Basic principles of programming

Lars Arvestad

2021-02-02

You typically learn programming by starting with small programming exercises to develop the fundamentals of programming. To solve “real” programming assignments, with practical relevance, significantly larger programs are needed that are harder to comprehend. Such programs need a good *program structure*, i.e., a good logical structure where the assignment is broken up in smaller pieces using functions, classes, and modules. As a beginner, it can be difficult to get started with with larger programs just because the task is daunting. What is a good first attempt at breaking down the problem in smaller pieces and how to you decide what parts you need?

Finding the right program structure is an art. Experienced programmers often find the distinction between good and bad code obvious, but it can be hard to teach that sense of right and wrong. It can also be hard to make active and planned design decisions, even if one has a good feeling for what is *good code*. You develop your sense of good coding by practicing, following others’ good examples, and writing code that you improve iteratively. This takes time.

One can also note that also good and experienced programmers can have a bad sense for coding and write programs that few can read without effort. That a person has written large and well-functioning software is no guarantee for the code to be well-written.

A suggest program structure does not have to be perfect to be useful. Also a suboptimal structure may be a good start for a prototype that yield insights useful for an even better structure. You learn by mistakes! Modern development practices in software engineering are about identifying bad design decisions early and immediately rectify them to avoid paying for the mistakes later in the project. Restructuring code is sometimes called *refactoring*, and aims to improve code step by step. This is such a central theme in software engineering that modern development environments have automatic or semiautomatic tools for common refactoring actions.

Good program structure is also important for being able to understand code. By avoiding bad design decisions you make your code more accessible for others, but also your future self! If you have a hard time to understand your thinking from three months earlier, how hard is it not for others to read your code? Neat and easy-to-read code is furthermore easier to find bugs in. Google has a style guide[1] for the C++ language, in which one aim is to “Optimize for the reader, not writer”. That is, it is more important the code is easy to read than it is to write, and it is clearly justified: Google employees put in much more time reading than writing code.

When I asked colleagues about where one reads about structuring code, several have mentioned the book *Code Complete* by Steve McConnell. This is an excellent recommendation, but probably not for beginners. And there is a lot of material for the

[1]Se ”Goals for Style Guide” i *Google C++ Style Guide*: `https://google.github.io/styleguide/cppguide.html`

more experienced programmer, but I think beginners will have a hard time finding simple guidelines. To help beginners get going with coding, I have collected some principles for programming. These principles are by no means rules, and there are probably many programmers that do not agree with them, but I believe they are a good start for beginners.

Even if the principles are written with Python in mind, they are quite general and applicable to other programming languages. Bear in mind that programming languages have different cultures and expectations for how to express one self and design software. These differences are often due to various strengths and weaknesses in languages and one should follow the conventions for the language you are using, not the conventions you are used to. A nice way to learn about preferred programming styles is to watch videos with presentations from conferences and workshops organized by various interest groups.

The central theme with the principles in this text is that *code should be readable*. By making it easy to read your code, whether it is about using blank lines, choosing identifiers, or deciding what functions to implement, it will be easier to figure out how computations are supposed to work.

Principles for program structure

- **A function should fit a screen page.** This enables the reader to get a quick overview of a self-sufficient piece of code without having to burden a busy brain with details like whether and how a variable has been initialized or what parameters a function takes. With a full overview of a function it is often easy to determine what a function does (or is supposed to do) without understanding its details.

  If a function has grown larger than a screen page, try to identify code blocks that can be factored out as independent units and make those blocks to new helper functions.

- **Use functions to group lines of code.** Programmers often create structure in long blocks of code by inserting blank lines and comments, but functions are a stronger tool for imposing structure. The purpose of a block of code and its dependencies are often made more clear of one uses several short functions instead of one long function partitioned in blocks of code. Yet another advantage with functions is that they are easier to verify in unit testing.

- **Separate *calculation* from *interaction*.** A function that calculates should not have print and input statements (or similar). As an example: if you write an equation solver with a lot of diagnostic output, its utility in other projects is reduced. Similarly, a function can be made useless if it interactively asks a user for input instead of taking a parameter for that same information. See Fig 1 for a small example.

- **A function should do one thing and do it well.** Consider the example of writing a small program to compute and present some statistics from several datasets. Such a task can be solved by writing a single block of code, without functions, but it the code becomes easier to understand if you have:

  - a function for reading data,

```
def birthdate():
    swed_id = input()
    parts = swed_id.split('-')
    return part[0]
```

(a) A bad example.

```
def birthdate(swed_id):
    parts = swed_id.split('-')
    return part[0]
```

(b) A better example.

Figur 1: Two examples of small functions for extracting the birth date out of a Swedish social security number. (a) A bad example with at least two mistakes: the function breaks the principle of separating user interaction and calculation, and the principle of of providing data by parameters. (b) A better solution: you assume that the social security number is entered somewhere else. This solution is generally more useful.

  - a function for calculating the statistics, and
  - a function for looping over input files and summarize/present the results.

  This program structure has several advantages.

  - The code will be easier to overview when it is broken down in smaller logical units.
  - It is easy to read a single short function and figure out what it does.
  - It is easy to verify that functions do what they are supposed to do, when their purpose an utility is limited and comprehensible.

- **Choose informative identifiers.** Naming functions to `start`, `compute`, or `f` is bad practice because the identifier gives no hint about the purpose of the function. Identifiers like `remove_outliers`, `newton_raphson`, and `compute_integral` on the other hand communicate quite clearly what to be done or even *how* it should be done (Newton-Raphson is the name of an algorithm for finding solutions to equations).

  It can be tempting to use abbreviations for variable identifier, but that should then be established abbreviations. Also try to avoid identifiers that are already used in Python's standard modules and popular third-party modules. It is, for example, inadvisable to use the identifier `map`, since that is a function in Python. It is allowed to redefine functions and classes in Python (but not the reserved words[2]), but one should be careful to avoid unfortunate surprises.

  A sign that a variable identifier is not good is that you need a comment to explain the variable. For example:

  ```
  nc = 1.7        # normalization constant
  ```

  In this case you are better of calling the variable `normalization_constant`.

  There are occasions when an informative identifier is not so important. It is for example accepted practice that `main` is the function that serves as the starting point of a program. When writing a function that takes a function as a parameter, it often makes sense to name the parameter `f` (as an abbreviation of "function").

[2]see *Keywords* in the chapter *Lexical analysis* in Python's reference manual, `https://docs.python.org/3/reference/lexical_analysis.html`

Code changes and it is the important to remember that identifiers might need to change too. If a function is named `print_integral`, but it doesn't do any printing (as it did in a first draft), it causes confusion. Maybe it should be called `compute_integral` instead?

- **A function should only depend on its parameters.** It should be possible to understand a function without looking at other parts of a program. By letting a function access global variables, a function's dependencies become less clear and a reader may need to carefully study the whole function just to figure out what the true input of the function is. It is almost always inappropriate to manipulate global variables (i.e., use the reserved word **`global`**) because it makes it hard to understand when a variable changes.

  There are occasions when using global variables is defensible, but the broadly accepted principle is to avoid globals.

- **Pay attention to what your functions return.** To have a **`return`** statement (or several!) makes your function clear and easy to read. Remember that you can have multiple return values!

  There are however various mistakes to avoid regarding return values:

  - Do not return numerical values as strings, even if you know that they will be put in a string later on. It is more "future proof" to return a numerical value as a numerical value.
  - Do not return "special values" to signify errors or problems. Instead, use exceptions because that is the established tool for handling errors and issues. If you write code to iteratively solve an equation, then `return 1000000` is a bad notification for "did not converge". It is then better to write `raise Exception("Did not converge")`.

    It is, in general, problematic with values carrying a special meaning that the user of a function has to keep track of. There are some circumstances when it is acceptable; for example is `None` a value that can signify lack of results.
  - Do not fund your own ways of packaging several values into a return value. In particular, it is not good to store numerical values in a string and later extract them when needed in computations. Problems with conversion easily appear, and it is also relative slow to make such conversions. There is an established and practical way to return several values in return statement: just write `return value1, value2, value3` and similar constructs. Read about "multiple return" values in the course book!

- **Every function should have a docstring.**

- **Variables are better than literals.** A literal like "10" does not hint about what the value represents. Some meaning is given to the reader if the literal appears in a line of code like `n_iterations = 10`. The code becomes more readable if it refers to variables like `n_iterations` than if the corresponding literal is used, even if it is well known. Another advantage is that your program becomes easier to tweak: instead of replacing all (relevant) occurrences of the literal 10, we can simply change one assignment.

Indications of problems

It is easy to ”accidentally” write bad code, even with a good understanding of good programming practice. Being aware of some simple indicators of bad code can be helpful, so let us look at some such indicators.

- **Deep indentation** indicates an unnecessarily complicated implementation of nested loops and/or conditionals. Code becomes hard to read when there are many combinations of conditions to keep track of. Try to ”flatten” the code. Figure 2 has an example.

  A common cause of deep indentation is that you have several logic cases — create functions for those cases. That way, the case analysis becomes clear and the various functions indicate what should happen in the different cases.

- **Do functions miss parameters?** The purpose of functions is to take some data and make a computation. The computation does not need to be mathematical and can be any kind of treatment of data. It is much less common that functions completely lack parameters. There are valid exception, of course, like printing a menu or initializing a data structure. It is not uncommon that beginners find it convenient to put data in one or several global variables and have functions work with them. Such a construct makes it hard to see how functions depend on each other; you have to read the functions very carefully to identify program flow.

- **Does a function have many parameters?** That may indicate that you are putting too many tasks into a single function. That may make the function hard to code, but also that it becomes hard to correct errors in the code. The function becomes harder to use, since there are more parameters to understand.

- **Is it hard to write unit tests?** That may be a sign that you need to restructure your code. You may not have enough simple functions, may lack return values, or have may have too strong dependencies of global variables.

Think modularly

If you have written useful code, then it is not surprising if others want to use or continue building on your work. It is then good if the central functionality, what makes your code interesting, is also available as a module. It is a pity if the functionality is *only* available when running your program, but not accessible as a module.

The value of modules is true for all programming languages, but it is worth noting that some modern programming languages, and Python is such an example, makes sharing code and modules easier than others. The key is to have a common and public repository of packages and make sharing easy. Take a look at the site `pypi.org`! There are thousands of packages and modules available there, and you can share your code there too, if you follow some guidelines.

```
def get_valid_records(data_records):
    good_records = []
    for item in data_records:
        if item.x >= 0:
            if item.y > 0:
                if item.val > 0:
                    if item.location:
                        good_records.append(item)
        elif item.y < 0:
            if item.val > 0:
                if item.location:
                    good_records.append(item)
    return good_records
```

(a) An example of ”deep indentation”. Can you easily read out what makes data record ”good”?

```
def get_valid_records(data_records):
    good_records = []
    for item in data_records:
        if record_quality(good_records, item):
            good_records.append(item)
    return good_records

def record_quality(good_records, item):
    '''Return True for a "good" item, False otherwise.'''
    if not item.location:
        return False

    if item.x >= 0 and item.y >= 0 and item.val > 0:
        return True
    elif item.x < 0 and item.y < 0 and item.val < 0:
        return True
    else:
        return False
```

(b) This code is equivalent to (a), but structured differently. Iteration and handling of co-de cases are separated from classification of records into good and bad. The conditions in `record_quality` are not nested, but kept together (with **and**) to clearly show when a data record is good. An early decision is taken if `item.location` happens to be empty.

Figur 2: Two ways of writing code for iterating through data records and collecting those that fulfill some necessary conditions. Which variant is easier to read, (a) or (b)?