Readings for Unit 6

Back to Top

Licensing Information

PDF of these readings also available to download: 6s090_reading6.pdf

Table of Contents

1) Introduction

Throughout the course, we've been getting familiar with functions. In this reading, we'll examine a particular way of using functions: having them *call themselves*. This is called recursion. We'll see some examples of when recursion is useful.

We'll also learn more about NumPy, which you'll use in this unit's exercises. Finally, we will spend some time more formally developing a strategy for designing programs.

2) Recursion

> *"People often joke that in order to understand recursion, you must first understand recursion."*
> -John D. Cook

As we have seen before, it is possible for one function to call another. It turns out that it is also possible for a function to call itself! It may not be immediately obvious why that is a good thing, but it turns out that certain types of computations are most easily expressed this way. Let's start with a small example, and by the end of this section, we will have built up to some more realistic and interesting examples.

Back to Top

```
def countdown(n):
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
```

If `n` is 0 or negative, this function prints the word "Blastoff!" Otherwise, it prints `n` and then calls a function named `countdown` (that is, it calls *itself*), passing `n-1` as an argument.

What happens if we call this function like this?

```
countdown(3)
```

The execution of this program proceeds as follows:

- The execution of our original call to `countdown` begins in a new frame with `n=3`, and since `n` is greater than 0, it outputs the value 3, and then calls itself...
    - In the process of running our original call, we call `countdown`. As before, the execution of this new call to `countdown` begins in a new frame with `n=2`, and since `n` is greater than 0, it outputs the value 2, and then calls itself...
        - The execution of yet another call to `countdown` begins in a new frame with `n=1`, and since `n` is greater than 0, it outputs the value 1, and then calls itself...
            - The execution of another call to `countdown` begins in a new frame with `n=0`, and since `n` is not greater than 0, it outputs the word "Blastoff!" and then returns (in this case, because there was no `return` statement, it simply returns `None`).
        - The function call of `countdown` with `n=1` then returns.
    - The function call of `countdown` with `n=2` then returns.
- The function call of `countdown` with `n=3` then returns.

And then you're back in the main body of the program. So, the total output looks like this:

```
3
2
1
Blastoff!
```

A function that calls itself is *recursive*, and the process by which it executes is called *recursion*.

2.1) Another Example

As another example of recursion, we can write a function that prints a string `n` times:

Back to Top

```python
def print_n(s, n):
    if n <= 0:
        return  # note: if we don't give a value to the return statement, it returns None
    print(s)
    print_n(s, n-1)
```

If `n <= 0` the *return statement* exits the function. The flow of execution immediately returns to the caller, and the remaining lines of the function don't run.

The rest of the function is similar to `countdown`: it displays `s` and then calls itself to display `s` $n-1$ additional times. So the number of lines of output is `1 + (n - 1)`, which adds up to `n`.

2.2) A Duality

Note that the function from above produces the exact same output as a program that we could have written with a `for` loop:

```python
def print_n(s, n):  # "recursive" solution
    if n <= 0:
        return
    print(s)
    print_n(s, n-1)



def print_n(s, n):  # "iterative" solution (using loops)
    for i in range(n):
        print(s)
```

You might be tempted to ask yourself: is iteration better, or is recursion?

You could make the argument that, in this case, the iterative solution is better: it's shorter, and it's easier to read. But later on, we'll see examples where the opposite is true! For better or for worse, there is no right answer to whether iteration or recursion is better *in the general sense*; it depends both on the program being written, and on the programmer's personal preferences.

Some people prefer to write in an *iterative* style (using loops), some prefer to write in a *recursive* style (using recursion), and almost all will write in one style or the other if a particular computation is easier to think about or express in one style.

In general, it is possible to rewrite any iterative program using recursion, and *vice versa*. But sometimes it is much easier to express a program in one style than the other. Over time, you will develop a sense of which works better for you in which situations.

2.3) Fibonacci Example

Some mathematical computations are (relatively) easily specified via *induction*, which is very closely related to recursion: they have some set of base cases, and the rest of the results are based on those base cases.

Consider, for example, the Fibonacci sequence: $0, 1, 1, 2, 3, 5, 8, 13, \ldots$.

This sequence can be specified in the following form, where $F[n]$ represents the n^{th} Fibonacci number:

$$F[0] = 0$$

Back to Top

$$F[1] = 1$$

$$F[n > 1] = F[n-1] + F[n-2]$$

The first two lines (specifying $F[0]$ and $F[1]$) are the *base cases*, and they are expressed without recursion; the last line specifies the *general case*, which is recursive. This general form holds: all recursive programs should have one or more *base cases* and one or more *general cases*.

A recursive implementation of a program to compute the n^{th} Fibonacci number mirrors this form:

```python
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n-1) + fib(n-2)
```

The way Python will proceed with evaluation is largely the same way a human might when using the mathematical form above. If we were to think, for example, about computing $F[3]$ by hand, we could proceed as follows, by always expanding the left-most term that can be expanded (shown in blue on each line):

$$\begin{aligned} F[3] &= F[2] + F[1] \\ &= F[1] + F[0] + F[1] \\ &= 1 + F[0] + F[1] \\ &= 1 + 0 + F[1] \\ &= 1 + 0 + 1 \\ &= 2 \end{aligned}$$

Python will evaluate `fib(3)` in a similar way. In order to figure out how exactly each function is being invoked and its results computed, we could draw out an environment diagram. However, here, we are interested in a higher-level behavior, and so we'll abstract away the internal details of the `fib` function below, focusing instead on end-to-end behavior.

- When we call `fib(3)`, Python eventually reaches a statement that tells it to return `fib(2) + fib(1)`. In order to do that, it must evaluate `fib(2)` and `fib(1)`.
    - Evaluating `fib(2)`, Python eventually reaches a statement that tells it to return `fib(1) + fib(0)`. In order to do that, it must evaluate `fib(1)` and `fib(0)`.
        - Evaluating `fib(1)` returns 1, and
        - evaluating `fib(0)` returns 0.
    - With these results, Python knows that `fib(2)` will be `1`.
    - Now, evaluating `fib(1)`, Python sees that it returns `1`.

- With these results, Python knows that `fib(3)` returns their sum, or `2`.

2.4) Base Cases and Infinite Recursion

Back to Top

Notice that each of the recursive functions we've looked at so far has had a *base case* with no recursive call. If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates. This is known as *infinite recursion*, and it is generally not a good idea. Here is a minimal program with an infinite recursion:

```
def recurse():
    recurse()
```

In most programming environments, a program with infinite recursion does not really run forever. When running this program, we in fact see an error:

```
  File "broken_recursion.py", line 2, in recurse
  File "broken_recursion.py", line 2, in recurse
  File "broken_recursion.py", line 2, in recurse
                .
                .
                .
  File "broken_recursion.py", line 2, in recurse
RuntimeError: Maximum recursion depth exceeded
```

By default, Python limits itself to recursive calls that go 1000 "layers" deep. So when the error occurs, there are 1000 `recurse` frames in existence!

If you encounter an infinite recursion by accident, review your function to confirm that there is a base case that does not make a recursive call. And if there is a base case, check whether you are guaranteed to reach it.

2.5) Global Variables

As we've seen with functions before, variables assigned inside of the function are *local variables*. They cannot be accessed from outside of the function. Additionally, if there's a variable of the same name outside of the function, then the local variable will be separate, so assigning a value to the local variable won't alter the external one. For example, running the code below will print `3` and then `2` since the external `a` variable is not changed when the local variable `a` is assigned.

```
a = 2

def foo(x):
    a = x
    print(a)

foo(3)
print(a)
```

However, occasionally we want to be able to alter external variables. For instance if we want to set some global flag to be `True` if any function call sees a certain value. We can use a *global variable* for this, and to mark that a variable should be global instead of local, we use the `global` keyword, as such:

```
a = 2

def foo(x):
    global a
    a = x
    print(a)

foo(3)
print(a)
```

Back to Top

Now running this function will print `3` and `3`. Typically people write the global declaration on the first lines of the function body. The `global` declaration needs to come before we assign to the global variable, and so this code would produce an error:

```
a = 2

def foo(x):
    a = x
    global a
```

The concept of global variables is not restricted to recursive functions (as we saw in the examples above), but it can be handy if we want our functions to all be able to edit some shared variable. For example, maybe we want our fibonacci example to keep track of the number of times we called it with `n=4`. Then we could write:

```
num_4 = 0

def fib(n):
    global num_4
    if n == 4:
        num_4 += 1
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)

fib(10)
print(num_4) # fib(4) was called 13 times! If only there were a way to keep from having to recalculate it
every time...
```

As a note, global variables are not commonly used in practice, since style standards typically expect functions to only modify their inputs and return any results that need to be used externally.

3) NumPy 101

Now let's dive deeper into NumPy, a widely-used library that provides fast operations for mathematical and numerical routines. NumPy, pronounced num-PIE, is an acronym for **Num**erical **Py**thon. NumPy is extraordinarily efficient because it is highly optimized for vector/matrix operations "under the hood" in ways that regular Python objects are not. In fact, for matrix solves, NumPy can be many thousands of times faster!

The central feature of NumPy is the `array` type, which is similar to a regular Python `list`, except that:

- all the elements within an `array` must have the same type[1] (such as `float`, `int`, or `bool`), and
- `array`s have different built-in features.

Back to Top

Let's start by looking at a one-dimensional array. In order to make an `array` object, we'll first need to import the `numpy` module[2]:

```
import numpy
```

After having run that line, we can create new `array` objects by using `numpy.array`, as shown in the example below (which creates a 1-dimensional array):

```
a = numpy.array([9, 3, 7, 6])
```

`array` objects can also be multidimensional. Below we create a 2-dimensional `array`. Notice that we create it by giving it a list of lists, of the same form we used in earlier 2-D "array" exercises.

```
a2 = numpy.array([[1, 2, 3], [4, 5, 6]])
```

NumPy `array` objects are frequently used to represent vectors and matrices. For example, `a2` has an interpretation as this matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

3.1) Array Properties and Operations

There are a few pieces of information we can quickly grab from `array` objects, including:

```
a2.shape  # a tuple of the dimensions (dim1, dim2, ...); here, (2, 3)
a2.size  # the total number of elements; here, 6
a2.T  # the transpose; here, numpy.array([[1, 4], [2, 5], [3, 6]])
```

NumPy conveniently overrides many of the built-in operators so that they behave as we would expect. For example, the `+` operator performs array addition, and `-` performs matrix subtraction.

```
a3 = numpy.array([[9, 10, 11], [12, 13, 14]])

print(a2 + a3) # prints numpy.array([[10, 12, 14], [16, 18, 20]])
```

Perhaps counter-intuitively, the `*` operator performs *element-wise multiplication* instead of matrix multiplication. To perform a matrix multiplication, you can use the `@` operator (the "at sign"). For example, to compute $a_2 * a_3^T$, we could use:

```
a2 @ a3.T
```

Back to Top

3.2) Array Accessing

Unlike with Python lists, different axes of a NumPy `array` are accessed by putting multiple values (separated by commas) in the square brackets used for indexing. For example, we can grab a few different elements from the array:

```
a2 = numpy.array([[1, 2, 3], [4, 5, 6]])
print(a2[0, 0])  # this grabs the element in row 0, column 0 (so 1 will be printed)
print(a2[1, 2])  # this prints 6 (row 1, column 2)
```

We can grab other pieces of `array`s via "slicing," which works on each dimension separately. The use of a single `:` in a dimension indicates that the slice should include everything in that dimension. For example:

```
print(a2[1, :])  # this will grab all columns in row 1, so it will print array([4, 5, 6])
print(a2[:, 2])  # this will grab all rows in column 2, so it will print array([3, 6])
```

Notice that the resulting slices are themselves NumPy `array` objects (not Python lists). You can convert any `array` object back to a list by calling its `tolist` method:

```
l = a2.tolist()  # l is now a list-of-lists [[1, 2, 3], [4, 5, 6]]
```

3.3) Other Tools

NumPy also makes functionalities beyond arrays available. As you read about the following selected functionalities, you should cross-reference them with the NumPy documentation, which has extensive explanations and examples for these and *many* other NumPy capabilities. (We recommend bookmarking this link!)

- `numpy.random.choice` generates random samples. It takes a 1-D `array` or list of elements from which to sample, *or* an integer. (If an integer, the sample returned is from the set of integers 0 up to, and not including, that integer.)

  This function also takes a few optional keyword arguments. These are arguments that need not be passed in; if they are, they can be specified by naming the argument in the function call. The optional arguments are:

  - `size`, the shape of the array of samples to return (an integer or tuple)
  - `replace`, a boolean indicating whether or not to sample with replacement, and
  - `p`, an array of the same shape as the given array, containing probabilities with which each element should be drawn

```
# gives an integer chosen uniformly from 0, 1, 2, or 3
numpy.random.choice(4)

# gives a 1x10 array where each element is a chosen uniformly from 1, 2, 3, 4, 5, or 6 (sampled with
replacement from a2)
a4 = numpy.array([1, 2, 3, 4, 5, 6])
numpy.random.choice(a4, size=10)
```

```
# same as above, but with five / six elements sampled without replacement
numpy.random.choice(a4, size=5, replace=False)
```

Back to Top

```
# gives a 1x10 array where each element is 2, 3, or 4, drawn with probabilities 0.25, 0.5, and 0.25
respectively
numpy.random.choice([2, 3, 4], p=[0.25, 0.5, 0.25], size=10)
```

- `numpy.random` also makes common probability distributions available. You can see a list of available distributions here. For example, you can draw random samples from a normal distribution with mean 1 and standard deviation 0.3 like so:

```
s1 = numpy.random.normal(1, 0.3) # get one number (one sample)
s300 = numpy.random.normal(1, 0.3, 300) # get an array of 300 samples
```

- `numpy.mean` quickly computes the mean of a given array. Optionally, you can ask if for the mean(s) *along a given axis*, where axis 0 runs down the rows, axis 1 runs along the columns, and so on for higher-dimensional arrays.

  Recall our definition of `a2` from before:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

```
a2 = numpy.array([[1, 2, 3], [4, 5, 6]])
numpy.mean(a2) # 3.5
numpy.mean(a2, axis=0) # numpy.array([2.5, 3.5, 4.5])
numpy.mean(a2, axis=1) # numpy.array([2.0, 5.0])
```

- Many other NumPy functions use this notion of axes. For example, `numpy.cumsum` also optionally takes an axis along which to compute the cumulative sum:

```
a2 = numpy.array([[1, 2, 3], [4, 5, 6]])
# 1-D array with cumulative sums from row-major order traversal of a2
# here, numpy.array([1, 3, 6, 10, 15, 21])
numpy.cumsum(a2)


# array of same size as a2, with cumulative sums along all columns
# here, numpy.array([[1, 2, 3], [5, 7, 9]])
numpy.cumsum(a2, axis=0)


# array of same size as a2, with cumulative sums along all rows
# here, numpy.array([[1, 3, 6], [4, 9, 15]])
numpy.cumsum(a2, axis=1)
```

4) Designing Programs

> *"There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies."*
> -C.A.R. Hoare

Back to Top

So far, we've spent a lot of time introducing the building blocks that Python offers us, and relatively little time talking actively about how to design a good program. We've done this for what we believe is a good reason— namely, that it is difficult to understand how to go from a problem statement to a well-designed working program without a sufficiently advanced understanding of Python.

In this section, we'll talk a bit about what constitutes good style in a program, and then we'll shift focus to talking about a strategy for desigining programs. The good news is that you've already got some practice under your belts in both of these areas, and so we'll try here to build on that experience.

4.1) Style

> *"A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away."*
> -Antoine de Saint-Exupery

When we talk about programming *style* in this text, we're typically not referring to writing code that *looks* pretty, but rather about writing code that is as easy to read, write, understand, and debug as possible.

What constitutes good style and good design is, in some ways, a subjective question. However, there are a few notions related to style that are reasonably well accepted:

- **Names Matter**: Choosing good names for functions, parameters, and other variables will help make your code much easier to understand. Names of functions should describe what they do. Variable names should describe what they represent (not just their types!). Single letter names are okay in some situations, but use such names sparingly.

- **Don't Repeat Yourself (DRY)**: (also known as *Less is More*) Multiple fragments of code should not describe redundant logic. Instead, that logic should be simplified into a loop, function, or variable (depending on the particular kind of repetition you're dealing with). If you find yourself re-writing the same short expression over and over, that might be a sign that you can store that in a variable as an intermediate result. If you find yourself copy/pasting a block of code to compute a result, that might be an opportunity to define a function.

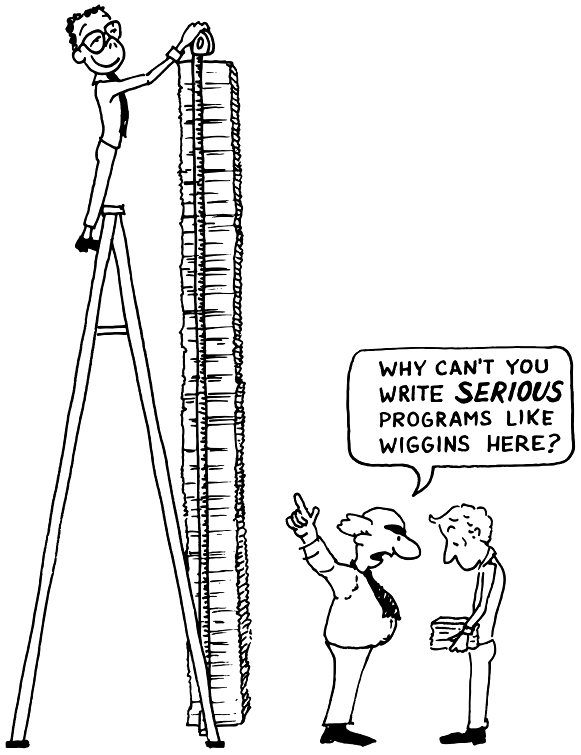
- **Generality Wins**: It is best to define functions and programs generally, and to let their inputs handle specific cases. For example, the `square` function is not defined in the `math` module, in part because it is a specific use case for the `pow` function (exponentiation), which *is* in the `math` module.

- **Plan for Change**: In many ways, this is an extension of *Generality Wins*. Oftentimes, when writing programs, you may find that the requirements for the program you're writing may change (or you may find that the problem you *actually* wanted to solve isn't the one solved by the program you're writing). As such, whenever possible, it is important to attempt to make programs that are (relatively) easy to change, should the need arise.

These guidelines will help to:

- improve the readability of your code
- reduce the number of errors in your code
- make it easier to make changes to your program if you need to
- minimize the amount of code you write

Indeed, this last goal is an important one, even though "Conventional Wisdom Reveres Complexity":[3]



Back to Top

You may hear of people talking about their programs in terms of "lines of code" as a metric, but *in the wrong direction*. It turns out that a good goal is usually to *minimize* the amount of code written to solve a particular problem (people are often surprised at how small a powerful program can be!).

> "*Simplicity is prerequisite for reliability.*"
> -Edsger W. Dijkstra

4.1.1) Refactoring Example: An "Averaging Filter"

In this example, we'll talk through successively refining a program to improve it in terms of style, a process referred to as refactoring. We'll start with a piece of code that has a number of issues, and gradually improve it throughout this section.

Imagine writing a program to apply an "averaging filter" to a list of numbers. We can think about this as applying a moving average: that is, if we have an input list with n numbers in it, $x_0, x_1, \ldots x_n$, we want to compute new values $y_k, y_{k+1}, \ldots, y_{n-(k-1)}$ such that each value in the output list is the average of the previous k values in the input list:

$$y_i = \frac{x_i + x_{i-1} + \ldots + x_{i-(k-1)}}{k}$$

Let's start with computing the running average for the following list, with $k = 3$:

```
input_list = [100, 27, 93, 94, 107, 10]
```

Because $k = 3$, we can't compute the appropriate values for the first two inputs, so our output list should be 4 elements long in this case.

Here is a way that one might get started writing a program to compute the desired result:

```python
## Program number 1
input_list = [100, 27, 93, 94, 107, 10]

averaged_result = [0] * 4

averaged_result[0] = (input_list[0] + input_list[1] + input_list[2]) / 3
averaged_result[1] = (input_list[1] + input_list[2] + input_list[3]) / 3
averaged_result[2] = (input_list[2] + input_list[3] + input_list[3]) / 3
averaged_result[1] = (input_list[3] + input_list[4] + input_list[5]) / 3

print(averaged_result)
```

Back to Top

This is the kind of code that might come about from writing one line of code, and then copy/pasting it, making small changes to account for the differences in what we wanted to compute at each step.

If you look closely at the code, you'll notice that two bugs managed to creep in to the pasted versions: the last result is not being stored in the proper location, and the second-to-last result is not being computed correctly!

This class of errors (forgetting to make a change or making an incorrect change on copy/pasted code) is a relatively common occurance, referred to as a "Copy/Paste Error". We can often avoid these kinds of errors by restructuring to avoid repetition.

This is a case where we can notice with relatively little effort that we are repeating ourselves a lot. Here, we are performing an operation several times for different elements in a list, and so we can "refactor" this code to make use of a loop:

```python
## Program number 2
input_list = [100, 27, 93, 94, 107, 10]

averaged_result = []
for i in range(0, 4):
    this_avg = (input_list[i] + input_list[i + 1] + input_list[i + 2]) / 3
    averaged_result.append(this_avg)

print(averaged_result)
```

In many ways, this is a much nicer piece of code. But now let's imagine that we wanted to perform this same computation on a *second* list of numbers. One option available to use is to copy/paste this structure and to make changes based on the properties of the new list:

```python
## Program number 3
input_list = [100, 27, 93, 94, 107, 10]

averaged_result = []
for i in range(0, 4):
    this_avg = (input_list[i] + input_list[i + 1] + input_list[i + 2]) / 3
    averaged_result.append(this_avg)

print(averaged_result)

input_list2 = [94, 38, 96, 20, 18, 100, 108]
```

```
averaged_result2 = []
for i in range(0, 5):
    this_avg = (input_list2[i] + input_list2[i + 1] + input_list2[i + 2]) / 3
    averaged_result2.append(this_avg)

print(averaged_result2)
```

Back to Top

We're repeating ourselves once again! We're performing the exact same computation on two different inputs. Here, it seems like we should be able to generalize this operation using a function! And indeed we can, though we'll need to think carefully about a few details of that function. The end result is shown here:

```
## Program number 4
def averaging_filter(inputs):
    result = []
    for i in range(len(inputs)-2):
        avg_val = (inputs[i] + inputs[i+1] + inputs[i+2]) / 3
        result.append(avg_val)
    return result

input_list = [100, 27, 93, 94, 107, 10]
print(averaging_filter(input_list))

input_list2 = [94, 38, 96, 20, 18, 100, 108]
print(averaging_filter(input_list2))
```

Why should this code be considered any better?

1. One could argue that the overall flow of the program is easier to read and understand now, because it is perhaps more obvious what operation we're performing on each of the two lists.

2. If we wanted to run this computation on more lists, it is much easier to do so now (particularly since our function can also account for the varying lengths of the lists).

3. If we discovered a bug in our implementation of the averaging procedure, we only have implement the fix in one place, rather than in multiple places (just like with copy/paste errors, it's easy to forget to make the change in one of the necessary places!).

This code has come a long way, but one could perhaps argue that another step is worthwhile. We saw earlier that the behavior of the averaging filter can change based on the value of k (the number of samples we average). As such, favoring generality (in case we decided tomorrow that we wanted $k = 5$ instead of $k = 3$, or that we wanted to use different k values on each of the lists), we could make a change to account for this, modifying our function to take another parameter:

```
## Program number 5
def averaging_filter(inputs, k):
    result = []
    for i in range(len(inputs)-k+1):
        total = 0.0
        for index in range(i, i+k):
            total = total + inputs[index]
        result.append(total / k)
    return result
```

```
input_list = [100, 27, 93, 94, 107, 10]
print(averaging_filter(input_list, 3))

input_list2 = [94, 38, 96, 20, 18, 100, 108]
print(averaging_filter(input_list2, 3))
```

Back to Top

Here, we've increased the generality of a piece of code, but that function has become harder to read. Here is where a well-placed comment might help, describing what the function is designed to do.

You might also notice that using a loop to compute the total is what the Python built-in `sum` function is designed to do. We can use it to make the code more concise:

```
## Program number 6
def averaging_filter(inputs, k):
    result = []
    for i in range(len(inputs)-k+1):
        total = sum(inputs[i: i+k])
        result.append(total / k)
    return result

input_list = [100, 27, 93, 94, 107, 10]
print(averaging_filter(input_list, 3))

input_list2 = [94, 38, 96, 20, 18, 100, 108]
print(averaging_filter(input_list2, 3))
```

At this point, you may notice that our for-loop is calculating and appending a single element. We could turn this into a for-loop comprehension as follows:

```
## Program number 7
def averaging_filter(inputs, k):
    """
    Calculates the averages of every sequence of k adjacent numbers in a list.

    Parameters:
        inputs: list of numbers; len(inputs) >= k
        k: integer number of elements to find the average of

    Returns:
        List of average numbers of size len(inputs)-k+1
        Ex: averaging_filter([2, 4, 6, 8], 2) -> [3.0, 5.0, 7.0]
    """
    return [sum(inputs[i:i+k]) / k for i in range(len(inputs)-k+1)]

input_list = [100, 27, 93, 94, 107, 10]
print(averaging_filter(input_list, 3))

input_list2 = [94, 38, 96, 20, 18, 100, 108]
print(averaging_filter(input_list2, 3))
```

With a descriptive docstring, the purpose and use of the function is clear without needing to look at the example code below. While the code is now only one line long, it may actually be harder to read. Determining when making code concise has gone too far is also something to watch out far.

Back to Top

Through this example, we've tried to illustrate iteratively improving on a program to improve it not in terms of correctness, but in terms of style, according to the principles laid out above. Over the course of this section, we have moved from a program that is very difficult to read, understand, modify, extend, and debug; to a program that is much easier in most of these aspects.

Importantly, most of the changes we made between different versions of the program were small. You can often avoid the dangerous cycle of endlessly fixing problems by iteratively making small improvements and testing them to make sure the program still works instead of trying to make many changes all at once.

4.2) Design and Planning

It is often the case that experienced programmers are able to envision elegant solutions to problems, while beginning programmers have difficulty doing so. Do not despair. **This has very little to do with talent or ability**; it's all about experience, and it's a skill that can be learned (although it takes lots of time and practice!).

It's always important to have a plan first. Here we'll discuss a principled way to go about formulating a plan. I like to break things down into roughly the following steps:[4]

1. Understand the Problem

   - What problem are you trying to solve?
   - What is the input, and what is the output? How can we represent these in Python?
   - What are some example input/output relationships? Come up with a few specific examples you can use to test later.

2. Make A Plan

   - Look for the connection between the input and the output. What are the high-level operations that need to be performed to produce the output? How can you construct the output using those operations? How can you test the operations?
   - What information, beyond the inputs, will you need to keep track of? What types of Python objects are useful to that end?
   - Have you read or written a related program before? If so, pieces of that solution might be helpful here.
   - Can you break the problem down into simpler cases? If you can't immediately solve the proposed problem, try first solving a subpart of the problem, or a related but simpler problem (sometimes, a more general or more specific case).
   - Does your plan make use of all the inputs? Does it produce all the proper outputs?

3. Implement the Plan

   - Translate your plan into Python.
   - You may be able to implement several of the important high-level operations as individual functions.
   - As you are going, consider the style guidelines above. If you find yourself repeating a computation, you may want to reorganize now (rather than at the end).
   - As you are going, check each step. Can you clearly see that the step is correct? Can you prove that it is correct? How can you use that result as part of the larger program?

4. Look Back

   - Test both for correctness and style.

- For each of the test cases you constructued earlier, run it and make sure you see the result you expect. Are there other test cases you should consider?
- Could you have solved the problem a different way? If so, what are the benefits and drawbacks of the s Back to Top chose?
- Can you use the result for some other problem? Can you use similar programming structures for some other problem?
- Look for opportunities to improve the style of your code according to the rules discussed in the previous section.
  - Are the names of your functions and variables concise and descriptive?
  - Are you repeating a computation anywhere?
  - Are there functions or other pieces of your code that could be generalized?

A big part of this outline is about breaking a big problem down into smaller, more manageable pieces.

It's worth noting that this still does not necessarily make formulating a plan *easy*, but, over time, the process will become second nature.

4.2.1) Testing and Debugging

As you program outside of the course (with its helpful autograder), testing will become an important part of making sure your code is correct. It is difficult to verify that code is correct without creating test cases, and it is important to choose good ones. However, it is not always easy to know what constitutes a good test case. Here are two guidelines that come pretty close to being universal:

1. Where possible, your test cases should cover every possible *branch* in your code.

   - If you have a conditional, make sure that you have at least one test case that executes each block associated with that conditional.
   - If you have a `while` loop, make sure you have one test case that skips the loop entirely, one that goes through the loop once, and one that goes through the loop multiple times.
   - If you have not yet written the code, you can still think about similar things: which types of input are likely to be handled differently? Make sure you include those types of inputs in your tests.

2. Whenever possible, each of your tests should try to test for *one particular thing*.

   - If one of your tests fails, it should not only tell you that *something* is wrong, but it should help you find *what part of your program* is wrong.
   - To this end, it helps to test not only your overall program, but any helper functions you defined along the way.

Some programmers find it useful to use test-driven development, which involves writing test cases for your function *before* you implement it. Regardless of what particular method you use, it is important to have a concrete idea of what your program is supposed to do and implement tests to check for various failure methods.

5) Summary

In this reading, we learned a bit about recursion, and saw some example recursive functions. We also introduced some more functionality from the powerful NumPy library. For the bulk of the time, we talked in depth about style and strategy for designing programs, which you will have a chance to apply in this unit's exercises.

In the next unit, we will introduce the notion of *classes*, which provide a way to create custom types of Python objects.

Next Exercise: Ackermann

Back to exercises

Back to Top

Footnotes

[1] The restriction that `array` objects can only contain values of the same type is somewhat limiting, but for many applications where we are dealing exclusively with numbers, this tradeoff is worth the incredible gains we make in terms of speed.

[2] If `numpy` feels like too long of a name for you to reference throughout your code, you can import it like `import numpy as np` instead, which renames it to `np` (or any name of your choosing).

[3] This image, from *Thinking Forth*, is licensed under a Creative Commons BY-NC-SA 2.0 Generic License.

[4] The high-level outline here comes from George Polya's book *How To Solve It*.