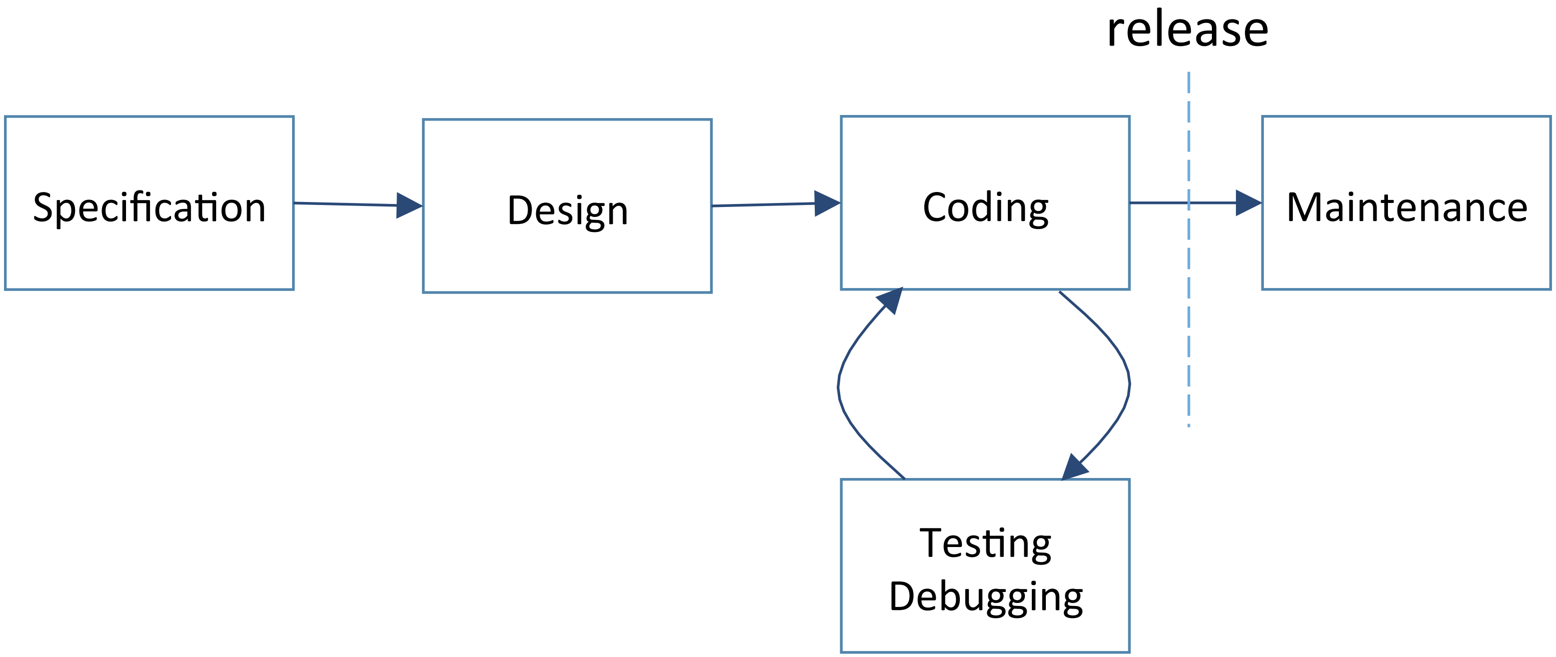
CSc 120
Introduction to Computer Programming II

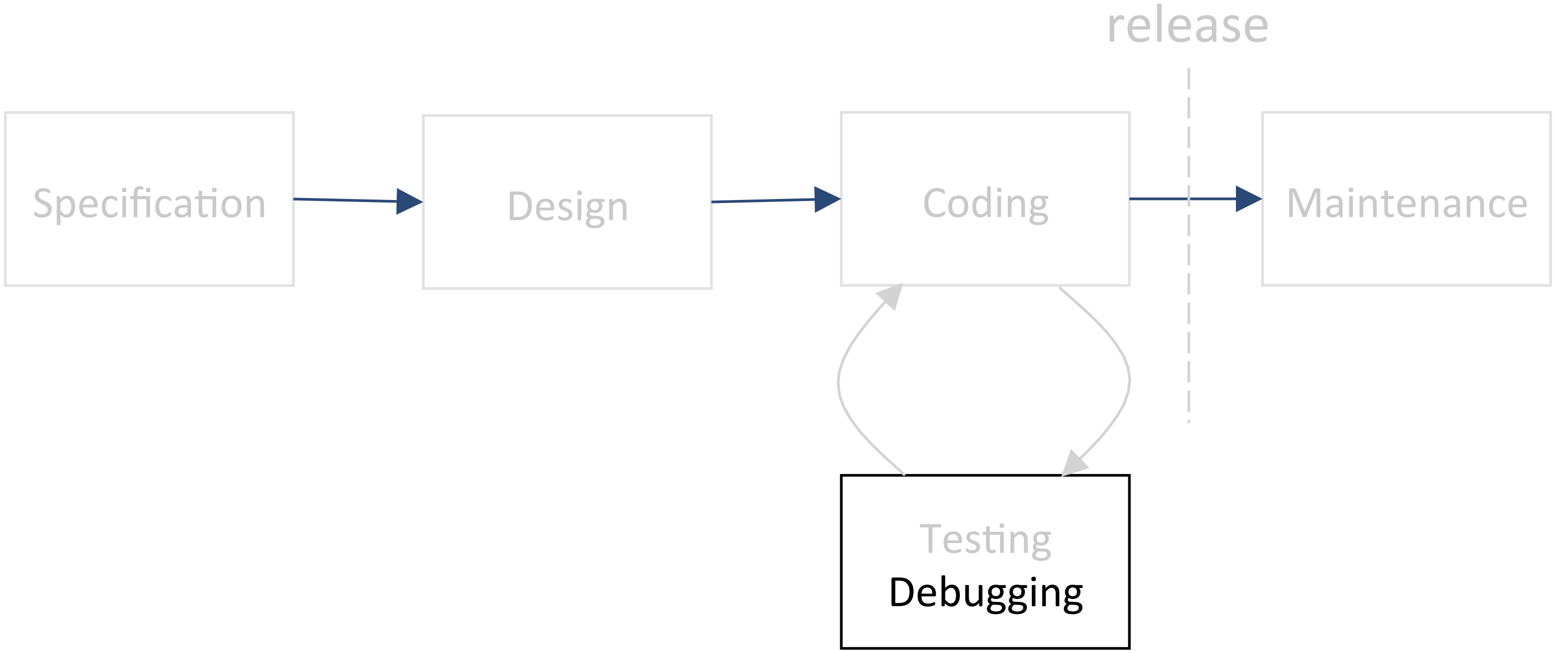
Adapted from slides by
Dr. Saumya Debray

06: Debugging

Steps in software development



Steps in software development



debugging

Invariants (reprise)

An *invariant* is a predicate about the program state that should always be true if the program is correct

- the programmer should know what invariants should hold where based on the intended functionality

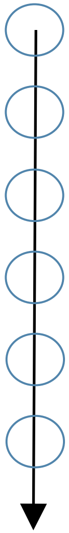
⇒ If all invariants hold, everywhere in the program, on all runs, then the program ***must*** be correct

⇒ If a program is *not* working correctly, some (intended) invariant somewhere does not hold

Buggy vs non-buggy code

An invariant's-eye view of buggy vs non-buggy program execution:

○ = program states during execution



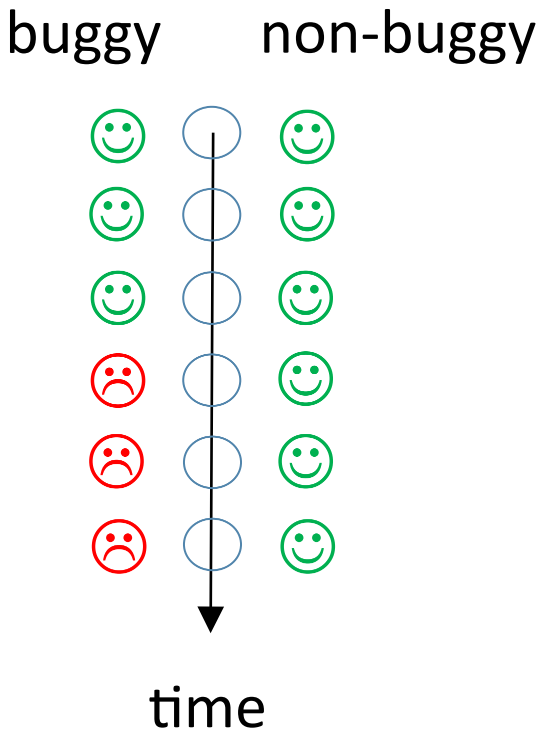
Buggy vs non-buggy code

An invariant's-eye view of buggy vs non-buggy program execution:

○ = program states during execution

☺ = all invariants hold

☹ = some invariant does not hold



debugging:

☹ → ☺

What is a bug?

A bug: a divergence between expectation and reality

Example:

I expect this program to print a sum of (non-negative) integers.

It's printing 0.

the debugging process

The debugging process

1. Find the earliest point where an invariant is not satisfied

2. Understand why the invariant fails to hold

3. Fix the code so that the invariant holds

The debugging process

- Programs that need debugging often:
  - involve a lot of code
  - process a lot of data
  - use complex logic
  - (some or all of the above)

- Figuring out the earliest point where an invariant is broken may not be easy
  - anything you can do to speed up this step is very useful
    - **assert**s in the code
    - “shrinking the search”

The debugging process

0. Find the smallest input and code that causes the bug to show up (“*shrinking the search*")

1. Find the earliest point where an invariant is not satisfied

2. Understand why the invariant fails to hold

3. Fix the code so that the invariant holds

shrinking the search

The debugging process

0. Minimize what you have to search through
   - Find the smallest input and code that shows the bug

1. Locate the bug
   - Find the first place where an invariant is not satisfied

2. Understand the problem
   - Understand why the invariant fails to hold

3. Fix the code

0. Shrinking the search

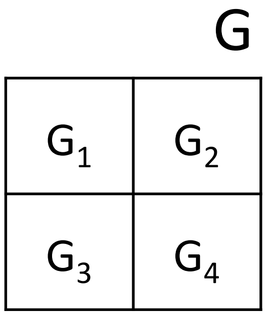
- Goal: get the bug to show up in a smaller (shorter) run of the program
  - reduce the size (or complexity) of the <u>input data</u> while still getting the bug to show up
  - Example 1: the input is a list L of 40,000 words.
    - cut L into two pieces, L1 and L2, of about 20,000 words each
    - **if** the bug shows up when input is L1:
      - *repeat the process using L1*
    - **elif** the bug shows up when input is L2:
      - *repeat the process using L2*
    - **else**:
      - *repeat the process using a middle piece of L*

discard irrelevant input

0. Shrinking the search

- Goal: get the bug to show up in a smaller (shorter) run of the program
    - reduce the size (or complexity) of the <u>input data</u> while still getting the bug to show up

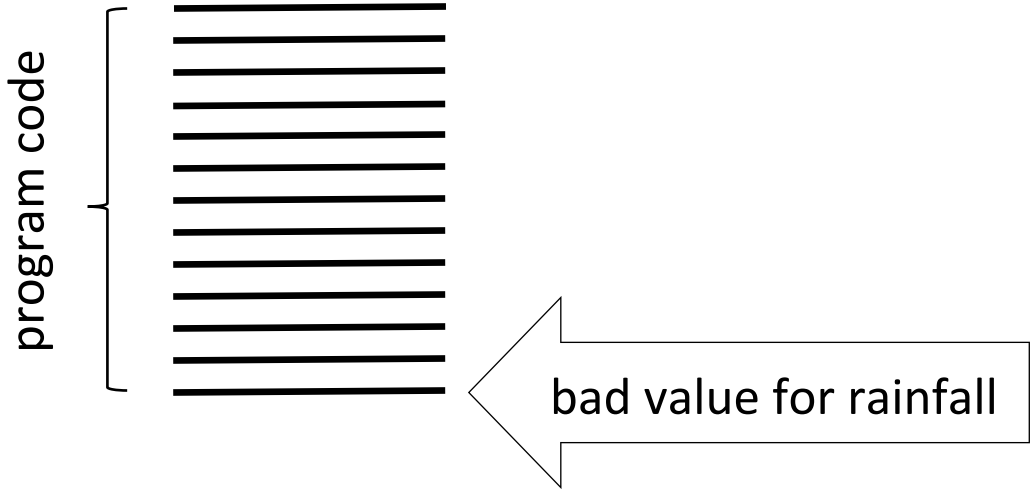
    - Example 2: The input is a 20 x 20 grid of letters G
        - divide G into smaller pieces G_1, G_2, G_3, G_4
        - for each smaller piece G_i:
            - *if G_i causes the bug to show up: repeat using G_i*
            - *else: try using a piece from the middle*



if the bug does not show up on (some of) the smaller pieces: this can itself give clues to the problem

0. Shrinking the search

- Goal: get the bug to show up in a smaller (shorter) run of the program
  - reduce the size (or complexity) of the program code while still getting the bug to show up
  - Example 3: Consider a program to analyze rainfall and temperature data, with a bug in the rainfall analysis

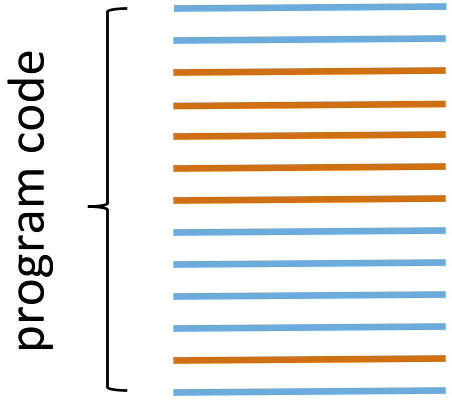


program code

bad value for rainfall

0. Shrinking the search

- Goal: get the bug to show up in a smaller (shorter) run of the program
  - reduce the size (or complexity) of the program code while still getting the bug to show up
  - Example 3: Consider a program to analyze rainfall and temperature data, with a bug in the rainfall analysis

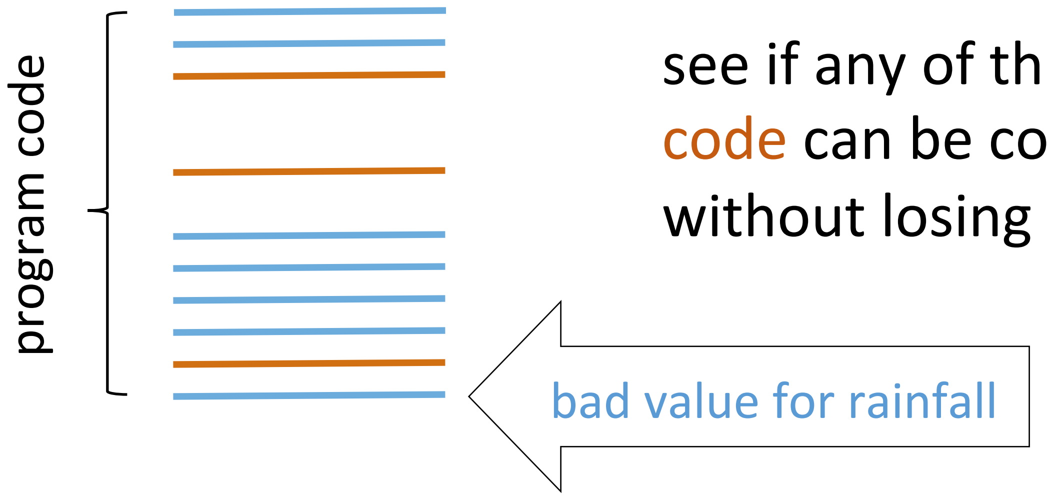


see if any of the temperature code can be commented out without losing the bug

bad value for rainfall

0. Shrinking the search

- Goal: get the bug to show up in a smaller (shorter) run of the program
  - reduce the size (or complexity) of the program code while still getting the bug to show up
  - Example 3: Consider a program to analyze rainfall and temperature data, with a bug in the rainfall analysis



see if any of the temperature code can be commented out without losing the bug

(be careful)

finding the bug

The debugging process

0. Minimize what you have to search through
   - Find the smallest input and code that shows the bug

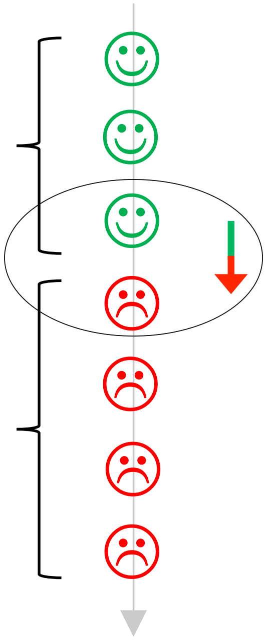
1. Locate the bug
   - Find the first place where an invariant is not satisfied

2. Understand the problem
   - Understand why the invariant fails to hold

3. Fix the code

1. Locating the bug

- Goal: Find the earliest place in the code where an invariant is not true



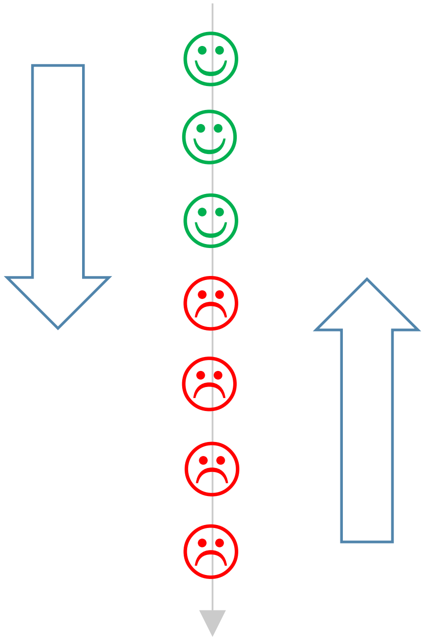
good states : all invariants are true

bad states : some invariants are false

We are looking for the transition from good states to bad states

1. Locating the bug

- Goal: Find the earliest place in the code where an invariant is not true
  - we can work forwards, or backwards, or a combination of both



work forward from a good state to find the first bad state

work backward from a bad state to find the last good state

Inspecting program state

- To figure out whether an invariant is true at some point in the code at runtime:
  - need to look at the program's state*
- Common ways of inspecting program state:
  - use print statements
  - use a debugger
    - pause the program's execution at specific points
      - *step through the program's execution; or*
      - *set breakpoints at the desired program statements*
    - inspect the program's state in the debugger

* program state = values of variables, data structures

Working forward vs. backward

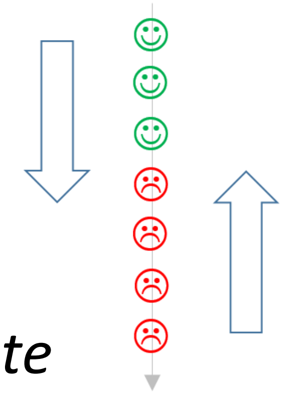
- Working forward:

  starting at a good state, identify a (later) bad state

  + matches the direction of execution ⇒ easier

  – the program's state may be large and complex

    ○ we may not know which part(s) to focus on



- Working backward

  starting at a bad state, identify a (earlier) good state

  + easier to know which part(s) of the program's state to focus on

  – does not match direction of execution

    ○ use breakpoints; move them backwards on successive runs

Working backward

- Given: a bad state (i.e., some invariant is broken)
- Approach 1:
  - think of possible reasons for the broken invariant
    - e.g.: *"we didn't look at the last element of the list"*
  - do experiments to accept or reject each hypothesis
  - the outcome of these experiments indicates whether some earlier state is good or bad

- Approach 2: (if Approach 1 is difficult to apply)
  - look at an earlier state to see if it is also bad
    - e.g.: if a function's arguments have bad values, look at the values of variables at the call site

Locating the bug: example

```
…
x = 0
for …
    …
    x += …
…
y = 0
while …
    y += …
…
z = x * y
assert  z > 0
```

Locating the bug: example

```
…
x = 0
for …
    …
    x += …
…
y = 0
while …
    y += …
…
z = x * y
assert z > 0
```

Invariant: z > 0
Observation: z == 0

Locating the bug: example

```
   1   …
   2   x = 0
   3   for …
   4       …
   5       x += …
   6   …
   7   y = 0
   8   while …
   9       y += …
  10   …
  11   z = x * y
☹ 12   assert  z > 0
```

Invariant: z > 0
Observation: z == 0

z has an incorrect value at line 11

so something is wrong somewhere in this range of code

Locating the bug: example

```
   1   …
   2   x = 0
   3   for …
   4       …
   5       x += …
   6   …
   7   y = 0
   8   while …
   9       y += …
  10   …
  11   z = x * y
☹ 12   assert  z > 0
```

Hypothesis 1:
x == 0 at line 11

Locating the bug: example

```
   1   …
   2   x = 0
   3   for …
   4       …
   5       x += …
   6   …
   7   y = 0
   8   while …
   9       y += …
  10   …
  11   z = x * y
☹ 12   assert z > 0
```

Hypothesis 1:
x == 0 at line 11

Experiment 1:
check x's value at line 11

Locating the bug: example

```
   1   …
   2   x = 0
   3   for …
   4       …
   5       x += …
   6   …
   7   y = 0
   8   while …
   9       y += …
  10   …
☹ 11   z = x * y
  12   assert z > 0
```

Hypothesis 1:
x == 0 at line 11

Experiment 1:
check x's value at line 11

Observation 1:
x == 0 at line 11
Invariant: x ≠ 0 at line 11

Locating the bug: example

```
   1   …
   2   x = 0
   3   for …
   4       …
   5       x += …
   6   …
   7   y = 0
   8   while …
   9       y += …
  10   …
☹ 11   z = x * y
  12   assert z > 0
```

Hypothesis 1:
x == 0 at line 11

Experiment 1:
check x's value at line 11

Observation 1:
x == 0 at line 11
Invariant: x ≠ 0 at line 11

Possible reasons:

- x == 0 at line 6; or
- x was set to 0 in lines 7-10

Locating the bug: example

```
  1   …
  2   x = 0
  3   for …
  4       …
  5       x += …
  6   …
  7   y = 0
  8   while …
  9       y += …
 10   …
☹ 11   z = x * y
 12   assert z > 0
```

Observation 1:
x == 0 at line 11
Invariant: x ≠ 0 at line 11

Hypothesis 2:
x == 0 at line 6

Experiment 2:
check x's value at line 6

Locating the bug: example

Scenario

1

```
   1    …
   2    x = 0
   3    for …
   4        …
   5        x += …
   6    …
   7    y = 0
   8    while …
   9        y += …
  10    …
☹ 11    z = x * y
  12    assert  z > 0
```

Observation 1:
x == 0 at line 11
Invariant: x ≠ 0 at line 11

Hypothesis 2:
x == 0 at line 6

Experiment 2:
check x's value at line 6

Observation 2:
x == 0 at line 6
invariant: x ≠ 0 at line 6 ☹

Locating the bug: example

Scenario

1

Invariant: x ≠ 0
Observation: x == 0 ☹

```
1    …
2    x = 0
3    for …
4        …
5        x += …
6    …
7    y = 0
8    while …
9        y += …
10   …
11   z = x * y
12   assert  z > 0
```

Observation 1:
x == 0 at line 11
Invariant: x ≠ 0 at line 11

Hypothesis 2:
x == 0 at line 6

Experiment 2:
check x's value at line 6

Observation 2:
x == 0 at line 6
invariant: x ≠ 0 at line 6 ☹

Locating the bug: example

Scenario

1

```
1   …
2   x = 0
3   for …
4       …
5       x += …
6   …
7   y = 0
8   while …
9       y += …
10  …
11  z = x * y
12  assert z > 0
```

x has an incorrect value at line 6

so something is wrong somewhere in this range of code

Invariant: x ≠ 0
Observation: x == 0

☹

Repeat what we just did, this time starting with the value of x at line 6

Locating the bug: example

Scenario

2

```
   1    …
   2    x = 0
   3    for …
   4        …
   5        x += …
☺  6    …
   7    y = 0
   8    while …
   9        y += …
  10    …
☹ 11    z = x * y
  12    assert  z > 0
```

Observation 1:
x == 0 at line 11
Invariant: x ≠ 0 at line 11

Hypothesis 2:
x == 0 at line 6

Experiment 2:
check x's value at line 6

Observation 2:
x ≠ 0 at line 6
invariant: x ≠ 0 at line 6

Locating the bug: example

Scenario

2

```
   1   …
   2   x = 0
   3   for …
   4       …
   5       x += …
☺  6   …
   7   y = 0
   8   while …
   9       y += …
  10   …
☹ 11   z = x * y
  12   assert z > 0
```

Observation 1:
x == 0 at line 11
Invariant: x ≠ 0 at line 11

Observation 2:
x ≠ 0 at line 6
invariant: x ≠ 0 at line 6

x has a correct value at line 6 but an incorrect value at line 11

so something is wrong somewhere in this range of code

Locating the bug: example

Scenario

2

```
   1   …
   2   x = 0
   3   for …
   4       …
   5       x += …
☺  6   …
   7   y = 0
   8   while …
   9       y += …
  10   …
☹ 11   z = x * y
  12   assert z > 0
```

Observation 1:
x == 0 at line 11
Invariant: x ≠ 0 at line 11

Observation 2:
x ≠ 0 at line 6
invariant: x ≠ 0 at line 6

To find where the value of x becomes 0:

1. We can work forward from line 6; or
2. we can work backward from line 11

Locating the bug: example

Scenario

3

```
   1   …
   2   x = 0
   3   for …
   4       …
   5       x += …
   6   …
   7   y = 0
   8   while …
   9       y += …
  10   …
☹ 11   z = x * y
  12   assert z > 0
```

Hypothesis 1:
x == 0 at line 11

Experiment 1:
check x's value at line 11

Observation 1:
x ≠ 0 at line 11
Invariant: x ≠ 0 at line 11 ☺

Locating the bug: example

Scenario

3

```
 1  …
 2  x = 0
 3  for …
 4      …
 5      x += …
 6  …
 7  y = 0
 8  while …
 9      y += …
10  …
11  z = x * y   ☹
12  assert z > 0
```

Hypothesis 1:
x == 0 at line 11

Experiment 1:
check x's value at line 11

Observation 1:
x ≠ 0 at line 11
Invariant: x ≠ 0 at line 11 ☺

Experiment 2:
check y's value at line 11

Locating the bug: example

Scenario

3

```
   1   …
   2   x = 0
   3   for …
   4       …
   5       x += …
   6   …
   7   y = 0
   8   while …
   9       y += …
  10   …
☹ 11   z = x * y
  12   assert z > 0
```

Hypothesis 1:
x == 0 at line 11

Experiment 1:
check x's value at line 11

Observation 1:
x ≠ 0 at line 11
Invariant: x ≠ 0 at line 11

Experiment 2:
check y's value at line 11

Observation 2:
y == 0 at line 11
Invariant: y ≠ 0 at line 11 ☹

Locating the bug: example

Scenario

3

```
   1   …
   2   x = 0
   3   for …
   4       …
   5       x += …
   6   …
☺  7   y = 0
   8   while …
   9       y += …
   10  …
☹  11  z = x * y
   12  assert z > 0
```

Observation 1:
x == 0 at line 11
Invariant: x ≠ 0 at line 11 ☺

Observation 2:
y == 0 at line 11
Invariant: y ≠ 0 at line 11 ☹

Check: is y initialized correctly?
Suppose that it is ☺

Locating the bug: example

Scenario

3

```
    1   …
    2   x = 0
    3   for …
    4       …
    5       x += …
    6   …
☺   7   y = 0
    8   while …
    9       y += …
   10   …
☹  11   z = x * y
   12   assert z > 0
```

Observation 1:
x == 0 at line 11
Invariant: x ≠ 0 at line 11 ☺

Observation 2:
y == 0 at line 11
Invariant: y ≠ 0 at line 11 ☹

something is wrong with the computation of y somewhere in this range of code

Locating the bug: summary

- Find the earliest point A in the program where there is a bad state ☹
  - i.e., **assert** failed or incorrect value observed
- Identify a variable x whose value at A is incorrect
- Find the latest point where the value of x is correct ☺
- Repeat:
  - narrow the range of code where x's value changes from correct to incorrect

until you see the problem or cannot narrow further

understanding the bug

The debugging process

0. Minimize what you have to search through
   - Find the smallest input and code that shows the bug

1. Locate the bug
   - Find the first place where an invariant is not satisfied

2. Understand the problem
   - Understand why the invariant fails to hold

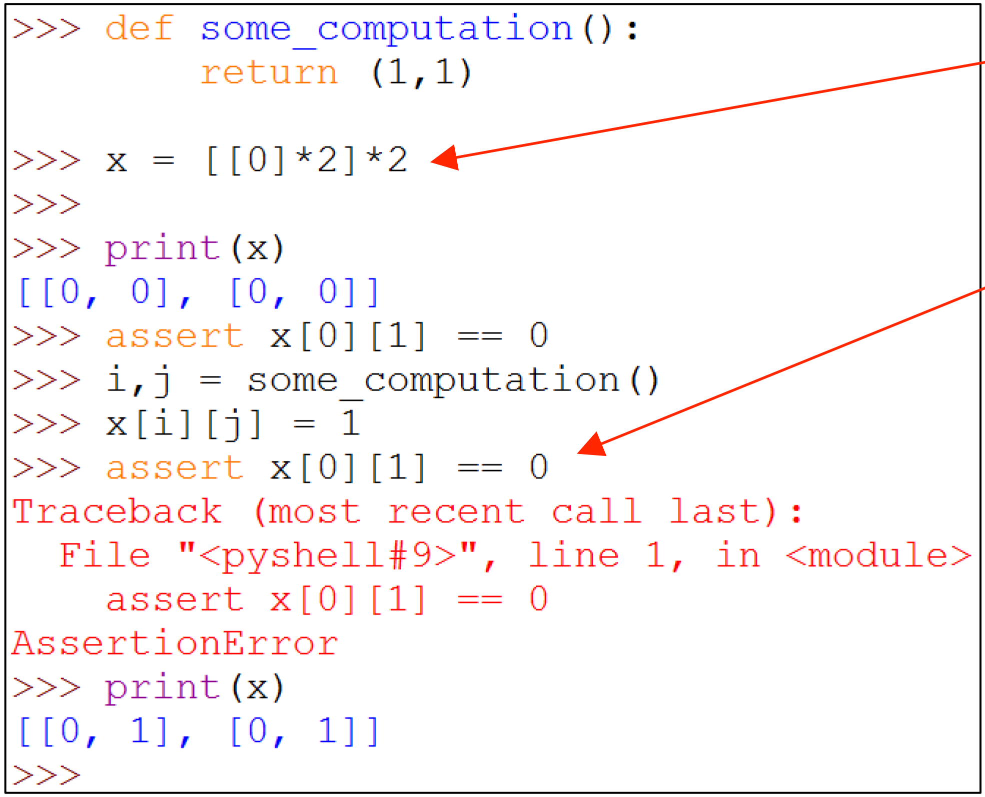
3. Fix the code

2. Understanding the problem

- An observed bug may arise due to many different underlying reasons
- Unless you understand the reason, you cannot be sure that your changes will in fact fix the problem
  - recall test cases that may pass "accidentally"
- Understanding the reason for a problem may involve more hypotheses and experiments
  - often becomes easier with experience

Understanding the problem

Example 1



```
>>> def some_computation():
        return (1,1)

>>> x = [[0]*2]*2
>>>
>>> print(x)
[[0, 0], [0, 0]]
>>> assert x[0][1] == 0
>>> i,j = some_computation()
>>> x[i][j] = 1
>>> assert x[0][1] == 0
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    assert x[0][1] == 0
AssertionError
>>> print(x)
[[0, 1], [0, 1]]
>>>
```

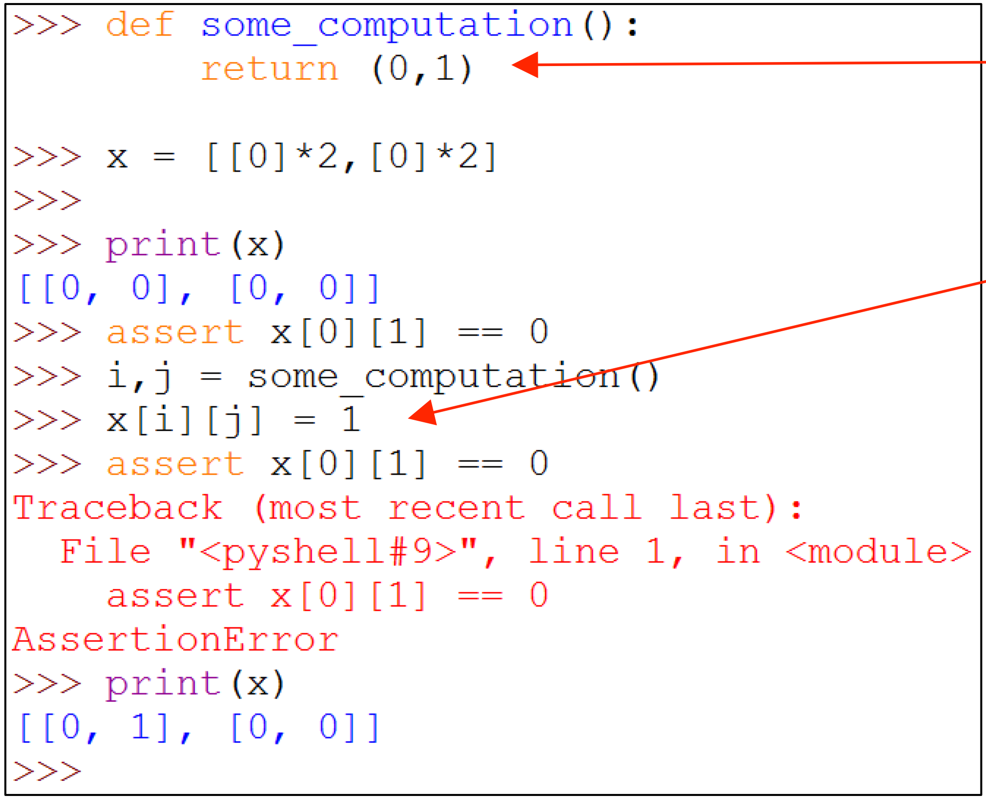
shallow copying creates "aliases"

as a result, an assignment to x[1][1] also changes the value of x[0][1]

the problem is in the structure of x, not in the values of i and j

Understanding the problem

Example 2



```
>>> def some_computation():
        return (0,1)

>>> x = [[0]*2,[0]*2]
>>>
>>> print(x)
[[0, 0], [0, 0]]
>>> assert x[0][1] == 0
>>> i,j = some_computation()
>>> x[i][j] = 1
>>> assert x[0][1] == 0
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    assert x[0][1] == 0
AssertionError
>>> print(x)
[[0, 1], [0, 0]]
>>>
```

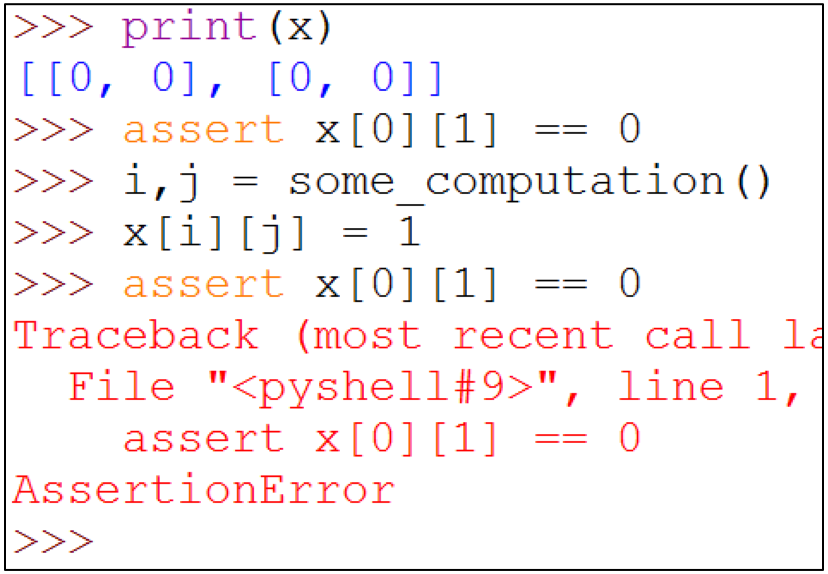
the values assigned to i,j are incorrect

as a result, this assignment changes the value of x[0][1] (the failed **assert** suggests that this was not intended)

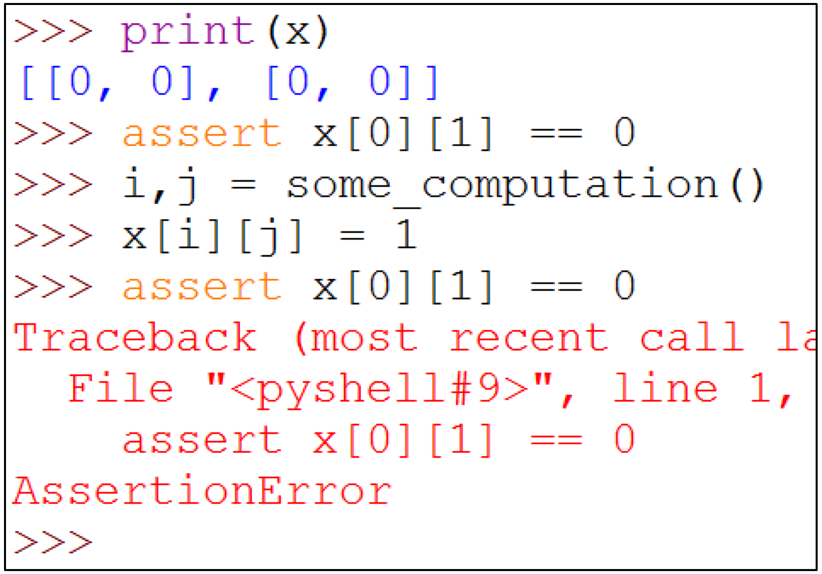
the problem is in the values computed for i and j

Understanding the problem

- But the location and behavior of the buggy code are very similar in both cases



```
>>> print(x)
[[0, 0], [0, 0]]
>>> assert x[0][1] == 0
>>> i,j = some_computation()
>>> x[i][j] = 1
>>> assert x[0][1] == 0
Traceback (most recent call la
  File "<pyshell#9>", line 1,
    assert x[0][1] == 0
AssertionError
>>>
```



```
>>> print(x)
[[0, 0], [0, 0]]
>>> assert x[0][1] == 0
>>> i,j = some_computation()
>>> x[i][j] = 1
>>> assert x[0][1] == 0
Traceback (most recent call la
  File "<pyshell#9>", line 1,
    assert x[0][1] == 0
AssertionError
>>>
```

- Without understanding the reason for the problem, we can't fix it!

Understanding the problem

- Without understanding the reason for the problem, we can't fix it

- Once you have a hypothesis for the underlying reason for a bug, it may be worth doing experiments to confirm it
  - think of other observations (possibly on other inputs) that would support or reject your hypothesis

The debugging process

0. Minimize what you have to search through
   - Find the smallest input and code that shows the bug

1. Locate the bug
   - Find the first place where an invariant is not satisfied

2. Understand the problem
   - Understand why the invariant fails to hold

3. Fix the code

fixing the bug

Fixing the code

- At this point, you should have figured out:
  - the location of the bug; and
  - the underlying reason for the problem
- Think of what changes to the code will remove the problem, i.e., fix the bug
- If you can't figure out a fix, you may want to:
  - dig deeper to understand the problem better
  - possibly consider different data structures or algorithms