TESTING, DEBUGGING

PROGRAMMING CHALLENGES

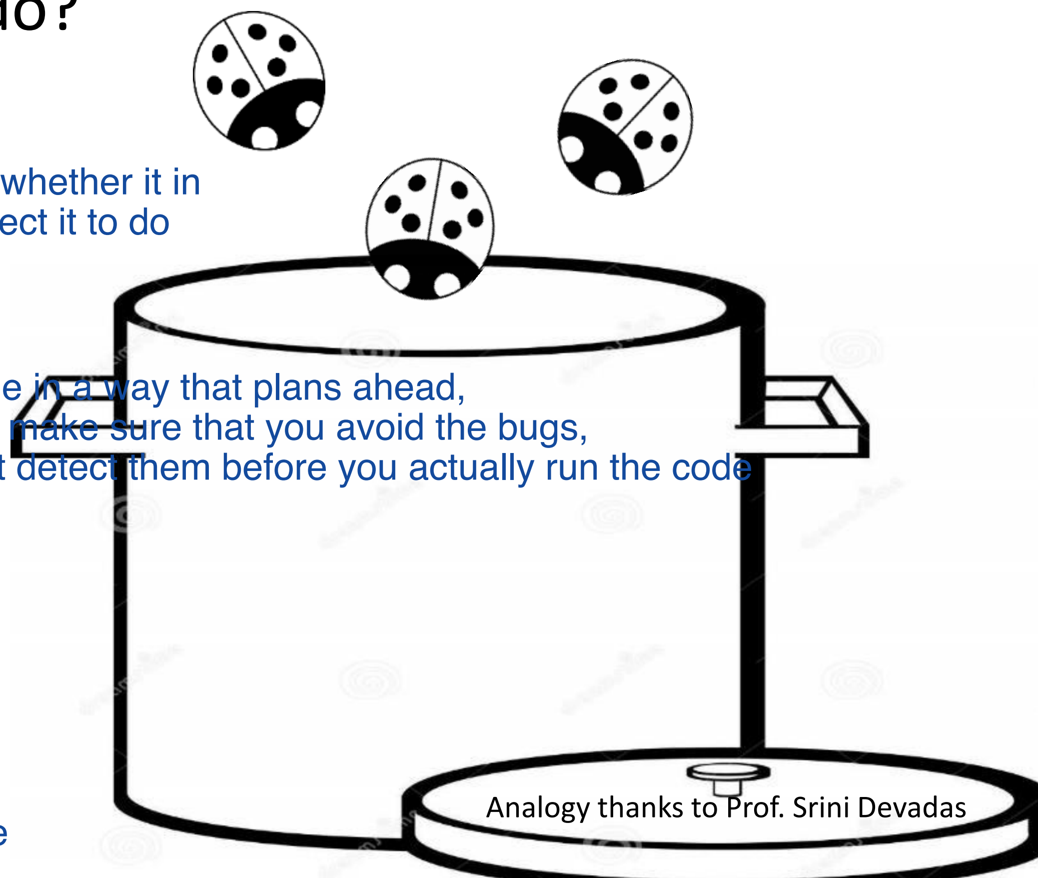
EXPECTATION **REALITY**

What you want the program to do **What the program actually does**

WE AIM FOR HIGH QUALITY – AN ANALOGY WITH SOUP

You are making soup but bugs keep falling in from the ceiling. What do you do?

- check soup for bugs
  - **testing** test my code to see whether it in fact does what I expect it to do
- keep lid closed
  - **defensive programming** write code in a way that plans ahead, to try and make sure that you avoid the bugs, or at least detect them before you actually run the code
- clean kitchen
  - **eliminate source of bugs - debugging** eliminate the source of bugs. Do the debugging to get it done



Analogy thanks to Prof. Srini Devadas

DEFENSIVE PROGRAMMING

- Write **specifications** for functions

  docstrings that clarify arguments and return

- **Modularize** programs

  Don't write one really long huge code, instead: Break it up into selfcontained pieces and test each piece

- Check **conditions** on inputs/outputs (assertions)

  to formally do that

TESTING/VALIDATION

- **Compare** input/output pairs to specification
- "It's not working!"
- "How can I break my program?"

DEBUGGING

- **Study events** leading up to an error
- "Why is it not working?"
- "How can I fix my program?"

- Given a set of functions I want to test, I should write a list of a set of example inputs,and what I expect in each case as an output.

- What are the right kinds of inputs to use to make sure I test all of the different cases that are going to be important, ensuring that my program is doing the right thing?

SET YOURSELF UP FOR EASY TESTING AND DEBUGGING

- from the **start**, design code to ease this part
- break program into **modules** that can be tested and debugged individually

also going to guide creation of a particular test cases to make sure that your assumptions are in fact valid, or that the code is being entered in a manner that supports those assumptions

- **document constraints** on modules

what do you expect the input to be? the output to be?

- **document assumptions** behind code design

thinking process when you created this code?

"Motherhood and apple pie" approach:
Something that cannot be questioned because it appeals to universally-held, wholesome values

WHEN ARE YOU READY TO TEST?

- ensure **code runs**
  - remove syntax errors
  - remove static semantic errors   don't form a well-formed expression
  - Python interpreter can usually find these for you
- have a **set of expected results**
  - an input set
  - for each input, the expected output

so that when you run the code, you can spot the places where it isn't doing the right kind of thing

CLASSES OF TESTS

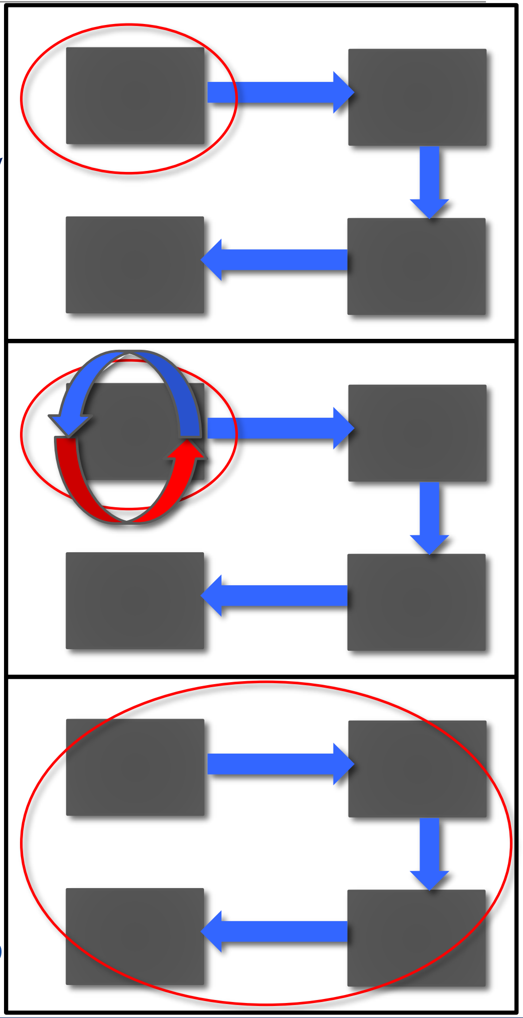
start and
go back to

- **Unit testing**
  - validate each piece of program
  - **testing each function** separately
- **Regression testing**
  - add test for bugs as you find them in a function
  - **catch reintroduced** errors that were previously fixed

by after each fix,
testing again
the same unit

- **Integration testing**
  - does **overall program** work?
  - tend to rush to do this

finally, after having debugged each of the pieces, now you need to make sure that they hand off information correctly to each other.



TESTING APPROACHES

- **intuition** about natural boundaries to the problem

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

  can you come up with some natural partitions?

- if no natural partitions, might do **random testing**

  probability that code is correct increases with more tests

  better options below

- **black box testing**

  explore paths through specification

- **glass box testing**

  explore paths through code

method of software testing that tests the functionality of an application
look at both: possible paths through the specification, possible boundary cases

BLACK BOX TESTING

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

- designed **without looking** at the code
- can be done by someone other than the implementer to avoid some implementer **biases**
- testing can be **reused** if implementation changes
- **paths** through specification
  - build test cases in different natural space partitions
  - also consider boundary conditions (empty lists, singleton list, large numbers, small numbers)

BLACK BOX TESTING

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

CASE	x	eps
boundary	0	0.0001
Perfect square	25	0.0001
Less than 1	0.05	0.0001
Irrational square root	2	0.0001
extremes	2	1.0/2.0**64.0
extremes	1.0/2.0**64.0	1.0/2.0**64.0
extremes	2.0**64.0	1.0/2.0**64.0
extremes	1.0/2.0**64.0	2.0**64.0
extremes	2.0**64.0	2.0**64.0

small numb ers

large numbers

test cases that go through every possible path in the code
know how the function you are testing is defined. Thus you can use this definition to figure out how many different paths through the code exist, and then pick a test suite based on that knowledge.

GLASS BOX TESTING

- **use code** directly to guide design of test cases
- called **path-complete** if every potential path through code is tested at least once
- what are some **drawbacks** of this type of testing?
    - can go through loops arbitrarily many times
    - missing paths
- guidelines
    - branches → exercise all parts of a conditional
    - for loops → loop not entered; body of loop executed exactly once; body of loop executed more than once
    - while loops → same as for loops, cases that catch all ways to exit loop

GLASS BOX TESTING

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

- a path-complete test suite could **miss a bug**
- path-complete test suite: 2 and -2
- but abs(-1) incorrectly returns -1
- should still test boundary cases

good glass box test suite would try to test a good sample of all the possible paths through the code
In glass box testing, we try to sample as many paths through the code as we can. In the case of loops, we want to sample three general cases:

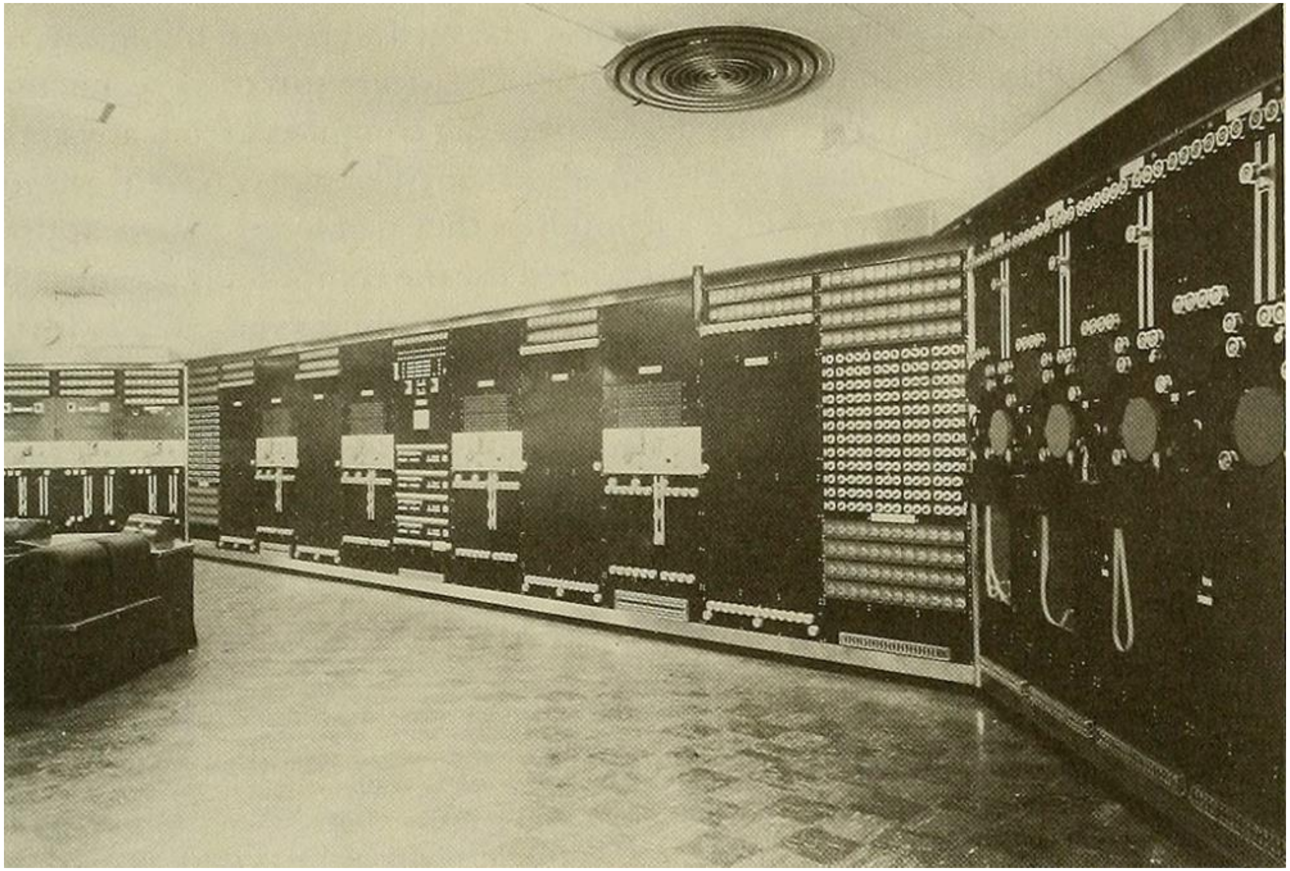
Not executing the loop at all.
Executing the loop exactly once.
Executing the loop multiple times.

BUGS

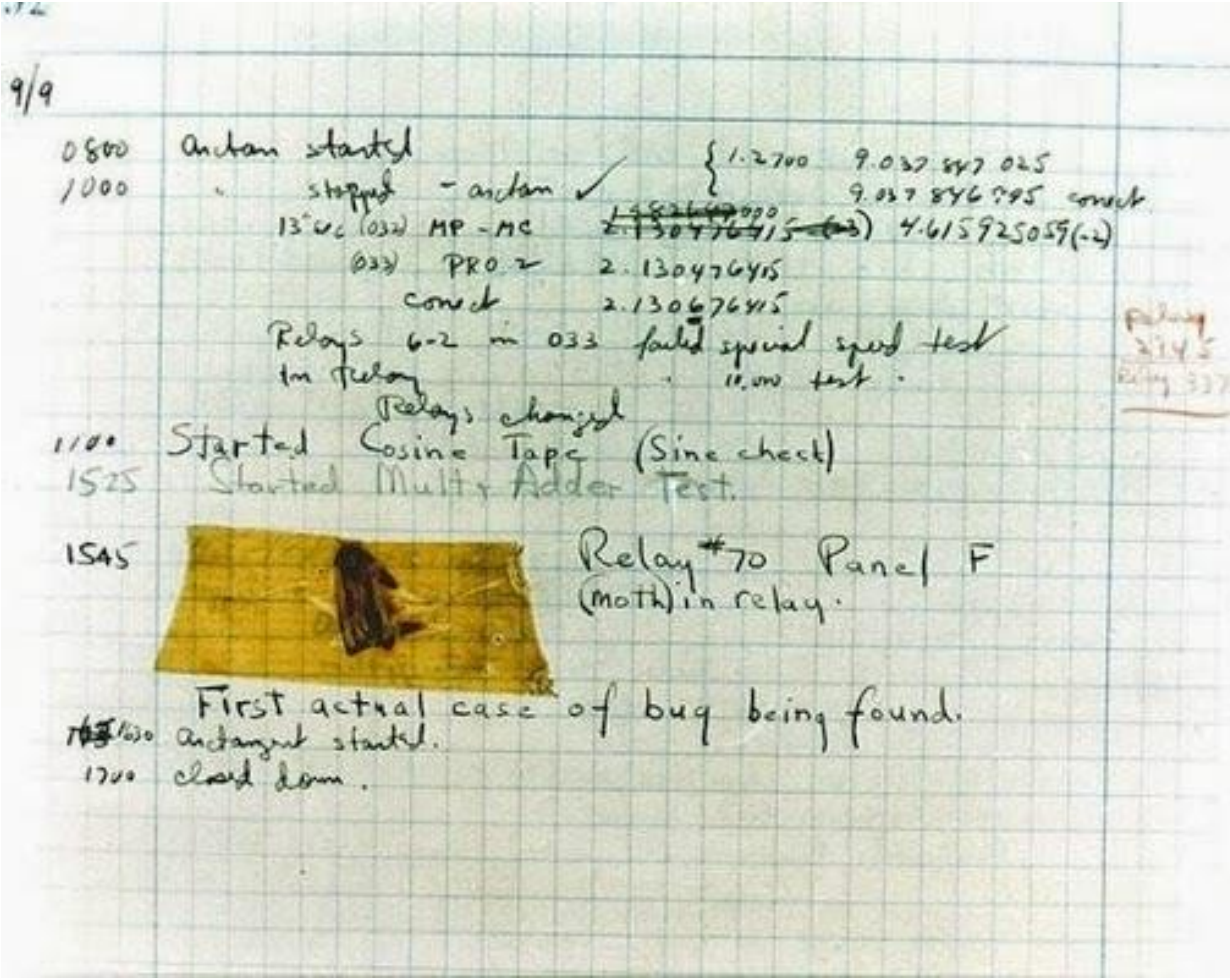
- once you have discovered that your code does not run properly, you want to:
  - isolate the bug(s)
  - eradicate the bug(s)
  - retest until code runs correctly

September 9, 1947

- Mark II Aiken Relay Computer



Admiral Grace Murray Hopper


9/9
0800 Anctan started
1000 stopped - anctan
1.2700 9.037 847 025
9.037 846 995 correct
MP - MC 4.615925059(-2)
(033) PRO 2 2.130476415
correct 2.130676415
Relays 6-2 in 033 failed special speed test
In relay 10,000 test.
Relays changed
1100 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 Antangent started.
1700 closed down.

RUNTIME BUGS

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

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

force bugs 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

- steep learning curve
- goal is to have a bug-free program
- tools
  - **built in** to IDLE and Anaconda
  - **Python Tutor**
  - `print` statement
  - use your brain, be **systematic** in your hunt

Insert statements at different points in your code that will print out, here's what I'm expecting, here's what I'm seeing.-> help you isolate where the code may be going wrong

PRINT STATEMENTS

- good way to **test hypothesis**
- when to print
  - enter function
  - parameters
  - function results
- use **bisection method**
  - put print halfway in code
  - decide where bug may be depending on values

ERROR MESSAGES - EASY

let the error messages actually guide what you're looking for in terms of the bug

- trying to access beyond the limits of a list

  `test = [1,2,3]` then `test[4]` → `IndexError`

  check the bounds of what I'm using as indices to look at structures

- trying to convert an inappropriate type

  `int(test)` → `TypeError`

- referencing a non-existent variable

  `a` → `NameError`

- mixing data types without appropriate coercion

  `'3'/4` → `TypeError`

- forgetting to close parenthesis, quotation, etc.

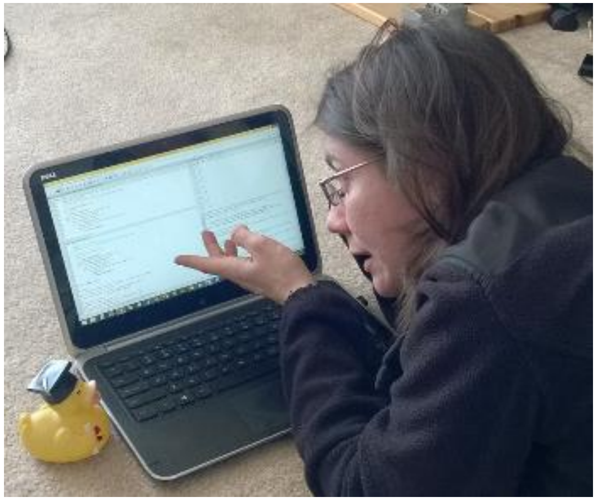
```
a = len([1,2,3]
print a
```
→ `SyntaxError`

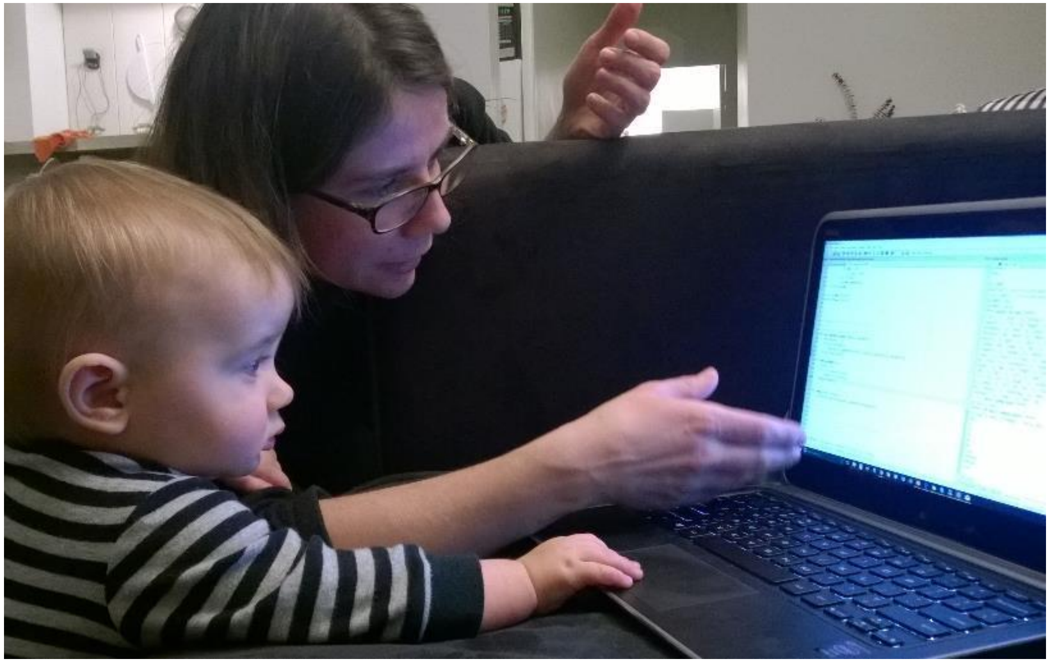
LOGIC ERRORS - HARD

- **think** before writing new code
- **draw** pictures, take a break
- **explain** the code to

  someone else

  a rubber ducky





DEBUGGING STEPS

- **study** program code
  - ask how did I get the unexpected result
  - don't ask what is wrong
  - is it part of a family?
- **scientific method**
  - study available data
  - form hypothesis what might be causing that particular error to occur.
  - repeatable experiments
  - pick simplest input to test with

DON'T

DO

DON'T		DO
Write entire program Test entire program Debug entire program	→	Write a function Test the function, debug the function Write a function Test the function, debug the function *** Do integration testing ***
Change code Remember where bug was Test code Forget where bug was or what change you made Panic	→	Backup code Change code Write down potential bug in a comment Test code Compare new version with old version

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

tell me whether something is a palindrome or not

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

*take the list,
take a reversed version of the list, and then
compare them, Because if they're the same, that's
another way of testing if it's a palindrome or not

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

type in a set of characters
up to that number of characters.
Use that to pull together a list

*call isPal

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’ bug!
- 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 = input('Enter element: ')
        result.append(elem)
    print(result)
    if isPal(result):
        print('Yes')
    else:
        print('No')
```

after loop

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, this time inside the loop

```python
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 = input('Enter element: ')
        result.append(elem)
        print(result)
    if isPal(result):
        print('Yes')
    else:
        print('No')
```

inside loop

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 = input('Enter element: ')
        result.append(elem)
        print(result)
    if isPal(result):
        print('Yes')
    else:
        print('No')
```

list moved outside the loop

inside loop, list was reset each time through the loop

STEPPING THROUGH

- 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; remove the earlier print statement

```python
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 = input('Enter element: ')
        result.append(elem)
    if isPal(result):
        print('Yes')
    else:
        print('No')
```

take x which should be a list of these characters.
I'm creating a temporary version of list x and I'm trying to reverse it.
So I'd like to check at this point is temp different from x?

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

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

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

STEPPING THROUGH

- 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

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

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

now temp and x were the same before hand, now both of them are reversed.

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

(mutable list)

```python
def isPal(x):
    assert type(x) == list
    temp = x[:]
    print('before reverse', temp, x)
    temp.reverse()
    print('after reverse', temp, x)
    if temp == x:
        return True
    else:
        return False

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

easy fix: make a copy of list, so now x and temp are lists that are equal, but not the same object

STEPPING THROUGH

- 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 -> binary search!
- explain the problem to someone else
- don't believe the documentation
- take a break and come back to the bug later