6.009
Fundamentals of Programming

Week 3 Lecture:

Designing and Debugging Programs
(Part II)

Adam Hartz
hz@mit.edu

6.009: Goals

Our goals involve helping you develop as a programmer, in multiple aspects:

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

So we will spend time discussing:

- High-level design strategies
- Ways to manage complexity
- Details and "goodies" of Python
- A mental model of Python's operation
- Testing and debugging strategies

...but discussion only goes so far!

6.009: Pedagogy

Learning to program is a lot like learning a musical instrument or a sport. How does one learn those things?

Just like with music/sports, practice is key!
To improve as a programmer, you have to program.
And 6.009 asks you to program...a lot!

- Labs give opportunities to practice new techniques/skills to solve interesting problems.
- Lectures/tutorials equip you with tools useful for attacking those problems.

6.009: Pedagogy

Getting the Most out of 6.009

Lectures/Tutorials:

- Step 1: Come to lecture/tutorial!
- Take notes *in your own words* and review them later
- **Ask questions!** We want to have a conversation.

Labs:

- Start early (labs are week-long assignments)
- Formulate a plan before writing code
  - Try to understand the problem thoroughly before writing code
  - When things go wrong, revisit the plan
- Work through problems on your own
- Ask for help when you need it!
  - Labs are intentionally challenging
  - Bugs are a natural part of life
  - Lots of opportunities for help (office hours / Piazza)

Today

- Review of Data Structures
- Designing Programs, Style (w/ Examples)
- Debugging (w/ Examples)

Choose the Right Structure for the Job

Python offers several types of *collections*:

- `list`
- `tuple`
- `set`
- `frozenset`
- `dict`
- ...

What properties to these different structures have? When might we want to use one, versus the other?

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.

Example: Lab 0

Cellular Automaton (gas simulation)

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" 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: “Flood Fill”

Let’s do some programming!

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.

Example: “Flood Fill”

Let’s do some debugging!