Course Notes: Designing Programs

You are not logged in.

If you are a current student, please Log In for full access to the web site.
Note that this link will take you to an external site (`https://shimmer.csail.mit.edu`) to authenticate, and then you will be redirected back to this page.

> These notes are new for the Spring 2020 semester, and they are very much a work in progress, and they'll likely be updated/augmented/refined throughout the semester. If you notice mistakes (big or small), if you have questions, if anything is unclear, if there are things not covered here that you'd like to see covered, or if you have any other suggestions; please get in touch either by posting to the course notes section of the forum, or by email to `6.009-staff@mit.edu`.

Table of Contents

1) Introduction

It is often the case that experienced programmers are able to envision elegant solutions to problems before writing any code, whereas beginning programmers may have a more difficult time. There is no need to be discouraged by this! It's important to remember that this has very little to do with intrinsic 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!).

In part, experienced programmers are able to arrive at correct, efficient solutions more quickly because experienced developers have more "tools in their toolbox," so to speak; by having seen and solved more problems, they have a larger collection of past experience upon which to draw for experience. One of the things we'll try to do in 6.009 is to expose you to a variety of different problems (as well as specific techniques to solve different kinds of problems), in an effort to help you build up that collection of experiences as well.

2) A General Framework for Program Design

Beyond the specific experiences one acquires with time, it can also be helpful to have in mind a general framework for thinking about designing programs.

In order to make this process as smooth and efficient as possible, it is a good idea to **avoid the temptation to write code until you have a plan in mind**. We all know from experience that it can be tempting to start writing code immediately when

presented with a new problem, but spending a little bit of time constructing a plan and designing a solution can save a lot of time and heartache down the line.

Some people like to formalize this more or less than others, but one way to think about the process is to break it down into the four steps outlined in the following sections (inspired by based on George Polya's wonderful book, *How To Solve It* (Princeton University Press, 1945), which some regard as the best book ever written about teaching and problem solving).

Note that a big part of this outline is about breaking a big problem down into smaller, more manageable pieces (which can often be implemented and tested independently).

It's also worth noting that following this process still does not necessarily make formulating a plan for any given program *easy* in an absolute sense, but the hope is that it makes the process *easier* than it otherwise would be; and, over time, the process will become second nature, but for now it can still be very helpful work through the outline below in detail every time you design and implement a program.

2.1) Understand the Problem

An important first step in designing a program to solve a problem is making sure you actually understand the problem you're trying to solve. As such, a good place to start is by asking yourself questions such as the following:

- 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 small, specific examples you can use to test later.
    - How do you, a human, solve those simple cases? Do those steps generalize? How can we break that down into steps small enough for the computer to understand?

2.2) Make A Plan

Once you have a good understanding of the problem in mind, it is time to formulate a plan by asking yourself questions such as the following (and, indeed, it is a good idea to think through this before writing any code):

- 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 ways to represent that information?
- 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?
- Thinking back to your understanding of the problem, are there any interesting "edge cases" that should be considered? Does your plan account for those?

2.3) Implement the Plan

It is only after understanding the problem and formulating a plan that it makes sense to start putting that plan into action by translating it into Python code, and it can be helpful to consider the following as you are doing so:

- You may be able to implement several of the important high-level operations as individual "helper" functions.
- As you are going, consider the style guidelines outlined here. 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 and each helper function:

    - Can you clearly see that the step is correct in a general sense?
    - Can you prove that it is correct in a general sense?
    - Does the step pass your test cases from earlier?
    - How can you use that result as part of the larger program?

2.4) Look Back

When you are reasonably certain that you are finished, it is still worth taking a step back, so to speak, and looking at things again:

- Test both for correctness and style.
- For each of the test cases you constructed 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 solution you 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?

3) Organizing the Design Process

Even when following the suggestions above, it is not always apparent how one should organize the process. And there is no real right answer, but many people find it helpful to do one or more of the following things:

1. Sketch out a high-level outline on paper, in any format that is helpful to you. This can involve working through specific cases by hand, outlining useful helper functions, drawing diagrams, writing "pseudocode," or anything else that is helpful to you.

   It is often the case that, walking by an experienced programmer's office as they are working, you will find papers (or whiteboards or chalkboards) full of diagrams, code outlines, etc.

2. Explicitly write an outline using comments. Particularly once the high-level design starts to take shape a bit, it can be helpful to organize your thoughts using comments in an actual Python file, even before writing any actual code.

   For example, you might start by writing functions that are empty except for doctsrings detailing their behavior. From there, you might write comments within those functions that mirror the structure you expect your code will eventually take, outlining high-level steps that need to be taken. That way, when it comes time to actually implement the plan, you already have a high-level plan explicitly written out, and you can think about writing smaller pieces of code at a time, since the comments have hopefully broken things down a bit for you.