How to Think About Debugging
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
- 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!
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 issue
- 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.
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 that string to a number
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?
  - (Four!)
Example: Defective Design (2)

Let’s write a Python program to do this:

```python
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 $%.2f left.' % budget
```

How many candies can you buy?

Caltech/LEAD CS: Summer 2011
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!)

• How to determine actual cause of failure?
Simple Examples

• Code: Defect and failure are on the same line.
  
  ```python
  def f1(a, b):
      return 'a' + b
  
  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:**
  ```python
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

  - …could also be code executed anytime before the failure occurred
Preemptive Bug Detection

- Want defects to manifest as failures quickly!
  - Don’t want much time to pass, or else they will be harder to find
- 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

- Had 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
Python Assertions

- Python provides an `assert` statement
- Form 1: `assert cond`
  - Tests `cond`; if it’s false then an `AssertionError` is reported
- Form 2: `assert cond, mesg`
  - Similar to Form 1, but `mesg` is included in the `AssertionError` if it fails
Example:

```python
def count_exact_matches(code, guess):
    assert len(code) == 4
    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
```

Tell us which assertion failed
Python 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:

```python
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
```

```python
>>> 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:
  ```python
  from types import *
  ```
- Module is designed to be imported this way
- Allows us to create assertions like this:
  ```python
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 arg
Aside: Argument Types (3)

- 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!
Python Assertions (4)

- 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`
Detective Work

- Once you have observed a failure in your program, you have a mystery to solve!
- 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
• Identify the issue first. Then make the fix.
  • Hunt bugs with a rifle, not with a shotgun.
  • Not as likely to introduce other defects, too.
• A major problem with bug-fixing: causing regressions
  • In the process of fixing a bug, you actually introduce other new bugs! (Very depressing.)
Detective Work (3)

- Limiting the effects of code-changes is yet another reason to keep functions small
  - (and to avoid using global variables)
- Functions usually perform computations in isolation from each other
- Fixing a bug in a small function is far less likely to affect other parts of the program, than fixing a bug in a very large function
Step 1: 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 be to re-run your program
  • As your programs become more complex, reproducing failures becomes more involved
Step 1: 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!
Step 1: Reproduce the Failure (2)

- Reasons for reproducing the failure:
  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!

- None of these reasons require special tools
- **Debugging tools** can help with steps 1 and 2
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
• Extremely effective for helping focus your thoughts and ideas

Caltech/LEAD CS: Summer 2011
The “first” bug:
This is part of a debugging log
And it’s displayed at Smithsonian in Washington DC
Maybe your debug logs will become just as notable one day!
Technique: Debugging Log (3)

- A debugging log also allows you to set aside an issue, and pick it up later.
- It’s generally a very bad idea to debug while exhausted:
  - Thought processes aren’t clear. You can’t reason effectively about the bug.
  - Tend to revert to “monkeys on a typewriter” mode.
  - Put it down and walk away. Come back later when you’re fresh.
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

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 700MB 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 2.1GB.
  - 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
  - Should make it much easier to identify and resolve the defect
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, 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)

- Python is great for isolating failures!
  - Can load your program into the interpreter, and then call various operations individually:
    ```python
    python -i stupid_broken_program.py
    ```

- Mastermind, cont.
  - Load your program, then test the important functions individually:
    ```python
    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 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 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
Given our 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:

```python
def foo():
    ''' This is my awesome function that doesn't work. '''
    bar()
    # TODO: this may be buggy...
    # abc('def')  # Do something amazing!
    return xyz()
```
Technique: Disabling Code (2)

- Can also wrap the code in triple-quotes

```python
def foo():
    ''' This is my awesome function that doesn't work. '''
    ''' TODO: 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/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  
  - (Sadly, it doesn’t find the bugs for you, yet…)
- Using a debugger is a more powerful, less intrusive way of watching your code run  
  - It is definitely possible to introduce other bugs while adding your debug output
- Really only essential for compiled languages  
  - Python has a debugger, but often unnecessary
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

If you’re going to add debug output, might as well print out everything that might be relevant
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…

• 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
  
  ```python
  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:
  
  ```python
  print '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
- 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
- Be a good programmer 😊
  - It will save you tons of time and frustration!
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 of the problem.
  - 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.
Next Time

- Dictionaries and files!