Extracted from:

A Common-Sense Guide to Data Structures and Algorithms in Python, Volume 1

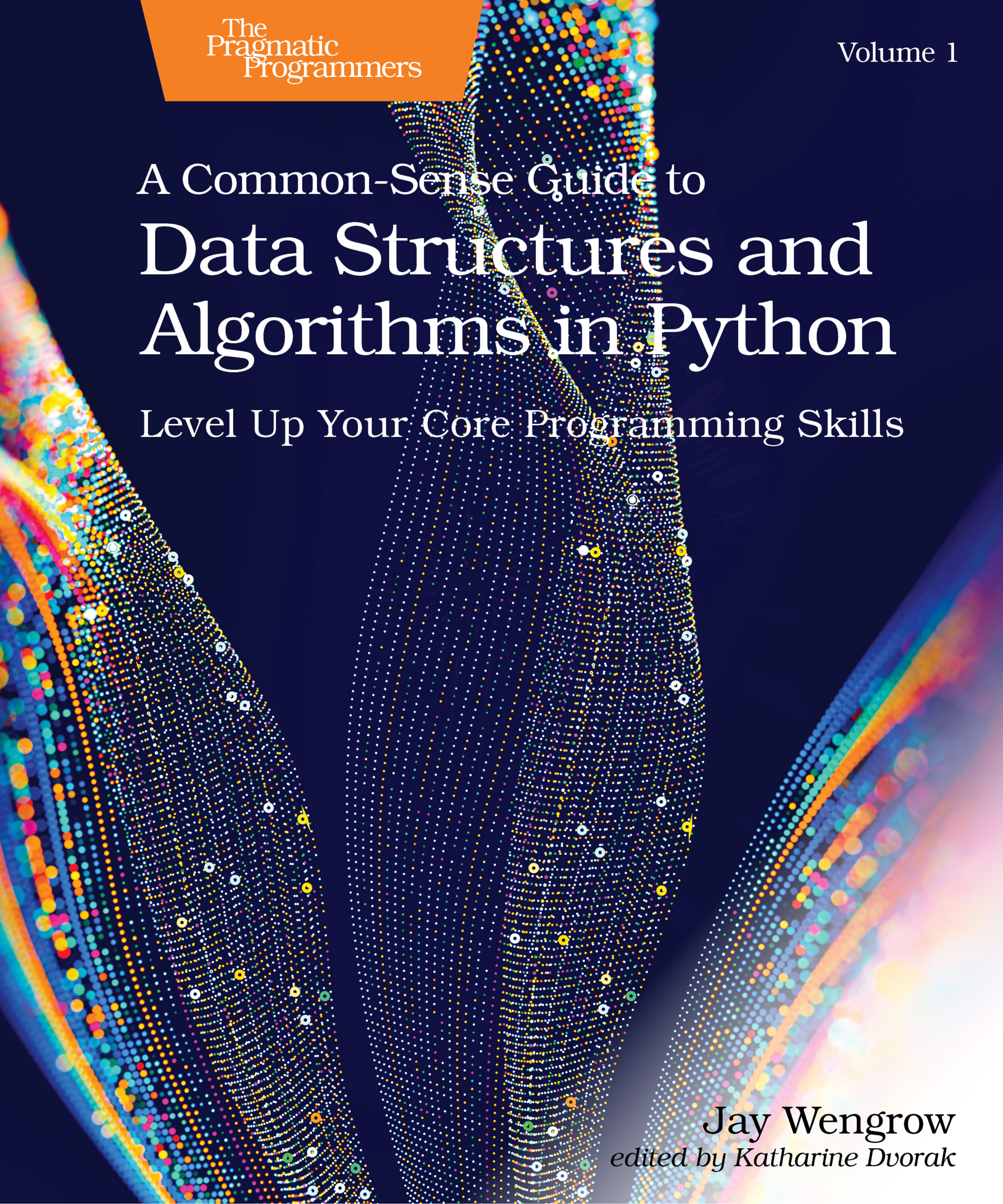
Level Up Your Core Programming Skills

The Pragmatic Bookshelf

Dallas, Texas

The Pragmatic Programmers

Volume 1



A Common-Sense Guide to

Data Structures and Algorithms in Python

Level Up Your Core Programming Skills

Jay Wengrow

edited by Katharine Dvorak

A Common-Sense Guide to Data Structures and Algorithms in Python, Volume 1

Level Up Your Core Programming Skills

Jay Wengrow

The Pragmatic Bookshelf

Dallas, Texas

The team that produced this book includes:

Publisher: Dave Thomas
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Katharine Dvorak
Copy Editor: L. Sakhi MacMillan
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

ISBN-13: 979-8-88865-035-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—December 2023

CHAPTER 11

Learning to Write in Recursive

In the previous chapter, you learned what recursion is and how it works. I found on my own learning journey that even once I understood how recursion works, I still struggled to write my own recursive functions.

Through deliberate practice and taking note of various recursive patterns, I discovered some techniques that helped me to learn to "write in recursive" more easily, and I'd like to share them with you. Along the way, you'll discover additional areas where recursion shines.

Note that we won't be discussing the efficiency of recursion in this chapter. Recursion can actually have a terribly negative impact on the time complexity of an algorithm, but this is the subject of the next chapter. For now, we're just going to focus on developing the recursive mindset.

Recursive Category: Repeatedly Execute

Over the course of tackling various recursive problems, I began to find that there are various categories of problems. Once I learned an effective technique for a certain category, when I found another problem that belonged to the same category, I was able to apply the same technique to solve it.

The category that I found to be the easiest was one in which the goal of the algorithm was to repeatedly execute a task.

The NASA spacecraft countdown algorithm from the previous chapter is a great example. The code prints a number such as 10, and then 9, then 8, all the way down to 0. While the number the function prints is different each time, we boil down the code's essence to the fact that it is repeatedly executing a task—namely, the printing of a number.

This was our implementation of that algorithm:

```
def countdown(number):
    print(number)

    if number == 0:
        return
    else:
        countdown(number - 1)
```

I found that for problems of this category, the last line of code in the function was a simple, single call to the function again. In the previous snippet, this takes the form of countdown(number - 1). This line does one thing: it makes the next recursive call.

The directory-printing algorithm from the previous chapter is another example of this. This function repeatedly executes the task of printing directory names.

Our code looked like this:

```
import os

def print_subdirectories(directory_name):
    for filename in os.listdir(directory_name):
        path = os.path.join(directory_name, filename)
        if os.path.isdir(path):
            print(path)
            print_subdirectories(path)
```

Here as well, the last line of code is print_subdirectories(path), which is a simple call to the recursive function, triggering it again.

Recursive Trick: Passing Extra Parameters

Let's try our hand at another problem of the "repeatedly execute" category. We're going to write an algorithm that takes an array of numbers and doubles each of the numbers within the array. Note that we're not going to produce a new array; instead, we're going to modify the array in place.

This algorithm, too, is one in which we repeatedly execute a task. Specifically, we repeatedly double a number. We start with the first number and double it. We move onto the second number and double it—and so on.

Let's take a stab at writing this algorithm in Python, which we'll call double_array(). We know that our final line will be the recursive call, so let's include that line:

```
def double_array(array):
    double_array(array)
```

In-Place Modifications

Let's take a look at the concept of *in-place* modifications, just in case it's not clear.

Generally speaking, there are two basic approaches to manipulating data. Let's take the example of doubling the values in an array. If I had the array [1, 2, 3, 4, 5] and wanted to "double" it to produce the array [2, 4, 6, 8, 10], I could do one of two things.

The first option is to create a new array that contains the doubled data but leave the original array alone. Take the following code:

```
a = [1, 2, 3, 4, 5]
b = double_array(a)
```

Since the double_array function is creating and returning a brand-new array, if we were to check the values of a and b, we'd get the following:

```
a # [1, 2, 3, 4, 5]
b # [2, 4, 6, 8, 10]
```

The original array a was unmodified, and b holds a brand-new array.

The second option is called *in-place* modification, meaning the function actually changes the *original array* passed into function.

With in-place modification, if we now check a and b, we'd find this:

```
a # [2, 4, 6, 8, 10]
b # [2, 4, 6, 8, 10]
```

The in-place function modifies a, and b is just pointing to the very same array that a is.

Whether we choose to create a new array or modify the original one in place is up to us and depends on the context of our project. We're going to talk more about in-place algorithms in *Dealing with Space Constraints*.

Next, we need to add the code that will actually double the number. But which number will we double? Let's try to double the first number:

```
def double_array(array):
    array[0] *= 2
    double_array(array)
```

Okay, so we've doubled the number at index 0, but how do we proceed to double the number at index 1?

Now, if we were to use a loop instead of recursion, we would have used a variable to keep track of the index and continuously increased it by 1, something along these lines:

```
def double_array(array):
    index = 0

    while index < len(array):
        array[index] *= 2
        index += 1
```

In our recursive version, though, the only argument to our function is the array. We need some way to keep track of and increment an index. How do we pull this off?

And now for our next trick…

Let’s pass in extra parameters!

Let’s modify the beginning of our function so that it accepts *two* arguments—the array itself and an index to keep track of. Here’s the code:

```
def double_array(array, index):
```

As things stand right now, when we call this function, we need to pass in the array and the starting index, which will be 0:

```
double_array([1, 2, 3, 4, 5], 0)
```

Once we have the index as a function argument, we have a way of incrementing and tracking the index as we make each successive recursive call. Here’s the code for this:

```
def double_array(array, index):
    array[index] *= 2
    double_array(array, index + 1)
```

In each successive call, we pass in the array again as the first argument, but we also pass along an incremented index. This allows us to keep track of an index just as we would in a classical loop.

Our code isn’t perfect just yet, though. Our function will throw an error once the index goes past the end of the array and tries to multiply a nonexistent number. To solve this, we need our base case:

```
def double_array(array, index):
    # Base case: when the index goes past the end of the array
    if index >= len(array):
        return

    array[index] *= 2
    double_array(array, index + 1)
```

We can test this function out with the following code:

```
array = [1, 2, 3, 4]
double_array(array, 0)
print(array)
```

Our recursive function is now complete. However, if our programming language supports default arguments as Python does, we can make things even prettier.

Right now, we need to call the function like this:

```
double_array([1, 2, 3, 4, 5], 0)
```

Admittedly, passing in that 0 as a second parameter isn't beautiful—it's just so we can achieve our trick of maintaining an index. After all, we *always* want to start our index off at 0.

However, we can use default parameters to allow us to simply call the function the original way:

```
double_array([1, 2, 3, 4, 5])
```

Here's our updated code to make this work:

```
def double_array(array, index=0):
    # Base case: when the index goes past the end of the array
    if index >= len(array):
        return

    array[index] *= 2
    double_array(array, index + 1)
```

All we updated here was setting a default argument of `index=0`. This way, the first time we call the function, we don't have to pass in the index parameter. However, we still get to use the index parameter for all successive calls.

The "trick" of using extra function parameters is a common technique in writing recursive functions, and a handy one.