

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 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

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 $x = 0, y \neq 0$ into $x = 0, y$ positive and $x = 0, y$ negative
 - Same for $x \neq 0, y = 0$

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
```

```
         $x \geq 0$ 
```

```
         $\text{eps} > 0$ 
```

```
    returns res such that
```

```
         $x - \text{eps} \leq \text{res} * \text{res} \leq x + \text{eps}$ """
```

- 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

Debugging

- The “history” of debugging
 - Often claimed that first bug was found by team at Harvard that was working on the Mark II Aiken Relay Calculator
 - A set of tests on a module had failed; when staff inspected the actual machinery (in this case vacuum tubes and relays), they discovered this:

92

9/9

0800 Antan started
 1000 " stopped - antan ✓
 1300 (032) MP-MC ~~1.582147000~~
 (033) PRO 2 2.130476415
 correct 2.130676415

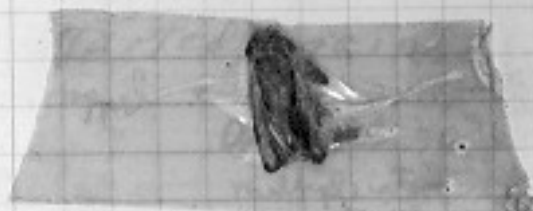
{ 1.2700 9.037847025
 9.037846995 correct
 4.615925059(-2)

Relays 6-2 in 033 failed special speed test
 in relay 10.000 test -

Relay
 214
 Relay 3

1100 Relays changed
 Started Cosine Tape (Sine check)
 1525 Started Mult+ Adder Test.

1545



Relay #70 Panel F
 (moth) in relay.

First actual case of bug being found.

1630 Antan started.
 1700 closed down.

A real bug!

- However, the term bug dates back even earlier:
 - *Hawkin's New Catechism of Electricity, 1896*
 - “The term ‘bug’ is used to a limited extent to designate any fault or trouble in the connections or working of electrical apparatus.”

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

```
def isPal(x):  
    assert type(x) == list  
    temp = x  
    temp.reverse  
    if temp == x:  
        return True  
    else:  
        return False
```

```
def silly(n):  
    for i in range(n):  
        result = []  
        elem = raw_input('Enter element: ')  
        result.append(elem)  
    if isPal(result):  
        print('Yes')  
    else:  
        print('No')
```

Stepping through the tests

- Suppose we run this code:
 - We try the input 'abcba', which succeeds
 - We try the input 'palinnilap', which succeeds
 - But we try the input 'ab', which also 'succeeds'
- Let's use binary search to isolate bug(s)
- Pick a spot about halfway through code, and devise experiment
 - Pick a spot where easy to examine intermediate values

```
def isPal(x):  
    assert type(x) == list  
    temp = x  
    temp.reverse  
    if temp == x:  
        return True  
    else:  
        return False
```

```
def silly(n):  
    for i in range(n):  
        result = []  
        elem = raw_input('Enter element: ')  
        result.append(elem)  
    print(result)  
    if isPal(result):  
        print('Yes')  
    else:  
        print('No')
```



Stepping through the tests

- At this point in the code, we expect (for our test case of 'ab'), that result should be a list ['a', 'b']
- We run the code, and get ['b'].
- Because of binary search, we know that at least one bug must be present earlier in the code
- So we add a second print

```
def isPal(x):  
    assert type(x) == list  
    temp = x  
    temp.reverse  
    if temp == x:  
        return True  
    else:  
        return False
```

```
def silly(n):  
    for i in range(n):  
        result = []  
        elem = raw_input('Enter element: ')  
        result.append(elem)  
        print(result)  
    if isPal(result):  
        print('Yes')  
    else:  
        print('No')
```



Stepping through

- When we run with our example, the print statement returns
 - ['a']
 - ['b']
- This suggests that result is not keeping all elements
 - So let's move the initialization of result outside the loop and retry

```
def isPal(x):  
    assert type(x) == list  
    temp = x  
    temp.reverse  
    if temp == x:  
        return True  
    else:  
        return False
```

```
def silly(n):  
    result = []  
    for i in range(n):  
        elem = raw_input('Enter element: ')  
        result.append(elem)  
        print(result)  
    if isPal(result):  
        print('Yes')  
    else:  
        print('No')
```



Stepping through

- So this now shows we are getting the data structure result properly set up, but we still have a bug somewhere
 - A reminder that there may be more than one problem!
 - This suggests second bug must lie below print statement; let's look at isPal
 - Pick a point in middle of code, and add print statement again

```
def isPal(x):  
    assert type(x) == list  
    temp = x  
    temp.reverse  
    print(temp, x)  
    if temp == x:  
        return True  
    else:  
        return False
```

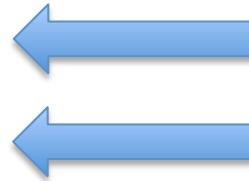


```
def silly(n):  
    result = []  
    for i in range(n):  
        elem = raw_input('Enter element: ')  
        result.append(elem)  
    if isPal(result):  
        print('Yes')  
    else:  
        print('No')
```

Stepping through

- At this point in the code, we expect (for our example of 'ab') that x should be ['a', 'b'], but temp should be ['b', 'a'], however they both have the value ['a', 'b']
- So let's add another print statement, earlier in the code

```
def isPal(x):  
    assert type(x) == list  
    temp = x  
    print(temp, x)  
    temp.reverse  
    print(temp, x)  
    if temp == x:  
        return True  
    else:  
        return False
```

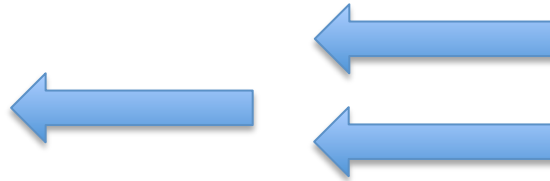


```
def silly(n):  
    result = []  
    for i in range(n):  
        elem = raw_input('Enter element: ')  
        result.append(elem)  
    if isPal(result):  
        print('Yes')  
    else:  
        print('No')
```

Stepping through

- And we see that `temp` has the same value before and after the call to `reverse`
- If we look at our code, we realize we have committed a standard bug – we forgot to actually invoke the `reverse` method
 - Need `temp.reverse()`
- So let's make that change and try again

```
def isPal(x):  
    assert type(x) == list  
    temp = x  
    print(temp, x)  
    temp.reverse()  
    print(temp, x)  
    if temp == x:  
        return True  
    else:  
        return False
```



```
def silly(n):  
    result = []  
    for i in range(n):  
        elem = raw_input('Enter element: ')  
        result.append(elem)  
    if isPal(result):  
        print('Yes')  
    else:  
        print('No')
```

Stepping through

- But now when we run on our simple example, both x and temp have been reversed!!
- We have also narrowed down this bug to a single line. The error must be in the reverse step
- In fact, we have an aliasing bug – reversing temp has also caused x to be reversed
 - Because they are referring to the same object

```
def isPal(x):
```

```
    assert type(x) == list
```

```
    temp = x[:]
```

```
    print(temp, x)
```

```
    temp.reverse()
```

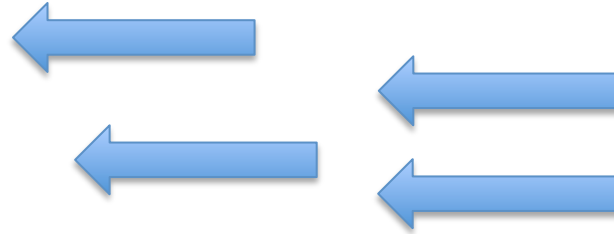
```
    print(temp, x)
```

```
    if temp == x:
```

```
        return True
```

```
    else:
```

```
        return False
```



```
def silly(n):
```

```
    result = []
```

```
    for i in range(n):
```

```
        elem = raw_input('Enter element: ')
```

```
        result.append(elem)
```

```
    if isPal(result):
```

```
        print('Yes')
```

```
    else:
```

```
        print('No')
```


Stepping through

- And now running this shows that before the reverse step, the two variables have the same form, but afterwards only temp is reversed.
- We can now go back and check that our other tests cases still work correctly

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