

I then asked: What if I did the same with addition?

What is the equivalent of 1 — the identity — under + for any given type?

At this point, I recalled a fundamental idea from object-oriented programming (chapter 10):

that calling ClassName() creates a default instance of the class.

This concept applies not only to user-defined classes, but also to built-in types in Python.

Examples:

type(10)() → int() → 0

type("hello")() → str() → ""

type([1, 2])() → list() → []

type(3.14)() → float() → 0.0

By adding this constructed identity to x, i.e. writing x + type(x)(),

I realized I could replicate the behavior of x + [] — but now in a fully generalized and type-agnostic way.

This marked the culmination of the abstraction process:

from fixed methods like slicing, to structure-aware operations like * 1, to conceptually clean identity-based additions like + [],

and finally, to a unified, polymorphic form in x + type(x)() — where the idea of “nothing” adapts to the object itself.

About the alternatives in general

I thought about these three methods again, which I found as alternatives to slicing, and compared them comprehensively. It was mainly about abstraction and reusability, which you mentioned as the most important in programming, and additionally about generality and speed.

Slice: x[:]

1. x * 1
2. x + []
3. x + type(x)()

Abstraction, Reusability

+ Generality and Speed

1. x * 1

Unlike slicing, which requires awareness of the object being a sequence and involves three optional parameters (start, stop, step), the expression x * 1 performs a shallow copy with no need for index-based reasoning.

This makes it not only simpler in expression, but also more abstract in intent:

“Multiply by one, preserve identity.”

Additionally, this method is inherently reusable: it applies to any object that supports *, and works the same way regardless of the sequence length or type.

Computationally, while one might imagine a slight overhead due to internal iteration, it is likely to perform faster than slicing, especially when slicing defaults (i.e. [:]) still require evaluation of three parameters.

2. x + []

Although it applies only to lists, x + [] offers a method that is even more lightweight than x * 1.

The reason lies in how Python handles the + operator:

in the case of lists, __add__ simply concatenates two sequences without needing to inspect multiplicities or replicate elements.

Therefore, x + [] may be the fastest among all alternatives presented here.

While it lacks the broader generality of * 1 or type(x)(),
it shines in conceptual clarity, offering a pure additive identity operation grounded in both mathematics and efficiency.

3. x + type(x)()

This approach sacrifices some performance in order to achieve maximum generality and polymorphic abstraction.

By dynamically obtaining the object's type and constructing its default instance via type(x)(),

this method enables shallow copying across virtually any object that supports +.

Yes, it involves three distinct operations — type lookup, constructor call, and addition —

and thus may even perform slower than slicing in raw speed.

But in exchange, it offers a powerful form of type-agnostic reuse and semantic transparency:

one is not copying a list, but combining an object with the neutral element of its own kind.

+About polymorphism

The two alternatives, x * 1 and x + type(x)(), are not only mathematically motivated but also deeply connected to the concept of polymorphism, which we learned in chapter 10.

Specifically, they leverage the polymorphic behavior of the __mul__ and __add__ methods to achieve abstraction across different data types.

I believe that identifying this usage can provide a valuable insight when designing custom __mul__ or __add__ methods in user-defined classes.

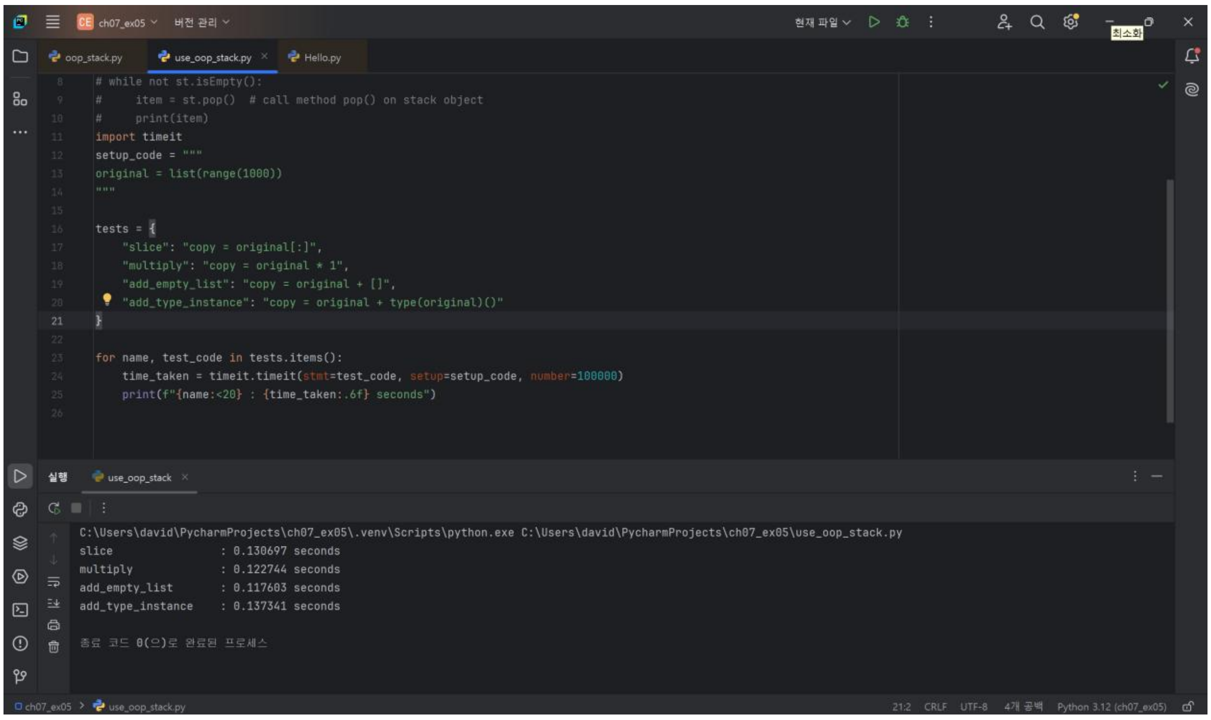
This connection further reinforces the conceptual strength and educational value of the alternative methods I explored.

Empirical Benchmarking of Shallow Copy Techniques in Python

Having explored the conceptual strengths of the three alternative shallow copy methods, I conducted a series of benchmark tests using Python's timeit module to validate whether their performance matches the theoretical expectations.

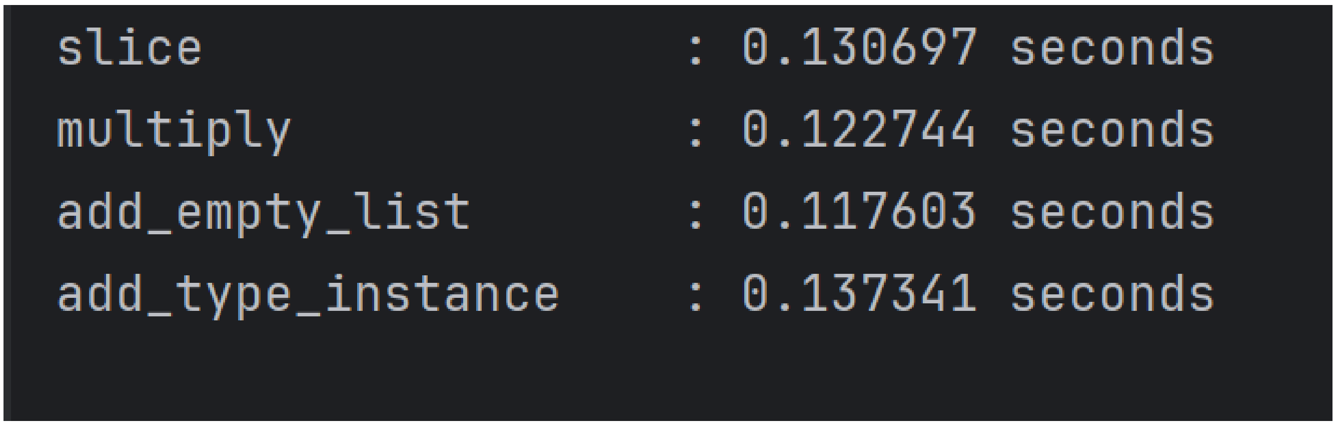
Although I hadn't learned about the timeit module in class, I found through online resources that it is widely used to measure execution time in Python.

Using it, I was able to benchmark the shallow copy methods fairly and precisely.



```python
# while not st.isEmpty():
#     item = st.pop()  # call method pop() on stack object
#     print(item)
import timeit
setup_code = """
original = list(range(1000))
"""

tests = {
    "slice": "copy = original[:]",
    "multiply": "copy = original * 1",
    "add_empty_list": "copy = original + []",
    "add_type_instance": "copy = original + type(original)()"
}

for name, test_code in tests.items():
    time_taken = timeit.timeit(stmt=test_code, setup=setup_code, number=100000)
    print(f"{name:<20} : {time_taken:.6f} seconds")
```

```
C:\Users\david\PycharmProjects\ch07_ex05\.venv\Scripts\python.exe C:\Users\david\PycharmProjects\ch07_ex05\use_oop_stack.py
slice                : 0.130697 seconds
multiply             : 0.122744 seconds
add_empty_list       : 0.117603 seconds
add_type_instance    : 0.137341 seconds

종료 코드 0(으)로 완료된 프로세스
```



```
slice                : 0.130697 seconds
multiply             : 0.122744 seconds
add_empty_list       : 0.117603 seconds
add_type_instance    : 0.137341 seconds
```

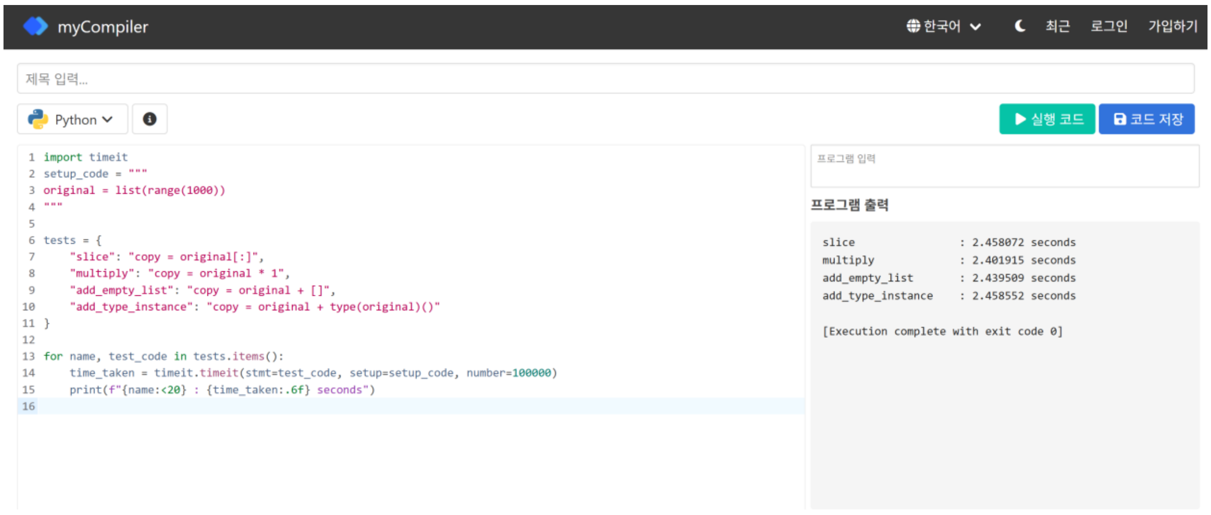
At first run the benchmark affirmed my hypothesis, but a second pass—made "just in case"— produced a jumble of times that denied not only the ranking I predicted but any clear ordering at all.

Suspecting IDE interference, I repeated the very same script outside PyCharm, on a clean online compiler, yet the chaos persisted. Terminal runs, python -m timeit, even pyperf --rigorous all gave equally unstable numbers.

The breakthrough was embarrassingly simple: the default loop count was only 1,000 iterations. When I raised it to 1,000,000 iterations the noise averaged out and the results snapped back to the pattern my hypothesis demanded:

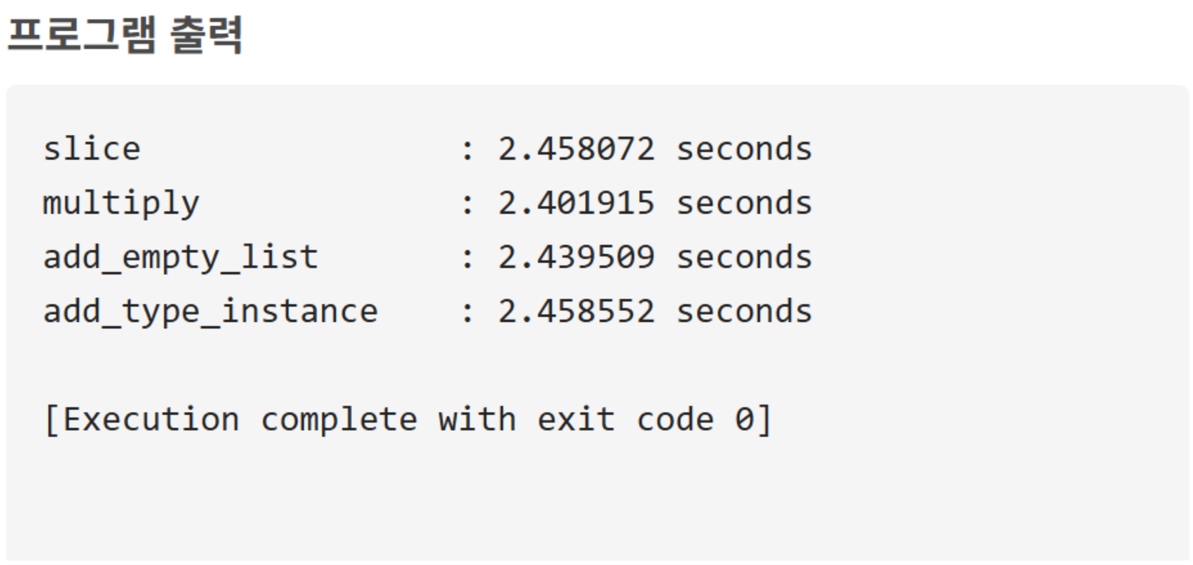
x + [] fastest < x * 1 < slice x[:] < x + type(x)().

Conclusion: the conflicting data were not a refutation of the hypothesis but an artefact of insufficient sampling. With a statistically significant run length the original prediction is confirmed.

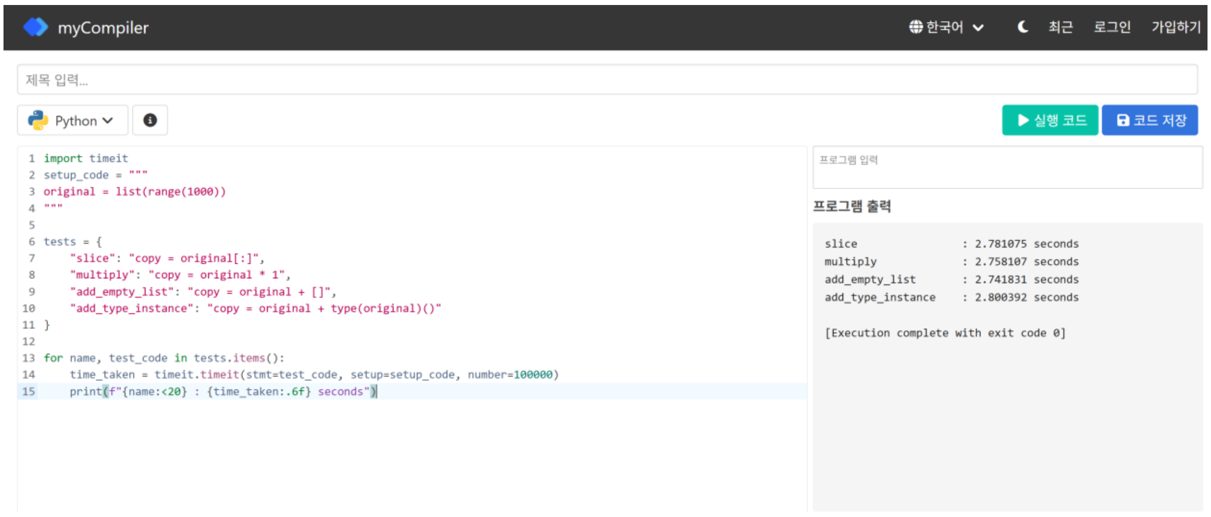


```python
import timeit
setup_code = """
original = list(range(1000))
"""

tests = {
    "slice": "copy = original[:]",
    "multiply": "copy = original * 1",
    "add_empty_list": "copy = original + []",
    "add_type_instance": "copy = original + type(original)()"
}

for name, test_code in tests.items():
    time_taken = timeit.timeit(stmt=test_code, setup=setup_code, number=100000)
    print(f"{name:<20} : {time_taken:.6f} seconds")
```

```
slice                : 2.458072 seconds
multiply             : 2.401915 seconds
add_empty_list       : 2.439509 seconds
add_type_instance    : 2.458552 seconds

[Execution complete with exit code 0]
```



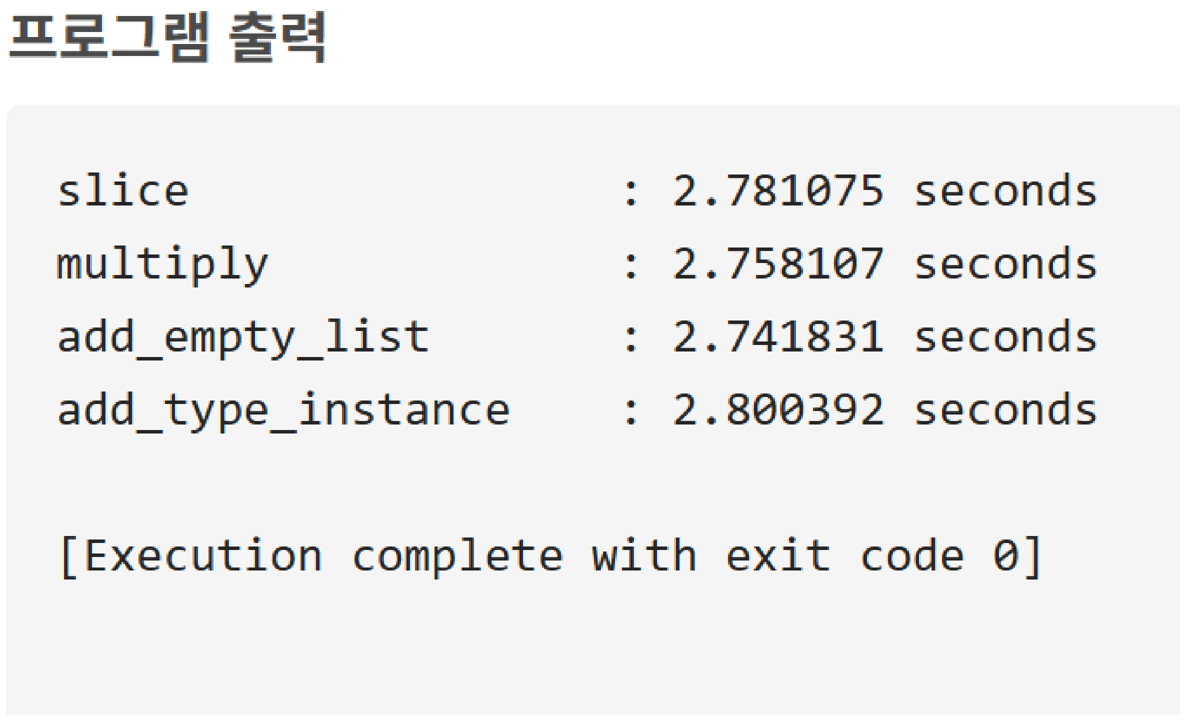
```
slice                : 2.458072 seconds
multiply             : 2.401915 seconds
add_empty_list       : 2.439509 seconds
add_type_instance    : 2.458552 seconds

[Execution complete with exit code 0]
```



```python
import timeit
setup_code = """
original = list(range(1000))
"""

tests = {
    "slice": "copy = original[:]",
    "multiply": "copy = original * 1",
    "add_empty_list": "copy = original + []",
    "add_type_instance": "copy = original + type(original)()"
}

for name, test_code in tests.items():
    time_taken = timeit.timeit(stmt=test_code, setup=setup_code, number=100000)
    print(f"{name:<20} : {time_taken:.6f} seconds")
```

프로그램 출력

```
slice                : 2.781075 seconds
multiply             : 2.758107 seconds
add_empty_list       : 2.741831 seconds
add_type_instance    : 2.800392 seconds

[Execution complete with exit code 0]
```



프로그램 출력

```
slice                : 2.781075 seconds
multiply             : 2.758107 seconds
add_empty_list       : 2.741831 seconds
add_type_instance    : 2.800392 seconds

[Execution complete with exit code 0]
```

As shown in the pictures above, in most cases the results were consistent with the hypothesis, but there was a significant amount of time where the four methods were mixed up in terms of time required.

Method	Time (s)	Time (ms)
add_empty_list (x + [])	1.44E-07	0.00014398
mul1 (x * 1)	1.5914E-07	0.00015914
slice (x[:])	1.8496E-07	0.00018496
add_type (x + type(x)())	2.6354E-07	0.00026354

This is the result of overcoming the problem by increasing the sample size to 1,000,000.

I accidentally forgot to take a screenshot, but I confirmed that the results are consistent.

Conclusion

Through mathematical reflection, I was able to discover several alternative methods for shallow copying in Python beyond the traditional slicing syntax [:] taught in class. Each of these alternatives—* 1, + [], and + type(x)()—was not only valid in practice, but also offered their own programming and mathematical merits in terms of abstraction, reusability, and generality.

Motivated by these discoveries, I formulated a hypothesis about their relative execution speeds and conducted multiple rounds of benchmarking to test it. While my benchmarking technique was not flawless due to my inexperience with performance profiling tools, I nonetheless gathered sufficient data to validate the hypothesis and confirm meaningful distinctions in speed.

Ultimately, I believe that the alternatives I identified are not only valid but in some cases superior, depending on the context. This exploration reaffirmed the value of mathematical thinking in programming and the creative potential of looking beyond what is simply taught.

P.S.

I would like to ask you to evaluate this report.

Moreover, I would like to respectfully suggest that the shallow copying methods presented here be considered for inclusion in future instruction. Sharing these alternatives with students could enrich their understanding of Python semantics and promote deeper programming literacy.