

# CSc 120

## Introduction to Computer Programming II

Testing

# Why test?

- Mars Climate Orbiter
  - Purpose: to study the Martian climate and to serve as a relay for the Mars Polar Lander
  - Disaster: Bad trajectory caused it to disintegrate in the upper atmosphere of Mars
  - Why: Software bug - failure to convert English units to *metric* values (pound-seconds vs. newton-seconds) *as specified in the contract*



# Why test?

- THERAC-25 Radiation Therapy
  - 1985 to 1987: two cancer patients at the East Texas Cancer Center in Tyler received fatal radiation overdose (a total of 6 accidents) – *massive overdose*
  - Why: Software bug - mishandled race condition (i.e., miscoordination between concurrent tasks)



# Why test?

- Hive Thermostat
- February, 2016: customers were roasting at home
- the thermostat mysteriously began setting the temperature to 90 degrees F (32 C)
- Hive:
  - “We are aware of a temporary glitch...where a *certain sequence of commands* in the Hive iOS app can cause thermostat temperature to rise to 90 degrees F.”

# Why test?

- Hive user:

The image shows a tweet from Laura Adams (@AdamsLaura) and a screenshot of the Hive mobile app interface. The tweet, posted from an iPad, says: "Hey @hivehome @HiveHelper when are you going to stop letting skynet try to boil me alive? (2nd time now)". The app screenshot displays the "Heating control" screen. On the left is a "Menu" with options: Devices (expanded), Heating control, Heating schedule, Manage devices, Install devices, Settings, Help & Support, and Log out. The main screen shows a "Schedule" tab selected, with a large red circular temperature display showing 32° (target), 31.5° (current), and 19.5° (outside now). Below this is a "Heating schedule" section showing a 9° target for the next 12:00 - 20:45 period. The status bar at the top shows 08:39 and 13% battery.

Laura Adams  
@AdamsLaura

Follow

Hey @hivehome @HiveHelper when are you going to stop letting skynet try to boil me alive? (2nd time now)

Menu

- Devices
  - Heating control
  - Heating schedule
  - Manage devices
  - Install devices
- Settings
- Help & Support
- Log out

Heating control

Schedule Manual Off

19.5° outside now

32°

31.5°

9° Heating schedule next 12:00 - 20:45

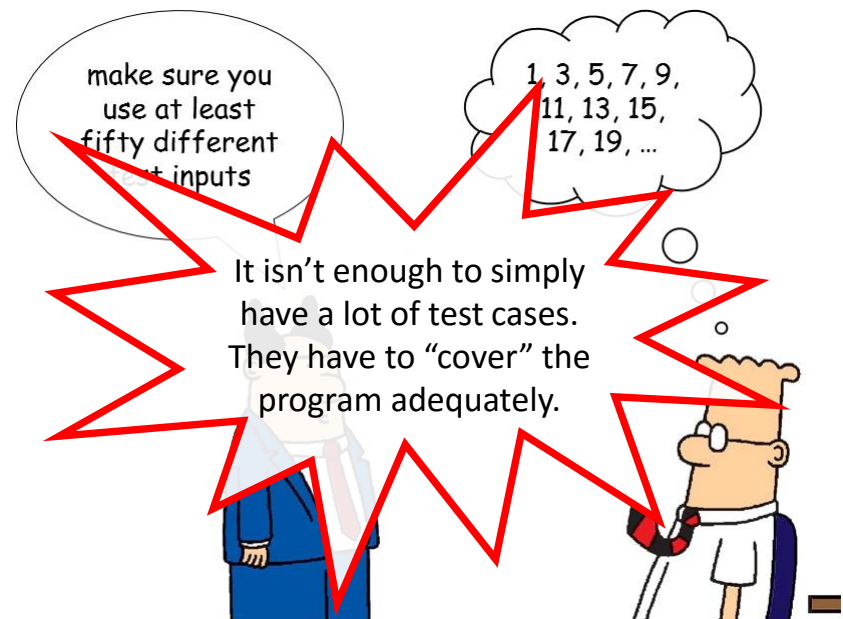
# Purpose of testing

- Every piece of software is written with some functionality in mind
- Testing aims to identify whether the program meets its intended functionality
  - "testing can only prove the presence of bugs, not their absence"
  - the more thoroughly your software is tested, the more confidence you can have about its correctness
  - "Test until fear turns into boredom." – Kent Beck

# Testing and test cases

"thoroughly"  $\neq$  lots of test cases

```
def main():  
    x = input()  
    if x % 2 == 1: # x is odd  
        do_useful_computation()  
    else:  
        delete_all_files()  
        send_rude_email_to_boss()  
        crash_computer()
```



# Approaches to testing

## Black-box testing

- Focuses only on functionality
  - does not look at how the code actually works
- Good for identifying missing features, misunderstandings of the problem spec

## White-box testing

- Focuses on the code
  - examines the code to figure out what tests to use
- Good for identifying bugs and programming errors



# black-box testing

# Black-box testing: what to test?

- Based purely on the desired functionality
  - shouldn't be influenced by the particular code you wrote (that's white-box testing)
- Aspects to consider:
  - expected outcome
    - normal vs error
  - characterizing values
    - edge cases vs “regular” values

# Black-box testing: Outcomes

- Choose tests for both *normal* and *error* behaviors
  - assumes that we know what the error situations are
- Desired program behavior:
  - on normal inputs: produce the expected behavior
  - on error inputs:
    - detect and indicate that an error occurred
    - then behave appropriately as required by the problem spec
- Passing a test:
  - the program *passes a test* if it shows the desired behavior for that test

# Black-box testing: Values

- Edge cases:
  - at or near the end(s) of the range of a value the program is supposed to operate on
  - Examples:
    - “zero-related” : 0, [], empty string, empty file, ...
    - “one-related” : 1, -1, list with one element, file with one line, ...
    - (maybe) large values
- “Regular” values:
  - not edge cases

# Example:

“Read a file containing integers and print the sum of the numbers that occur on odd-numbered lines.”

Sample input file:



```
9
4
8
2
3
```

# Example

“Read a file containing integers and print the sum of the numbers that occur on odd-numbered lines.”

*Testing for outcome (legal vs. error):*

## Normal behavior

- no. of numbers = 1
  - 0 adds
- no. of numbers = 3
  - 1 add; 1 skip in-between
- no. of numbers = 4
  - 1 add; 1 skip at end
- > 4 numbers
  - several add operations

## Error behavior

- input file does not exist (or is unreadable)
- file has non-numeric characters
- empty line
- more than one number on a line

# Example

“Read a file containing integers and print the sum of the numbers that occur on odd-numbered lines.”

*Testing for values (edge cases vs. regular values):*

## Edge cases

- empty file
- file with one number

## Regular values

- a file with several numbers, one per line

# Example

“Read a file containing integers and print the sum of the numbers that occur on odd-numbered lines.”

*Putting these together:*

## Normal behavior

- empty file
  - file with one number
  - a file with several numbers, one per line
- edge /
- regular

## Error behavior

- input file does not exist (or is unreadable)
- file has non-numeric characters
- empty line
- more than one number on a line



# EXERCISE

*Consider this program specification:*

*Write a program that reads a file name and computes (and prints out) the length of the longest line in that file.*

*Specify input files that exemplify each of the following:*

- a) two error cases*
- b) two edge cases*
- c) one regular (normal) case*

# EXERCISE-sol

*Consider this program specification:*

*Write a program that reads a file name and computes (and prints out) the length of the longest line in that file.*

*Specify input files that exemplify each of the following:*

*a) two error cases*

the file does not exist

the file is readable but not organized into lines (it's a JPEG,...)

*a) two edge cases*

the file has one line

the file is empty

*a) one regular (normal) case*

the file has many lines, each line containing readable values

# EXERCISE

*Consider the rhyming words assignment.*

*Specify input files that exemplify each of the following:*

*a) one error cases*

*b) one edge cases*

*c) one regular (normal) case*

# EXERCISE-sol

*Consider the rhyming words assignment.*

*Specify input files that exemplify each of the following:*

*a) two error cases*

*An input file that has more than one pronunciation per line*

*An empty input file*

*b) one edge case*

*A small input file with correct pronunciations, but no two words in the file rhyme.*

*c) one regular (normal) case*

*An input file that has two words, where the pronunciation of each has phonemes meet the rules for rhyming*

# EXERCISE/ICA

*Consider this program specification:*

*Write a program that reads a (possibly empty) file containing only numbers (and whitespace) and prints out the difference between the smallest and largest numbers. An empty input file should generate no output.*

*Specify input files that exemplify each of the following:*

*a) two error cases*

*a) two edge cases*

*b) one regular (normal) case*

# white-box testing

# White-box testing: what to test?

- Ideally, that every path through the code works correctly
  - but this can be prohibitively difficult and expensive

unit testing

- Instead, what we often do is:
  - check that the individual pieces of the program work properly
  - use **asserts** of pre/postconditions to check that the pieces interact properly

# Unit testing

- Tests individual units of code, e.g., functions, methods, or classes
  - e.g.: given specific test inputs, does the function behave correctly?
    - CloudCoder!
  - useful for making programmers focus on the exact behavior of the function being tested
    - e.g., preconditions, postconditions, invariants
  - helps find problems early
- Isolate a unit and validate its correctness
- Often automated, but can be done manually



# Unit testing

*# grid\_is\_square(arglist) – returns True if arglist  
# has the shape of a square grid, i.e.,  
# the length of each element ("row") of arglist is  
# equal to the number of rows of arglist*

```
def grid_is_square(arglist):  
    num_rows = len(arglist)  
    for row in arglist:  
        if len(row) != num_rows:  
            return False  
    return True
```

# Unit testing

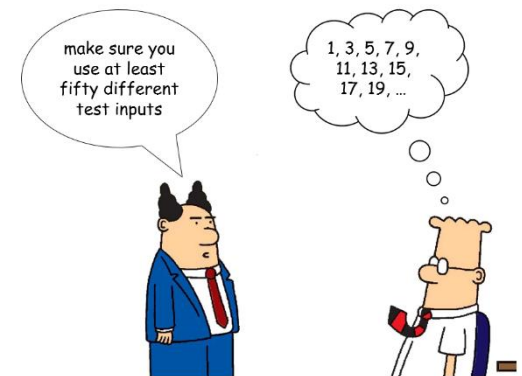
*# grid\_is\_square(arglist) – returns True if arglist  
# has the shape of a square grid, i.e.,  
# the length of each element ("row") of arglist is  
# equal to the number of rows of arglist*

```
def grid_is_square(arglist):  
    num_rows = len(arglist)  
    for row in arglist:  
        if len(row) != num_rows:  
            return False  
    return True
```

- Write three white box test cases (inputs) for this.
- (I.e., give the specific *arglist* that would be passed in to test the function.)

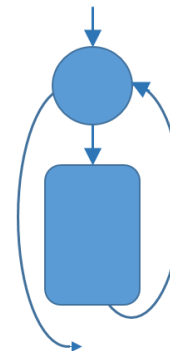
# Code coverage

- Code coverage refers to how much of the code is executed ("covered") by a set of tests
  - want to be at (or close to) 100%
  - coverage tools report which parts of the program were executed, and how much
    - e.g., Coverage.py
- Figuring out how to increase coverage often leads to testing edge cases



# Unit testing: practical heuristics

- Check both normal and error behaviors
- edge-case inputs:
  - zero values (0, empty list/string/tuple/file, ...)
  - singleton values (1, list/string/tuple/file of length 1, ...)
  - large values
- if statements: make sure each outcome (True/False) is taken
- Loops: test 0, 1, >1 iterations

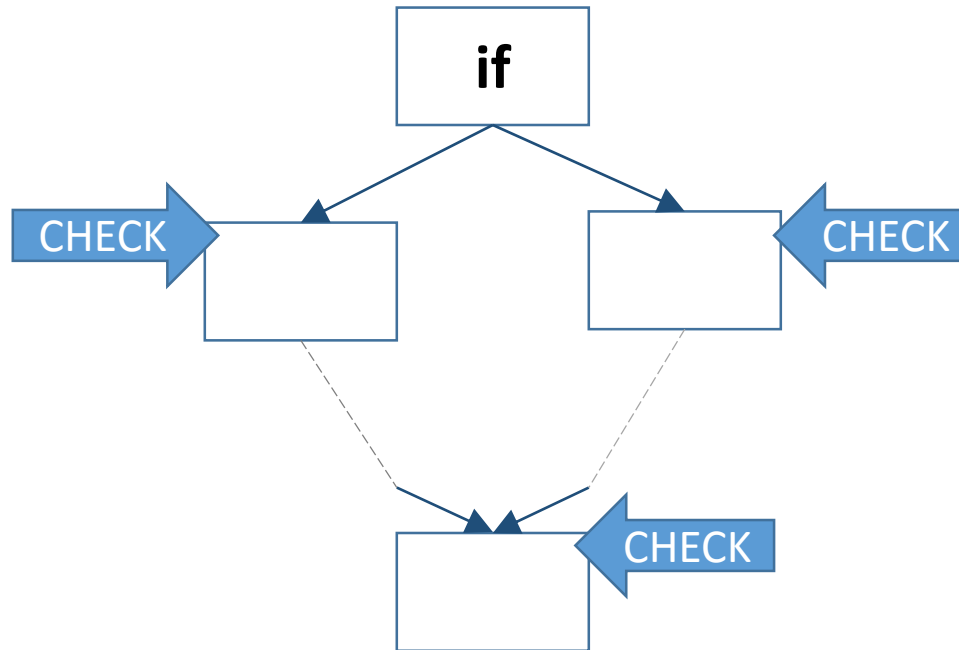


# Unit testing: what to check?

- Not just “output is what we expect”
  - remember “accidental” success
- Check that invariants hold at key points

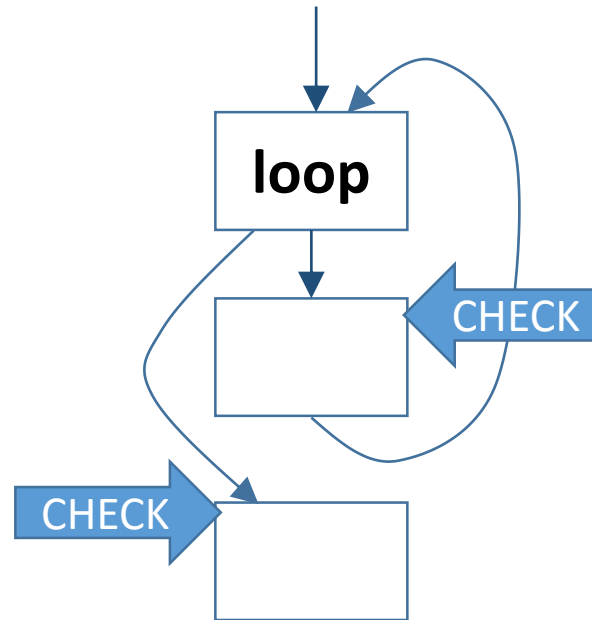
# Unit testing: what to check?

- Not just “output is what we expect”
  - remember “accidental” success
- Check that invariants hold at key points



# Unit testing: what to check?

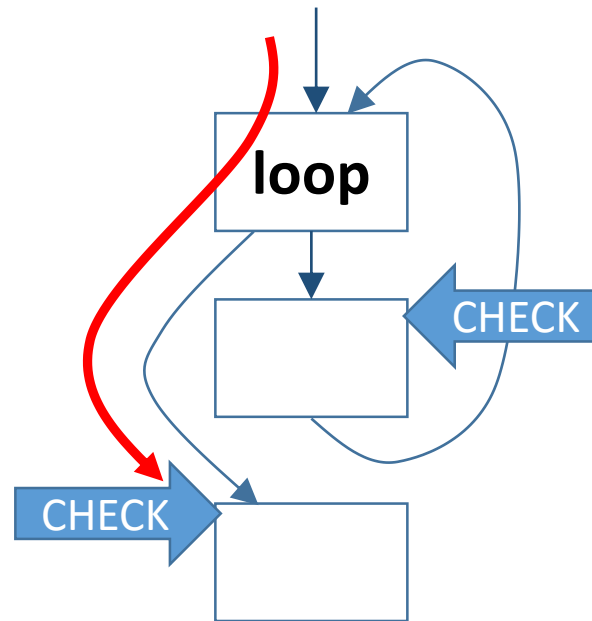
- Check that invariants hold at key points



# Unit testing: what to check?

- Check that invariants hold at key points

① Check that nothing breaks if the loop does not execute at all



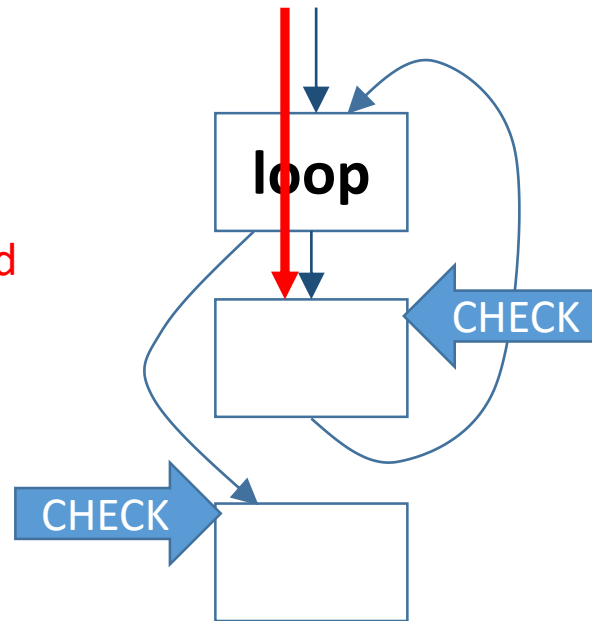


# Unit testing: what to check?

- Check that invariants hold at key points

① Check that nothing breaks if the loop does not execute at all

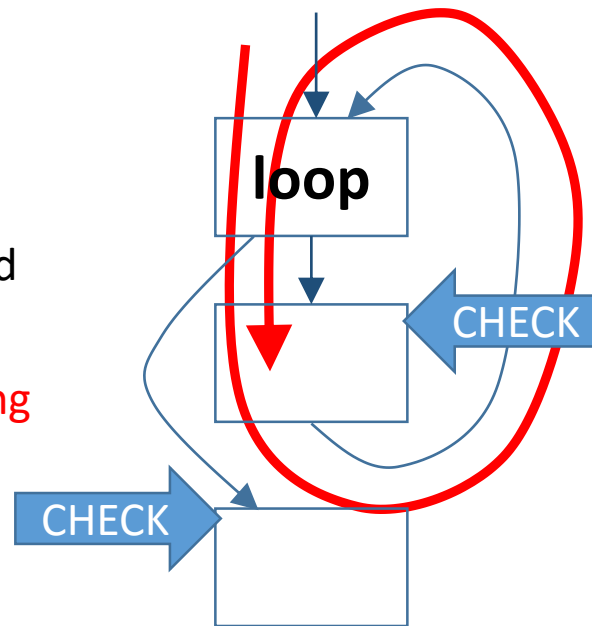
② Check that everything is initialized properly when the loop is first entered



# Unit testing: what to check?

- Check that invariants hold at key points

- ① Check that nothing breaks if the loop does not execute at all
- ② Check that everything is initialized properly when the loop is first entered
- ③ Check that everything is OK after going around the loop



# Unit testing: summary

- Test normal and error values, edge cases
- If statements: test all branches (if/elif/else)
- Loops: check invariants for:
  - 0 iterations
  - 1 iteration
  - >1 iteration
- Functions:
  - check return values

# Example: buggy list-lookup

*# lookup(string, lst) -- returns the  
# position where the given string  
# occurs in lst.*


```
def lookup(string, lst):  
    for i in range(len(lst)):  
        if string == lst[i]:  
            return i
```

# Example: buggy list-lookup

*# lookup(string, lst) -- returns the  
# position where the given string  
# occurs in lst.*

```
def lookup(string, lst):  
    for i in range(len(lst)):  
        if string == lst[i]:  
            return i
```

0, 1, >1 iterations  $\Rightarrow$  lists  
of length 0, 1, 2



# Example: (buggy) list-lookup

*# lookup(string, lst) -- returns the  
# position where the given string  
# occurs in lst.*

```
def lookup(string, lst):  
    for i in range(len(lst)):  
        if string == lst[i]:  
            return i
```

0, 1, >1 iterations  $\Rightarrow$  lists  
of length 0, 1, 2

both branches taken  $\Rightarrow$   
string is at positions 0, 1



# Example: (buggy) list-lookup

*# lookup(string, lst) -- returns the  
# position where the given string  
# occurs in lst.*

```
def lookup(string, lst):  
    for i in range(len(lst)):  
        if string == lst[i]:  
            return i
```

0, 1, >1 iterations  $\Rightarrow$  lists  
of length 0, 1, 2

both branches taken  $\Rightarrow$   
string is at positions 0, 1

some possible test inputs:

'a', []

'a', ['a']

'a', ['b', 'a']

# Example: (buggy) list-lookup

*# lookup(string, lst) -- returns the  
# position where the given string  
# occurs in lst.*

```
def lookup(string, lst):  
    for i in range(len(lst)):  
        if string == lst[i]:  
            return i
```

**Note:** this will  
catch the no-  
return-value bug

0, 1, >1 iterations  $\Rightarrow$  lists  
of length 0, 1, 2

both branches taken  $\Rightarrow$   
string is at positions 0, 1

some possible test inputs:

'a', []

'a', ['a']

'a', ['b', 'a']



# EXERCISE

*Write four unit tests for the function below:*

*# average(lst) -- returns the  
# average of the numbers in lst.*

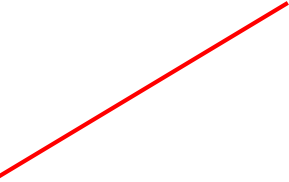
```
def average(lst):  
    sum = 0  
    for i in range(len(lst)):  
        sum += lst[i]  
    return sum/len(lst)
```

# EXERCISE

*# average(lst) -- returns the  
# average of the numbers in lst.*

```
def average(lst):  
    sum = 0  
    for i in range(len(lst)):  
        sum += lst[i]  
    return sum/len(lst)
```

0, 1, >1 iterations  $\Rightarrow$  lists  
of length 0, 1, 2



# EXERCISE

*# average(lst) -- returns the  
# average of the numbers in lst.*

```
def average(lst):  
    sum = 0  
    for i in range(len(lst)):  
        sum += lst[i]  
    return sum/len(lst)
```

0, 1, >1 iterations  $\Rightarrow$  lists  
of length 0, 1, 2

some possible test inputs:

`[]`

`[17]`

`[5, 12]`

# EXERCISE

*Write four unit tests for the function below:*

*# average(lst) -- returns the  
# average of the numbers in lst.*

```
def average(lst):  
    sum = 0  
    for i in range(len(lst)):  
        sum += lst[i]  
  
    return sum/len(lst)
```


0, 1, >1 iterations  $\Rightarrow$  lists  
of length 0, 1, 2

some possible test inputs:

[ ]

[17]

[5, 12]



**Note:** this will catch the  
divide-by-zero on empty list  
bug

# EXERCISE

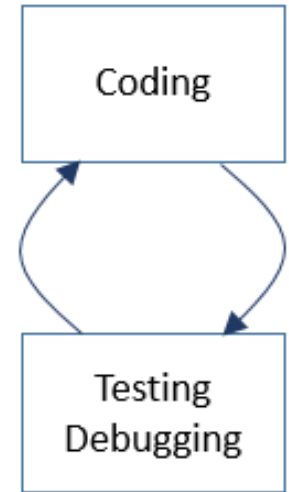
*Write four unit tests for the function below:*

*# Returns a list consisting of the strings in **wordlist**  
# that end with **tail**.*

```
def words_ending_with(wordlist, tail):  
    outlist = []  
    for item in wordlist:  
        if item.endswith(tail):  
            outlist.append(item)  
    return outlist
```

# Testing strategy

- Test as a part of program development
  - try out small tests even when the code is only partially developed (i.e., lots of stubs)
    - helps catch problems at function boundaries, e.g., number and types of arguments
    - can help identify bugs in the design, e.g., missing pieces
- Start with tiny test inputs (work your way up to small, then medium, then large)
  - problems found on tiny inputs are usually easier to debug



# REVIEW

- *In black-box testing, what does the tester know about the code being tested?*
- 

- *When black-box testing, what are some of the kinds of cases we should test?*

- \_\_\_\_\_
- \_\_\_\_\_
- \_\_\_\_\_

- *How does white-box testing differ from black-box testing?*
-