Stop using loops. Do this instead.

Take your code to the next level.

Shu Hasegawa Follow

Sep 6 · 7 min read ★

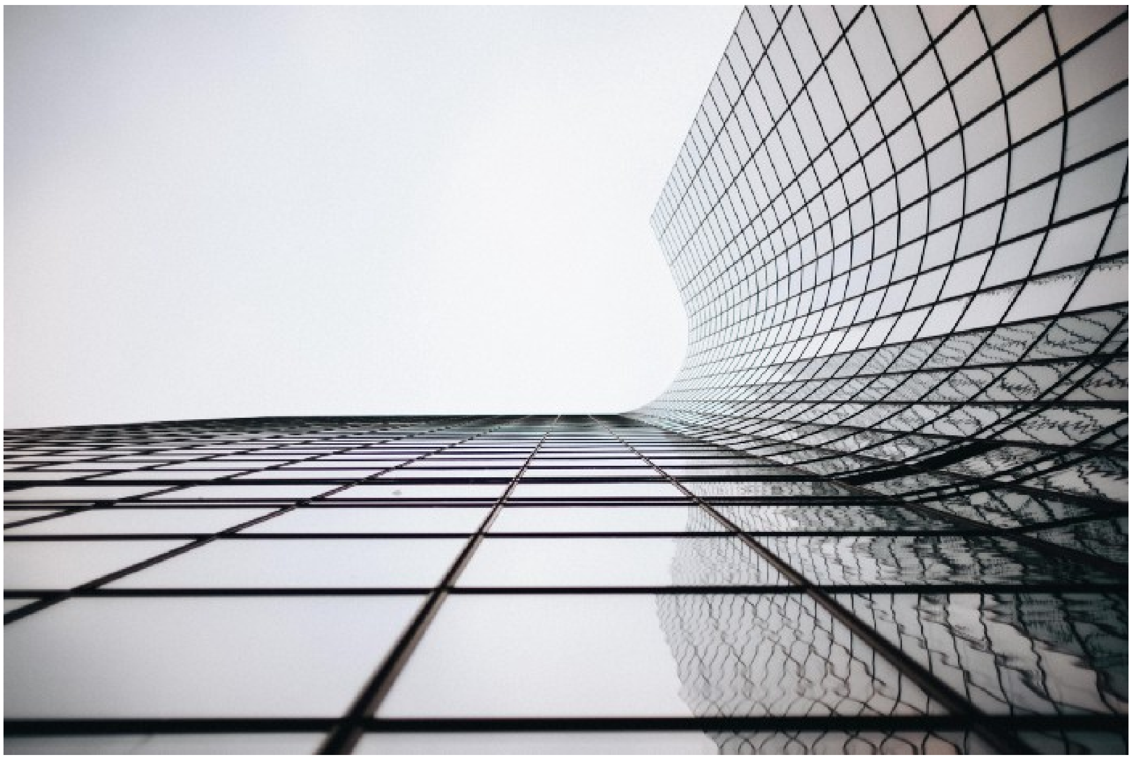


Photo by Etienne Boulanger on Unsplash

Years ago, when I was somewhat of an amateur programmer, I stumbled upon a programming problem I was unable to solve. The solution required processing the given data set through several complicated algorithms, and finally searching for and removing any duplicates in the final list.

Easy, right? Simply by using the `map()` function to process the data, and the `set()` function to remove any duplicates, I was finished. But when I proudly presented my solution to my mentor, he challenged me. I was told to solve the same problem, but without the use of both `set()` or `map()` . Oh, and I was also given a time limit for the amount of time my program can take to finish processing. My smile faded. My thoughts were blank. This was a problem.

```
nums = [1, 2, 2, 1, 3, 4, 4, 5, 5]
# casting the list nums into a set
unique_nums = set(nums)
```

The code above illustrates the usage of the `set()` function to remove duplicates. `set()` is a casting function that casts an object into a set, an iterable without duplicates. When this is done so with a list like `nums` , all the duplicates in the list are removed as it is cast into a set.

. . .

I found that the main difficulty of the problem was not writing the code, but the relatively short time given to my program to output the correct answer. `set()` and `map()` are both really fast functions, but without them, I had no idea how to make my program fast enough. My first attempt greatly exceeded the five-second limit.

Thus began my desperate attempts to accelerate my program. I implemented numerous built-in functions and even rewrote my entire solution on several occasions. Additionally, since my initial method of removing list duplicates was extremely slow, I decided on trying to speed that up first. I had an idea for reducing the number of comparisons the program has to make in order to find duplicates, which would subsequently reduce time.

Part of the given data set involved sets of integers, and I realized that for two sets of numbers to have identical values, they must at least have the same average. Therefore, by organizing all the sets with the same averages into different lists, and only comparing the values in those lists with each other, there would be a lot fewer comparisons. Let me quickly illustrate this.

Suppose there are 10 sets of numbers and that there are a total of 5 different averages the sets add up to, with every two sets having the same average. By organizing the sets we, therefore, have 5 different lists, and we compare the values in each list with each other. This results in a maximum of 10 comparisons. If we simply compare every value with the other, however, there can be up to 90. There would be a lot of repeated comparisons with the same sets as well as unnecessary ones. By organizing the sets into smaller groups, we are ruling out a lot of sets, which we know that is impossible to have the same numbers as the ones in another list because of varying averages.

I also found that the more specific the categories the sets are organized into, the fewer comparisons there are. For instance, using the same example above, if there are only 2 different averages instead, then we would have 2 lists, each containing 5 sets of values. This would result in a maximum of 40 comparisons.

Thus, I started organizing the sets not only in terms of their averages, but their medians and modes as well. What I did not realize was that the time it took to find the mean, median, and mode for every set of numbers actually ended up taking more time than the time it saved for the comparisons.

I know. I thought I was some sort of genius, but I just proved myself to be stupid. So I went back to my original method for a while.

. . .

However, in the end, I came up with a decent method where I repurposed the QuickSort algorithm to help me find duplicates faster.

My Original Method

```
# duplicate is a global variable used elsewhere in my code
for i in elements:
    # copying each element and removing it to avoid self comparison
    copy = i
    elements.remove(i)
    # checking if that item is repeated in the rest of the list
    if copy in elements:
        duplicate = "Duplicate found."
```

Short and simple, but very slow. Every element in the list is compared with every other element in the list, thus generating an overwhelming amount of comparisons.

Using QuickSort

```
def compare(sequence, length):
    global duplicate, elements_len

    # finding the pivot and preventing sort with single element list
    if length <= 1:
        return
    elif length == elements_len:
        pivot = next(sequence)
    else:
        pivot = sequence.pop()

    # declaring necessary variables for sorting
    # items greater and lower than the pivot
    items_greater, items_lower = [], []
    greater_append = items_greater.append
    lower_append = items_lower.append

    # sorting loop
    for item in sequence:
        # my key addition to the original QuickSort algorithm
        if item == pivot:
            duplicate = "Duplicate found."
            return
        elif item > pivot:
            greater_append(item)
        else:
            lower_append(item)

    # starting/continuing the recursion
    # the same sorting process is repeated with the two lists
    compare(items_lower, len(items_lower))
    compare(items_greater, len(items_lower))
```

The QuickSort algorithm drastically decreases the number of comparisons made. Yes, I know I used recursion, which can be rather slow. I never liked studying sorting algorithms, so QuickSort was pretty new to me and I found this particular implementation of it to be the simplest and cleanest, which is why I had only memorized this method. Don't follow my example if you're just starting out.

. . .

Ultimately, I was close, but I had utterly exhausted all my ideas. The remaining obstacles to my program's performance speed were its loops, and with the `map()` function restricted, they were impossible to replace.

This was when I discovered the concept of List Comprehension. It was comparable with the `map()` function, being nearly equivalent in speed and clarity. With this newfound technique, I replaced the loops within my program and finally solved the problem within the time limit.

With its numerous advantages, list comprehension is a concept all programmers should implement as often as possible. It had certainly assisted me in my studies and will prove of great value to any who uses it consistently and efficiently.

. . .

Basics

Here is everything I learned about list comprehension while solving that problem.

List comprehension consists of a short syntax and may be employed for defining lists and repeating code. It uses a simplified version of the `for` loop, the `for` statement, which is linked with an expression and an optional conditional statement. The entire line of code is encased within square brackets and may or may not be assigned to a variable (to define a list), depending on the use.

Should the purpose of using list comprehension be to assign values to elements in a list, in which the expression will be or return a value, an assignment would be necessary. If list comprehension is being used to conduct a certain action, in which the expression will not return any value, then an assignment is unnecessary.

Basic List Comprehension syntax

```
[expression for element in iterable]
```

Notice that the for statement has identical syntax to the definition of a `for` loop. The difference is that the expression, which would be repeatedly executed, is written before the definition.

Basic List Comprehension examples

```
# with assignment
numbers = [num for num in range(10)] # creates a list from 0-9

# without assignment
[print(num) for num in range(10)] # outputs all numbers from 0-9
```

In the first example, the expression, comprised only of the variable `num`, is a numerical value and will subsequently be assigned to an element to a list.

In the second example, the expression, a print statement, does not return a value and thus cannot be assigned to anything. It only repeatedly conducts the action of outputting `num`.

Conditions in List Comprehension

If Statements

List comprehension also allows the use of conditional statements in its syntax, which opens up the possibility of selecting specific values to work with in your code. The syntax of list comprehension with an `if` statement is as follows:

```
[expression for element in iterable if condition]
```

The conditional statement will go after the `for` statement, and the expression would be executed if the condition is true. Here is an example:

```
# print out all even numbers from 1-99
[print(num) for num in range(1, 100) if num % 2 == 0]
```

In the example, the program will iterate through the numbers 1 to 99, and for each number, check if it is even by seeing if it is divisible by 2 (hence the modulus, which returns the remainder of the expression). If the number is even, then the program will output it.

Else Statements

The case is different when an `else` statement is involved. Instead of the conditional statement going after the `for` statement, it will go before it, but after the expression.

```
[expression if condition else expression2 for element in iterable]
```

In summary, the `if` and `else` statement goes in between the expression and the `for` statement.

Example:

```
# print "e" for all even numbers, "o" for all odd numbers
[print("e") if n % 2 == 0 else print("o") for n in range(1, 100)]
```

Takeaway

Loops are commonly used in code by programmers to repeat a set of instructions and process large data sets. However, many fail to realize how taxing they are on a program's performance speed. Throughout my programming experience, I have learned that the usage of loops also severely complicates and extends code. What's worse is that even if a programmer recognizes the drawbacks of loops, they are often unaware of other alternatives. The key takeaway is that in spite of list comprehension being an underused programming concept, it is a great asset that will fundamentally enhance your code.

. . .

Thank you for reading. Follow and subscribe to Shu Hasegawa to get notified of new content that can help you get started on your programming career.

More programming articles:

14 Java Methods You Don't Know About

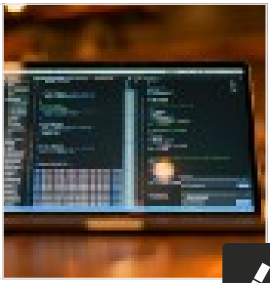
And how you can start using them

medium.com

Start Using These Python Functions Now

5 Python functions every programmer must know

medium.com



The Ultimate Guide To Python Functions

You may be familiar with using functions... what about writing them?

medium.com

More content at plainenglish.io

Python Programming Coding Technology Computer Science