Development Strategies, Function Design

BBM 101 - Introduction to Programming I

Hacettepe University
Fall 2016

Fuat Akal, Aykut Erdem, Erkut Erdem

Today

- **How to develop a program**
  - Program development strategy
- **More on Testing and Debugging**
  - Testing and debugging
  - Black box testing
  - Glass box testing
  - Integration testing and unit testing
  - Debugging approaches

Today

- **How to develop a program**
  - Program development strategy
- **More on Testing and Debugging**
  - Testing and debugging
  - Black box testing
  - Glass box testing
  - Integration testing and unit testing
  - Debugging approaches

Slides based on material prepared by Ruth Anderson, Michael Ernst and Bill Howe in the course CSE 140 University of Washington

Program development methodology: English first, then Python

1. Define the problem
2. Decide upon an algorithm
3. Translate it into code

Try to do these steps in order

Program development methodology: English first, then Python

1. **Define the problem**
   A. Write an English description of the input and output **for the whole program**. (Do not give details about *how you will compute* the output.)
   B. Create test cases **for the whole program**
      - Input *and* expected output
2. Decide upon an algorithm
3. Translate it into code

Try to do these steps in order

Program development methodology: English first, then Python

1. Define the problem
2. **Decide upon an algorithm**
   A. Implement it in English
      - Write the recipe or step-by-step instructions
   B. Test it using paper and pencil
      - Use small but not trivial test cases
      - Play computer, animating the algorithm
      - Be introspective
        – Notice what you really do
        – May be more or less than what you wrote down
        – Make the algorithm more precise
3. Translate it into code

Try to do these steps in order

Program development methodology: English first, then Python

1. Define the problem
2. Decide upon an algorithm
3. **Translate it into code**
   A. Implement it in Python
      - Decompose it into logical units (functions)
      - For each function:
        – Name it (important and difficult!)
        – Write its documentation string (its specification)
        – Write tests
        – Write its code
        – Test the function
   B. Test the whole program

Try to do these steps in order

Program development methodology: English first, then Python

1. Define the problem
2. Decide upon an algorithm
3. Translate it into code

Try to do these steps in order

– It's OK (even common) to back up to a previous step when you notice a problem
– You are incrementally learning about the problem, the algorithm, and the code
– "Iterative development"

The *Wishful Thinking* approach to implementing a function

- If you are not sure how to implement one part of your function, define a **helper function** that does that task
  - "I wish I knew how to do task X"
  - Give it a name and assume that it works
  - Go ahead and complete the implementation of your function, *using* the helper function (and assuming it works)
  - Later, implement the **helper function**
  - The helper function should have a simpler/smaller task

The *Wishful Thinking* approach to implementing a function

- Can you test the original function?
  - Yes, by using a stub for the **helper function**
  - Often a lookup table: works for only 5 inputs, crashes otherwise, or maybe just returns the same value every time

Why functions?

There are several reasons:

- Creating a new function gives you an opportunity to name a group of statements, which **makes your program easier to read and debug**.
- Functions **can make a program smaller** by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to **debug the parts one at a time** and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, **you can reuse it**.

Today

- **How to develop a program**
  - Program development strategy
- **More on Testing and Debugging**
  - Testing and debugging
  - Black box testing
  - Glass box testing
  - Integration testing and unit testing
  - Debugging approaches

Slides based on material prepared by E. Grimson, J. Guttag and C. Terman in MITx 6.00.1x

Testing and Debugging

- Would be great if our code always worked properly the first time we run it!
- But life ain't perfect, so we need:
  - Testing methods
    - Ways of trying code on examples to determine if running correctly
  - Debugging methods
    - Ways of fixing a program that you know does not work as intended

When should you test and debug?

- Design your code for ease of testing and debugging
  - Break program into components that can be tested and debugged independently
  - Document constraints on modules
    - Expectations on inputs, on outputs
    - Even if code does not enforce constraints, valuable for debugging to have description
  - Document assumptions behind code design

When are you ready to test?

- Ensure that code will actually run
  - Remove syntax errors
  - Remove static semantic errors
  - Both of these are typically handled by the Python interpreter
- Have a set of expected results (i.e. input- output pairings) ready

Testing

- Goal:
  - Show that bugs exist
  - Would be great to prove code is bug free, but generally hard
    - Usually can't run on all possible inputs to check
    - Formal methods sometimes help, but usually only on simpler code

"Program testing can be used to show the presence of bugs, but never to show their absence!"

– Edsger Dijkstra

Test suite

- Want to find a collection of inputs that has high likelihood of revealing bugs, yet is efficient
  - **Partition space of inputs** into subsets that provide equivalent information about correctness
    - Partition divides a set into group of subsets such that each element of set is in exactly one subset
  - **Construct test suite** that contains one input from each element of partition
  - **Run test suite**

Example of partition

```
def isBigger(x, y):
    """Assumes x and y are ints
    returns True if x is less than y
    else False"""
```

- Input space is all pairs of integers
- Possible partition
  - x positive, y positive
  - x negative, y negative
  - x positive, y negative
  - x negative, y positive
  - x=0,y=0
  - x=0,y!=0
  - x!=0,y=0

Why this partition?

- Lots of other choices
  - E.g., x prime, y not; y prime, x not; both prime; both not
- Space of inputs often have natural boundaries
  - Integers are positive, negative or zero
  - From this perspective, have 9 subsets
    - Split x = 0, y != 0 into x = 0, y positive and x =0, y negative
    - Same for x != 0, y = 0

Partitioning

- What if no natural partition to input space?
  - Random testing – probability that code is correct increases with number of trials; but should be able to use code to do better
  - Use heuristics based on exploring paths through the specifications – **black-box testing**
  - Use heuristics based on exploring paths through the code – **glass-box testing**

Black-box testing

- Test suite designed without looking at code
  - Can be done by someone other than implementer
  - Will avoid inherent biases of implementer, exposing potential bugs more easily
  - Testing designed without knowledge of implementation, thus can be reused even if implementation changed

Paths through a specification

```
def sqrt(x, eps):
    """Assumes x, eps floats
           x >= 0
           eps > 0
       returns res such that
           x-eps <= res*res <= x+eps"""
```

- Paths through specification:
  - x = 0
  - x > 0
- But clearly not enough

Paths through a specification

- Also good to consider boundary cases
  - For lists: empty list, singleton list, many element list
  - For numbers, very small, very large, "typical"

Example

- For our sqrt case, try these:
  - First four are typical
    - Perfect square
    - Irrational square root
    - Example less than 1
  - Last five test extremes
    - If bug, might be code, or might be spec (e.g. don't try to find root if eps tiny)

x	eps
0.0	0.0001
25.0	0.0001
.05	0.0001
2.0	0.0001
2.0	1.0/2.0**64.0
1.0/2.0**64.0	1.0/2.0**64.0
2.0**64.0	1.0/2.0**64.0
1.0/2.0**64.0	2.0**64.0
2.0**64.0	2.0**64.0

Glass-box Testing

- Use code directly to guide design of test cases
- Glass-box test suite is **path-complete** if every potential path through the code is tested at least once
  - Not always possible if loop can be exercised arbitrary times, or recursion can be arbitrarily deep
- Even path-complete suite can miss a bug, depending on choice of examples

Example

```
def abs(x):
    """Assumes x is an int
       returns x if x>=0 and -x otherwise"""
    if x < -1:
        return -x
    else:
        return x
```

- Test suite of {-2, 2} will be path complete
- But will miss `abs(-1)` which incorrectly returns -1
  - Testing boundary cases and typical cases would catch this {-2 -1, 2}

Rules of thumb for glass-box testing

- Exercise both branches of all if statements
- Ensure each except clause is executed
- For each for loop, have tests where:
  - Loop is not entered
  - Body of loop executed exactly once
  - Body of loop executed more than once
- For each while loop,
  - Same cases as for loops
  - Cases that catch all ways to exit loop
- For recursive functions, test with no recursive calls, one recursive call, and more than one recursive call

Conducting tests

- Start with **unit testing**
  - Check that each module (e.g. function) works correctly
- Move to **integration testing**
  - Check that system as whole works correctly
- Cycle between these phases

Test Drivers and Stubs

- **Drivers** are code that
  - Set up environment needed to run code
  - Invoke code on predefined sequence of inputs
  - Save results, and
  - Report
- Drivers simulate parts of program that use unit being tested
- **Stubs** simulate parts of program used by unit being tested
  - Allow you to test units that depend on software not yet written

Good testing practice

- Start with unit testing
- Move to integration testing
- After code is corrected, be sure to do **regression testing**:
  - Check that program still passes all the tests it used to pass, i.e., that your code fix hasn't broken something that used to work

"Most people, if you describe a train of events to them, will tell you what the result would be. They can put those events together in their minds, and argue from them that something will come to pass. There are few people, however, who, if you told them a result, would be able to evolve from their own inner consciousness what the steps were which led up to that result. This power is what I mean when I talk of reasoning backwards, or analytically." -- Sherlock Holmes (A Study in Scarlet, by Sir Arthur Conan Doyle)

Runtime bugs

- **Overt vs. covert:**
  - **Overt** has an obvious manifestation – code crashes or runs forever
  - **Covert** has no obvious manifestation – code returns a value, which may be incorrect but hard to determine
- **Persistent vs. intermittent:**
  - **Persistent** occurs every time code is run
  - **Intermittent** only occurs some times, even if run on same input

Categories of bugs

- Overt and persistent
  - Obvious to detect
  - Good programmers use **defensive programming** to try to ensure that if error is made, bug will fall into this category
- Overt and intermittent
  - More frustrating, can be harder to debug, but if conditions that prompt bug can be reproduced, can be handled
- Covert
  - Highly dangerous, as users may not realize answers are incorrect until code has been run for long period

Debugging skills

- Treat as a search problem: looking for explanation for incorrect behavior
  - Study available data – both correct test cases and incorrect ones
  - Form an hypothesis consistent with the data
  - Design and run a repeatable experiment with potential to refute the hypothesis
  - Keep record of experiments performed: use narrow range of hypotheses

Debugging as search

- Want to narrow down space of possible sources of error
- Design experiments that expose intermediate stages of computation (use `print` statements!), and use results to further narrow search
- Binary search can be a powerful tool for this

Some pragmatic hints

- Look for the usual suspects
- Ask why the code is doing what it is, not why it is not doing what you want
- The bug is probably not where you think it is – eliminate locations
- Explain the problem to someone else
- Don't believe the documentation
- Take a break and come back to the bug later