Dividing Python Code Into Functions
Version 1.1, Spring Term 2024

Craig Partridge

Colorado State University
craig.partridge@colostate.edu

Abstract. Guidance on dividing Python code into functions is provided.

Keywords: Python · functions · modules.

1 Introduction

The topic of dividing code into functions (and functions' relative, methods for Python classes) is generally considered a mysterious art. As a result, when teaching undergraduates, we typically provide platitudes but a lack of concrete guidelines.

The goal of this essay is to provide actionable guidance to undergraduates about how to divide their Python code into functions.

2 Five Rules

In the interests of ensuring this guidance is actionable, it starts with five "rules". Rules is in quotes because, alas, each rule will occasionally have to be broken due to special software requirements. But, with the possible exception of the injunction to "do one thing" in a function, breaking these rules is usually a sign of poor software design.

The five rules are presented briefly here, in the order of their ease of application, and then in detailed sections following this section.

2.1 Rule 1: Code that appears in more than one place should be a function

Code that appears in multiple places should be consolidated into a function. Simply from a software engineering perspective, this rule makes a lot of sense. Suppose the piece of code has a bug in it, or needs revision? It is simpler and more reliable to change that code in *one* place, namely the function, than to have to change multiple scattered instances of the same code.

2.2 Rule 2: A function should be less than 30 lines long.

This rule is derived from two perspectives. Functions should be focused on solving a single problem (see rule 5) and thus, one hopes, brief. And, a function should be easy for another programmer to understand[1], which means the function should easily fit on a single screen or sheet of paper. Having to scroll (or flip pages) causes the reader to lose context and makes it harder to understand the code.

2.3 Rule 3: A function should have three or fewer levels of conditional control

This rule complements rule 2. If we want to keep function code short, to ease understanding, we should also limit its complexity. This rule limits complexity by limiting the number of levels of conditional control, where conditional control includes loops, conditional statements, conditional expressions and exception wrappers.

2.4 Rule 4: Create a function when it improves readability

This point is best illustrated with an example. Suppose you are writing some mathematical software that only works with non-negative numbers. The code will likely have a lot of tests to ensure values are non-negative. You could simply write:

```
if (value >= 0):
    # do something
```

or you could write a simple function:

```
def non_negative(value):
    return (value>=0)
```

and then write the test as:

```
if (non_negative(value)):
    # do something
```

The version using `non_negative()` is clearer. It also meets the requirement of Rule 1 to create a function when code is repeated. Even though >= 0 may seem too trivial to make into a function, by improving readability, you are also avoiding errors where you accidentally type > 0 somewhere in your code when you meant >= 0

[1] All code should be written on the assumption that someone else is going to read it.

2.5 Rule 5: Do one thing.

The rule to "do one thing" in a function goes back a long way in computing.[2] Unfortunately it is sometimes hard to apply in practice, because when applied too zealously, it results in code that is harder to understand. This challenge is discussed in detail below.

3 Rule 1: Code that appears in more than one place should be a function

Taking a chunk of code that appears multiple times and converting it into a function is, for most software engineers, a needs-no-thought-just-do-it action. The one place where one may get some disagreement is when to convert small pieces of code: an expression or a couple of lines of code into a function. What follows are a couple of suggestions for such situations.

3.1 Use functions to avoid creating bugs

There are lots of cases of a few lines of repeated code that turn out to be easy to mistype. An example is swapping two values in a list (or in other languages, swapping pointers). The code is deceptively simple:

```
temp = lst[i]
lst[i] = lst[j]
lst[j] = temp
```

Yet programmers routinely flub it. A classic is to accidentally invert `i` and `j` in the middle line and type:

```
temp = lst[i]
lst[j] = lst[i]
lst[j] = temp
```

It is easy to do and surprisingly hard to find.

So there's no shame in creating a function `swap` to solve this problem:

```
def swap(i, j):
    return j, i


lst[i],lst[j] = swap(lst[i], lst[j])
```

[2] Many websites attribute the rule to Robert C. Martin's *Clean Code*[5], but you will also find it said in similar ways in the works of software pioneers such as Niklaus Wirth.

3.2 Use functions to show intent

Another reason to use a function is to show why your code is doing something. For instance, suppose you are working with 16-bit numbers and routinely have some code along the lines of the following:

```
HI_BIT = 0x8000
if (value & HI_BIT) != 0:
```

Is this code looking to see if the value is negative? Or is setting the high bit deemed a case of an overflow or exceeding a maximum desired value? This code does not tell you. Whereas defining a function `overflow` that tested the high bit would.

You can view this point as repeating the readability rule (section 6) or as a slightly richer goal. In either case, the point is that repeated code as small as an expression may benefit from being made into a function.

4 Rule 2: A function should be less than 30 lines long

The motivation for this rule is to make it easy for someone to see the entire function on a screen or page, so they can immediately see how parts interact, without having to bounce from screen to screen or page to page.

In the early days of computing, screens could only hold 24 lines of code and that was often considered the informal function length limit. Screens continue to grow in size, so one might argue the screen derived limit should grow too, but my experience is that most folks find it easiest to comprehend a modest number of lines, so I'm sticking with 30.[3]

What if the code really needs to be longer? The most common complaint about this rule is that a programmer feels their code really needs to be longer. That may be true. But in many cases, it may be a case of using an older version of Python or a failure to look for libraries.

For instance, CSV (Comma Separated Value) files used to be a nuisance to parse in Python. Consider parsing a line of several values, separated by commas, and checking that each value is correctly formatted and has a sane value. The code can quickly grow long and it is not clear how to break it up (e.g., splitting the parser into two functions `parse_first_five_fields` and `parse_last_five_fields` does not substantially improve the readability of the code). But now there are good Python libraries such as `csv` and `pandas` that make parsing CSV files much easier and enable compact functions.

[3] Google's Python Style guide, section 3.18, puts the limit at 40 lines.[2]

Is 30 lines too long? There is a school of software design that believes functions should be small, at most a few lines. See, for instance, Martin's chapter on functions in *Clean Code*. I have seen a number of programs and systems written in this style and know some programmers who use it. My assessment is that it is a style of programming that, in the hands of a skilled programmer, leads to extraordinary productivity. The programmers who use small functions are often capable of producing large amounts of code with very few bugs. The approach appears especially well-suited to file processing and web interfaces. The one downside is that the high degree of nesting of functions can lead to inefficient code, as complex operations are smeared across multiple functions, such that the inefficiencies are hidden from the programmer even when using a performance profiler.

5 Rule 3: A function should have three or fewer levels of conditional control

Just as limiting the number of lines in a function and stating a function should do one thing seek to make functions simpler and easier to understand, limiting the levels of conditional control seeks to manage the complexity of functions. Intuitively, a function whose core is code such as:

```
if (condition1):
    if (condition1_1):
        while (condition1_1_1):
            # do something
    else:
            # do something different
else: # implicitly condition2 is true
    if (condition2_1):
        if (condition_2_1_1):
            # do even something else
        else:
            # do still something else
    else:
        # and yet another posibility
```

is hard to understand and should be broken up into more easily understood chunks.[4] This rule seeks to avoid putting too much complexity in one function.

Conditional control is a loop (`for` or `while`), a conditional statement (`if` or `match`), a conditional expression (`x if C else y`), or an exception wrapper (`try/except/else/finally`). Observe that `while` loops that contain a conditional that updates the `while`'s condition, such as

[4] A coding tip that can also help reduce how complex the code is to read is to choose the condition such that the shorter block of code comes first. So the code after the `if` is shorter than the code after the `else`. This makes it easier for the reader to see that there is an `else` clause balancing the `if`.

```
while (i < limit):
    ...
    if (condition):
        i += 1
    else:
        i += 2
```

are *two* controls, the `while` and the `if/else`. That's true even if the loop is rewritten to use a conditional expression, as in

```
while (i < limit):
    ...
    i = (i+1) if (condition) else (i+2)
```

for the simple reason that the complexity of the loop has not changed.[5]

Why is three the right limit? One might wonder why three levels of control is the right limit. The Goldilocks-style answer is that three felt right to me (and there are those who disagree). Two levels of control was too few. Four levels of control was clearly too much.

The example I find particularly insightful is to consider the common situation of a three-dimensional matrix representing region, such as a body of water. You are simulating changing conditions, such as temperature variations, in the matrix based on events. A natural inclination is to write code like this

```
for i in range(0, x);
    for j in range(0, y):
        for k in range(0, z):
            new_m[i][j][k] = update(old_m, i, j, k, ev)
```

where `update` uses `i, k, j` as the center of a region in `old_m` that is affected by an event `ev`.

There is no obvious way to usefully break this routine into additional functions, which would be required if the levels of control were limited to two. Furthermore, the code is short, easy to read, and clearly doing a single thing. Reasoning along these lines led to the limit being three.

I also observed that some classic algorithms, such as Boyer-Moore and Knuth-Morris-Pratt, lend themselves to an implementation of a loop combined with two levels of `if/else`. Again this suggests a limit of three levels of control made sense.

Spreading complexity? The one downside to this rule is that it can lead to code such as this:

[5] Use of the so-called walrus operator (e.g. `(i := i+1)`) may or may not add a level of control depending on context.

```
def lots_of_work(argument1, argument2):
    _lots_of_work_part1(argument1, argument2)
    _lots_of_work_part2(argument1, argument2)
    return _lots_of_work_part3(argument1, argument2)
```

Clearly this code does not improve readability. It probably just smears the complexity of the routine across three parts. It makes the function harder to understand. So don't do it. Instead, concentrate on dividing the work into functions that do distinct tasks and can be named to reflect the tasks they perform.

6 Rule 4: Create a function when it improves readability

Don't comment bad code, rewrite it. Kernighan and Pike[3]

Writing code that is easy to read is important. Since most coding is a collaborative activity among teams, writing code that teammates can understand and, if necessary, change is important. Thus, using functions to make your code more readable matters.

A couple of examples illustrate the point. First, consider testing to determine if a number is odd. You can do this a number of ways, including

```
if (value % 2) == 1:
    # do something odd

LOW_BIT = 0x1
if (value & LOW_BIT) == LOW_BIT:
    # do something odd
```

Now imagine your colleague is reading your code and comes across one of these lines. How do they know that the purpose of the code is to test if the value is odd?

One might suggest including a comment such as `# testing for odd` solves the problem. But a programmer should not rely on a comment to make up for code that is not clear. Rather you should define a function whose name describes its purpose and use that, as in:

```
def is_odd(val):
    return (val % 2) == 1

if is_odd(value):
    # do something odd
```

As a second example, consider looking at the top of a stack. Most programmers create a stack using Python lists and the built-in methods `append` and `pop`. But there's no built-in method `top` to peek at the top of the stack.

The easy answer is to just look at the top of the stack. The following code implements such an approach and will print the value "orange":

```
stack = []
stack.append(``apple'')
stack.append(``banana'')
stack.append(``orange'')
print(stack[-1])  # print top of the stack
```

This approach is easier because it doesn't require you to create a new class for a stack and define the missing method `top`. But the comment shows it is the wrong approach. Clearly a programmer encountering `stack[-1]` will have to think for a moment about why it is being accessed and be thankful for the explanatory comment. Observe how the following code obviates the need for a comment:

```
stack.append(``apple'')
stack.append(``banana'')
stack.append(``orange'')
print(stack.top())
```

Adding the method is the right approach.

7 Rule 5: Do One Thing

The rule to "do one thing" in a function is probably the most widely cited guidance for creating functions. Unfortunately, it is also easy to misapply.

The challenge is what is "one thing"? Two examples illustrate the point.

Suppose you were writing a function `palindrome` that checked to see if a string or list was a palindrome.[6] Further suppose, you sketched out the function as having two steps:

1. Check argument is valid (e.g. not `None`); and
2. Check whether the input is a palindrome.

Following the "do one thing" rule, you might then be tempted to write the following function:

```
def palindrome(p):
    if (p is None):
        return False
    return _palindrome_(p)
```

The idea behind this code is that checking the argument is "a thing" and thus you should do that, and then call another function to actually test for the palindrome. While this code may look odd, some programmers do write code similar to this example.

On the flip side, suppose you chose to implement the Boyer-Moore pattern matching algorithm. (Boyer-Moore precomputes a table, such that, when the pattern and text don't match, the algorithm uses the table to shift position in the pattern to a value where the text could match. See [1]). Now suppose, on the

[6] A palindrome is a word, phrase or list that reads the same backwards and forwards, e.g. *madam*

notion that the function does one thing, namely implements the Boyer-Moore algorithm, you wrote this:

```
 1 def bm_find_pat(text, pattern):
 2     if (text is None) or (pattern is None):
 3         return -1
 4
 5     loc_tab = [-1]*256
 6     for i in range(0, len(pattern)):
 7         loc_tab[pattern[i]] = i
 8
 9     pat_len = len(pattern)
10     text_len = len(text)
11     i = pat_len-1  # tracks in text
12     j = pat_len-1  # j tracks in pattern
13
14     while i < text_len:
15         if text[i] == pattern[j]:
16             if j == 0:
17                 return i
18             else:  # haven't compared all chars
19                 i = i-1
20                 j = j-1
21         else:  # chars don't match
22             i = i + pat_len - min(j, loc_tab[text[i]]+1)
23             j = pat_len - 1
24     return -1
```

You might observe that the function is only 24 lines long and thus fits the rules well.

My view is that neither function embodies the spirit of the ”do one thing” rule. I believe the spirit of the “do one thing” rule is more fully expressed as: do one *major* thing *towards solving the problem.*

By that logic, the version of `palindrome` falls short because a trivial argument check does not move substantially towards testing for a palindrome. The code in `_palindrome_` should be in `palindrome` proper.

The version of `bm_find_pat` fails in the other direction. It does (1) a simple argument check [lines 2-3]; (2) builds the auxiliary table [lines 5-7]; and then (3) actually does the Boyer-Moore search [lines 9-24]. It is doing at least two non-trivial things without placing either of them in a separate function.

8 An Extended Example

As an extended example, let’s look at the Boyer-Moore function in section 7 and seek to improve it to better fit the five rules.

I find it useful to seek to examine code I am trying to restructure with fresh eyes (either trying to revitalize my own perspective or asking a friend to read it). The fresh assessment is:

– The original code does three things: check inputs, generate a table and then run a complex loop with two inter-related indicies (`i, j`) one of which is driven by the table.
– The return values are messy. The function returns `None` if the inputs are bad, a positive integer offset if the pattern is found and -1 if the pattern is not found. This sort of mixed error code is properly condemned by Maguire ([4], p. 104).[7] Another hazard is that while -1 is an improper index in most programming languages, in Python it has a well-defined meaning as the index of the last element in a list.
– The input checking leaves something to be desired. Notably what if the length of either the text or pattern is zero? Text is perhaps obvious, there's no match to the pattern. But does a zero-length pattern match? It isn't clear.

The code below seeks to address these issues. It separates out the complex table-driven code into two subroutines: one to build the table and one to update `i,j`. Following Martin ([5], pp. 103-105), the code no longer returns a result when inputs are bad but throws an exception. If the pattern does not match, the return value is now an invalid index (`None`). And the input checking is improved.

```
def bm_make_loc_tab(pattern):
    loc_tab = [-1]*256
    for i in range(0, len(pattern)):
        loc_tab[pattern[i]] = i
    return loc_tab


def bm_next_ij(i, j, match, pattern_len, offset):
    if (match):
        return i-1, j-1
    i = i + pattern_len - min(j, offset+1)
    j = pattern_len - 1
    return i, j


def find_pat(text, pattern):
    assert ((text is not None) and (pattern is not None))
    assert (len(pattern) > 0)
    loc_tab = bm_make_loc_tab(pattern)
    pat_len = len(pattern)
```

[7] Maguire notes muddled return codes are often a case of mission creep, which is precisely what happened here. I wrote the code to match the algorithm in a textbook and then added argument checking.

```
    text_len = len(text)
    i = pat_len-1  # tracks in text
    j = pat_len-1  # j tracks in pattern
    while i < text_len:
        if (match := (text[i] == pattern[j])) and (j == 0):
            return i
        i, j = bm_next_ij(i, j, match, pat_len, loc_tab[text[i]])
    return None
```

Viewed from the perspective of the rules, this new version is an improvement. The main function is smaller (13 lines rather than 24 lines) and it has two levels of conditional control, down from three levels.

But the new version also has some challenges. Dividing the code at the point of deciding how to update `i,j` causes `bm_next_ij` to have *five* arguments. Good software practice generally frowns on functions with more than three arguments. Good software practice also suggests it is important for the function arguments to reveal what information the function requires to do its computation.[8]. A reasonable middle-point in this dilemma is to conclude that `bm_next_ij` is the wrong logical place to divide code between functions.

One possible solution is the revised code below. It puts the entire complex loop in its own function. So the logic of the main routine becomes (1) check arguments; (2) compute table; and (3) call routine to do search, with the computed table. There's a clear progression towards a result and no routine is longer than 15 lines.

```
def make_loc_tab(pattern):
    loc_tab = [-1]*256
    for i in range(0, len(pattern)):
        loc_tab[pattern[i]] = i
    return loc_tab

def bm_find_pat(text, pattern, loc_tab):
    pat_len = len(pattern)
    text_len = len(text)
    i = pat_len-1  # i tracks in text
    j = pat_len-1  # j tracks in pattern
    while i < text_len:
        if text[i] == pattern[j]:
            if j == 0:
```

[8] This summary intentionally ducks a massive programming style argument. Martin proposes that functions should have no more than three and preferably zero arguments, and ducks the issues that zero arguments cause, such as programming with side-effects ([5], p. 40). Maguire would likely argue that I'm cheating by using the walrus assignment `:=` to avoiding passing `text` and `pattern` to `bm_next_ij` and point out that passing boolean arguments usually means the function implementation is confused ([4], p. 102-103). Because of this disagreement among experts there is no rule in this document about the number of arguments to a function.

```
                return i
            else:  # haven't compared all chars
                i = i-1
                j = j-1
        else:  # chars don't match
            i = i + pat_len - min(j, loc_tab[text[i]]+1)
            j = pat_len - 1
    return None


def find_pat(text, pattern):
    assert ((text is not None) and (pattern is not None))
    assert (len(pattern) > 0)
    loc_tab = make_loc_tab(pattern)
    return bm_find_pat(text,pattern,loc_tab)
```

References

1. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Commun. ACM **20**(10), 762–772 (oct 1977). https://doi.org/10.1145/359842.359859, https://doi.org/10.1145/359842.359859
2. Google: Google python style guide. https://google.github.io/styleguide/pyguide.html, [Accessed 11-02-2024]
3. Kernighan, B.W., Pike, R.: The practice of programming. Addison-Wesley professional computing series, Addison Wesley, Boston, MA (Feb 1999)
4. Maguire, S.: Writing solid code. Microsoft Press, Redmond, WA (Jan 1993)
5. Martin, R.C.: Clean code. Prentice Hall, Philadelphia, PA (Aug 2008)