# 6.009
# Fundamentals of Programming

Week 2 Lecture:
Designing and Debugging Programs

Adam Hartz
`hz@mit.edu`

## 6.009: Staff

**Instructors:**

Duane Boning

Pete Szolovits

Adam Hartz

Karl Berggren

## 6.009: Web Site

Just about everything in 6.009 happens via the web site:

`http://mit.edu/6.009`

## 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

## 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

## 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?

## 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: A typical week

A typical week centers around a lab assignment, supplemented by instructor presentations and with lots of help available.

- **Lecture:** Mon 11-12:30, 26-100
- **Tutorials**: Wed 9-11 or 1-3
  (first 30-60 minutes of session)
- **Office Hours:**
  - Wed 9-11 or 1-3
    (last 60-90 minutes of session)
  - Fri 9-11, 10-12, 1-3, 2-4

## Labs: the Heart of 6.009

**Logistics:**

- Issued Fridays at 8am
- Typically a mix of conceptual questions and writing code (Python 3.5+)
- Some questions are due Mondays at 10am
- Bulk of the lab is due the following Friday at 4pm
- Checkoff meetings are due on Wednesday at 10pm
- Lateness policy described on web site

**Cool Problems!**

- Image Processing, Minesweeper, SAT Solver, LISP Interpreter, Platforming Game, ...

## 6.009 Grading

- 9 labs (40% of grade)
- 3 Quizzes (15%, 20%, 25%)
- See web site for more details and points-to-grade mapping.

## Getting the Most out of 6.009

## 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.

## 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)

## Getting the Most out of 6.009

## Collaboration Policy

Our goal is that *every student* develops these skills throughout the course. Collaborating too closely with others (or outsourcing pieces to other students or StackOverflow) can rob you of an opportunity to develop those skills in yourself.

Please read our collaboration policy carefully.

(see "Information/Policies" on the web site)

# Understanding Python

We will devote a good amount of time in 6.009 to developing a *notional machine* for Python.

## Check Yourself

What happens when the following program is run?

```
functions = []
for i in range(5):
    def func(x):
        return x + i
    functions.append(func)

for f in functions:
    print(f(12))
```

0. It prints 12, then 13, then ..., then 16
1. It prints 13, then 14, then ..., then 17
2. It prints 16, then 15, then ..., then 12
3. It prints 17, then 16, then ..., then 13
4. A Python error occurs
5. Something else

## 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).
- **Generality Wins**: Define functions and program generally so they can be used for multiple specific cases.
- **Plan for Change**: Requirements/expectations sometimes change! It should be easy to adjust code.

## 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

## Comments Matter

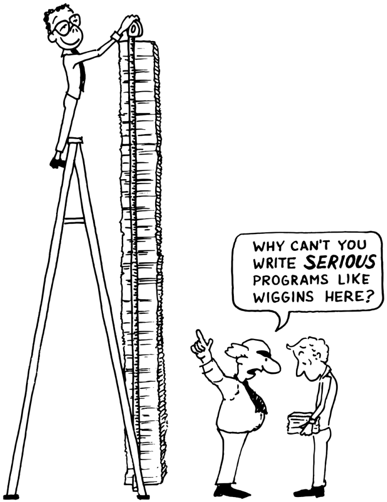
*"Commenting your code is like cleaning your bathroom - you never want to do it, but it really does create a more pleasant experience for you and your guests."*
-Ryan Campbell

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?"

## Design Principles: DRY, Generality Wins



*Conventional winsdom reveres complexity*

Image from *Thinking Forth*, licensed under a CC BY-NC-SA 2.0 Generic License

## 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”

## 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!

## 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 *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

- **Reproduce:**
  Find a way to reliably and conveniently reproduce the problem on demand. Try to find the simplest case that exhibits the broken behavior.
- **Diagnose:**
  Construct hypotheses, and test them by performing experiments until you are confident that you have identified the underlying bug.
- **Fix:**
  Design and implement changes that fix the problem (and don't break anything else!)
- **Reflect:**
  Learn the lessons of 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 tool known to humankind is built in to Python!
- The "rubber duck" technique.

## Example: “Flood Fill”

Let’s do some programming!