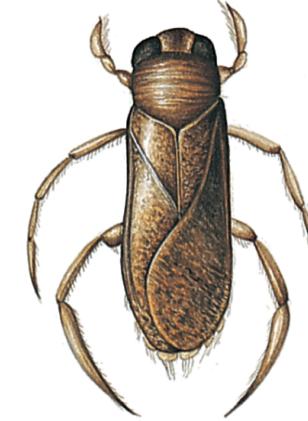


# CS I

# Introduction to Computer Programming

*Lecture 17: November 13, 2015*

## How to think about debugging



# Today

- Background on bugs
  - defects vs. failures
- Finding bugs (defects)
  - assertions, unit tests, localization
- Fixing bugs
  - understanding the system
  - refactoring the design
- Examples



# Program Failures

- So, you wrote your program.
- And, guess what? It doesn't work. ☹
- Your program has a **bug** in it.
- Somehow, you must track down the bug and fix it.
- You need to **debug** the program.



# Program Failures (2)

- True origin of the term “bug” is unclear, but computers have failed because of actual bugs!
- 1947: Harvard Mark II relay-based computer stopped working
- A *moth* was found wedged between a relay's contacts!

(033) PRO 2 2.130476415  
cosine 2.130676415  
Relays 6-2 in 033 failed special speed test  
in relay 10.000 test.

1100 Started Cosine Tape (Sine check)  
1525 Started Multi Adder Test.

1545 Relay #70 Panel  
(moth) in relay.

1630 First actual case of bug being found.  
1700 Antagonist started.  
closed down.



# Bugs = Defects + Failures

- Helpful to break down concept of a “bug” into two more fundamental components:
- Program contains a defect in the code, due to a design or implementation mistake
- This defect is manifested as a failure of some kind
  - Program reports an error when you run it, or it produces an incorrect result, it crashes, etc.



# Bugs = Defects + Failures (2)

- Several different kinds of errors
- **Syntax errors**: didn't use language correctly
  - The Python interpreter simply doesn't understand what you tried to say
  - e.g. you forgot a closing-quote on a string constant
- **Semantic or logical errors**: the interpreter understands your code, but implementation is still incorrect in some way
  - e.g. you tried to add a string to a number
  - or you just got a wrong answer!



# Bugs = Defects + Failures (3)

- Errors are the *failures*, but often the *defect* occurred much earlier
  - Stored incorrect data in a data structure
  - Passed invalid data to a function
  - etc.
- For a given failure, need to
  - 1. find the corresponding defect
  - 2. figure out how to fix the code so defect doesn't occur!



# Finding Bugs

- The majority of the work in fixing a bug is usually *finding the defect that produces it*
- Usually, once you identify the defective code, the proper fix becomes obvious
- If the overall *design* is defective, this can be much more difficult
  - Can require redesigning and reimplementing a large portion of the program



# Example: Defective Design

- Math Department Candy Store:
  - First candy costs ten cents
  - Each subsequent candy costs ten cents more than the previous candy
  - You have one dollar to spend
- How many candies can you buy?
  - 4 (right?)



# Example: Defective Design (2)

- Let's write a Python program to do this:

```
price = 0.10    # First candy is 10 cents
budget = 1.00   # I have one dollar
count = 0        # Start with no candies
while budget >= price:
    count += 1      # Buy another candy!
    budget -= price
    price += 0.10    # Price goes up

print 'I bought %d candies.' % count
print 'I have ${%f} left.' % budget
```

- How many candies can you buy?



# Example: Defective Design (3)

- This program produces the wrong result!  
`I bought 3 candies.`  
`I have $0.400000 left.`
- Problem: floating-point numbers are an approximation
  - In fact, cannot exactly represent numbers like 0.1 in floating-point!
- Using floats to represent monetary amounts is a *defective design*
  - Can use integers to represent all monetary values in cents, or use other decimal representations



# Finding Bugs (2)

- The defect is only the *cause* of the failure, but it is not the failure itself!
  - The defect will immediately begin to affect the program's state...
  - ...but the effects may not become visible for some time
- The greater the separation between defect and failure, the harder it is to diagnose the defect from the failure



# Finding Bugs (3)

- If the defect *directly* causes the failure:
  - Simply need to identify the location where the program fails. Can usually fix bug very quickly.
- Normally the defect and failure are separated from each other
  - Often in different functions
  - Sometimes separated by a significant amount of execution time (hours, days, weeks!)



# Simple Examples

- Code:

```
def f1(a, b):  
    return 'a' + b
```

Defect and failure are  
on the same line.

```
f1(3, 4)
```

- Error:

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
  File "<stdin>", line 2, in f1  
TypeError: cannot concatenate 'str'  
          and 'int' objects
```



# Simple Examples (2)

- Code:

```
def f1(a, b):  
    return a + b Failure occurs here...  
def f2(a):  
    return f1('a', 10) Defect is here!  
f2(6)
```

- Error:

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
  File "<stdin>", line 2, in f2  
    File "<stdin>", line 2, in f1  
TypeError: cannot concatenate 'str' and  
'int' objects
```



# Simple Examples (3)

- Traceback tells you where to start looking
  - Traceback (most recent call last):*  
*File "<stdin>", line 1, in <module>*  
*File "<stdin>", line 2, in f2*  
*File "<stdin>", line 2, in f1*  
***TypeError: cannot concatenate 'str' and 'int' objects***
- Last line is where the *failure* occurred
- But the actual *defect* could be anywhere along the line, or...
  - ...could also be code executed any time before the failure occurred



# Finding defects

- *You want defects to manifest as failures quickly!*
  - Don't want much time to pass, or else they will be harder to find
  - Most of the time spent debugging is time spent *tracking down* the defect, not fixing it
- If you can find the defect quickly, you have typically eliminated 80-90% of your debugging time



# Finding defects

- One common technique for causing defects to manifest quickly is using assertions
- Frequently have conditions that you expect to be true at certain points in your program
  - Explicitly state these in an assertion, in your code
  - At runtime, the assertion is checked: if it's false, the program is stopped immediately!



# Example: Mastermind Functions

- Have to write two functions for Mastermind:
  - `count_exact_matches(code, guess)`
  - `count_letter_matches(code, guess)`
- Expectations for the inputs?
  - Both `code` and `guess` are strings of length 4
- Expectations for the outputs?
  - Result will always be between 0 and 4
- These can be enforced as *assertions*
  - If they are violated as our program runs, we'll be notified immediately



# Technique: Assertions

- Python provides an **assert** statement
- Form 1: **assert cond**
  - Tests **cond**; if it's false then an **AssertionError** exception is raised
- Form 2: **assert cond, mesg**
  - Similar to Form 1, but **mesg** is included in the **AssertionError** if it fails



# Assertions (2)

- Example:

```
Line 1: def count_exact_matches(code, guess):  
Line 2:     assert len(code) == 4  
Line 3:     assert len(guess) == 4  
          ... # rest of implementation
```

```
>>> count_exact_matches('ABCD', 'ABCDE')  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
  File "<stdin>", line 3, in  
    count_exact_matches  
AssertionError
```

Tells us which assertion failed



# Assertions (3)

- A bit difficult to figure out location, plus we don't know what we actually got passed!
- Update the assertion to include more detail:

```
def count_exact_matches(code, guess):  
    assert len(code) == 4, \  
        'Invalid code: %s' % code  
    assert len(guess) == 4, \  
        'Invalid guess: %s' % guess  
    ... # rest of implementation  
  
>>> count_exact_matches('ABCD', 'ABCDE')  
AssertionError: Invalid guess: ABCDE
```



# Aside: Argument Types

- Python doesn't enforce constraints on the types of arguments, variables, etc.
- `count_exact_matches(code, guess)`
  - Can pass any valid Python type for `code` and `guess`!
  - Can't enforce the expected types in the function declaration
- But there is a way to enforce argument types using assertions



# Aside: Argument Types (2)

- Python provides a special **types** module:  
`from types import *`
  - Module is designed to be imported this way
- Allows us to create assertions like this:

```
def count_exact_matches(code, guess):  
    assert type(code) == StringType, \  
        "code %s isn't a string!" % code  
    assert len(code) == 4  
    ... # etc.
```

- **type()** built-in function returns type of argument



# Aside: Argument Types (3)

- We could also write this code as follows:

```
def count_exact_matches(code, guess):  
    assert type(code) is str, \  
        "code %s isn't a string!" % code  
    assert len(code) == 4  
    ... # etc.
```

- Common type checks include:
  - `assert type(x) is int`
  - `assert type(x) is float`
  - `assert type(x) is list`
  - `etc.`



# Aside: Argument Types (4)

- It's actually somewhat discouraged in Python to enforce types too strictly
  - Part of Python's strength is its flexibility
- However, if a function definitely requires its arguments to be specific types, this mechanism is very effective
  - The moment you misuse the function, you will be alerted with an error message!



# Assertions (4)

- What assertions do is turn *incorrect program states* into *failures*
- This doesn't mean that the assertion pinpoints the *defect*...
- ... but the defect will probably occur *much* closer to the assertion failure than to whatever would fail if there were no assertions
- Assertions reduce defect→failure distance!



# Assertions (5)

- Definitely use assertions in your code!
  - Saves so *much* time when debugging!!!
- You should even be willing to perform large, slow tests in assertions
  - Doesn't matter how fast your program is, if it produces the wrong answers
- Assertions can be turned off once you are confident your program works properly
  - **python -O program.py**



# Technique: Unit tests

- Assertions are great, but even more useful are *unit tests*
- Unit tests document your *exact expectations* of how a function/method should behave
- Unit tests typically test a *single function* in isolation, so if anything goes wrong, you know *exactly* where to look (saves *tons* of debugging time!)



# Unit tests (2)

- With unit tests, you control the inputs
- Normally only give known valid inputs
  - unless you're checking to see if error handling works correctly!
- Therefore, if a failure occurs, the defect is either
  - in the actual code of the function being tested
  - in code that is called from the function being tested
- Again, reduces defect→failure distance



# Unit tests (3)

- Guidelines for writing good unit tests:
  - 1. Write them! For every non-trivial function!
  - 2. Investigate all "corner cases"
    - list functions → empty list
    - string functions → empty string
    - etc.
  - 3. Test cases with incorrect input too
    - assert that the correct error occurs
    - Python **nose** module makes this easy
  - 4. Update unit tests if new bugs found
    - get better *code coverage* over time



# Unit tests (3)

- One school of thought argues that programmers should write tests *before* writing code to be tested
  - "*Test-driven development (TDD)*"
- Tests serve both as a debugging tool and as a **specification** of what the functions need to be able to do



# Unit tests (4)

- The best thing about writing unit tests is that they *force you to write functions that are easy to test!*
- That means
  - small functions
  - well-defined inputs and outputs
  - no global variables
  - no odd "side-effects"
  - "do one thing and do it well"
- Debugging is *much easier* in this case!



# Failure → Defect

- Assertions and unit tests will show you where a failure occurs
- Unit tests also control for inputs, so defect must be local too
- However, in many cases the defect may not be localized to where the failure occurs
- In that case, you have some detective work ahead of you...



# Debugging attitude (I)

- Nobody really loves debugging
- However, good programmers don't fear it
- Debugging is an interesting intellectual challenge, and you should treat it as (potentially) a fun problem to solve
- Doing so will make your life as a programmer much less stressful and more rewarding



# Debugging attitude (2)

- However, there are some conditions which make debugging miserable, and you need to know how to avoid them
- Examples:
  - Assignment/exam is due in one hour, gotta go!
  - My boss is screaming at me to fix this right away, or the business is going to collapse!
  - This bug is one of the really nasty kinds of bugs that take forever to track down



# Debugging attitude (3)

- Problems with deadlines reflect either poor time management or poor management!
  - maybe you need a better boss ☺
  - maybe you need a better job ☺
- Examples of the nastier kinds of bugs:
  - Concurrency-related bugs (multiple things happening at once)
  - Memory corruption bugs (C, C++)
  - For these, special tools may be required
  - Fortunately, most bugs are more mundane



# Detective Work

- Once you have observed a *failure* in your program, you have a mystery to solve!
  - *Where is the defect?*
- One thing a good detective never does:
  - Guess randomly!
- The programming version of this:
  - Make random guesses about the cause, and try various changes
  - Called “*the shotgun approach*”, and “*monkeys on a typewriter*”
  - (The results are just as good, too.)



# Detective Work (2)

- Instead, must consider the clues, and track down the defect from details of the failure
- We'll discuss strategies for this after the interlude



# Interlude

- More computer history!



# Step I: Reproduce the Failure

- Before you can do anything else, you must find a way to **reproduce the failure**
  - Recreate the steps that caused the program to fail
  - Was it specific data inputs? Was it a specific interaction with the program?
- At this point in your programming career, may simply need to re-run your program
  - As your programs become more complex, reproducing failures becomes more involved



# Step I: Reproduce the Failure

- Reproducing the failure is very important, for three major reasons:
  1. So you can watch it fail. See exactly what the program was doing as it crashed and burned.
  2. So you can **zero in on the cause**. If the program fails in some circumstances but not in others, this will give you *hints* as to what part of the program actually contains the defect.
  3. So you can test if you actually fixed it. If you can reliably cause the failure, and your fix makes it go away, you win!



# Technique: Keep a Debugging Log!

- When debugging nasty issues, it's extremely helpful to keep a record of your efforts (hand-written or typed, it doesn't matter)
  - A general description of the failure
  - Inputs or circumstances when the failure occurs
  - Ideas of potential causes, along with the efforts you made to verify your ideas, and indications for or against each theory



# Debugging Log (2)

- A debugging log is especially helpful when you don't have the time/energy to spend hours debugging right now
- Record everything you know, pick it up later when you are ready to seriously tackle the bug
- And when you aren't so frustrated!
  - Taking a short walk can be *incredibly* helpful for figuring out tough bugs!



# Step 2: Isolate the Failure

- So your program fails. And you know how to make it fail.
- Problem: the defect is only in a small part of your code
  - Need to zero in on the part of the code that's actually flawed



# Step 2: Isolate the Failure

- Goal: try to devise the **smallest possible scenario** that still causes the failure to occur
  - If the scenario is small, there won't be very much code involved
  - The less code that's involved, the easier it is to find the defect



# Example: Failure Isolation

- A common programming task: process very large data files
  - e.g. data collected from an experiment
  - Example: the Large Hadron Collider (LHC) produces 700 megabytes of data *per second!*
- But you have a bug:
  - Your program generally works, but...
  - ...one data file causes your program to crash
  - ...and the data file is 2.1 gigabytes long.
  - What do you do?



# Example: Failure Isolation (2)

- You have the offending file, so you can reproduce the failure
  - Still, a lot of code runs before the failure occurs
- Want to narrow in on the actual cause of the bug:
  - **Cut the file down until you have the minimal portion that still causes the crash**
  - Perhaps the program crashes because the file contains a few bad values
  - If you're lucky, might even identify a handful of input lines that causes the problem



# Example: Failure Isolation (3)

- What if you *can't* get the issue to reproduce by trying smaller parts of the file in isolation?
- That would be a much harder bug to track down
  - But that would indicate that the bug is triggered by some characteristic of the file as a whole
  - e.g. maybe the file's size causes some memory issue
  - e.g. maybe a *combination* of values read from the file causes the failure
- Here is the point: You learn more about the nature of the defect by trying to isolate it
  - Record these things in your debugging log!



# Step 2: Isolate the Failure (2)

- Previous example involved trying to isolate the *inputs* that cause the failure
- Can also try to isolate the *general area of your program's code* that contains the failure
- Example: Mastermind program
  - The program behaves oddly when you run it
  - You have found inputs that reproduce the behavior
  - Now, want to narrow down the specific area of your program's code with the failure



# Step 2: Isolate the Failure (3)

- Easiest way to do this: *write unit tests!*
- Mastermind, continued
  - Load your program, then **test the important functions individually** with unit tests:  
`make_random_code()`  
`count_exact_matches(code, guess)`  
`count_letter_matches(code, guess)`
  - Does each operation behave correctly?
  - If not, the code you need to review is very small!



# Technique: Understand the System!

- Knowing where to start with bugs really requires you to understand the system being debugged. For example:
  - What are the major operations of the program?
  - What parts of the code perform each of these functions?
- If a failure manifests only when a *specific* feature is used, you know what files or functions to start with



# Understand the System! (2)

- Are there parts of the code that are always executed?
  - If a failure manifests in a variety of scenarios, this would be a likely location for the defect
- Are there parts of the code that can be disabled or removed, and the test run again?
- If you disable part of the code, and the failure still occurs:
  - The defect is *not* in the code we disabled!



# Understand the System! (3)

- Can ask similar questions when analyzing the *inputs* to your programs
- Are there parts of the code that only run on specific inputs, or sequences of inputs?
  - In other words, can you trigger specific paths through your program with certain inputs?
- If you can exercise every path through your code, bugs won't be able to hide



# Example: Scoring a Blackjack Hand

- Given a blackjack hand-scoring function, how would we exercise all code paths?
  - Try the function with an empty hand
  - Try it with a hand containing no aces
  - Try it with a hand containing one ace that can be scored as 11 without busting
  - Try it with a hand containing one ace that must be scored as 1 to not bust
  - Try it with multiple aces, where some (but not all) aces must be scored as 1 to not bust
  - Try it with multiple aces, where all aces must be scored as 1 to not bust



# Understand the System! (4)

- Whole point of understanding the system:
  - You are performing **experiments** with your program, making observations of its behavior
  - You need to *correlate* the observed behaviors with various parts of your program's code
  - The more effective you are at doing this, the faster you will zero in on the location of bugs
- Debugging involves a lot of thinking...



# Understand the System! (5)

- Both positive *and negative* correlations are helpful!
- If you can say, “This behavior is definitely not caused by the code in this part of the program,” that also helps narrow down the source of the issue



# Technique: Disabling Code

- How to disable a portion of the code depends on the specific language being used, as well as the program's structure
- In Python, can comment out code:

```
def foo():  
    ''' This is my awesome function that  
    doesn't work. '''  
  
    bar()  
  
    # FIXME: this may be buggy...  
    # abc('def')  # Do something amazing!  
  
    return xyz()
```



# Technique: Disabling Code (2)

- Can also wrap the code in triple-quotes

```
def foo():

    ''' This is my awesome function that
    doesn't work. '''

    ''' FIXME: This may be buggy... '''

    bar()

    abc('def') # Do something amazing!

    ...

    return xyz()
```

- (This won't work if the code you're wrapping also contains triple-quotes...)



# Technique: Use Simple Test Cases

- If you can't chop down the input data, and you can't chop down the program itself:
- Can create very simple test cases to exercise simple paths through the code
- Have already seen this earlier:
  - Mastermind: test the individual operations for correctness
  - Blackjack scoring function: use a battery of tests to exhaustively exercise the code



# Step 3: Identify the Defect Itself

- If you have completed the first two steps:
  - You probably have defect's location narrowed down to a relatively small part of your code
- Now, need to identify possible origins of the failure, and eliminate them one by one
  - Requires that you reproduce failure yet again, and watch what your program does as it fails
  - If you can't peer into the program's execution, you cannot identify the exact origin of the bug



# Step 3: Identify the Defect Itself (2)

- Several different approaches for this step:
  - Add logging/printing/debugging output to your program
  - Or, run your program in a **debugger** to single-step through the failure scenario
- Both approaches have the same goal:
  - Examine the state-changes your program makes, correlated with the specific lines of code making those changes
  - Determine the exact point in time when your program begins to create invalid state



# Step 3: Identify the Defect Itself (3)

- What is a “debugger” ?!
  - A separate tool that allows you to execute your code line by line, and to examine the program’s state as it runs
- Really only essential for compiled languages
  - Python has a debugger, but often unnecessary



# Technique: Printing Out Details

- A very common approach for debugging programs:
  - Add `print` statements to the code, then rerun your failure scenario
  - Then, pore through the debug output to see what happened
  - Can print to terminal or to a log file



# Technique: Printing Out Details (2)

- Common scenario:
  - The program fails.
  - Programmer suspects a particular cause of the failure, and adds debug-output to the program to explore that specific cause.
  - Guess what? It's not that!
  - Programmer has to go back and print out more details...



# Technique: Printing Out Details (3)

- Tip: When adding debug output, print *all* details that could be useful to know!
  - Allows you to evaluate multiple potential causes in less time



# Technique: Printing Out Details (3)

- Also, make sure each output line is unique

```
def foo(n) :  
    b = 0  
    for i in range(n) :  
        print b  
        b = bar(i, b)  
        print b  
        xyz(b)
```

- When you run this function, how do you tell what output is from what line?
- Want to make each line unique somehow:

```
print 'CASE 1: b = %d' % b
```



# Step 4: Fix Defect, Verify Fix

- Once the actual defect is found, usually straightforward to fix...
  - ...unless it's a design issue. This is why it's always good to design up front.
- You aren't finished fixing the bug until you verify the fix
- By this point, you should have a way to reproduce the failure...
  - Retry your tests and see if the failure is gone
  - If no more failures, you're done!



# Step 4: Fix Defect, Verify Fix (2)

- Bug fixes can introduce new defects into the code
  - Such defects are called *regressions*
- Usually occurs when:
  - The actual cause of the bug wasn't fully understood
  - Or the impact of the bug fix isn't fully understood



# Step 4: Fix Defect, Verify Fix (3)

- Good programmers also check for regressions:
  - Verify that the original bug is fixed
  - Also perform other tests to ensure that **no new bugs were introduced**
- Advantage of having a big suite of unit tests:
  - testing for regressions is trivially easy!



# Some Notes about Fixing Bugs

- Believe your observations, not your suspicions.
- A common scenario:
  - The program fails
  - The programmer *suspects* a particular issue, and focuses his attention there
  - However, there is no indication that the suspected cause is the *actual* cause
  - Sometimes this will cause you to miss obvious details that clearly indicate the actual source of the problem



# Some Notes about Fixing Bugs (2)

- **If you didn't fix it, it isn't fixed!**
- Another common scenario:
  - An intermittent failure is occurring
  - You made a few stabs at fixing it, but you still really don't know what causes the problem or how to reproduce it
- You definitely can't correlate any of your “fixes” with the problem



# Some Notes about Fixing Bugs (3)

- **If you didn't fix it, it isn't fixed!**
- It is very appealing to assume that if a bug hasn't occurred recently, it must be fixed
  - You may even avoid focused testing on that issue
    - (mainly because you really don't want to know...)
- Normally, these problems usually come back
  - Usually during a demo, or when your code is being graded



# Summary of debugging tips

- Use assertions to catch bugs early
- Check types on arguments
- Write unit tests!
  - Write easily tested functions!
- Be systematic when debugging (not random)
- Reproduce the failure!
- Keep a debugging log for tough cases
- Isolate the failure!



# Summary of debugging tips

- Reduce inputs down to the smallest possible example that fails
- Reduce code down to the smallest possible example that fails
- Disable code that's not part of the code that is causing the bug
- Test functions individually
- Use simple test cases
- Add debugging `print` statements to code to show the state of the program
- Re-run all tests after fixing the bug



# Next Time

- More exception handling
- The runtime stack

