# 6.009: Fundamentals of Programming

## Week 3 Lecture: Designing and Debugging Programs

Adam Hartz
hz@mit.edu

*21 September 2020*

## 6.009: Goals

Our goals involve helping you develop your programming skills, in multiple aspects:

- **Programming:** analyzing problems, developing plans
- **Coding:** translating plans into Python
- **Debugging:** developing test cases, verifying correctness, finding and fixing errors

Ultimately, we would like to:

- improve the readability of our code
- reduce the number of errors, and make them easier to find/fix
- make it easier to change the program if needed
- minimize the amount of code we write

## Style

“Style” can mean many different things.

On a trivial level, it refers to the structure of the characters in your code file
(search for “Python PEP8” for the official Python style guide).

But it also means more than that!
Careful organization of code can make every step of the programming process easier.

## Design Principles: Style

- **Names Matter**: Choose *meaningful*, *concise* variable names.

- **Comments Matter**: Include comments in your code, particularly for complicated logic or notes to the reader. Avoid comments that are redundant with the code.

- **DRY**: Don't repeat yourself (multiple pieces of code should not describe redundant logic).

- **Avoid Monolithic Functions**: Separate big, complicated programs into smaller pieces that can be tested, debugged, and verified independently.

## Names Matter

Let’s look at some examples of name choices.

## Comments Matter

*"Commenting your code is like cleaning your bathroom - you never want to do it, but it really does create a more pleasant experience for you and your guests."*
-Ryan Campbell

Good comments:

- Document non-obvious features of the code
- Are not redundant with the code itself
- Describe clever algorithms / design decisions

*"But who will ever read my code?"*

## Example: Averaging Filter

Example of successivey refining a program for *style* and *readability*.

Applying an averaging filter to a list of numbers:
a list with $n$ numbers $x_0, x_1, x_2, \ldots, x_{n-1}$, compute output values $y_0, y_1, y_2, \ldots, y_{n-k}$ where:

$$y_i = \frac{x_i + x_{i+1} + x_{i+2} + \ldots + x_{i+(k-1)}}{k}$$

## Designing for Style...

...takes time, experience, and practice!

When starting, this kind of iterative refinement is often necessary.
With more experience, will be able to “see” more of these improvements before implementing the original version.

Make a plan and implement something that works, then *then* look for these opportunities. Eventually, your plans will start to include these patterns.

## The Design Process

The next few slides are adapted from George Polya's *How To Solve It*.

*How to Solve It* suggests the following steps when solving a problem:

- First, you have to understand the problem.
- After understanding, make a plan.
- Carry out the plan.
- Look back on your work. How could it be better?

## The Design Process

- **Understand the Problem**:

  What problem are you solving? What is the input, and what is the expected output? How can these be represented in Python? What are some example input/output relationships? (come up with a few examples you can use to test later)

- **Formulate a Plan (on Paper!)**:

  Look for the connection between the input and output; what are the high-level operations you need to perform in order to compute the output? How can you test those pieces? How do they fit together to form the overall solution? What do you need to keep track of (aside from the input), and how can those data be represented in Python? Have you read/written a related program? If so, are techniques from that program applicable here? Can you break the problem down into simpler pieces/cases? Try solving a subpart of subproblem first.

  Are you convinced your plan will work? If so, move on to the next step.

## The Design Process

- **Implement the Plan**

  Translate the plan into Python. Implement and test high-level operations on their own. Consider our tips for "style," and reorganize when you see opportunities. Check each step as you go; is each step correct?

- **Look Back:**

  Look for correctness, style, and efficiency. For each test case you constructed earlier, run it and make sure you see what you expect. Are there other cases you should consider? Could you have solved the problem a different way? What is good or bad about your approach?

## The Design Process

The high-level message:

*"First solve the problem. Then write the code."*
-John Johnson

## Example: Caesar Cipher

Simple form of encryption where each letter in the plaintext is replaced by a letter some (configurable) fixed number of positions down the alphabet. This produces an encrypted message that cannot be easily read by someone who doesn't know the shift value.

For example, shifting `dog` by 4 would produce `hsk`.

If the shifted value goes beyond the end of the alphabet, it wraps back around to `a`.

Many variations exist, but in our version we will:

- shift letters as described above
- shift digits in a similar way (but wrapping after `9` instead of `z`)
- leave all other character unchanged

## Debugging is a Part of Life

*“By June 1949 people had begun to realize that it was not so easy to get programs right as at one time appeared.*

*...the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs.”*

-Maurice Wilkes, *Memoirs of a Computer Pioneer*, MIT Press, 1985, p. 145.

## Debugging

What's the most effective strategy for debugging?

## Debugging

What's the most effective strategy for debugging?

Avoid writing bugs in the first place!

Following the advice on the previous slides is a good place to start, and can help you design programs that are less error-prone.

...but bugs will happen anyway! How do we deal with them?

## Debugging

What's the most effective strategy for debugging?

Avoid writing bugs in the first place!

Following the advice on the previous slides is a good place to start, and can help you design programs that are less error-prone.

...but bugs will happen anyway! How do we deal with them?

**Resist the temptation to start changing things around just to see if it works!**

## Debugging

Effective debugging requires taking these steps:

- Work out *how* and *why* the software is behaving unexpectedly.
- Fix the problem.
- Avoid breaking anything else in the process.
- Maintain and/or improve the overall quality (readability, etc) of the code to avoid future bugs.
- Ensure that the same problem does not occur elsewhere and cannot happen again.

**Step 1 is crucial!**

## An Empirical Approach to Bug Finding

- **Hypothesize and Localize:**
  Read and interpret error messages or invalid output. Add print and/or assert statements to compare results/conditions against expectations.
  Construct hypotheses, and test them by performing experiments until you are confident that you have identified the underlying bug.
- **Reproduce the Issue:**
  Find a way to reliably and conveniently reproduce the problem on demand. Try to find the simplest case that exhibits the broken behavior.
- **Fix the Issue:**
  Design and implement changes that fix the problem (and that don't break anything else!)
- **Reflect:**
  Learn from the bug. Where did things go wrong? Are there any other examples of the same problem that also need fixing? What can you do to ensure that the same problem doesn't happen again?

## Other Advice

- Make sure you're getting enough sleep!
- Step away from the computer, take a walk. And/or revisit your plan on paper.
- The greatest debugging tools known to humankind are built in to Python!
- The "rubber duck" technique.

## Example: Caesar Cipher

Let’s return to our Caesar Cipher example.