Bitter Lessons from 6.02 Lab 1: A Guide to Debugging

Kanak Kshetri
(Edited by Hari Balakrishnan)

September 17, 2010

1 Introduction

The purpose of this guide is to show you some techniques to answer the immortal question: "Why isn't my code working?"

There are two levels of being wrong:

1. Your code runs, but the result is incorrect.
2. Your code doesn't run at all.

We'll focus on the #2 to begin with.

2 High-level strategy

1. Figure out **what** the problem is.
2. Figure out **where** the problem is coming from.
3. Solve the problem.
4. We discuss five strategies: reading stack traces, sprinkling print statements carefully, using assertions, writing your own test cases, and getting familiar with a real debugger.

3 Strategy 1: Reading stack traces

3.1 Intro

A stack trace is the nasty thing that Python spits out when your code is incorrect and doesn't run correctly. Here's one from when I was doing Task #3 of Lab #1.

```
Traceback (most recent call last):
  File "/afs/athena.mit.edu/user/k/a/kanak/602/lab1/lab1_3.py",
line 130 , in <module>
    rcvd_image = lab1.bits_to_farray(rcvd_bits)
  File "/afs/athena.mit.edu/user/k/a/kanak/602/lab1/lab1.py",
line 111, in bits_to_farray
```

```
    assert len(bits) % nbits == 0, "bits not a multiple of %d"
% nbits
AssertionError: bits not a multiple of 8
```

3.2 Meaning

The stack trace actually has a lot of useful information that can help you figure out what's wrong.

3.2.1 Last Line

- The `AssertionError: bits not a multiple of 8` tells you what went wrong: somewhere, the length of a list was supposed to be a multiple of 8, but it wasn't in the code that executed.
- But you really want to know where exactly did things go awry?

3.2.2 Reading the stack trace backwards

- The line right above the Assertion Error (`assert len(...)`) tells you where it went wrong.
- But that line isn't something I wrote!
- Where is that line coming from?
- If you look at the line in the trace above it, it tells you the filename ("lab1.py") and the exact line number ("111"). It even tells you the function name ("bits_to_farray")
- But I didn't write anything in lab1.py...
- Let's keep going upwards.
- The line above tells us that line 130 in lab1_3.py, is where the problem started.
- So rcvd_bits, which is the result of **my** receive function, is not a multiple of 8.

3.2.3 Pinpointing the problem

- Now, I know what the problem is (I dropped some bits or added extra ones, because I don't have a multiple of 8), and I know where the problem is (somewhere in the receive function).
- I can now start looking at the places where I'm adding or removing things from my list to find the culprit.

3.3 Take-home lessons

- Read the last line of a stack trace to find out the error
- Work your way backwards till you find the place in your code that caused the mistake.

4 Strategy 2: Debugging with print statements

Debugging with print statements is usually a very good idea, not only to figure out why things aren't running, but (often) more importantly to figure out why the results aren't being produced correctly.

4.1 Example 1: Case of the never-ending program

- Suppose you're working on lab1_4, and for some reason the program just doesn't terminate.
- To add to your troubles, you have decided to write a mammoth 50+ line function called "receive" that does everything from digitizing to doing the modulo decoding to finding sync. (It is, in most cases, a bad idea to have functions that are longer than one screen in length.)
- Now which part of the function could be causing the problem?

4.1.1 Be liberal with prints, but not too liberal

- Add a print statement to the start of the function so that we know your function is being called.
- Examples of places where print statements may provide insight:
  - `"Entering receive"`
  - `"Entering receive, sample length:  "`
- Examples of what **not** to print because they don't tell you the context.
  - `"hello"`
  - `"hi"`
  - `"asdfasda"`
- Add a print statement to the end of your function that says something like `"Exiting receive"` so you know when your function is done.
- Now add a print statement in between each "component":
  - before and after digitizing
  - before and after doing modulo
  - before and after sync
- Ideally, each of these components would be a seperate function, but we have seen some cases of large monolithic functions being written by students (bad idea!).
- Run your program.
- You should see when each component runs... this will give you an idea about where the function is spending most of its time.

4.2 Example 2: Case of the empty message

- You're still working on lab1_4, and now your program returns, but it's always returning an empty list.
- What could the problem be?
- There are a lot of places where you're fooling around with the message: digitizing, modulo decoding, finding syncs etc.
- Let's find out what the message length is before and after each of these steps.

4.2.1 Debugging

- Add a print statement right after the function is called... `print "Length of original signal: ", len(samples)`
- Add a print statement before and after you do anything to it.
- Remember to give it a useful label (e.g. `"Length of signal after digitizing: "`) and **not** just `print len(samples)`.... You don't want to get 10 numbers out and have no idea what each number means.

4.3 Take-away lessons

- Adding print statements at strategic locations can help you better understand which part of a function is the culprit. But don't overdo the printing, you will be overwhelmed!
- Even knowing that "digitizer is discarding samples" is valuable information.

5 Strategy: 2 assertions are your friend

5.1 Example: The case of the ill-formed message

- The function that showed the image required a list of numbers, but it inexplicably gave an `Assertion Error: Cannot use += with types 'int' and 'list'`, complaining about the list that my receive function returned.
- What could the problem be?

5.2 What are the assumptions?

- I'm assuming that my receive is actually returning (a) a list, (b) a list whose each element is a number.
- Are these assumptions true?

5.3 Asserts to the rescue

5.3.1 Instances and assertions

- The function, `isinstance(x, y)` returns True if x is an instance of y, and False otherwise.
- Try the following out:
  - `isinstance("hello", str)` returns True
  - `isinstance("hello", int)` returns False
  - `isinstance("hello", (str, list, int))` returns True
- The assert statements allow you to make claims. If the claim doesn't hold, Python will throw an error.
- Try the following out:
  - `assert isinstance("hello", str)` I'm claiming that "hello" is a string. It is, so the statement has no effect.
  - `assert isinstance(2, str)` I'm claiming that 2 is a string. It is not, so you should see an "AssertionError"
- You can actually pass any expression that returns a boolean value to assert, such as the following (try these out too):
  - `assert 111 % 2 == 0` I claimed 111 is divisible by 2, it's not (ERROR)
  - `assert len([]) == 1` I claimed the length should be 1, it's not (ERROR)

5.3.2 Verifying my assumptions: Part 1

- Am I actually returning a list?
- I'm going to add `assert isinstance(bits, list)` before **every** `return` statement.
- This way, Python will yell at me if I do something crazy like return an int.

5.3.3 Verifying my assumptions: Part 2

- Am I actually returning a list of numbers?
- This is a little more tricky to do... Exercise: write a function that returns False whenever there is a non-int in a list. The answer is in this footnote, but don't look at it (or the bullet point below)until you try it out first![1]
- Now I can say: `assert all(isinstance(i, int) for i in bits)` where `bits` is the name of my list of numbers
- Again, if I'm doing something crazy like returning "None" in my list, Python will yell at me (figuratively!).

[1] Here's a one-liner I came up with: `all(isinstance(i, int) for i in lst)`

5.4 Take-away lessons

- A lot of people made the mistake of using `append` (which adds a value to the end of list) when they should have used `extend` (which joins two lists together)
- Usually, this resulted in the image drawing function complaining about ints and lists and what not
- Using asserts would have warned you that your assumption (you're returning a list of numbers) is not true.

6 Strategy 4: Use YOUR own test cases

- There's nothing stopping you from writing your own test cases to see if your functions are working correctly.
- For example, let's say you're not 100% sure that your sync-finder is working.
- Don't just keep running the program with the test image-signal and hope that it finds the sync! (That generally won't work out too well for you, unless you have some reason to believe that the bug is non-deterministic and you need more information to figure out what's going on!)
- Make up your own signal that is `random bits + SYNC + random bits`, and see if your function finds the sync.
- There are some seasoned software engineers who believe that you should actually write your test cases **before** you write your code because thinking about what inputs generate what outputs gives you a very good understanding of the problem. This method is also known as test-driven development.

7 Strategy 5: Using an industrial-strength (or at least, real) debugger

- Idle actually has a good debugger accessible through the Debug menu.
- Learning to use it effectively will, more often than not, help you debug programs faster than relying on print statements, particularly for more complex bugs. But the strategies we mentioned above will still be valuable.

8 Some common sources of bugs

We summarize here some common sources of bugs we saw in Lab 1.

8.1 Modifying the index variable in a for loop

- If you have a for loop, `for i in range(blah blah)` please, for the love of God, **do not modify** `i` **inside the body of the for loop**. For instance,

```
for i in range(10):
    i = 9
```

- If you need to control the way your index variable is changing, use `while` and take complete control of it. Changing i inside the loop as in the above example is asking for a huge amount of trouble.

8.2 Use "is not None" instead of "!= None"

- There are some places where this might be a problem.
- Read this if you're interested.

8.3 Writing your own code instead of relying on library functions

- Why write your own implementation of "mean" when you can use numpy's mean function?
- Why write your own implementation of "append" or "extend" or even list slicing?
- Whenever you're using a while loop, ask yourself: could I just use a for here?

The tongue-in-cheek lesson is: be lazy! (But please start the labs early.) Search for library functions and write your own only when you can't find one. Please feel free to ask around whether a certain library function is likely to be available or not. It's hard to write bug-free code, so don't add to the burden by writing code that exists in libraries that others have painstakingly debugged!

9 Conclusion

- Ultimately, nothing beats actually understanding what you're trying to do, verifying the assumptions that you're making, and writing clear code where each function does exactly a single task.
- An excellent resource to properly understand programming is How To Design Programs. You probably don't have the time to read the entire book (YOU SHOULD, AT SOME POINT, IF YOU'RE 6-3), but you should at the very least, read this page).
- It's better to take 5 minutes to think about the problem carefully before starting to code, than to immediately start writing code and spending hours debugging it.
- It is important to know how to debug, but you should try to write code that reduces the amount of time you spend debugging. (We know, that's asking for the moon.)