BBM 101

Introduction to Programming I

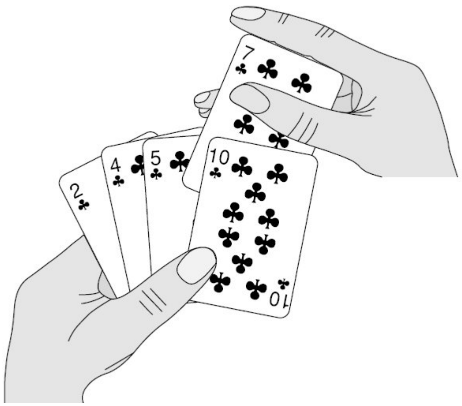
The Adventures of Sherlock Holmes (1939)

Lecture #08 – Testing and Debugging

HACETTEPE UNIVERSITY

Erkut Erdem, Aykut Erdem & Aydın Kaya // Fall 2017

Last time... **Sorting, List Comprehension, Visualization**



Sorting

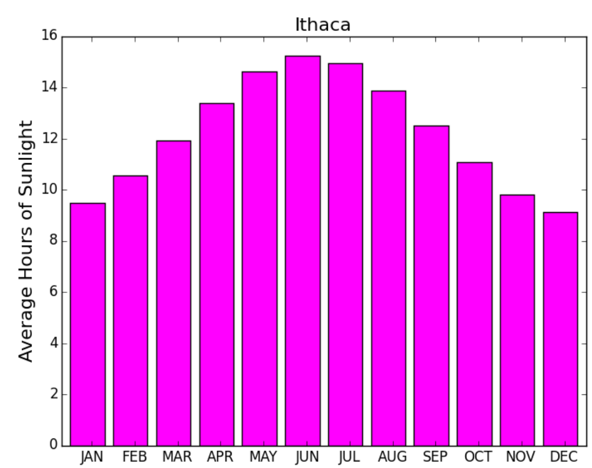
```
print("hamlet:", hamlet)

print("sorted(hamlet):",
sorted(hamlet))
print("hamlet:", hamlet)

print("hamlet.sort():",
hamlet.sort())
print("hamlet:", hamlet)
```

List comprehension

```
[i*2 for i in range(3)]
```

Data visualization

```
M = MonthAverages('Ithaca')
bar(range(12),M,facecolor='magenta')
xlim(-.2,12)
ylabel('Average Hours of Sunlight')
title(A.City,fontsize=16)
show()
```



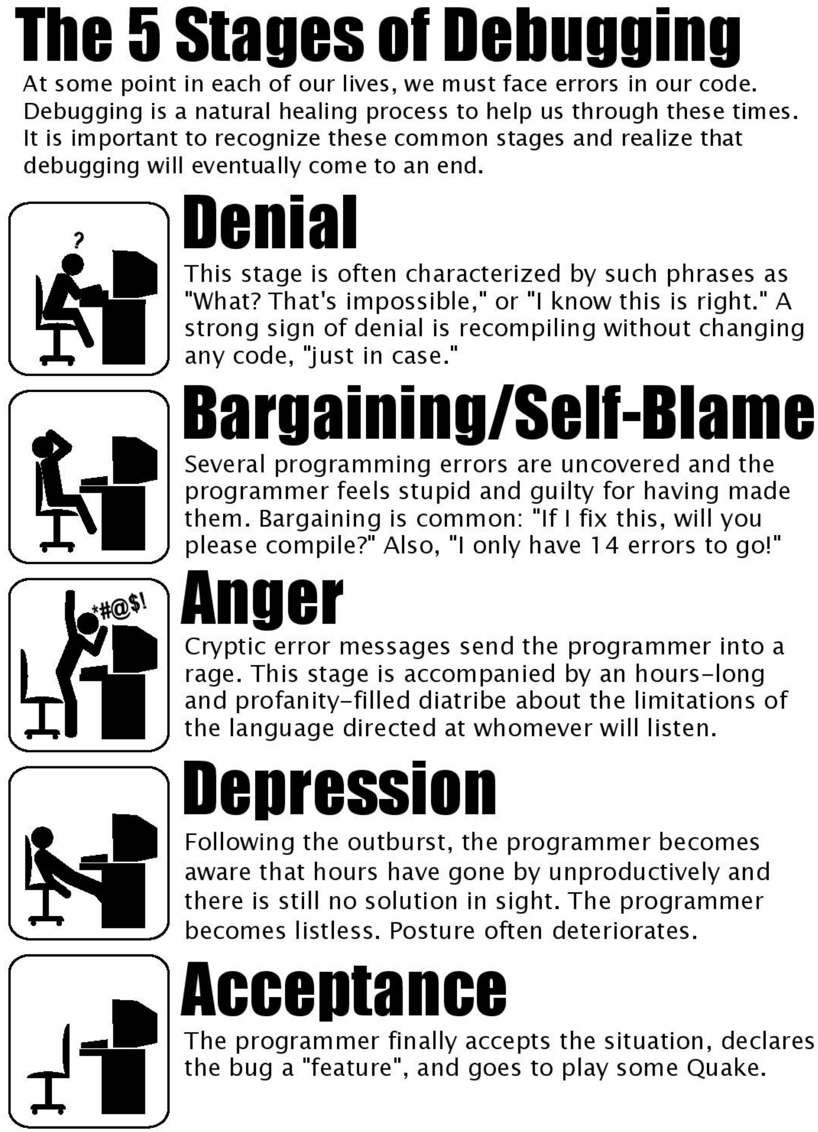
Lecture Overview

- Debugging
- Exception Handling
- Testing

Disclaimer: Much of the material and slides for this lecture were borrowed from
—R. Anderson, M. Ernst and B. Howe in University of Washington CSE 140

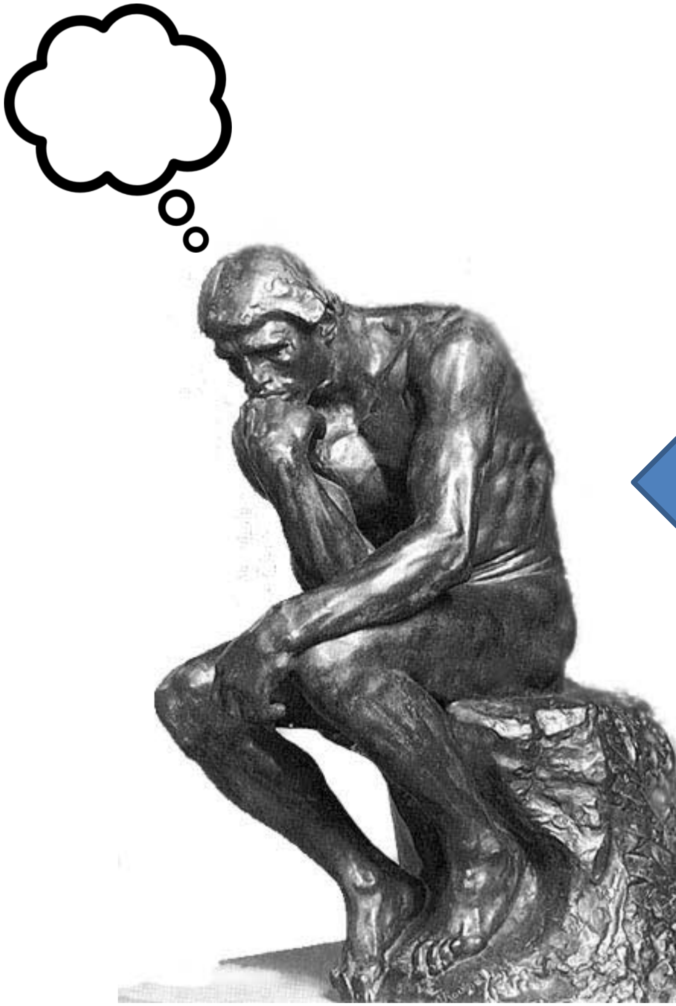
Lecture Overview

- Debugging
- Exception Handling
- Testing



https://www.reddit.com/r/ProgrammerHumor/comments/1r0cw7/the_5_stages_of_debugging/

The Problem



Not the same!

There is a bug!

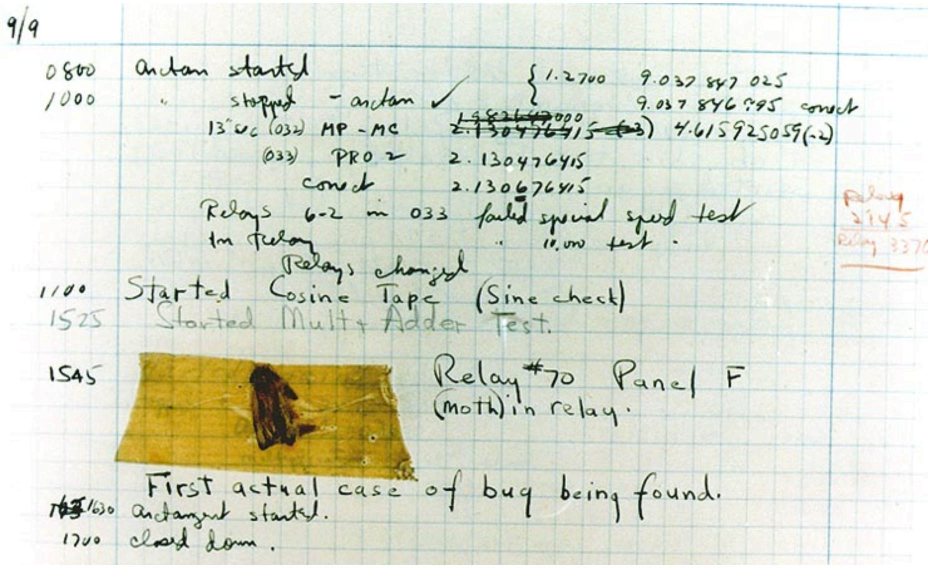
What you want
your program to do

What your program does

What is Debugging?

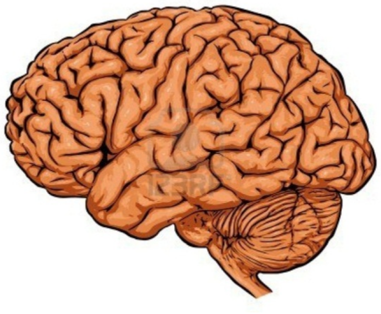
- Grace Hopper was one of U.S.'s first programmers
- She found a moth in the Mark I computer, which was causing errors, and called it a computer "bug"
- Thus, the word debugging is coined ☺

FIG. 1. The first computer bug



Debugging Tools

- Python error message
- **`assert`**
- **`print`**
- Python interpreter
- Python Tutor (http://pythontutor.com)
- Python debugger
- Best tool:



Two Key Ideas

1. The scientific method
2. Divide and conquer

If you master those, you will find debugging easy, and possibly enjoyable ;-)

The Scientific Method

1. Create a hypothesis
2. Design an experiment to test that hypothesis
   – Ensure that it yields insight
3. Understand the result of your experiment
   – If you don't understand, then possibly suspend your main line of work to understand that

The Scientific Method

Tips:

- Be systematic
  - Never do anything if you don't have a reason
  - Don’t just flail
    - Random guessing is likely to dig you into a deeper hole
- Don’t make assumptions (verify them)

Example Experiments

1. An alternate implementation of a function
   – Run all your test cases afterward

2. A new, simpler test case
   – Examples: smaller input, or test a function in isolation
   – Can help you understand the reason for a failure

Your Scientific Notebook

Record everything you do

- Specific inputs and outputs (both expected and actual)
- Specific versions of the program
  - If you get stuck, you can return to something that works
  - You can write multiple implementations of a function
- What you have already tried
- What you are in the middle of doing now
  - This may look like a stack!
- What you are sure of, and why

Your notebook also helps if you need to get help or reproduce your results.

Read the Error Message

```
Traceback (most recent call last):
  File "nx_error.py", line 41, in <module>
    print(friends_of_friends(rj, myval))
  File "nx_error.py", line 30, in friends_of_friends
    f = friends(graph, user)
  File "nx_error.py", line 25, in friends
    return set(graph.neighbors(user))#
  File "/Library/Frameworks/…/graph.py", line 978, in neighbors
     return list(self.adj[n])
TypeError: unhashable type: 'list'
```

First function that was called (**<module>** means the interpreter)

Second function that was called

Call stack or traceback

Last function that was called (this one suffered an error)

The error message: daunting but useful.
You need to understand:

- the literal meaning of the error
- the underlying problems certain errors tend to suggest

List of all exceptions (errors):
http://docs.python.org/3/library/exceptions.html#bltin-exceptions
Two other resources, with more details about a few of the errors:
http://inventwithpython.com/appendixd.html
http://www.cs.arizona.edu/people/mccann/errors-python

Common Error Types

- **`AssertionError`**
  - Raised when an assert statement fails.
- **`IndexError`**
  - Raised when a sequence subscript is out of range.
- **`KeyError`**
  - Raised when a mapping (dictionary) key is not found in the set of existing keys.
- **`KeyboardInterrupt`**
  - Raised when the user hits the interrupt key (normally Control-C or Delete).

Common Error Types

- **`NameError`**
  - Raised when a local or global name is not found.
- **`SyntaxError`**
  - Raised when the parser encounters a syntax error.
- **`IndentationError`**
  - Base class for syntax errors related to incorrect indentation.
- **`TypeError`**
  - Raised when an operation or function is applied to an object of inappropriate type.

Divide and Conquer

- Where is the defect (or “bug”)?
- Your goal is to find the one place that it is
- Finding a defect is often harder than fixing it

- Initially, the defect might be anywhere in your program
  - It is impractical to find it if you have to look everywhere
- Idea: bit by bit reduce the scope of your search
- Eventually, the defect is localized to a few lines or one line
  - Then you can understand and fix it

Divide and Conquer

- 4 ways to divide and conquer:
  - In the program code
  - In test cases
  - During the program execution
  - During the development history

Divide and Conquer in the Program Code

- Localize the defect to part of the program
  - e.g., one function, or one part of a function
- Code that isn't executed cannot contain the defect

Divide and Conquer in the Program Code

Three approaches:

1. Test one function at a time

Divide and Conquer in the Program Code

Three approaches:

2. Add assertions or print statements
   - The defect is executed before the failing assertion (and maybe after a succeeding assertion)

Divide and Conquer in the Program Code

Three approaches:

3. Split complex expressions into simpler ones

Example: Failure in

```
result = set({graph.neighbors(user)})
```

Change it to

```
nbors = graph.neighbors(user)
nbors_set = {nbors}
result = set(nbors_set)
```

The error occurs on the “nbors_set = {nbors}" line

Divide and Conquer in Test Cases

- Your program fails when run on some large input
  - It’s hard to comprehend the error message
  - The log of print statement output is overwhelming
- Try a smaller input
  - Choose an input with some but not all characteristics of the large input
  - Example: duplicates, zeroes in data, …

Divide and Conquer in Execution Time via Print (or “logging”) Statements

- A sequence of **`print`** statements is a record of the execution of your program
- The **`print`** statements let you see and search multiple moments in time
- Print statements are a useful technique, in moderation
- Be disciplined
  - Too much output is overwhelming rather than informative
  - Remember the scientific method: have a reason (a hypothesis to be tested) for each print statement
  - Don’t *only* use print statements

Divide and Conquer in Development History

- The code used to work (for some test case)
- The code now fails
- The defect is related to some line you changed

- This is useful only if you kept a version of the code that worked (use good names!)
- This is most useful if you have made few changes
- Moral: test often!
  - Fewer lines to compare
  - You remember what you were thinking/doing recently

A Metaphor About Debugging

If your code doesn't work as expected, then by definition you don't understand what is going on.

- You're lost in the woods.
- You're behind enemy lines.
- All bets are off.
- Don't trust anyone or anything.

Don't press on into unexplored territory -- go back the way you came! (and leave breadcrumbs!)

You're trying to "advance the front lines," not "trailblaze"

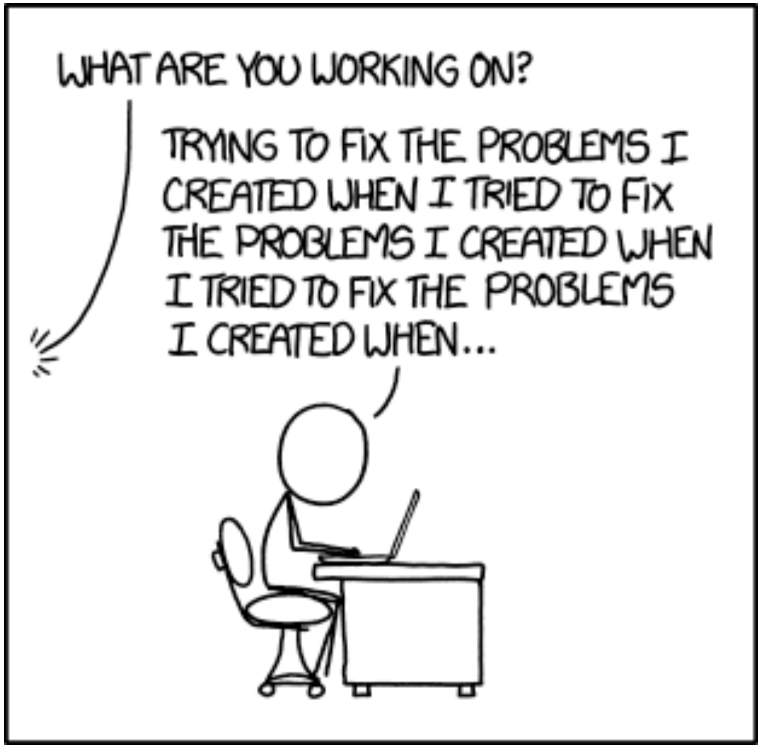
Time-Saving Trick: Make Sure You are Debugging the Right Problem

- The game is to go from "working to working"
- When something doesn't work, STOP!
  - It's wild out there!
- FIRST: Go back to the last situation that worked properly.
  - Rollback your recent changes and verify that everything still works as expected.
  - Don't make assumptions – by definition, you don't understand the code when something goes wrong, so you can't trust your assumptions.
  - You may find that even what previously worked now doesn't
  - Perhaps you forgot to consider some "innocent" or unintentional change, and now even tested code is broken

A Bad Timeline

- A works, so celebrate a little
- Now try B
- B doesn't work
- Change B and try again
- Change B and try again
- Change B and try again

…



https://xkcd.com/1739/

A Bad Timeline

- A works, so celebrate a little
- Now try B
- B doesn't work
- Change B and try again
- Change B and try again
- Change B and try again

…

from giphy.com

A Better Timeline

- A works, so celebrate a little
- Now try B
- B doesn't work
- *Rollback to A*
- Does A still work?
  - Yes: Find A' that is somewhere between A and B
  - No: You have *unintentionally changed something else*, and there's no point futzing with B at all!

These "innocent" and unnoticed changes happen more than you would think!

- You add a comment, and the indentation changes.
- You add a print statement, and a function is evaluated twice.
- You move a file, and the wrong one is being read
- You are on a different computer, and the library is a different version

Once You are on Solid Ground You can Set Out Again

- Once you have something that works and something that doesn't work, it is only a matter of time
- You just need to incrementally change the working code into the non-working code, and the problem will reveal itself.
- Variation: Perhaps your code works with one input, but fails with another. Incrementally change the good input into the bad input to expose the problem.

Simple Debugging Tools

`print`

- shows what is happening whether there is a problem or not
- does not stop execution

`assert`

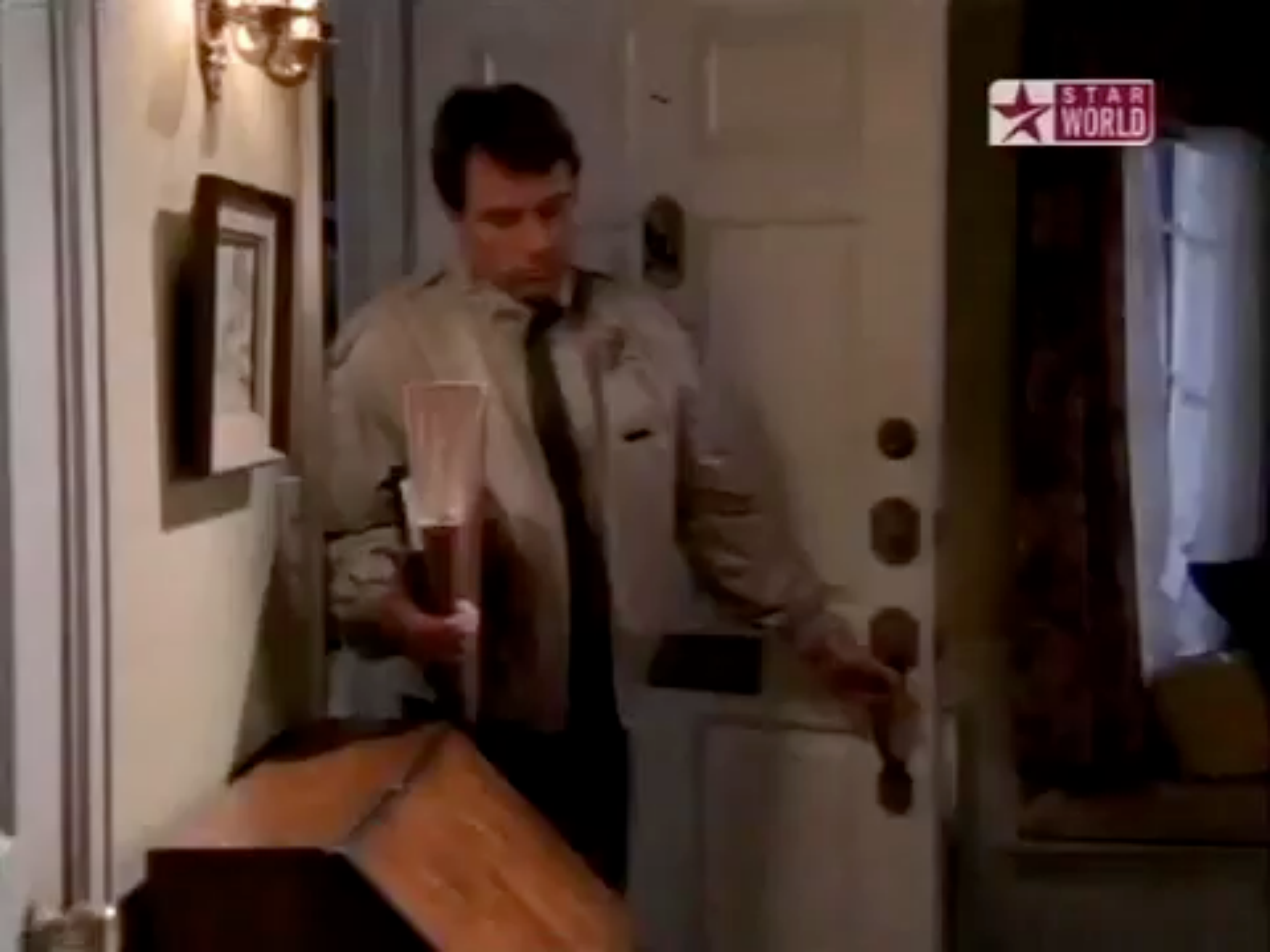
- Raises an exception if some condition is not met
- Does nothing if everything works
- Example: `assert len(rj.edges()) == 16`
- Use this liberally! Not just for debugging!

David Amador
@DJ_Link

when a "simple" bug is found and we start looking at the code

12:22 PM · Jul 3, 2017

2,181 Retweets **3,400** Likes


STAR
WORLD

Lecture Overview

- Debugging
- Exception Handling
- Testing

What is an Exception?

- An exception is an abnormal condition (and thus rare) that arises in a code sequence at runtime.
- For instance:
  - Dividing a number by zero
  - Accessing an element that is out of bounds of an array
  - Attempting to open a file which does not exist

What is an Exception?

- When an exceptional condition arises, an object representing that exception is created and thrown in the code that caused the error

- An exception can be caught to handle it or pass it on

- Exceptions can be generated by the run-time system, or they can be manually generated by your code

What is an Exception?

```
test = [1,2,3]
test[3]
```

```
IndexError: list index out of range
```

What is an Exception?

```
successFailureRatio = numSuccesses/numFailures
print('The success/failure ratio is',
successFailureRatio)
print('Now here')
```

```
ZeroDivisionError: integer division or
modulo by zero
```

What is an Exception?

```
val = int(input('Enter an integer: '))
print('The square of the number', val**2)
```

```
> Enter an integer: asd
```

```
ValueError: invalid literal for int() with
base 10: 'asd'
```

Handling Exceptions

- Exception mechanism gives the programmer a chance to do something against an abnormal condition.

- Exception handling is performing an action in response to an exception.

- This action may be:
  - Exiting the program
  - Retrying the action with or without alternative data
  - Displaying an error message and warning user to do something
  - ....

Handling Exceptions

```
try:
    successFailureRatio = numSuccesses/numFailures
    print('The S/F ratio is', successFailureRatio)
except ZeroDivisionError:
    print('No failures, so the S/F is undefined.')
print('Now here')
```

- Upon entering the **`try`** block, the interpreter attempts to evaluate the expression **`numSuccesses/numFailures`**.
- If expression evaluation is successful, the assignment is done and the result is printed.
- If, however, a **`ZeroDivisionError`** exception is raised, the print statement in the **`except`** block is executed.

Handling Exceptions

```
while True:
    val = input('Enter an integer: ')
    try:
        val = int(val)
        print('The square of the number', val**2)
        break #to exit the while loop
    except ValueError:
        print(val, 'is not an integer')
```

Checks for whether `ValueError` exception is raised or not

Keywords of Exception Handling

- There are five keywords in Python to deal with exceptions: **`try`**, **`except`**, **`else`**, **`raise`** and **`finally`**.

- **`try`**: Creates a block to monitor if any exception occurs.

- **`except`**: Follows the try block and catches any exception which is thrown within it.

Are There Many Exceptions in Python?

- Yes, some of them are…
  - **`Exception`**
  - **`ArithmeticError`**
  - **`OverflowError`**
  - **`ZeroDivisonError`**
  - **`EOFError`**
  - **`NameError`**
  - **`IOError`**
  - **`SyntaxError`**

List of all exceptions (errors):
http://docs.python.org/3/library/exceptions.html#bltin-exceptions

Multiple except Statements

- It is possible that more than one exception can be thrown in a code block.
  - We can use multiple **`except`** clauses
- When an exception is thrown, each **`except`** statement is inspected in order, and the first one whose type *matches* that of the exception is executed.
  - Type matching means that the exception thrown must be an object of the same class or a sub-class of the declared class in the **`except`** statement
- After one **`except`** statement executes, the others are bypassed.

Multiple except Statements

```
try:
    You do your operations here;
except Exception-1:
    Execute this block.
except Exception-2:
    Execute this block.
except (Exception-3[, Exception-4[,...ExceptionN]]]):
    If there is any exception from the given exception list,
    then execute this block.
```

```
except (ValueError, TypeError):
    …
```

The except block will be entered if <u>any of the listed exceptions</u> is raised within the try block

Multiple except Statements

```
try:
    f = open('outfile.dat', 'w')
    dividend = 5
    divisor = 0
    division = dividend / divisor
    f.write(str(division))
except IOError:
    print("I can't open the file!")
except ZeroDivisionError:
    print("You can't divide by zero!")
```

You can't divide by zero!

Multiple except Statements

```
try:
    f = open('outfile.dat', 'w')
    dividend = 5
    divisor = 0
    division = dividend / divisor
    f.write(str(division))
except Exception:
    print("Exception occured and handled!")
except IOError:
    print("I can't open the file!")
except ZeroDivisionError:
    print("You can't divide by zero!")
```

Exception occured and handled!

Multiple except Statements

```
try:
    f = open('outfile.dat', 'w')
    dividend = 5
    divisor = 0
    division = dividend / divisor
    f.write(str(division))
except:
    print("Exception occured and handled!")
except IOError:
    print("I can't open the file!")
except ZeroDivisionError:
    print("You can't divide by zero!")
```

```
SyntaxError: default 'except:' must be last
```

except-else Statements

```
try:
   You do your operations here
except:
   Execute this block.
else:
   If there is no exception, execute this block.
```

```
try:
       f = open(arg, 'r')
except IOError:
       print('cannot open', arg)
else:
       print(arg, 'has', len(f.readlines()), 'lines')
```

`finally` Statement

- **`finally`** creates a block of code that will be executed after a **`try/except`** block has completed and before the code following the **`try/except`** block

- **`finally`** block is executed whether or not exception is thrown

- **`finally`** block is executed whether or not exception is caught

- It is used to gurantee that a code block will be executed in any condition.

finally Statement

You can use it to clean up files, database connections, etc.

```
try:
   You do your operations here
except:
   Execute this block.
finally:
   This block will definitely be executed.
```

```
try:
   file = open('out.txt', 'w')
   do something…
finally:
   file.close()
   os.path.remove('out.txt')
```

Nested `try` Blocks

- When an exception occurs inside a **`try`** block;
  - If the **`try`** block does not have a matching except, then the outer **`try`** statement's except clauses are inspected for a match
  - If a matching except is found, that except block is executed
  - If no matching except exists, execution flow continues to find a matching except by inspecting the outer try statements
  - If a matching except cannot be found at all, the exception will be caught by Python's exception handler.

- Execution flow never returns to the line that exception was thrown. This means, an exception is caught and except block is executed, the flow will continue with the lines following this except block

Let’s clarify it on various scenarios

```
try:
    statement1
    try:
        statement2
    except Exception1:
        statement3
    except Exception2:
        statement4;
    try:
        statement5
    except Exception3:
        statement6
    statement7;
except Exception3:
    statement8
statement9;
```

Information: Exception1 and Exception2 are subclasses of Exception3

Question: Which statements are executed if

1- statement1 throws Exception1

2- statement2 throws Exception1

3- statement2 throws Exception3

4- statement2 throws Exception1 and statement3 throws Exception2

Scenario: statement1 throws Exception1

```
try:
    statement1
    try:
        statement2
    except Exception1:
        statement3
    except Exception2:
        statement4;
    try:
        statement5
    except Exception3:
        statement6
    statement7;
except Exception3:
    statement8
statement9;
```

Step1: Exception is thrown

Exception1

Step2: except clauses of the try block are inspected for a matching except statement. Exception3 is super class of Exception1, so it matches.

Step3: statement8 is executed, exception is handled and execution flow will continue bypassing the following except clauses

Step4: statement9 is executed

Scenario: statement2 throws Exception1

```
try:
    statement1
    try:
        statement2
    except Exception1:
        statement3
    except Exception2:
        statement4;
    try:
        statement5
    except Exception3:
        statement6
    statement7;
except Exception3:
    statement8
statement9;
```

Step1: Exception is thrown

Exception1

Step2: except clauses of the try block are inspected for a matching except statement. First clause catches the exception

Step3: statement3 is executed, exception is handled

Step4: execution flow will continue bypassing the following except clauses. statement5 is executed.

Step5: Assuming no exception is thrown by statement5, program continues with statement7 and statement9.

Scenario: statement2 throws Exception3

```
try:
    statement1
    try:
        statement2
    except Exception1:
        statement3
    except Exception2:
        statement4;
    try:
        statement5
    except Exception3:
        statement6
    statement7;
except Exception3:
    statement8
statement9;
```

Step1: Exception is thrown

Exception3

Step2: except clauses of the try block are inspected for a matching except statement. None of these except clauses match Exception3

Step3: except clauses of the outer try statement are inspected for a matching except . Exception3 is catched and statement8 is executed

Step4: statement9 is executed

Scenario: statement2 throws Exception1 and statement3 throws Exception2

```
try:
    statement1
    try:
        statement2
    except Exception1:
        statement3
    except Exception2:
        statement4;
    try:
        statement5
    except Exception3:
        statement6
    statement7;
except Exception3:
    statement8
statement9;
```

Step1: Exception is thrown

Exception1

Step2: Exception is catched and statement3 is executed.

Step3: statement3 throws a new exception

Exception2

Step4: Except clauses of the outer try statement are inspected for a matching except. Exception2 is catched and statement8 is executed

Step5: statement9 is executed

raise Statement

- You can raise exceptions by using the **raise** statement.

- The syntax is as follows:

```
raise exceptionName(arguments)
```

`raise` Statement

```
def getRatios(vect1, vect2):
    ratios = []
    for index in range(len(vect1)):
        try:
            ratios.append(vect1[index]/vect2[index])
        except ZeroDivisionError:
            ratios.append(float('nan')) #nan = Not a Number
        except:
            raise ValueError('getRatios called with bad arguments')
    return ratios

try:
    print(getRatios([1.0, 2.0, 7.0, 6.0], [1.0,2.0,0.0,3.0]))
    print(getRatios([], []))
    print(getRatios([1.0, 2.0], [3.0]))
except ValueError as msg:
    print(msg)
```

```
[1.0, 1.0, nan, 2.0]
[]
getRatios called with bad arguments
```

`raise` Statement

- Avoid raising a generic **`Exception`**! To catch it, you'll have to catch all other more specific exceptions that subclass it..

```
def demo_bad_catch():
  try:
    raise ValueError('a hidden bug, do not catch this')
    raise Exception('This is the exception you expect to handle')
  except Exception as error:
    print('caught this error: ' + repr(error))

>>> demo_bad_catch()
caught this error: ValueError('a hidden bug, do not catch this',)
```

raise Statement

- and more specific catches won't catch the general exception:..

```
def demo_no_catch():
  try:
    raise Exception('general exceptions not caught by specific handling')
  except ValueError as e:
    print('we will not catch e')

>>> demo_no_catch()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in demo_no_catch
Exception: general exceptions not caught by specific handling
```

Custom Exceptions

- Users can define their own exception by creating a new class in Python.

- This exception class has to be derived, either directly or indirectly, from Exception class.

- Most of the built-in exceptions are also derived form this class.

Custom Exceptions

```
class ValueTooSmallError(Exception):
   """Raised when the input value is too small"""
   pass

class ValueTooLargeError(Exception):
   """Raised when the input value is too large"""
   pass
```

Custom Exceptions

```
number = 10 # you need to guess this number

while True:
    try:
        i_num = int(input("Enter a number: "))
        if i_num < number:
            raise ValueTooSmallError
        elif i_num > number:
            raise ValueTooLargeError
        break
    except ValueTooSmallError:
        print("This value is too small, try again!")
    except ValueTooLargeError:
        print("This value is too large, try again!")

print("Congratulations! You guessed it correctly.")
```

Lecture Overview

- Debugging
- Exception Handling
- Testing

Testing

- Programming to analyze data is powerful
- It is useless if the results are not correct
- **Correctness is far more important than speed**

Famous Examples

- Ariane 5 rocket
  - On June 4, 1996, the maiden flight of the European Ariane 5 launcher crashed about 40 seconds after takeoff.
  - Media reports indicated that the amount lost was half a billion dollars
  - The explosion was the result of a software error

- Therac-25 radiation therapy machine
  - In 1985 a Canadian-built radiation-treatment device began blasting holes through patients' bodies.

Testing does not ***Prove*** Correctness

- Edsger Dijkstra: “Program testing can be used to show the presence of bugs, but never to show their absence!”

Testing = Double-Checking Results

- How do you know your program is right?
  - Compare its output to a correct output
- How do you know a correct output?
  - Real data is big
  - You wrote a computer program because it is not convenient to compute it by hand
- Use small inputs so you can compute by hand
- Example: standard deviation
  - What are good tests for **`std_dev`**? $s = \sqrt{\frac{1}{N-1} \sum_{i=1}^{N} (x_i - \overline{x})^2},$

Testing ≠ Debugging

- **Testing**: Determining whether your program is correct
  - Doesn't say where or how your program is incorrect

- **Debugging**: Locating the specific defect in your program, and fixing it

  2 key ideas:
  - divide and conquer
  - the scientific method

What is a Test?

- A test consists of:
  - an input: sometimes called “test data”
  - an oracle: a predicate (boolean expression) of the output

What is a Test?

- Example test for sum:
  - input: [1, 2, 3]
  - oracle: result is 6
  - write the test as: **`sum([1, 2, 3]) == 6`**
- Example test for sqrt:
  - input: 3.14
  - oracle: result is within 0.00001 of 1.772
  - ways to write the test:
    - **`-0.00001 < sqrt(3.14) - 1.772 < 0.00001`**
    - **`math.abs(sqrt(3.14) - 1.772) < 0.00001`**

Test Results

- The test passes if the boolean expression evaluates to **`True`**

- The test fails if the boolean expression evaluates to **`False`**

- Use the **`assert`** statement:
  - **`assert sum([1, 2, 3]) == 6`**
  - **`assert True`** does nothing
  - **`assert False`** crashes the program and prints a message

Where to Write Test Cases

- At the **top level**: is run every time you load your program

```
def hypotenuse(a, b):
    …
assert hypotenuse(3, 4) == 5
assert hypotenuse(5, 12) == 13
```

- In a **test function**: is run when you invoke the function

```
def hypotenuse(a, b):
    …
def test_hypotenuse():
    assert hypotenuse(3, 4) == 5
    assert hypotenuse(5, 12) == 13
```

Assertions are not Just for Test Cases

- Use assertions throughout your code
- Documents what you think is true about your algorithm
- Lets you know immediately when something goes wrong
  - The longer between a code mistake and the programmer noticing, the harder it is to debug

Assertions Make Debugging Easier

- Common, but unfortunate, course of events:
  - Code contains a mistake (incorrect assumption or algorithm)
  - Intermediate value (e.g., result of a function call) is incorrect
  - That value is used in other computations, or copied into other variables
  - Eventually, the user notices that the overall program produces a wrong result
  - Where is the mistake in the program? It could be anywhere.

- Suppose you had 10 assertions evenly distributed in your code
  - When one fails, you can localize the mistake to 1/10 of your code (the part between the last assertion that passes and the first one that fails)

Where to Write Assertions

- Function entry: Are arguments legal?
  - Place blame on the caller before the function fails
- Function exit: Is result correct?
- Places with tricky or interesting code
- Assertions are ordinary statements; e.g., can appear within a loop:

```
for n in myNumbers:
  assert type(n) == int or type(n) == float
```

Where *not* to Write Assertions

- Don't clutter the code
  - Same rule as for comments
- Don't write assertions that are certain to succeed
  - The existence of an assertion tells a programmer that it might possibly fail
- Don't write an assertion if the following code would fail informatively

```
assert type(name) == str
print("Hello, " + name)
```

- Write assertions where they may be useful for debugging

What to Write Assertions About

- Results of computations
- Correctly-formed data structures

```
assert 0 <= index < len(mylist)
assert len(list1) == len(list2)
```

When to Write Tests

- Two possibilities:
  - Write code first, then write tests
  - Write tests first, then write code

When to Write Tests

- If you write the code first, you remember the implementation while writing the tests
  - You are likely to make the same mistakes in the implementation

When to Write Tests

- If you write the tests first, you will think more about the functionality than about a particular implementation
  - You might notice some aspect of behavior that you would have made a mistake about
  - This is the better choice

Write the Whole Test

- A common **mistake**:
  1. Write the function
  2. Make up test inputs
  3. Run the function
  4. Use the result as the oracle

- You didn't write a test, but only half of a test
  – Created the tests inputs, but not the oracle

- The test does not determine whether the function is correct
  – Only determines that it continues to be as correct (or incorrect) as it was before

Testing Approaches

- **Black box testing** - Choose test data ***without*** looking at implementation

- **Glass box** (white box, clear box) **testing** - Choose test data ***with*** knowledge of implementation

Inside Knowledge might be Nice

- Assume the code below:

```
c = a + b
if c > 100
   print("Tested”)
print("Passed”)
```

- Creating a test case with a=40 and b=70 is not enough
  - Although every line of the code will be executed
- Another test case with a=40 and b=30 would complete the test

Tests might not Reveal an Error Sometimes

```
def mean(numbers):
  """Returns the average of the argument list.
     The argument must be a non-empty number list."""
  return sum(numbers)//len(numbers)

# Tests
assert mean([1, 2, 3, 4, 5]) == 3
assert mean([1, 2, 3]) == 2
```

This implementation is elegant, but wrong!

`mean([1,2,3,4])` → **would return 2.5!!!**

Last but not Least, Don’t Write Meaningless Tests

```
def mean(numbers):
  """Returns the average of the argument list.
     The argument must be a non-empty number list."""
  return sum(numbers)//len(numbers)
```

Unnecessary tests. Don’t write these:

```
mean([1, 2, "hello"])
mean("hello")
mean([])
```