Twitter Thread by Santiago



Santiago
@svpino

11 ways ChatGPT saves me hours of work every day, and why you'll never outcompete those who use AI effectively.

A list for those who write code:

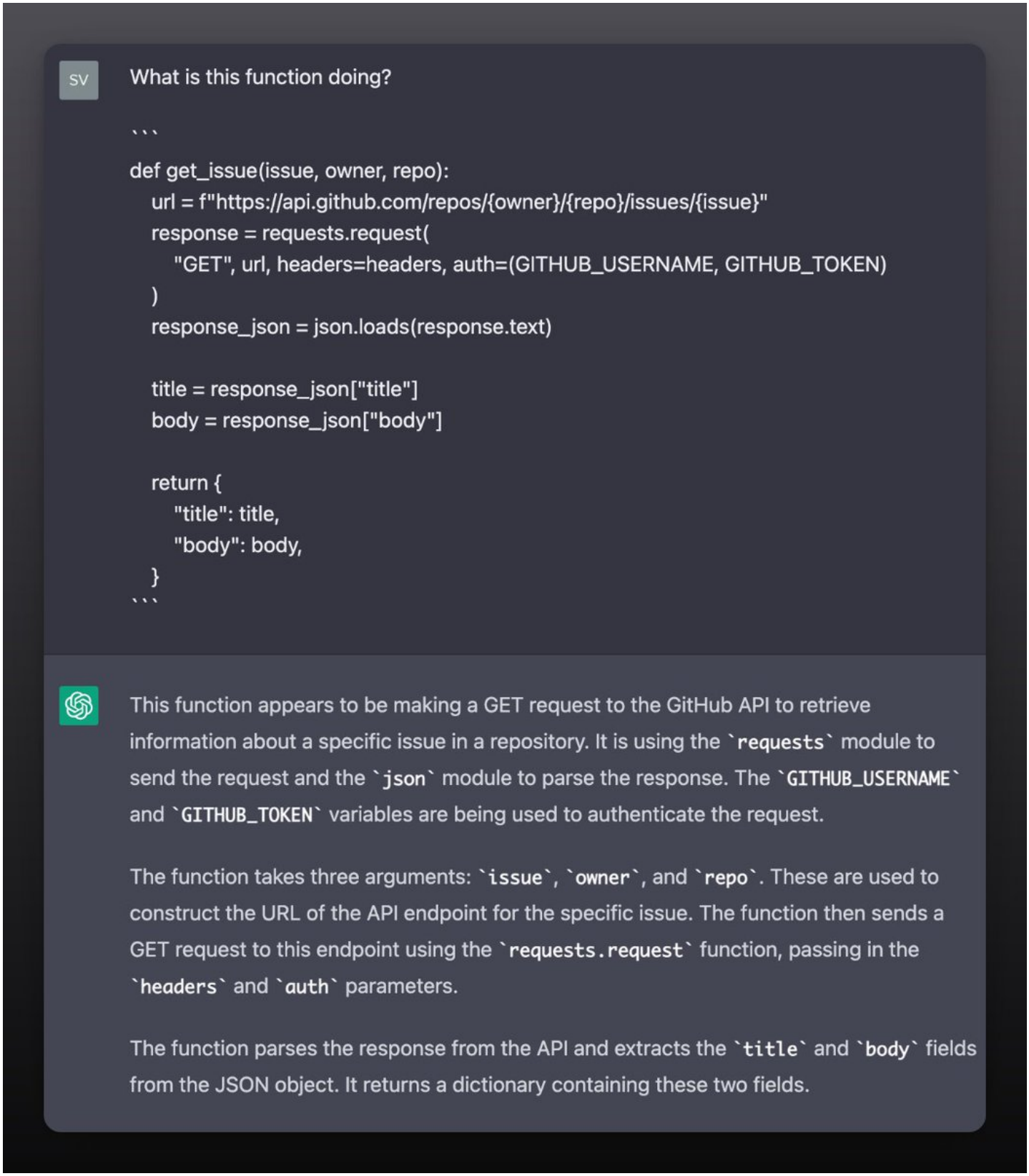
1 of 16

1. Explaining code

Take some code you want to understand and ask ChatGPT to explain it.

I've found explanations are very detailed. This is much quicker than trying to figure out convoluted code.

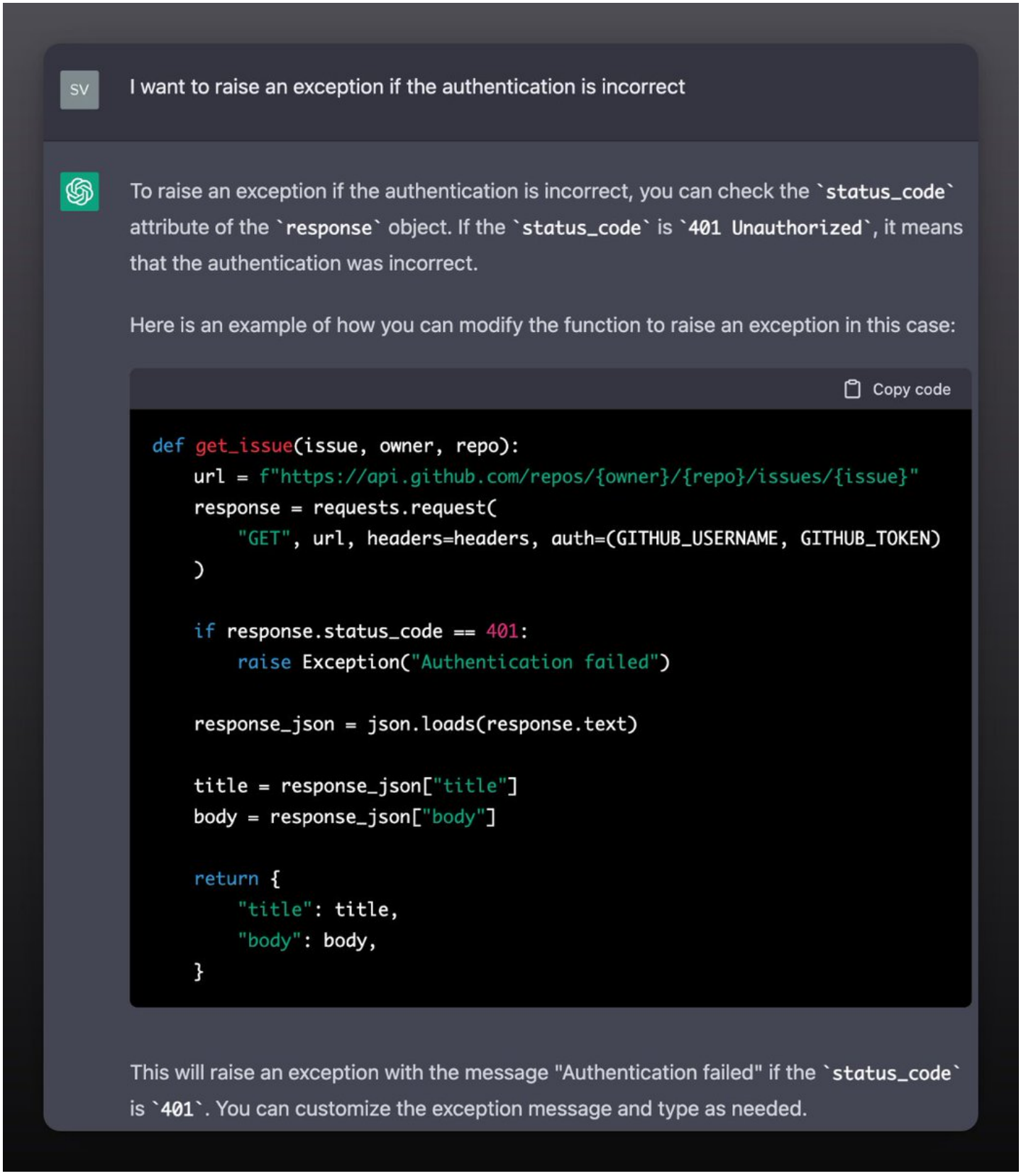
2 of 16



2. Improve existing code

Ask ChatGPT to improve existing code by describing what you want to accomplish.

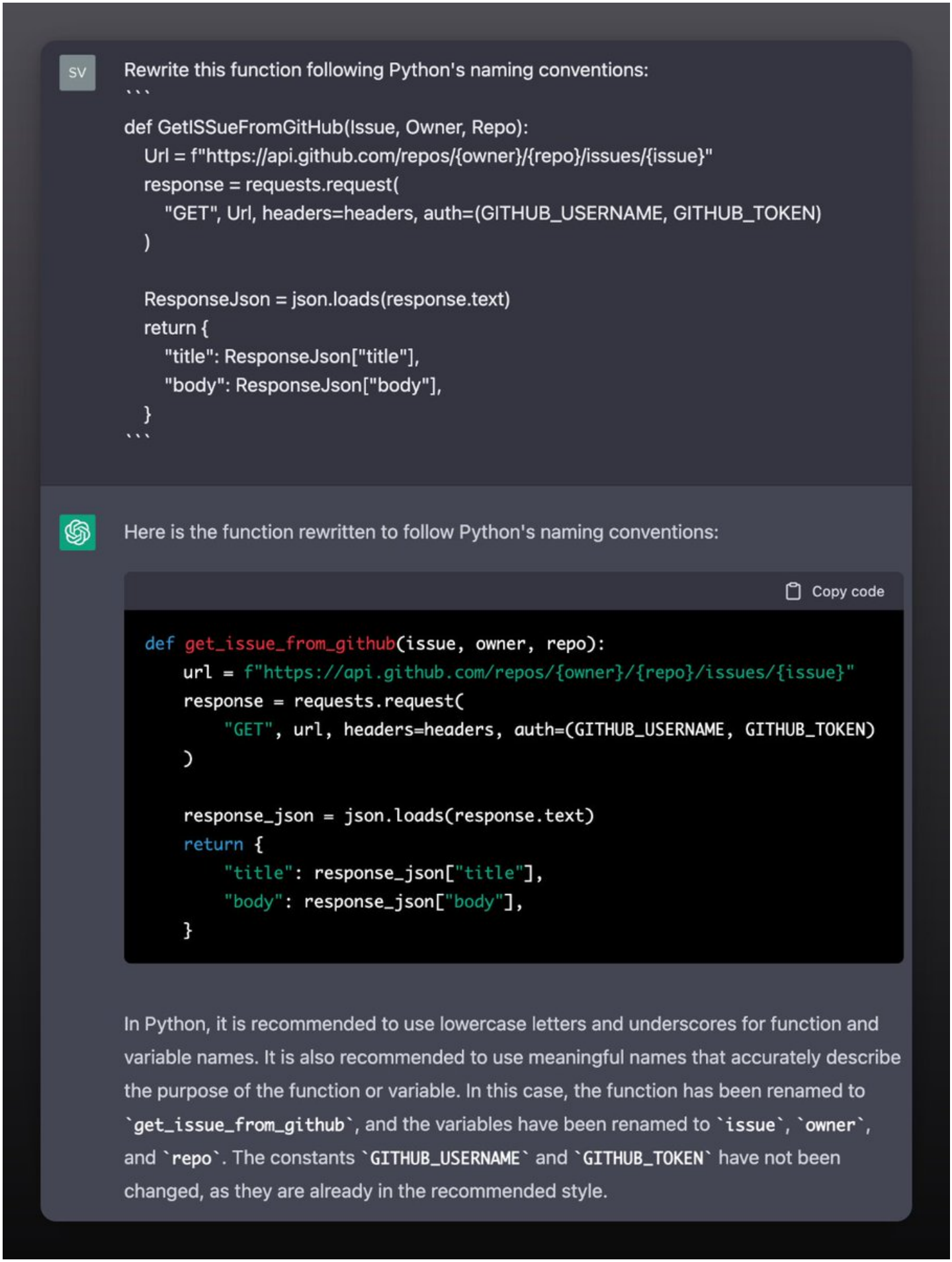
It will give you instructions about how to do it, including the modified code.



3. Rewriting code using the correct style

This is great when refactoring code written by non-native Python developers who used a different naming convention.

Notice how ChatGPT not only gives you the updated code; it also explains the reason for the changes.

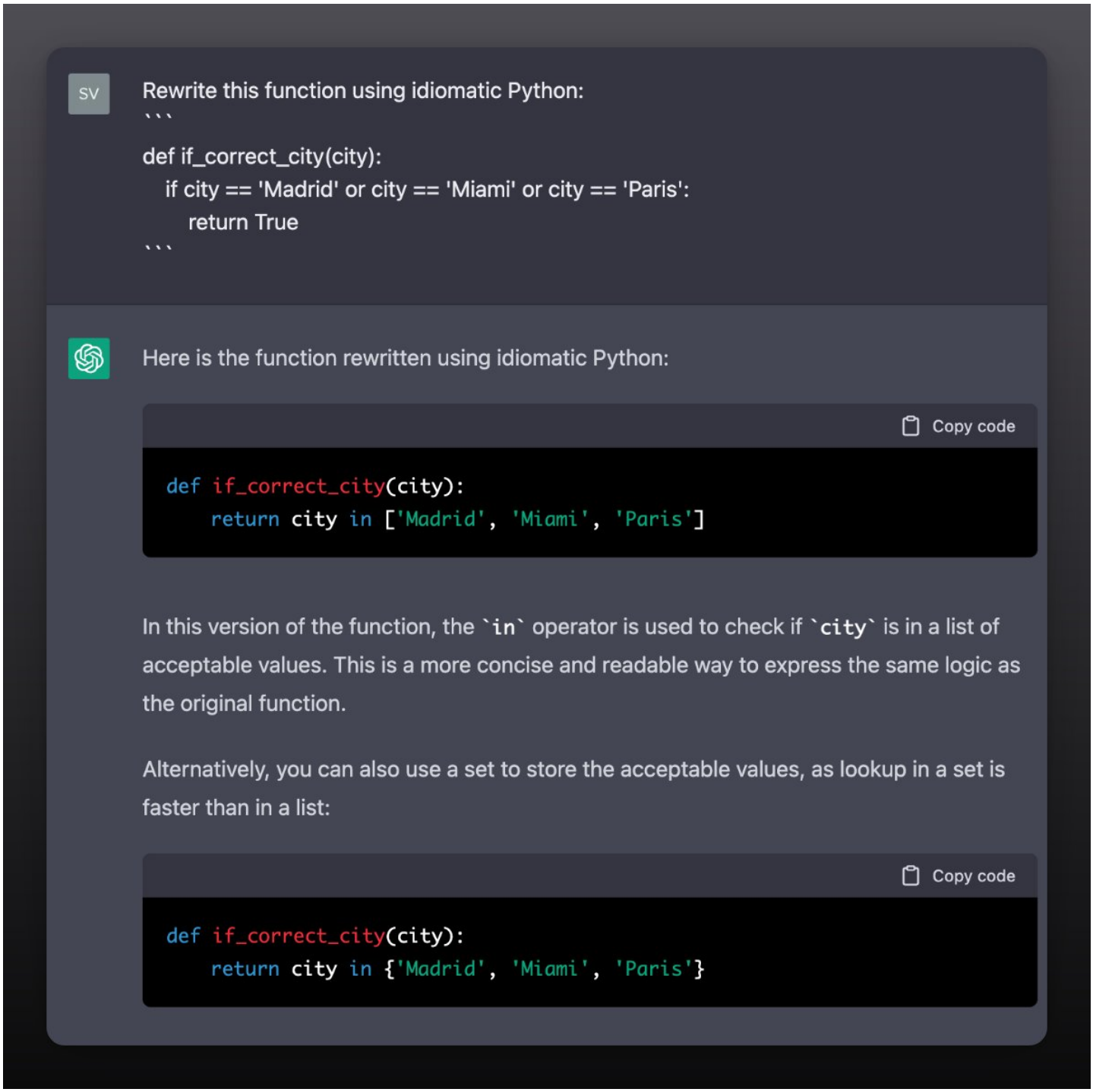


4. Rewriting code using idiomatic constructs

Very helpful when reviewing and refactoring code written by non-native Python developers.

ChatGPT knows the "Pythonic" way, and it will give you suggestions to improve your code and make it much more readable.

5 of 16



5. Simplifying code

This is one of my favorite tricks: Ask ChatGPT to simplify complex code.

The result will be a much more compact version of the original code.

Notice the explanation and how it tells us this is simpler but not the most efficient.

SV

Simplify the following code as much as possible:

```
def partition(array, low, high):
    pivot = array[high]
    i = low - 1
    for j in range(low, high):
        if array[j] <= pivot:
            i = i + 1
            (array[i], array[j]) = (array[j], array[i])
    (array[i + 1], array[high]) = (array[high], array[i + 1])
    return i + 1

def quickSort(array, low, high):
    if low < high:
        pi = partition(array, low, high)
        quickSort(array, low, pi - 1)
        quickSort(array, pi + 1, high)


data = [1, 7, 4, 1, 10, 9, -2]
print(data)

quickSort(data, 0, len(data) - 1)
print(data)
```

Here is the code simplified as much as possible:

Copy code

```
def quick_sort(array):
    if not array:
        return []
    pivot = array[0]
    less = [x for x in array[1:] if x <= pivot]
    greater = [x for x in array[1:] if x > pivot]
    return quick_sort(less) + [pivot] + quick_sort(greater)

data = [1, 7, 4, 1, 10, 9, -2]
print(quick_sort(data))
```

In this version of the code, the `partition` function has been removed and the quick sort logic has been simplified to a single recursive function. The pivot element is chosen as the first element in the array, and the list is partitioned into two sublists using list comprehensions. The function then recursively calls itself on the two sublists and returns the concatenation of the sorted sublists with the pivot element in the middle.

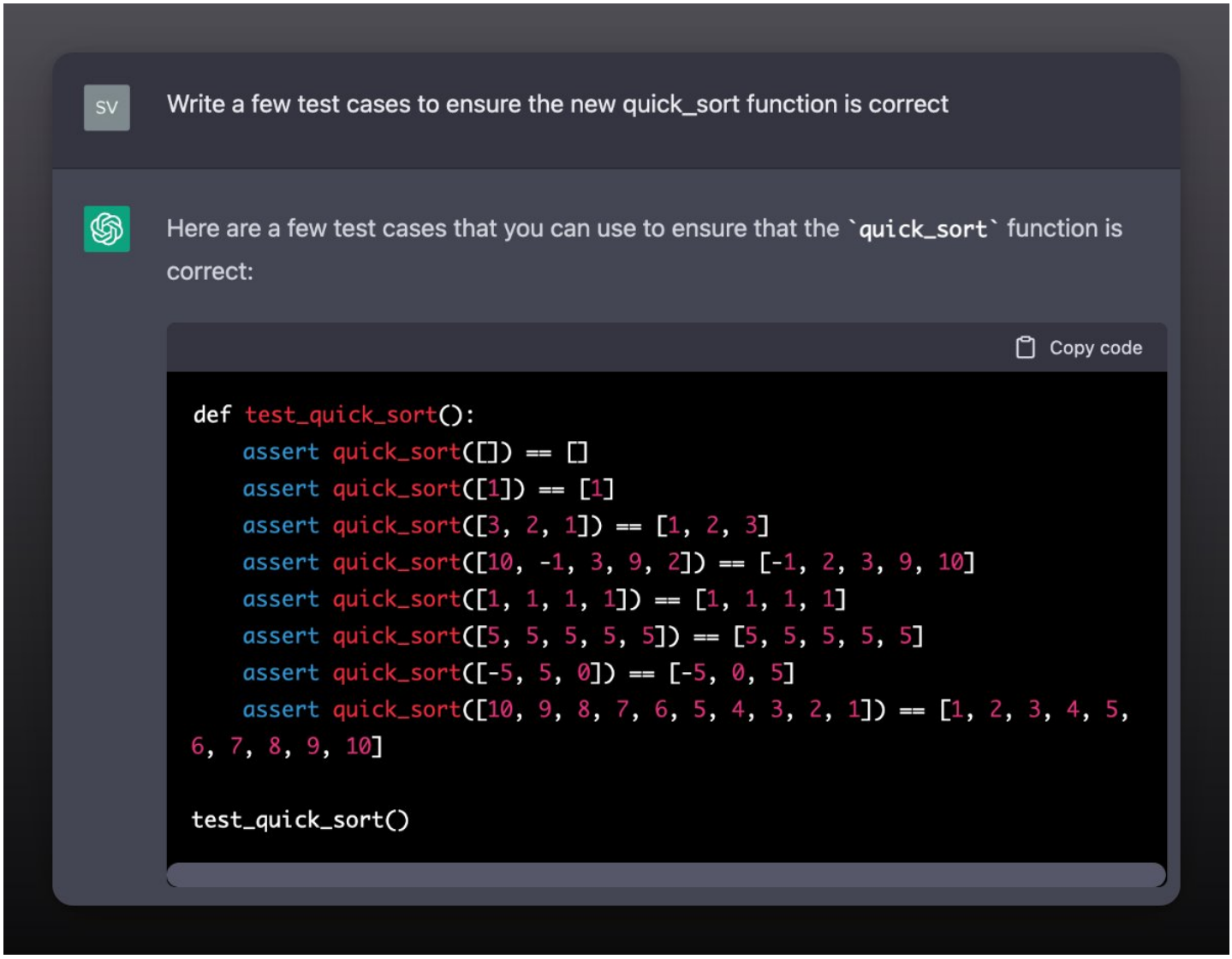
Note that this implementation is not the most efficient way to implement quick sort, but it is the simplest.

6. Writing test cases

This has become one of my favorite ChatGPT abilities: Ask it to help you test a function, and it will write test cases for you.

This example focuses on the quick_sort function from the previous example.

7 of 16



7. Exploring alternatives

ChatGPT told me its Quick Sort implementation wasn't the most efficient, so I asked for an alternative implementation.

This is great when you want to explore different ways to accomplish the same thing.

I've learned a ton from this!

8 of 16

SV What is a more efficient way to implement the quick_sort function in Python?

One way to make the `quick_sort` function more efficient is to use the `random` module to choose a random pivot element instead of always choosing the first element. This helps to avoid worst-case performance when the list is already sorted or nearly sorted.

Here is an example of how you can modify the `quick_sort` function to choose a random pivot element:

Copy code

```
import random

def quick_sort(array):
    if not array:
        return []
    pivot_index = random.randint(0, len(array) - 1)
    pivot = array[pivot_index]
    less = [x for i, x in enumerate(array) if i != pivot_index and x <=
pivot]
    greater = [x for i, x in enumerate(array) if i != pivot_index and x >
pivot]
    return quick_sort(less) + [pivot] + quick_sort(greater)
```

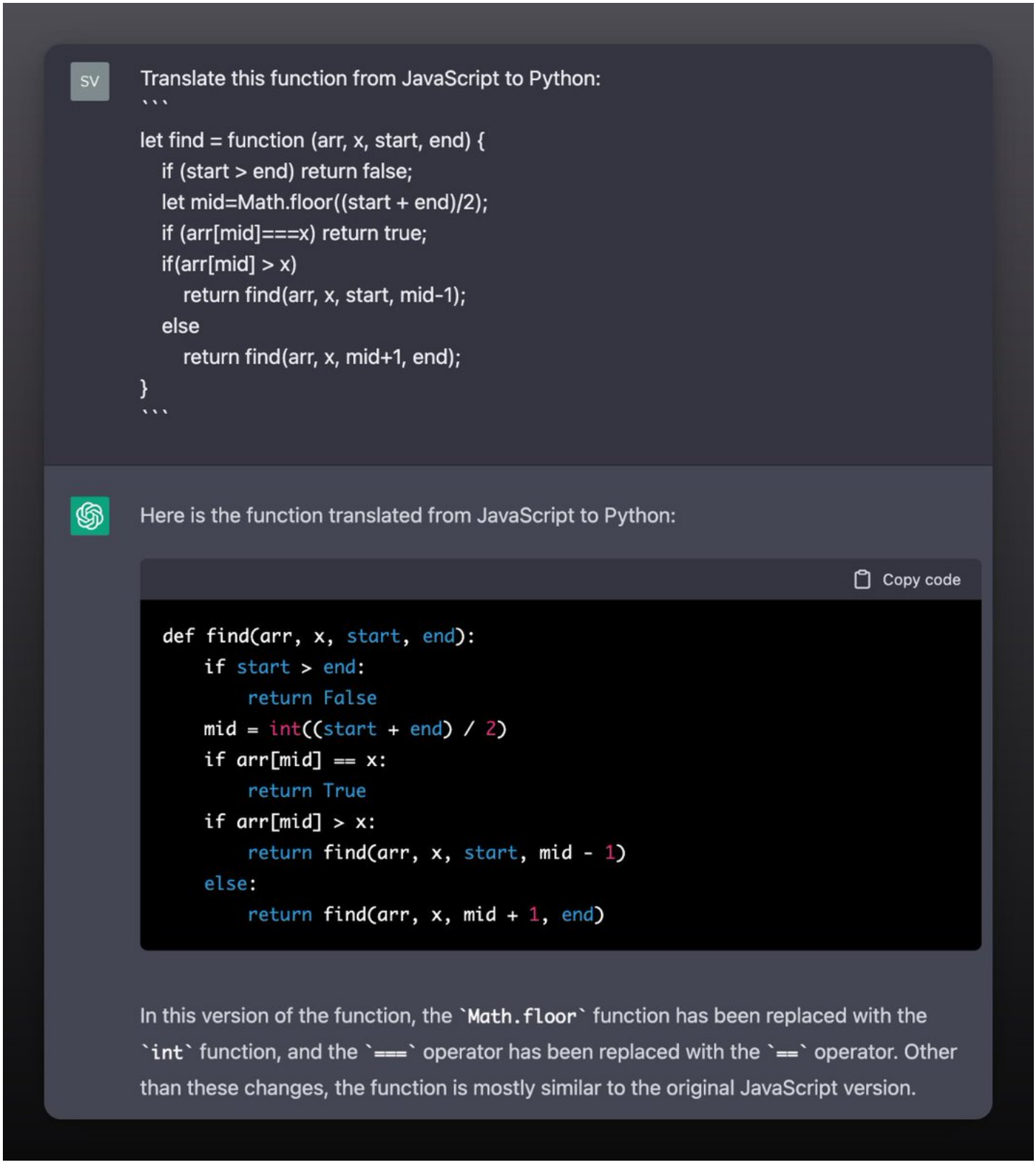
In this version of the function, the pivot element is chosen randomly using the `random.randint` function. The list is then partitioned into two sublists using list comprehensions, with the pivot element excluded from the sublists. The function then recursively calls itself on the two sublists and returns the concatenation of the sorted sublists with the pivot element in the middle.

This approach helps to ensure that the pivot element is not always the first or last element in the list, which can lead to slower performance in certain cases. However, it is important to note that this implementation is still not the most efficient way to implement quick sort in Python, as it has a time complexity of O(n^2) in the worst case. There are more efficient algorithms available, such as the introsort algorithm, which has a time complexity of O(n*log(n)) in the worst case.

8. Translating code

Anytime you want to port some code from one language to another, ask ChatGPT to help you.

9 of 16



9. Writing documentation

This is another one of my favorite tricks.

Ask ChatGPT to write the documentation for a piece of code, and it usually does a great job.

It even includes usage examples as part of the documentation!

SV Write documentation for this function

Here is some documentation for the `find` function:

Copy code

```
def find(arr, x, start, end):
    """
    Finds whether a given value `x` exists in the given list `arr`. The
search is performed within the specified range `start` to `end`
(inclusive).
    The function returns a boolean value indicating whether `x` was found
in `arr`.

    Parameters:
    - arr (list): The list to search in.
    - x (int): The value to search for.
    - start (int): The start index of the range to search in.
    - end (int): The end index of the range to search in.

    Returns:
    - bool: `True` if `x` was found in `arr`, `False` otherwise.

    Example:

    find([1, 2, 3, 4, 5], 3, 0, 4) -> True
    find([1, 2, 3, 4, 5], 6, 0, 4) -> False
    find([1, 2, 3, 4, 5], 2, 2, 4) -> False
    """

    if start > end:
        return False
    mid = int((start + end) / 2)
    if arr[mid] == x:
        return True
    if arr[mid] > x:
        return find(arr, x, start, mid - 1)
    else:
        return find(arr, x, mid + 1, end)
```

This documentation provides a brief description of the function, a list of parameters with their types and descriptions, a return value with its type and description, and an example of how to use the function. It also provides a more detailed description of the function's behavior and how it performs the search. This information should be sufficient for users of the function to understand how to use it and what it does.

10. Tracking down bugs

If you are having trouble finding a bug in your code, ask ChatGPT for help.

It took ChatGPT seconds to find the bug in the attached example. I don't know about you, but it would have taken me much longer than that.

SV

Find the bug in the following function that finds whether a supplied value x exists in an array.

```
def find(arr, x, start, end):
  if start > end:
    return False
  mid = int((start + end) / 2)
  if arr[mid] == x:
    return True
  if arr[mid] > x:
    return find(arr, x, start, mid)
  else:
    return find(arr, x, mid + 1, end)
```

The bug in the `find` function is that it does not update the end index of the range when calling itself recursively with the `arr`, `x`, `start`, and `mid` arguments. This can cause the function to enter an infinite loop if the value of `mid` is not updated.

To fix this bug, you can update the end index of the range to `mid - 1` when calling the function recursively with the `arr`, `x`, `start`, and `mid` arguments, like this:

Copy code

```
def find(arr, x, start, end):
    if start > end:
        return False
    mid = int((start + end) / 2)
    if arr[mid] == x:
        return True
    if arr[mid] > x:
        return find(arr, x, start, mid - 1)
    else:
        return find(arr, x, mid + 1, end)
```

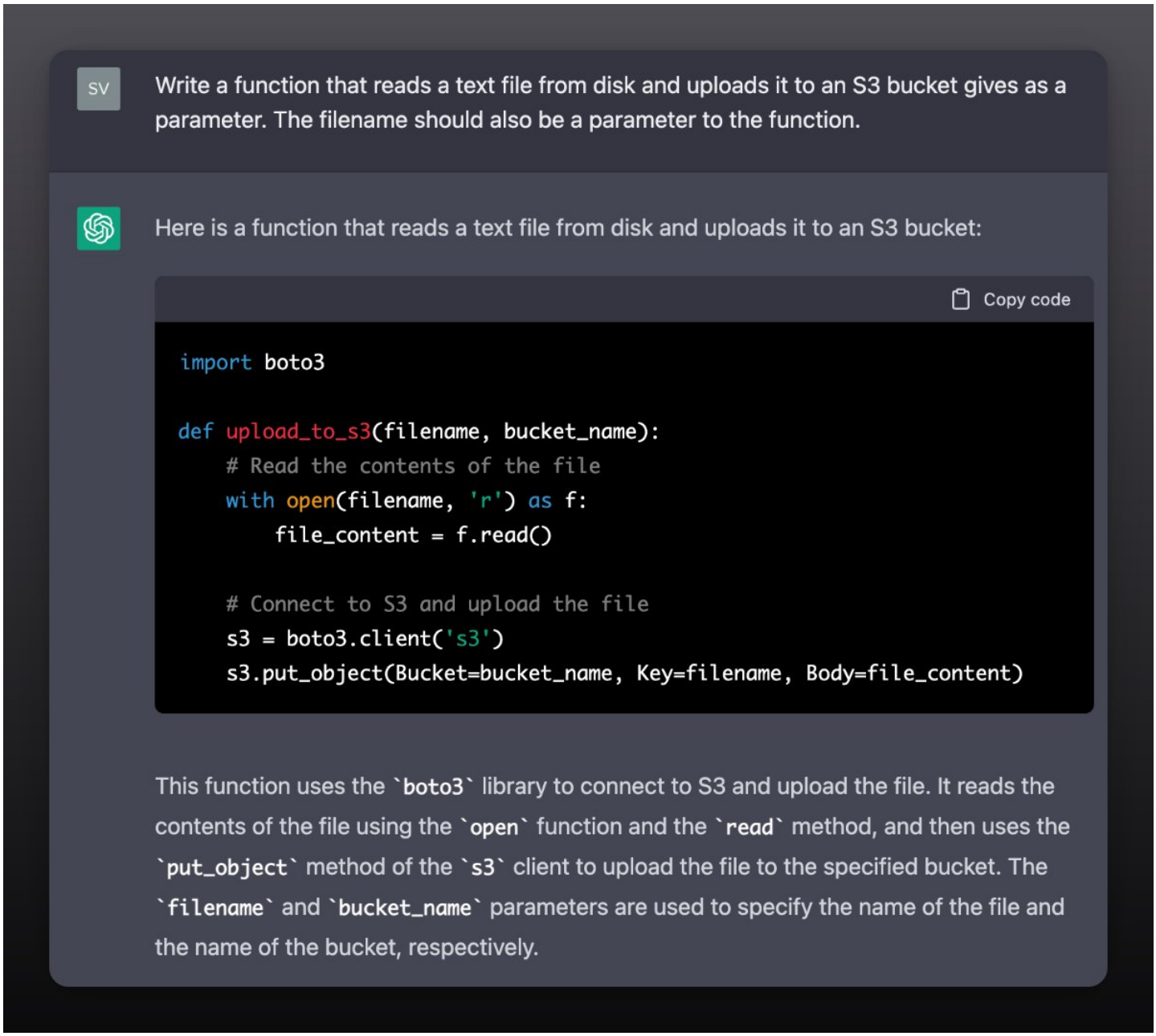
This fixes the bug by correctly updating the end index of the range when the function is called recursively. The function will now terminate correctly and return the correct result.

11. Scaffolding

Probably the way I use it the most is to kick off the structure of anything new I want to write. GitHub Copilot does a great job at this as well.

An example where this is very useful is when interacting with a RESTful API.

12 of 16



Something to keep in mind:

I have 2+ decades of programming experience. I like to think I know what I'm doing.

I don't trust people's code (especially mine,) and I surely don't trust ChatGPT's output.

13 of 16

This is not about letting ChatGPT do my work. This is about using it to 10x my output.

ChatGPT is flawed. I find it makes mistakes when dealing with code, but that's why I'm here: to supervise it.

Together we form a more perfect Union. (Sorry, couldn't help it)

14 of 16

Developers who shit on this are missing the point.

The story is not about ChatGPT taking programmers' jobs. It's not about a missing import here or a subtle mistake there.

The story is how, overnight, AI gives programmers a 100x boost.

Ignore this at your own peril.

15 of 16

Next time, I'll show you how to build an end-to-end deep learning model using ChatGPT.

I'll only ask questions.

It'll blow your mind.

Follow me @svpino to ensure you don't miss it.

16 of 16