CSc 120
Introduction to Computer Programming II

Debugging

Some warmup exercises

Exercise 1

Source code

```
# compute the sum of the elements
# of a list L
def sumlist(L):
    print("Entered sumlist")
    sum = 0
    i = 0
    while i < len(L):
        sum += L[i]
    print("Leaving sumlist")
    return sum
```

Execution behavior

```
>>> sumlist([1,2,3,4])
Entered sumlist
```

What's the problem?

Exercise 1

Source code

```
# compute the sum of the elements
# of a list L
def sumlist(L):
    print("Entered sumlist")
    sum = 0
    i = 0
    while i < len(L):
        sum += L[i]
    print("Leaving sumlist")
    return sum
```

The Problem

```
>>> sumlist([1,2,3,4])
Entered sumlist
```

infinite loop

Exercise 1

Source code

```
# compute the sum of the elements
# of a list L
def sumlist(L):
    print("Entered sumlist")
    sum = 0
    i = 0
    while i < len(L):
        sum += L[i]
    print("Leaving sumlist")
    return sum
```

The Problem

```
>>> sumlist([1,2,3,4])
Entered sumlist
```

infinite loop

because this statement is not executing as intended

Exercise 1

Source code

```
# compute the sum of the elements
# of a list L
def sumlist(L):
    print("Entered sumlist")
    sum = 0
    i = 0
    while i < len(L):
        sum += L[i]
    print("Leaving sumlist")
    return sum
```

The Problem

```
>>> sumlist([1,2,3,4])
Entered sumlist
```

infinite loop

because this statement is not executing as intended

because this variable is not being updated correctly

Exercise 1

Source code

```
# compute the sum of the elements
# of a list L
def sumlist(L):
    print("Entered sumlist")
    sum = 0
    i = 0
    while i < len(L):
        sum += L[i]
    print("Leaving sumlist")
    return sum
```

The Problem

```
>>> sumlist([1,2,3,4])
Entered sumlist
```

infinite loop

immediate cause

because this statement is not executing as intended

root cause

because this variable is not being updated correctly

Exercise 2

Source code

```
1  # compute the sum of the elements
2  # of a list L
3  def sumlist(L):
4      print("Entered sumlist")
5      sum = 0
6      i = 0
7      while i < len(L):
8          i += 1
9          sum += L[i]
10     print("Leaving sumlist")
11     return sum
```

Execution behavior

```
>>> sumlist([1,2,3,4])
Entered sumlist
  File "sumlist.py", line 9
      sum += L[i]
  IndexError: list index out of range
```

What's the problem?

Exercise 2

Source code

```
1  # compute the sum of the elements
2  # of a list L
3  def sumlist(L):
4      print("Entered sumlist")
5      sum = 0
6      i = 0
7      while i < len(L):
8          i += 1
9          sum += L[i]
10     print("Leaving sumlist")
11     return sum
```

The Problem

```
>>> sumlist([1,2,3,4])
Entered sumlist
  File "sumlist.py", line 9
    sum += L[i]
IndexError: list index out of range
```

Step 1: Use error message to locate and identify the problem

- line number
- type of error

Exercise 2

Source code

```
1  # compute the sum of the elements
2  # of a list L
3  def sumlist(L):
4      print("Entered sumlist")
5      sum = 0
6      i = 0
7      while i < len(L):
8          i += 1
9          sum += L[i]
10     print("Leaving sumlist")
11     return sum
```

The Problem

```
>>> sumlist([1,2,3,4])
Entered sumlist
  File "sumlist.py", line 9
    sum += L[i]
IndexError: list index out of range
```

this value is incorrect

Exercise 2

Source code

```
1  # compute the sum of the elements
2  # of a list L
3  def sumlist(L):
4      print("Entered sumlist")
5      sum = 0
6      i = 0
7      while i < len(L):
8          i += 1
9          sum += L[i]
10     print("Leaving sumlist")
11     return sum
```

The Problem

```
>>> sumlist([1,2,3,4])
Entered sumlist
  File "sumlist.py", line 9
    sum += L[i]
  IndexError: list index out of range
```

this value is incorrect

because the order of these statements is incorrect

Exercise 2

Source code

```
1  # compute the sum of the elements
2  # of a list L
3  def sumlist(L):
4      print("Entered sumlist")
5      sum = 0
6      i = 0
7      while i < len(L):
8          i += 1
9          sum += L[i]
10     print("Leaving sumlist")
11     return sum
```

The Problem

```
>>> sumlist([1,2,3,4])
Entered sumlist
  File "sumlist.py", line 9
      sum += L[i]
  IndexError: list index out of range
```

immediate cause

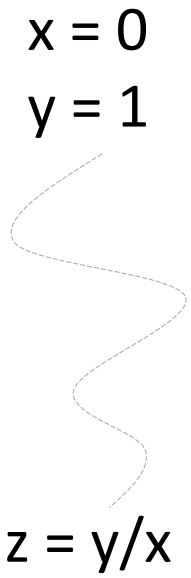
this value is incorrect

root cause

because the order of these statements is incorrect

Bugs: “immediate” vs. “root” cause

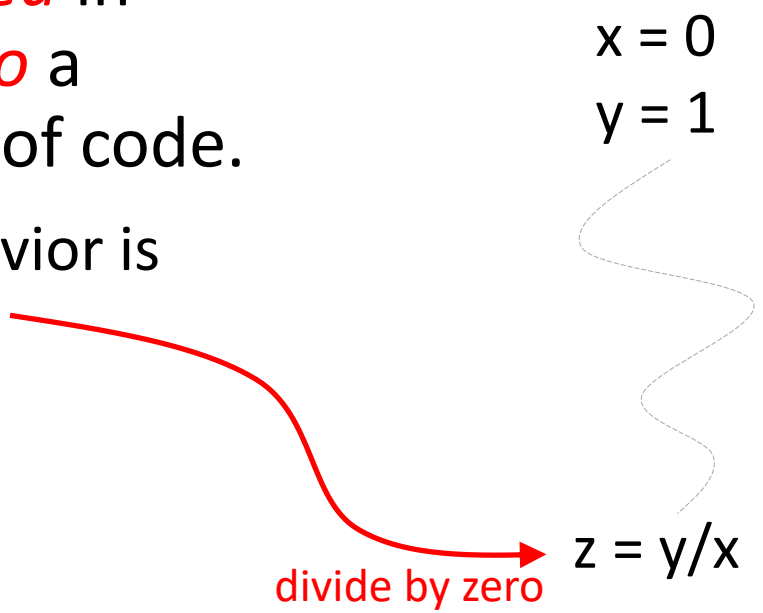
Often, buggy behavior *observed* in one piece of code *arises due to* a problem in some other piece of code.



Bugs: “immediate” vs. “root” cause

Often, buggy behavior *observed* in one piece of code *arises due to* a problem in some other piece of code.

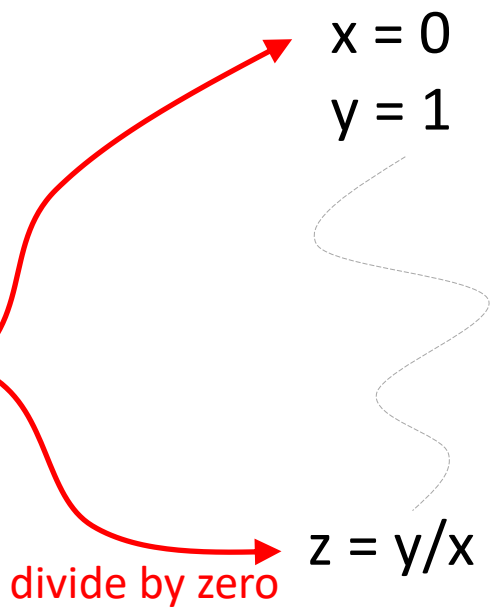
- the place where buggy behavior is observed: *immediate cause*



Bugs: “immediate” vs. “root” cause

Often, buggy behavior *observed* in one piece of code *arises due to* a problem in some other piece of code.

- the place where buggy behavior is observed: *immediate cause*
- the code that gave rise to that behavior: *root cause*



Immediate vs. root cause

Bugs: “immediate” vs. “root” cause

- “Immediate cause” : the most recently executed code that resulted in the problem showing up
- “Root cause” : the actual problem in the code that led to the immediate cause

Examples	**Immediate cause** (what you see)	**Root cause** (what you figure out)
Exercise 1	infinite loop: while loop does not terminate	the variable i is not being updated correctly
Exercise 2	array index out of bounds: variable i has incorrect value	order of statements is incorrect

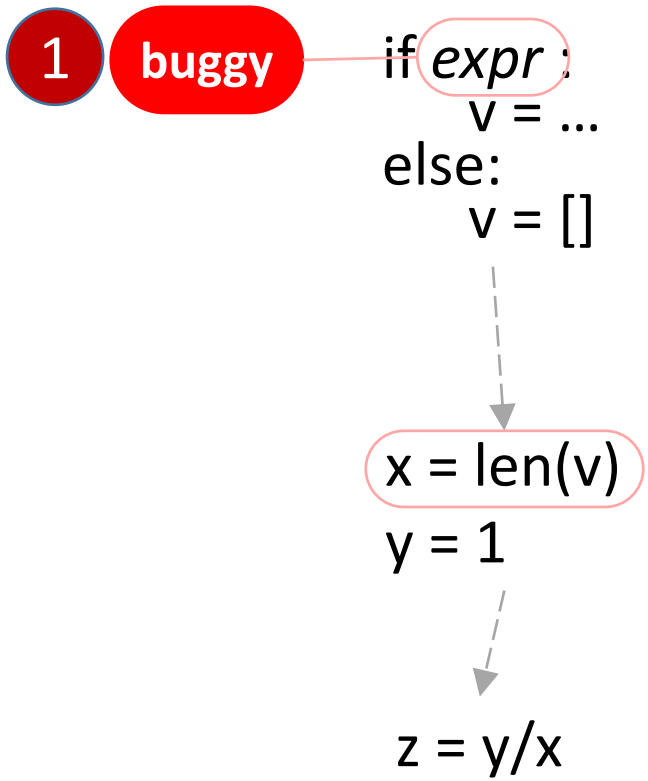
Bugs: “immediate” vs. “root” cause

The root cause for a problem may itself be due to a bug somewhere else in the code

```
if expr :
    v = …
else:
    v = []

x = len(v)
y = 1

z = y/x
```

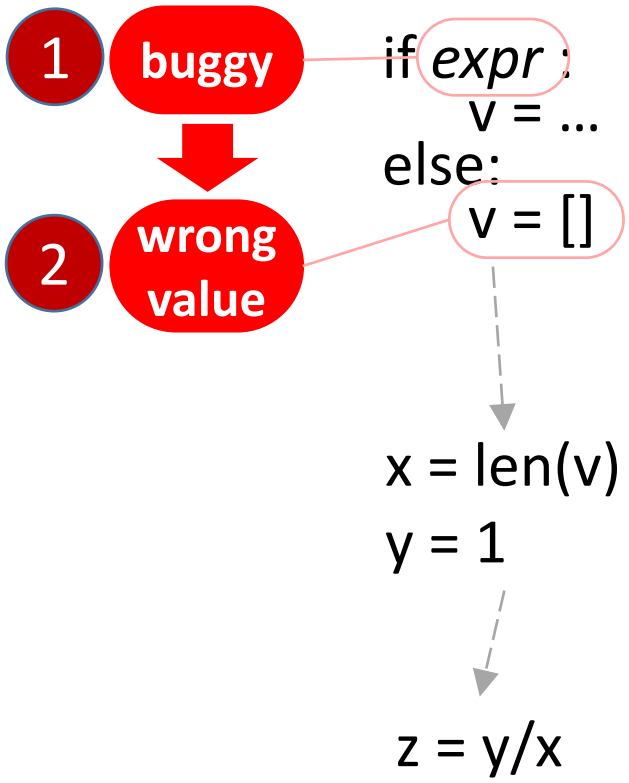
Bugs: “immediate” vs. “root” cause

The root cause for a problem may itself be due to a bug somewhere else in the code



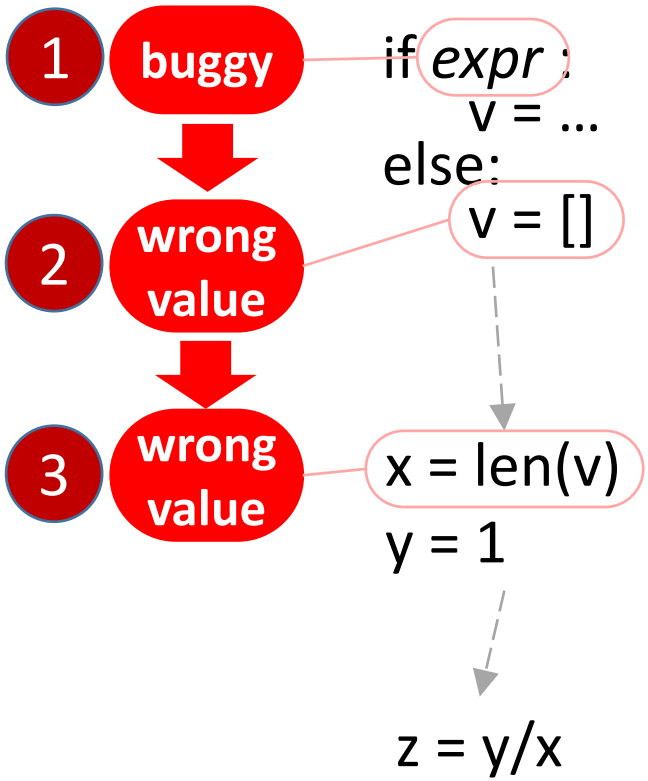
Bugs: “immediate” vs. “root” cause

The root cause for a problem may itself be due to a bug somewhere else in the code



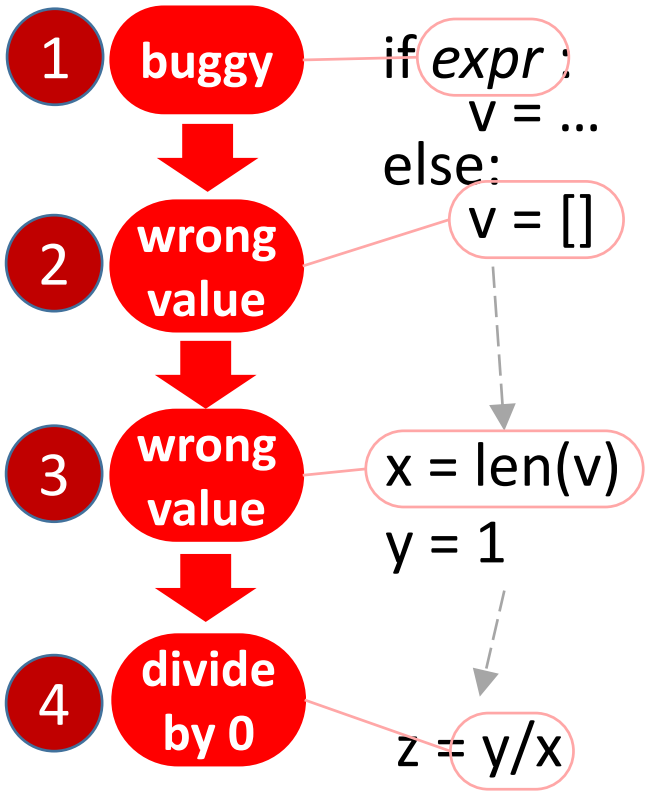
Bugs: “immediate” vs. “root” cause

The root cause for a problem may itself be due to a bug somewhere else in the code



Bugs: “immediate” vs. “root” cause

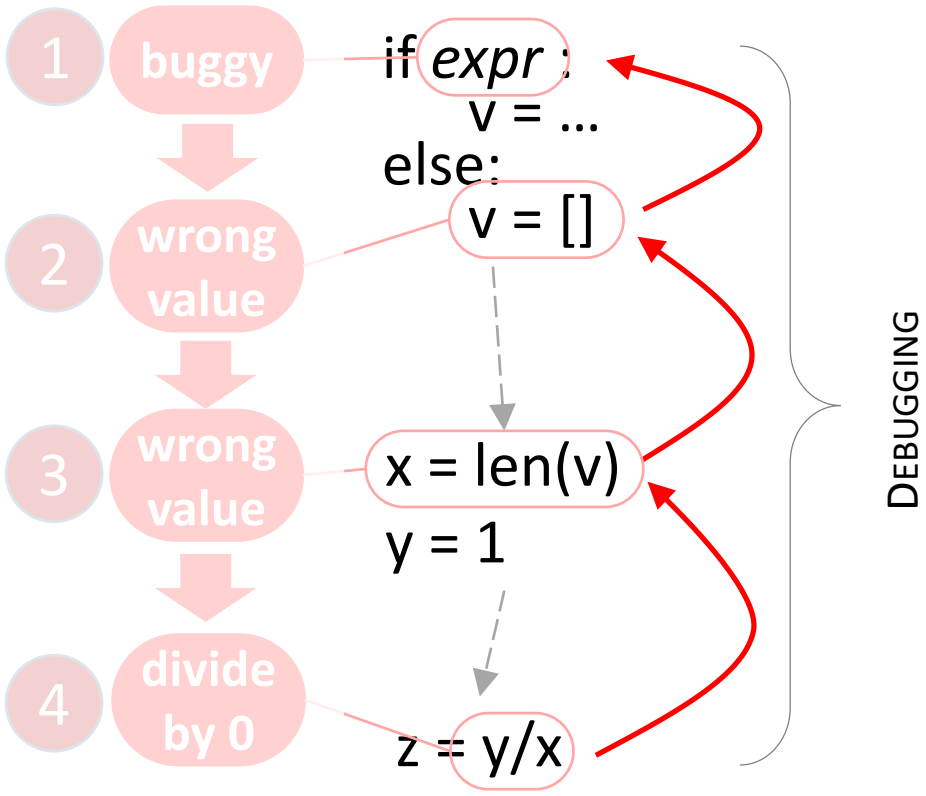
The root cause for a problem may itself be due to a bug somewhere else in the code



Bugs: “immediate” vs. “root” cause

The root cause for a problem may itself be due to a bug somewhere else in the code

- to fix the bug, we have to work back to figure out where the problem began



Exercise 3

Source code

```
def average(L):
    return sum(L)/len(L)

def main():
    L = …some computation…
    avg = average(L)
    print(avg)
```

Execution behavior

```
>>> main()
  File "average.py", line 2
      return sum(L)/len(L)
ZeroDivisionError: division by zero
```

1. What's the problem?
2. What's the immediate cause?
3. What's the root cause?

Exercise 3

Source code

```
def average(L):
    return sum(L)/len(L)

def main():
    L = …some computation…
    avg = average(L)
    print(avg)
```

The Problem

```
>>> main()
  File "average.py", line 2
    return sum(L)/len(L)
ZeroDivisionError: division by zero
```

Problem: divide-by-zero error

Exercise 3

Source code

```
def average(L):
    return sum(L)/len(L)

def main():
    L = …some computation…
    avg = average(L)
    print(avg)
```

The Problem

```
>>> main()
  File "average.py", line 2
    return sum(L)/len(L)
ZeroDivisionError: division by zero
```

Problem: divide-by-zero error

immediate cause

because L == [ ]

Exercise 3

Source code

```
def average(L):
    return sum(L)/len(L)

def main():
    L = …some computation…
    avg = average(L)
    print(avg)
```

The Problem

```
>>> main()
  File "average.py", line 2
    return sum(L)/len(L)
ZeroDivisionError: division by zero
```

Problem: divide-by-zero error

immediate cause

because L == [ ]

root cause

because this computation returned an empty list

EXERCISE

Source Code

```
def write_to_file(fname, data):
    if not fname.endswith(".txt"):
        fname += ".txt"
    fname.write(data)
```

Execution behavior

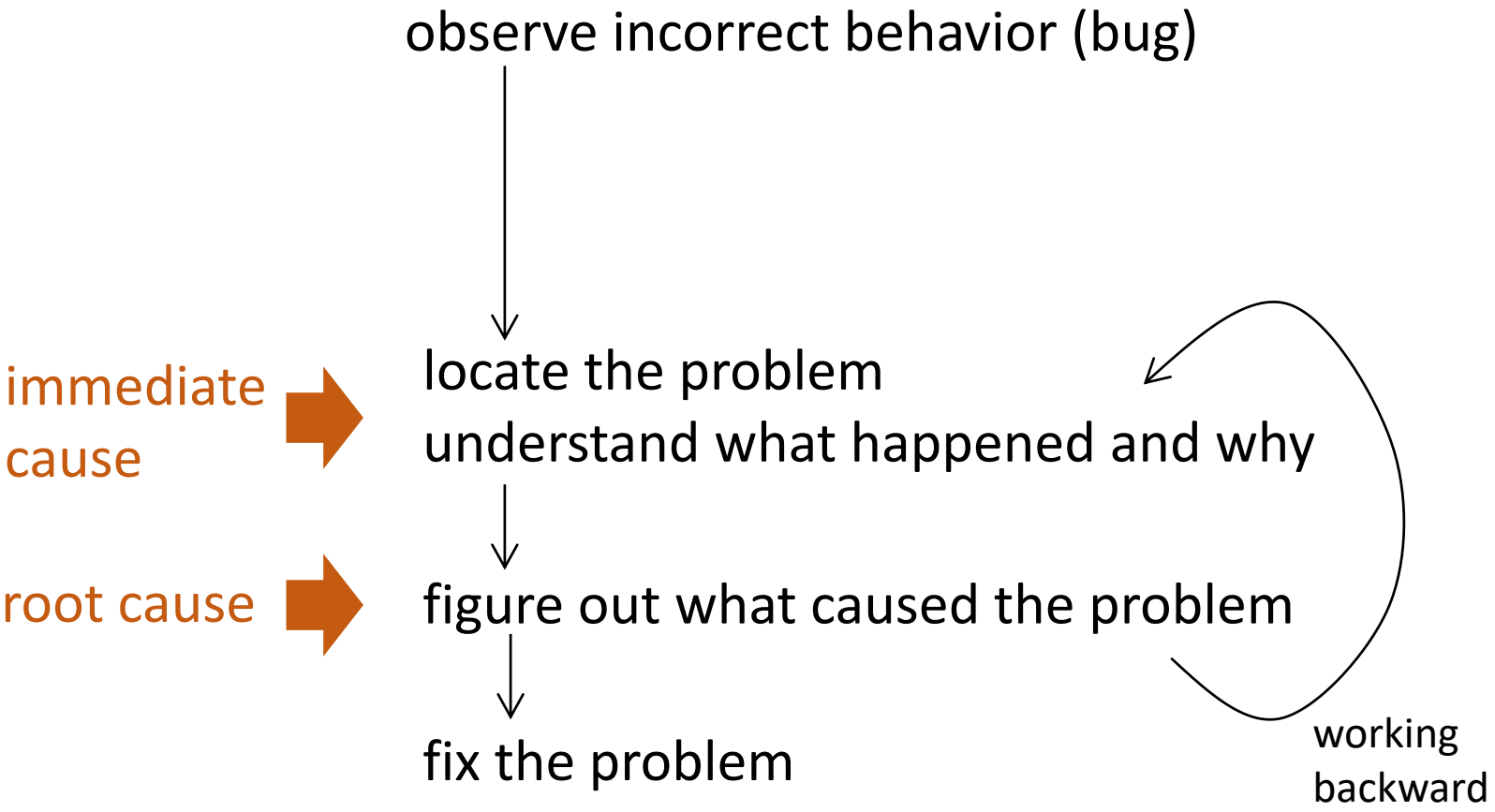
```
>>> write_to_file("myfile", [1,2,3])
File "myprog.py", line 8
    fname.write(data)
AttributeError: 'str' object has no attribute 'write'
```

QUESTION:

1. *What's the problem?*
2. *What's the immediate cause?*
3. *What's the root cause?*

The debugging process

The debugging process



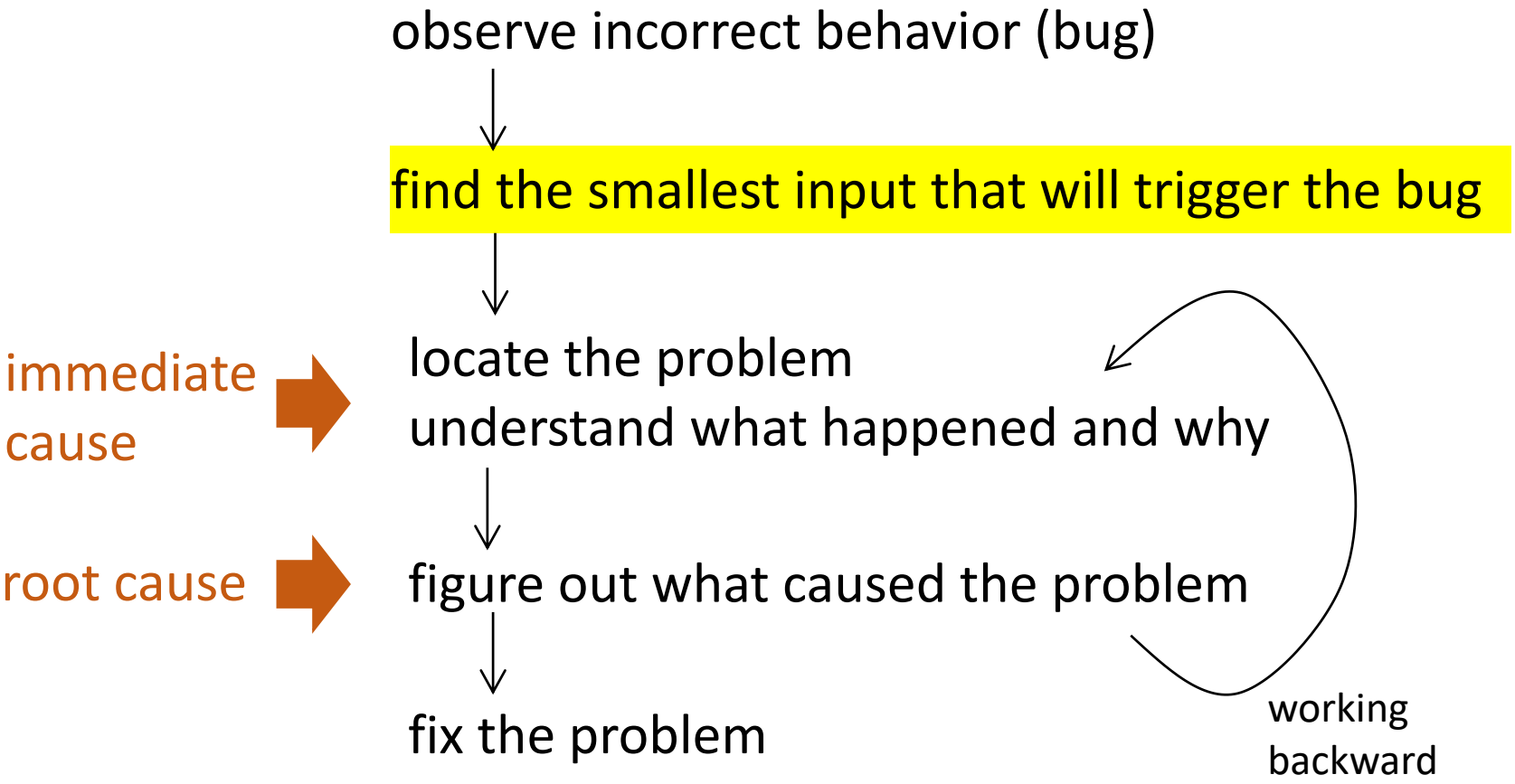
Effective debugging

- If we had infinite time, debugging would be easy(er)
  - but we don't
- Effective debugging ≡ finding and fixing bugs *quickly*
- Programs that need debugging often:
  - involve a lot of code
  - process a lot of data
  - use complex logic
  - (some or all of the above)

searching through all this is what makes debugging difficult and time-consuming

- *Effective debugging minimizes the amount of search necessary*

The debugging process



Shrinking the input

Goal: find the smallest* input that triggers the *same* bug

- we can do this mechanically

Idea: Identify and get rid of computation that is *irrelevant* to the bug we're working on

- It's important that the smaller input trigger the same bug as the original input

* In practice, we usually stop when the input is small enough to be "manageable" and/or when we've spent enough time

EXERCISE

A program *P* that reads in and processes a data file

- *P* gives a divide-by-zero error on line 513 when run on an input file *F* containing 100,000 data items
- we divide *F* into several pieces and run *P* on each:
  - input F_1 (12,000 items) $\Rightarrow$ index-out-of-bounds error on line 602
  - input F_2 (32,000 items) $\Rightarrow$ divide-by-zero error on line 676
  - input F_3 (51,000 items) $\Rightarrow$ divide-by-zero error on line 513
  - input F_4 (35,000 items) $\Rightarrow$ no errors

***Question**: which of these inputs is acceptable for debugging the original problem? Why?*

SOLUTION

A program *P* that reads in and processes a data file

- *P* gives a divide-by-zero error on line 513 when run on an input file *F* containing 100,000 data items

- we divide *F* into several pieces and run *P* on each:
  - input F_1 (12,000 items) $\Rightarrow$ index-out-of-bounds error on line 602
  - input F_2 (32,000 items) $\Rightarrow$ divide-by-zero error on line 676
  - → input F_3 (51,000 items) $\Rightarrow$ divide-by-zero error on line 513
  - input F_4 (35,000 items) $\Rightarrow$ no errors

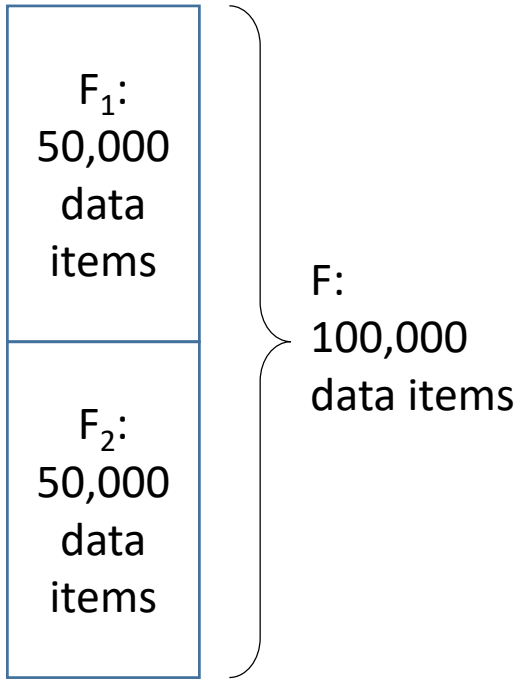
The smaller input should give the same error in the same place in the code

EXERCISE

We have a program *P* that reads in a data file and computes something about the data

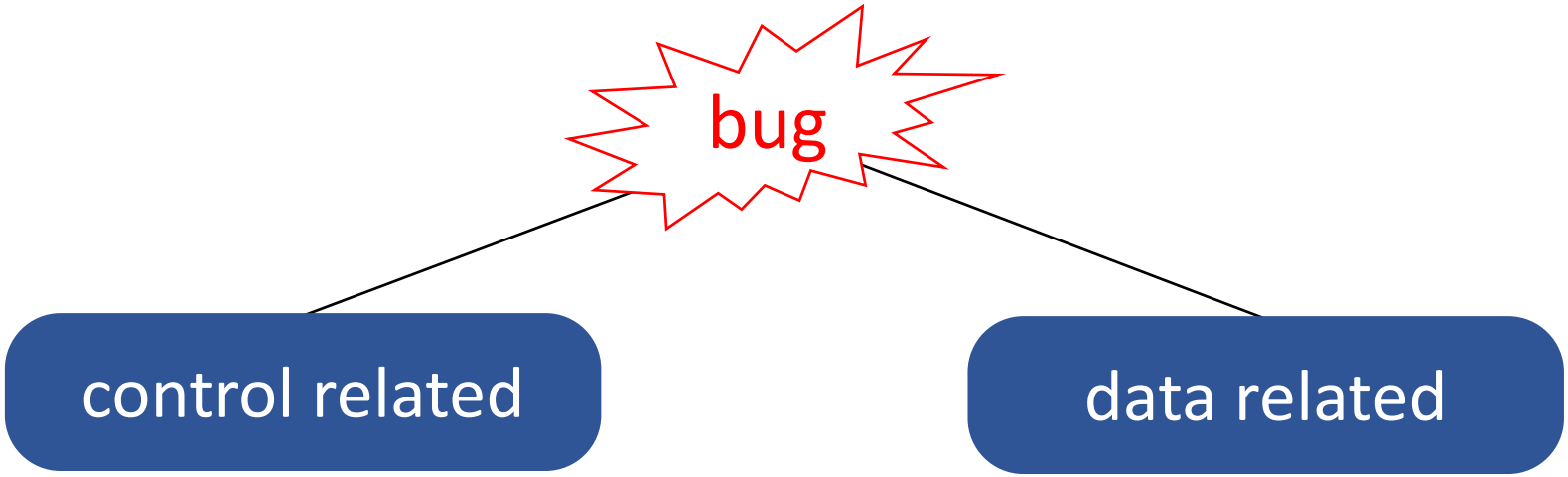
- *P* gives a divide-by-zero error on line 513 an input file *F* containing 100,000 data items
- we divide *F* into two halves F_1 and F_2
- we find that *P* works fine on both F_1 and F_2

***Question**: What can we do to shrink the input?*



Understanding bugs

Kinds of bugs



Statements are not executed in the right order

Data values are not correct

I. Control-related bugs

Identifying control-related bugs

Problem: program statements are executed when they shouldn't, or not executed when they should.

Check: Is the problem due to any of:

- code getting executed when it shouldn't?
- code not getting executed when it should?
- code getting executed in the wrong order?

Identifying control-related bugs

Problem: program statements are executed when they shouldn't, or not executed when they should.

Check: Is the problem due to any of:

- code getting executed when it shouldn't?
- code not getting executed when it should?
- code getting executed in the wrong order?

How?

- add print statements to your code to see which statements are being executed and in what order

EXERCISE

Source Code	Execution behavior

```
def write_to_file(fname, data):
    if not fname.endswith(".txt"):
        fname += ".txt"
    f = open(fname, "w")
    f.write(data)
```

```
>>> write_to_file("myfile.txt", [1,2,3])
```

Problem: nothing is written out

QUESTION:

- *Is this a control-related bug or a data-related bug?*
- *Why?*

EXERCISE

Source Code	Execution behavior

```
VOWELS = "aeiou"

# count the no. of vowels in string
count_vowels(string):
    count = 0
    for letter in string:
        if letter in VOWELS:
            count += 1
    return count
```

```
>>> count_vowels("Apple")
1
```

QUESTION:

- *Is this a control-related bug or a data-related bug?*
- *Why?*
- *What is the immediate cause?*
- *What is the root cause?*

Control-related bugs

Problem: program statements are executed when they shouldn't, or not executed when they should.

Infinite loop

```
i = 0
while i < len(L):
    sum += L[i]
```

- The loop body is executing when it shouldn't (infinitely)
- The statement after the loop is not getting executed

L == [] ⇒ Divide-by-0 exception

```
def average(L):
    return sum(L)/len(L)
```

- the expression in the return statement is getting evaluated when it shouldn't

Control-related bugs

Common* **reason**: incorrect data values

* common ≠ universal

- e.g., for expressions that control **if**- or **while-** statements

Infinite loop

```
i = 0
while i < len(L):
    sum += L[i]
```

- The values for this expression are incorrect
  - it doesn't become **False** when it should
  - reason: the values of i are incorrect

L == [] ⇒ Divide-by-0 exception

```
def average(L):
    return sum(L)/len(L)
```

- The division operation is performed even if the value of **len(L)** is bad

Control-related bugs: causes

Common* **cause**: incorrect data values

* common ≠ universal

- e.g., for expressions that control **if**- or **while-** statements

Infinite loop

```
i = 0
while i < len(L):
    sum += L[i]
```

- The values for this expression are incorrect
  - it doesn't become **False** when it should
  - reason: the values of i are incorrect

L == [] ⇒ Divide-by-0 exception

```
def average(L):
    return sum(L)/len(L)
```

- The division operation is performed even if the value of **len(L)** is bad

immediate cause: control-related

Control-related bugs: causes

Common* **cause**: incorrect data values

* common ≠ universal

- e.g., for expressions that control **if**- or **while-** statements

Infinite loop

```
i = 0
while i < len(L):
    sum += L[i]
```

- The values for this expression are incorrect
  - it doesn't become **False** when it should
  - reason: the values of i are incorrect

L == [] ⇒ Divide-by-0 exception

```
def average(L):
    return sum(L)/len(L)
```

- The division operation is performed even if the value of **len(L)** is bad

immediate cause: control-related

root cause: data-related

Debugging Control-related bugs

When the immediate cause of a bug is control-related, i.e.:

- code is being executed when it shouldn't, or
- not being executed when it should:

- Check the culprit code to figure out which variables control whether or not it gets executed

- Ask whether the values of these variables are causing the culprit code to:
  - being executed when it shouldn't; or
  - the culprit code to not being executed when it should

EXERCISE

Source Code	Execution behavior

```
VOWELS = "aeiou"

# count the no. of vowels in string
count_vowels(string):
    count = 0
    for letter in string:
        if letter in VOWELS:
            count += 1
    return count
```

```
>>> count_vowels("Apple")
1
```

QUESTION:

- *Where should we add* print() *statements to figure out what's going on?*
- *Why?*

EXERCISE

Source Code	Execution behavior

```
def f(x):
    ... body of function f ...

def g(x):
    ... body of function g ...

def h(x):
    ... body of function h ...

def main():
    u = f(1)
    v = g(u)
    w = h(v)
    print(w)
```

```
>>> main()
```

← nothing happens

QUESTION:

1. *What do you think might be the problem?*
2. *How can we identify the immediate cause?*

EXERCISE

Source Code	Execution behavior

```
def f(x):
    ... body of function f ...

def g(x):
    ... body of function g ...

def h(x):
    ... body of function h ...

def main():
    w = h(g(f(1)))
    print(w)
```

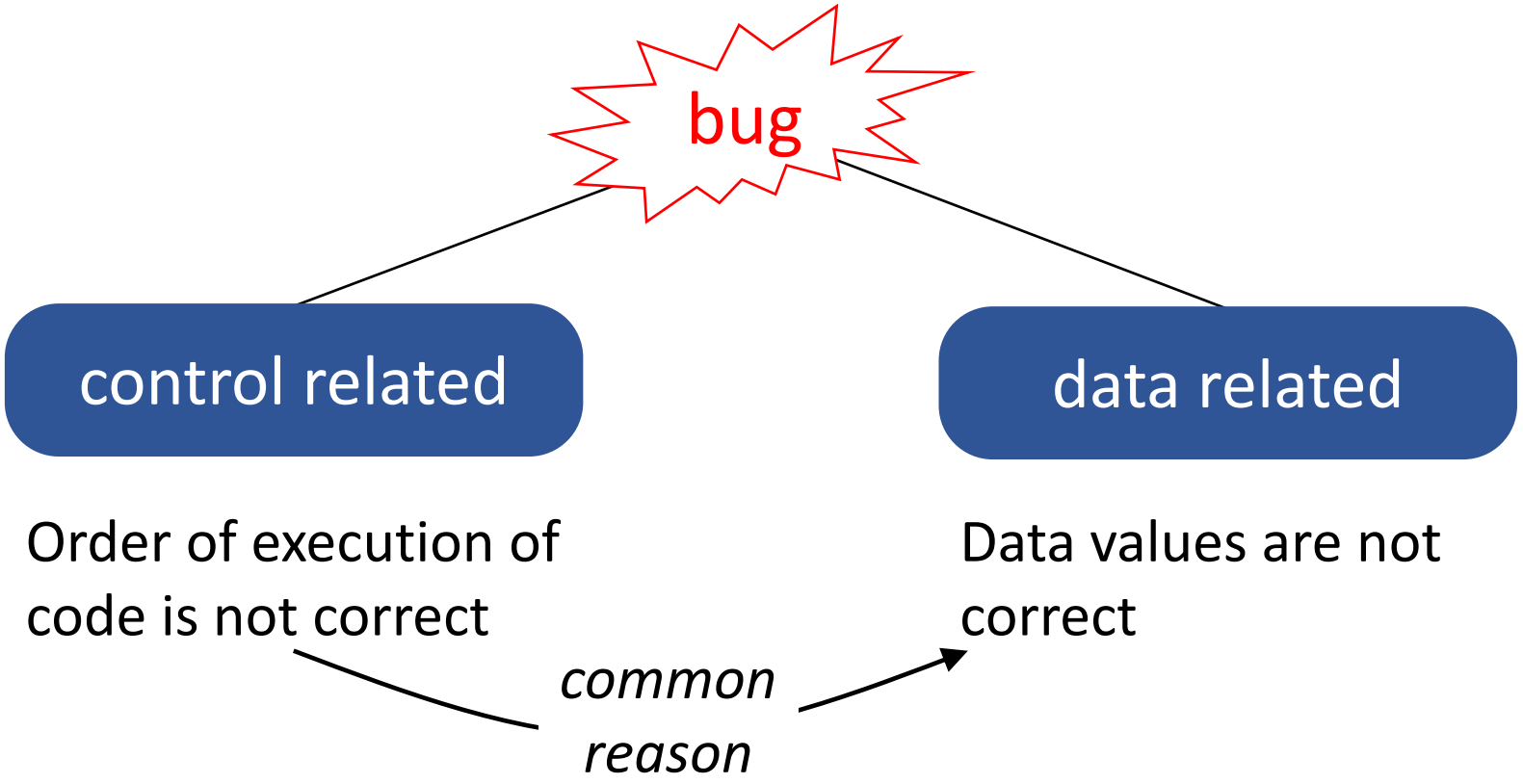
```
>>> main()
```

← nothing happens

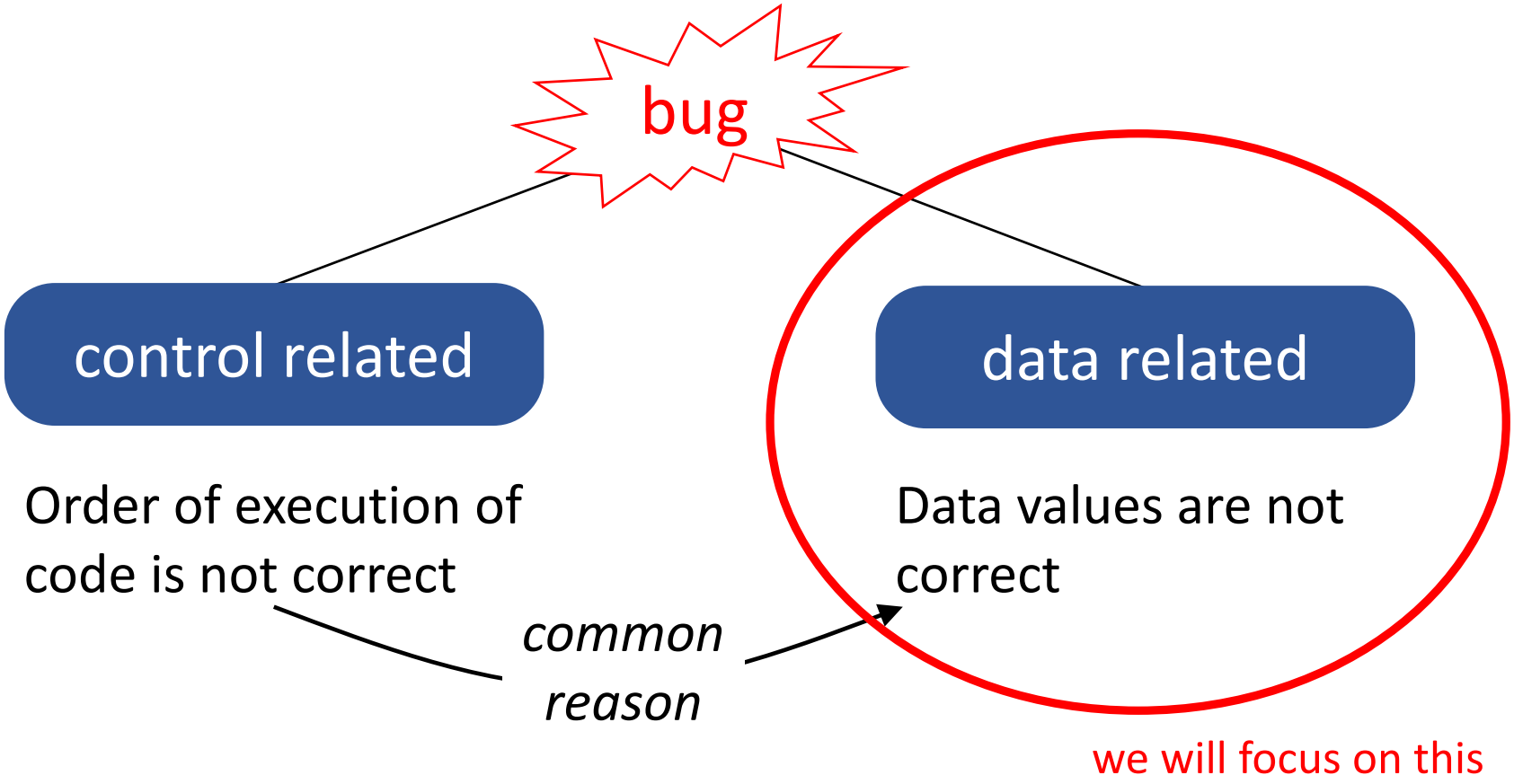
QUESTION:

1. *What do you think might be the problem?*
2. *How can we identify the immediate cause?*

Kinds of bugs



Kinds of bugs



II. Data-related bugs

Identifying data-related bugs

Problem: variables or expressions have the wrong value

Check: Is the problem due to any of:

- a variable or expression having the wrong value?
- a function being called with wrong argument values?

Identifying data-related bugs

Problem: variables or expressions have the wrong value

Check: Is the problem due to any of:

- a variable or expression having the wrong value?
- a function being called with wrong argument values?

How?

- for each variable/expression x we want to investigate:
  - figure out how to tell whether or not it has the right value
  - print the value of x at (i.e., just before) the problem point and check whether it has the right value

Identifying data-related bugs

Problem: variables or expressions have the wrong value

Check: Is the problem due to any of:

- a variable or expression having the wrong value?
- a function being called with wrong argument values?

How?

IMPORTANT!

- for each variable/expression x we want to investigate:
  - figure out how to tell whether or not it has the right value
  - print the value of x at (i.e., just before) the problem point and check whether it has the right value

EXERCISE

Source Code	Execution behavior

```
# Each line of infile is: first name
# followed by last name
def print_last_names(infile):
    f = open(infile)
    for line in f:
        name = line.split()
        last_name = name[1]
        print("@@ " + last_name)
    f.close()
```

```
>>> print_last_names("myfile")
@@ Asimov
@@ Heinlein
@@ Le
@@ Willis
```

Problem: there is no name in the input file with last name 'Le' (there is, however, the name 'Ursula Le Guin')

QUESTION:

- *Is this a control-related bug or a data-related bug?*
- *Why?*

EXERCISE

Source code	Execution behavior

```
# compute the sum of the elements
# of a list L
def sumlist(L):
    sum = 0
    i = 0
    while i < len(L):
        i += 1
        sum += L[i]
    return sum
```

```
>>> sumlist([1,2,3,4])
"Entered sumlist"
  File "sumlist.py", line 9
    sum += L[i]
IndexError: list index out of range
```

QUESTION:

- *Is this a control-related bug or a data-related bug?*
- *Why?*

Data-related bugs: causes

```
# get any leading digits indicating repetition
match = re.match("(\d+)(.+)", fmt)
if match is None:
    reps = 1
else:
    reps = int(match.group(1))
    fmt = match.group(2)

if fmt[0] == "(":  # process parenthesized format list recursively
    fmt = fmt[1:-1]
    fmt_list = fmt.split(",")
    rexp = self.gen_output_fmt(fmt_list)
else:
    if fmt[0] in "iI":  # integer
        sz = fmt[1:]
        gen_fmt = "{}"
        cvt_fmt = "{:" + str(sz) + "d}"
        rexp = [(gen_fmt, cvt_fmt, int(sz))]

    elif fmt[0] in "xX":
        gen_fmt = " "
        rexp = [(gen_fmt, None, None)]

    elif fmt[0] in "aA":
        gen_fmt = "{}"
        # the '*' in the third position of the tuple (corresponding to
        # field width) indicates that the field can be arbitrarily wide
        rexp = [(gen_fmt, "{}", "*")]

    elif fmt[0] in "eEfFgG":  # various floating point formats
        idx0 = fmt.find(".")
        sz = fmt[1:idx0]
        suffix = fmt[idx0 + 1 :]
        # The 'E' and G formats can optionally specify the width of
        # the exponent, e.g.: 'E15.3E2'.  For now we ignore any such
        # the exponent width -- but if it's there, we need to extract
        # the sequence of digits before it.
        m = re.match("(\d+).*", suffix)
        assert m is not None, f"Improper format? '{fmt}'"
        prec = m.group(1)
        gen_fmt = "{}"
        cvt_fmt = "{:" + sz + "." + prec + fmt[0] + "}"
        rexp = [(gen_fmt, cvt_fmt, int(sz))]

    elif fmt[0] in "pP":  # scaling factor
        # For now we ignore scaling: there are lots of other things we
        # need to spend time on.  To fix later if necessary.
        e
        rexp = [(gen_fmt, None, None)]

    elif fmt[0] == "/":  # newlines
        gen_fmt = "\\n" * len(fmt)
```

Possible Causes

The value of an expression is incorrect here (data-related bug)

y = some_expr(x)

Data-related bugs: causes

```
# get any leading digits indicating repetition
match = re.match("(\d+)(.+)", fmt)
if match is None:
    reps = 1
else:
    reps = int(match.group(1))
    fmt = match.group(2)

if fmt[0] == "(":  # process parenthesized format list recursively
    fmt = fmt[1:-1]
    fmt_list = fmt.split(",")
    rexp = self.gen_output_fmt(fmt_list)
else:
    if fmt[0] in "iI":  # integer
        sz = fmt[1:]
        gen_fmt = "{}"
        cvt_fmt = "{:" + str(sz) + "d}"
        rexp = [(gen_fmt, cvt_fmt, int(sz))]

    elif fmt[0] in "xX":
        gen_fmt = " "
        rexp = [(gen_fmt, None, None)]

    elif fmt[0] in "aA":
        gen_fmt = "{}"
        # the '*' in the third position of the tuple (corresponding to
        # field width) indicates that the field can be arbitrarily wide
        rexp = [(gen_fmt, "{}", "*")]

    elif fmt[0] in "eEfFgG":  # various floating point formats
        idx0 = fmt.find(".")
        sz = fmt[1:idx0]
        suffix = fmt[idx0 + 1 :]
        # The 'E' and G formats can optionally specify the width of
        # the exponent, e.g.: 'E15.3E2'.  For now we ignore any such
        # the exponent width -- but if it's there, we need to extract
        # the sequence of digits before it.
        m = re.match("(\d+).*", suffix)
        assert m is not None, f"Improper format? '{fmt}'"
        prec = m.group(1)
        gen_fmt = "{}"
        cvt_fmt = "{:" + sz + "." + prec + fmt[0] + "}"
        rexp = [(gen_fmt, cvt_fmt, int(sz))]

    elif fmt[0] in "pP":  # scaling factor
        # For now we ignore scaling: there are lots of other things we
        # need to spend time on.  To fix later if necessary.
        e
        y = some_expr(x)
        rexp = [(gen_fmt, None, None)]

    elif fmt[0] == "/":  # newlines
        gen_fmt = "\\n" * len(fmt)
```

Possible Causes

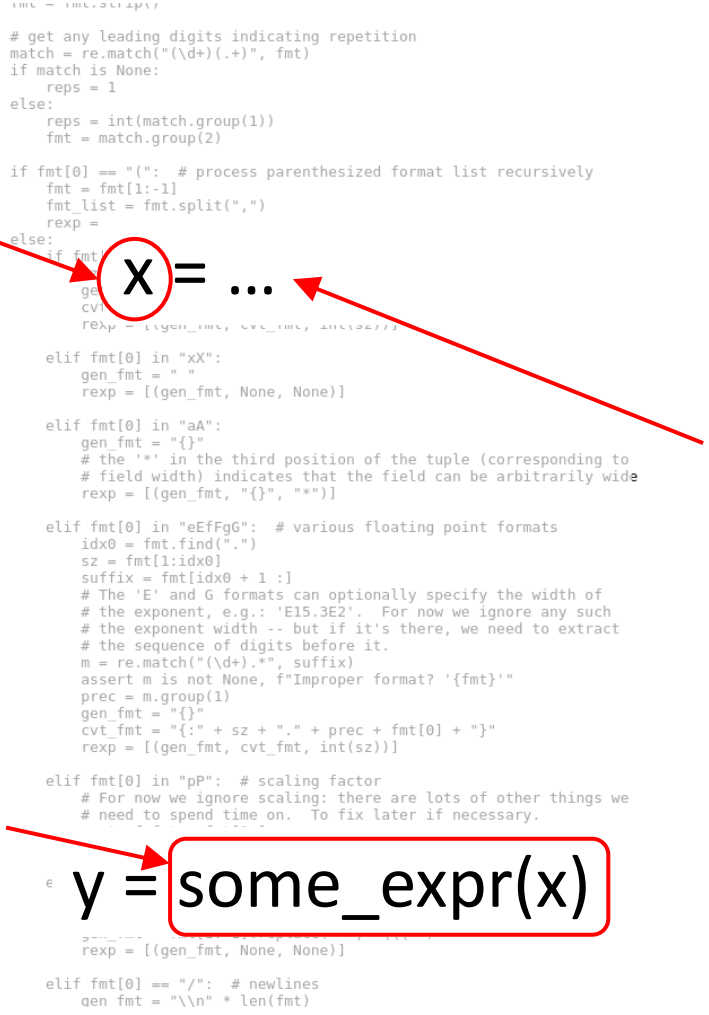
1. the expression is wrong;

The value of an expression is incorrect here (data-related bug)

Data-related bugs: causes

x is last assigned a value here

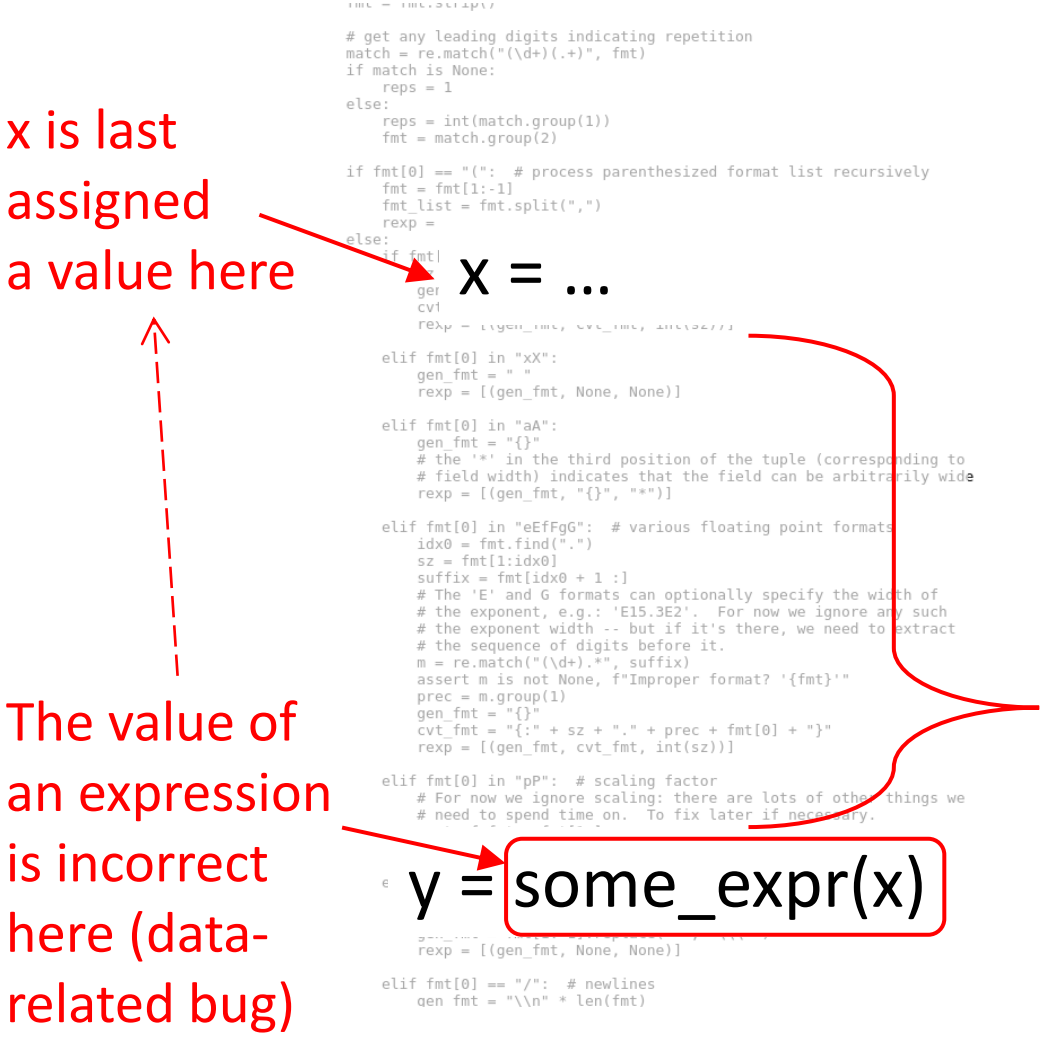
The value of an expression is incorrect here (data-related bug)



Possible Causes

1. the expression is wrong; or
2. the expression is right, but an operand x is assigned the wrong value; or

Data-related bugs: causes



Possible Causes

1. the expression is wrong; or
2. the expression is right, but an operand x is assigned the wrong value here; or
3. x is assigned the right value, but this is accidentally overwritten somewhere later

EXERCISE

Source Code

```
# Each line of infile is: first name
# followed by last name
def print_last_names(infile):
    f = open(infile)
    for line in f:
        name = line.split()
        last_name = name[1]
        print("@@ " + last_name)
    f.close()
```

Problem: prints out the last name 'Le' for the input 'Ursula Le Guin'

Possible causes

1. the expression is incorrect;
2. the expression is OK, but some operand was assigned the wrong value;
3. the expression is OK, its operands were assigned the right values, but some value got overwritten.

QUESTION:

Which of these best describes the cause of this bug?

EXERCISE

Source Code

```
# compute the sum of the elements
# of a list L
def sumlist(L):
    sum = 0
    i = 0
    while i < len(L):
        i += 1
        sum += L[i]
    return sum
```

Problem: "IndexError: list index out of range"

Possible causes

1. the expression is incorrect;
2. the expression is OK, but some operand was assigned the wrong value;
3. the expression is OK, its operands were assigned the right values, but some value got overwritten.

QUESTION:

Which of these best describes the cause of this bug?

EXERCISE

Source Code

```
# compute the sum of the elements
# of a list L
def sumlist(L):
    sum = 0
    i = 0
    while i <= len(L):
        sum += L[i]
        i += 1
    return sum
```

Problem: "IndexError: list index out of range"

Possible causes

1. the expression is incorrect;
2. the expression is OK, but some operand was assigned the wrong value;
3. the expression is OK, its operands were assigned the right values, but some value got overwritten.

QUESTION:

Which of these best describes the cause of this bug?

EXERCISE

Source Code

```
def process(data):
    assert   data[“gdp”] != 0
    assert   data[“pop”] != 0
    exports = get_export_info(data)
    imports = get_import_info(data)
    gdp = data[“gdp”]
    pop = data[“pop”]
    per_capita_gdp = gdp/pop

def main():
    db = read_db()
    process(db[“usa”])

main()
```

Problem: Divide-by-zero Error

here

Possible causes

1. the expression is incorrect;
2. the expression is OK, but some operand was assigned the wrong value;
3. the expression is OK, its operands were assigned the right values, but some value got overwritten.

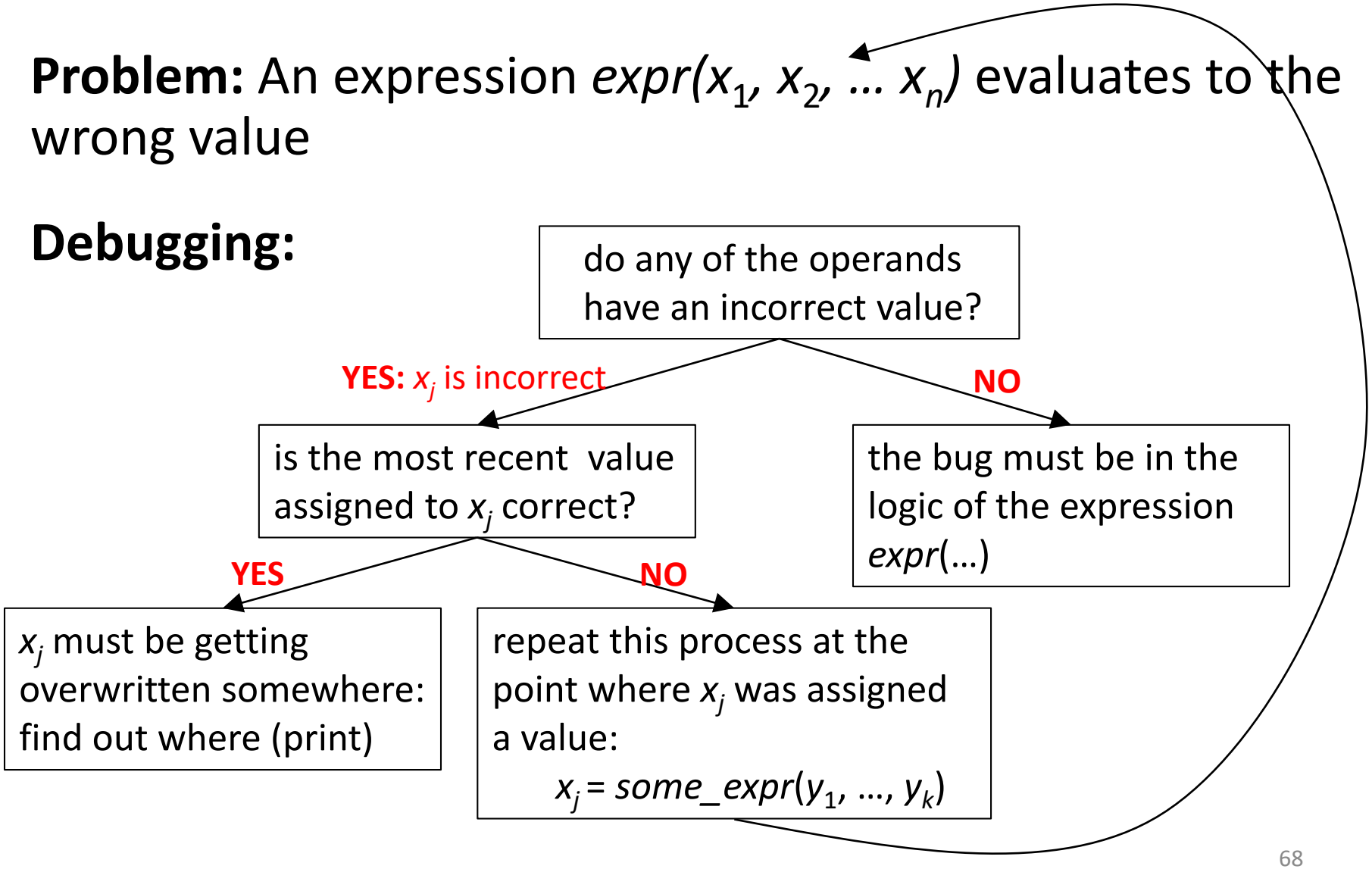
QUESTION:

Which of these best describes the cause of this bug?

Data-related bugs: Debugging

Problem: An expression *expr*(x_1, x_2, … x_n) evaluates to the wrong value

Debugging:



Debugging: working backwards

```
fmt = fmt.strip()

# get any leading digits indicating repetition
match = re.match("(\d+)(.+)", fmt)
if match is None:
    reps = 1
else:
    reps = int(match.group(1))
    fmt = match.group(2)

if fmt[0] == "(":  # process parenthesized format list recursively
    fmt = fmt[1:-1]
    fmt_list = fmt.split(",")
    rexp = self.gen_output_fmt(fmt_list)
else:
    if fmt[0] in "iI":  # integer
        sz = fmt[1:]
        gen_fmt = "{}"
        cvt_fmt = "{:" + str(sz) + "d}"
        rexp = [(gen_fmt, cvt_fmt, int(sz))]

    elif fmt[0] in "xX":
        gen_fmt = " "
        rexp = [(gen_fmt, None, None)]

    elif fmt[0] in "aA":
        gen_fmt = "{}"
        # the '*' in the third position of the tuple (corresponding to
        # field width) indicates that the field can be arbitrarily wide
        rexp = [(gen_fmt, "{}", "*")]

    elif fmt[0] in "eEfFgG":  # various floating point formats
        idx0 = fmt.find(".")
        sz = fmt[1:idx0]
        suffix = fmt[idx0 + 1 :]
        # The 'E' and G formats can optionally specify the width of
        # the exponent, e.g.: 'E15.3E2'.  For now we ignore any such
        # the exponent width -- but if it's there, we need to extract
        # the sequence of digits before it.
        m = re.match("(\d+).*", suffix)
        assert m is not None, f"Improper format? '{fmt}'"
        prec = m.group(1)
        gen_fmt = "{}"
        cvt_fmt = "{:" + sz + "." + prec + fmt[0] + "}"
        rexp = [(gen_fmt, cvt_fmt, int(sz))]

    elif fmt[0] in "pP":  # scaling factor
        # For now we ignore scaling: there are lots of other things we
        # need to spend time on.  To fix later if necessary.
                           1:]
                           tput_fmt_list(rest_of_fmt)

                           # character string
                           # -2 for the quote at either end
        # escape any double-quotes in the string
        gen_fmt = fmt[1:-1].replace('"', '\\"')
        rexp = [(gen_fmt, None, None)]

    elif fmt[0] == "/":  # newlines
        gen_fmt = "\\n" * len(fmt)
```

print(x)

wrong value printed:
data-related bug

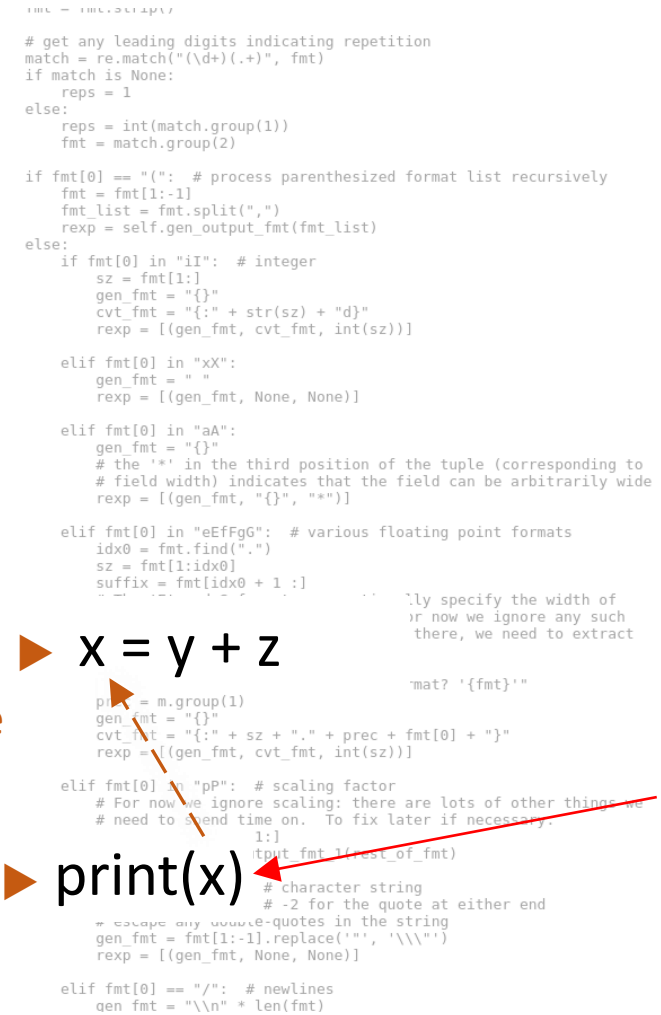
Debugging: working backwards

```
fmt = fmt.strip()

# get any leading digits indicating repetition
match = re.match("(\d+)(.+)", fmt)
if match is None:
    reps = 1
else:
    reps = int(match.group(1))
    fmt = match.group(2)

if fmt[0] == "(":  # process parenthesized format list recursively
    fmt = fmt[1:-1]
    fmt_list = fmt.split(",")
    rexp = self.gen_output_fmt(fmt_list)
else:
    if fmt[0] in "iI":  # integer
        sz = fmt[1:]
        gen_fmt = "{}"
        cvt_fmt = "{:" + str(sz) + "d}"
        rexp = [(gen_fmt, cvt_fmt, int(sz))]

    elif fmt[0] in "xX":
        gen_fmt = " "
        rexp = [(gen_fmt, None, None)]

    elif fmt[0] in "aA":
        gen_fmt = "{}"
        # the '*' in the third position of the tuple (corresponding to
        # field width) indicates that the field can be arbitrarily wide
        rexp = [(gen_fmt, "{}", "*")]

    elif fmt[0] in "eEfFgG":  # various floating point formats
        idx0 = fmt.find(".")
        sz = fmt[1:idx0]
        suffix = fmt[idx0 + 1 :]
        # The 'E' and G formats can optionally specify the width of
        # the exponent, e.g.: 'E15.3E2'.  For now we ignore any such
        # the exponent width -- but if it's there, we need to extract
        # the sequence of digits before it.
        m = re.match("(\d+).*", suffix)
        assert m is not None, f"Improper format? '{fmt}'"
        prec = m.group(1)
        gen_fmt = "{}"
        cvt_fmt = "{:" + sz + "." + prec + fmt[0] + "}"
        rexp = [(gen_fmt, cvt_fmt, int(sz))]

    elif fmt[0] in "pP":  # scaling factor
        # For now we ignore scaling: there are lots of other things we
        # need to spend time on.  To fix later if necessary.
                                1:]
                                tput_fmt_list(rest_of_fmt)

                                # character string
                                # -2 for the quote at either end
        # escape any double-quotes in the string
        gen_fmt = fmt[1:-1].replace('"', '\\"')
        rexp = [(gen_fmt, None, None)]

    elif fmt[0] == "/":  # newlines
        gen_fmt = "\\n" * len(fmt)
```

immediate cause: x's value ▶ print(x)

wrong value printed: data-related bug

Debugging: working backwards



root cause for x's value

suppose that y's value is incorrect, then:

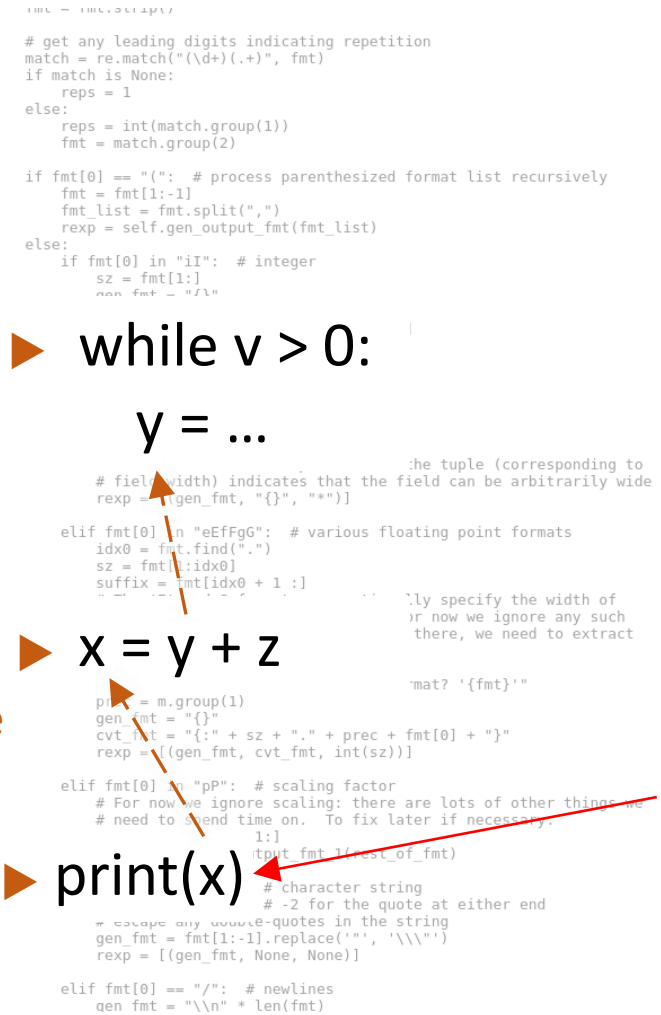
immediate cause: y's value

▶ x = y + z

immediate cause: x's value ▶ print(x)

wrong value printed: data-related bug

Debugging: working backwards



root cause for y's value
immediate cause: v > 0

▶ while v > 0:

y = …

root cause for x's value

suppose that y's value is incorrect, then:

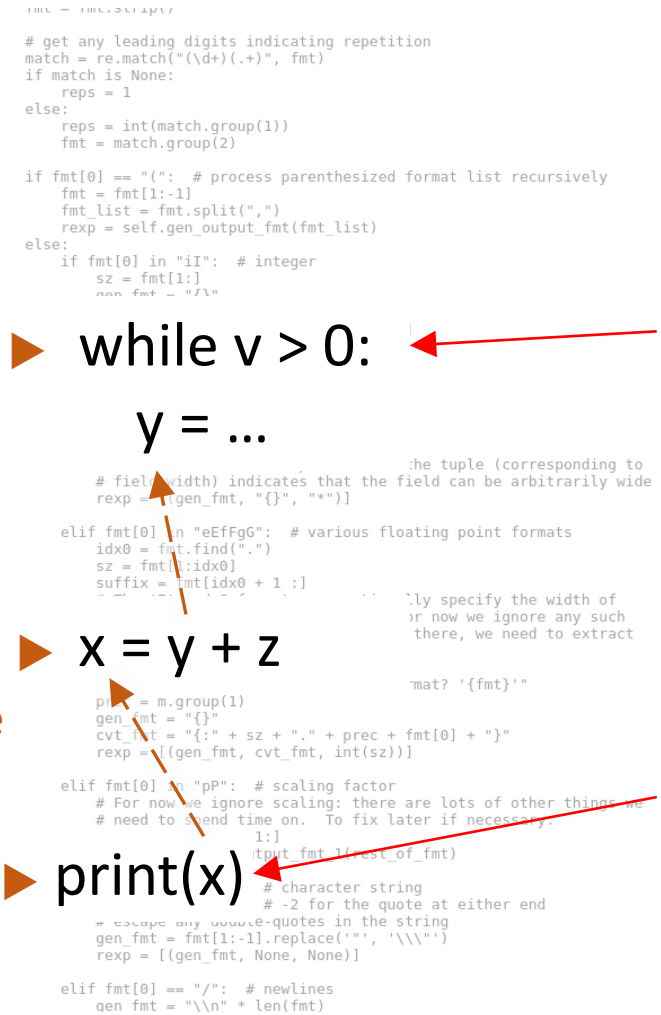
immediate cause: y's value

▶ x = y + z

immediate cause: x's value ▶ print(x)

wrong value printed:
data-related bug

Debugging: working backwards



root cause for y's value
immediate cause: v > 0

while v > 0:

y = …

control-related bug

root cause for x's value

suppose that y's value is incorrect, then:

immediate cause: y's value

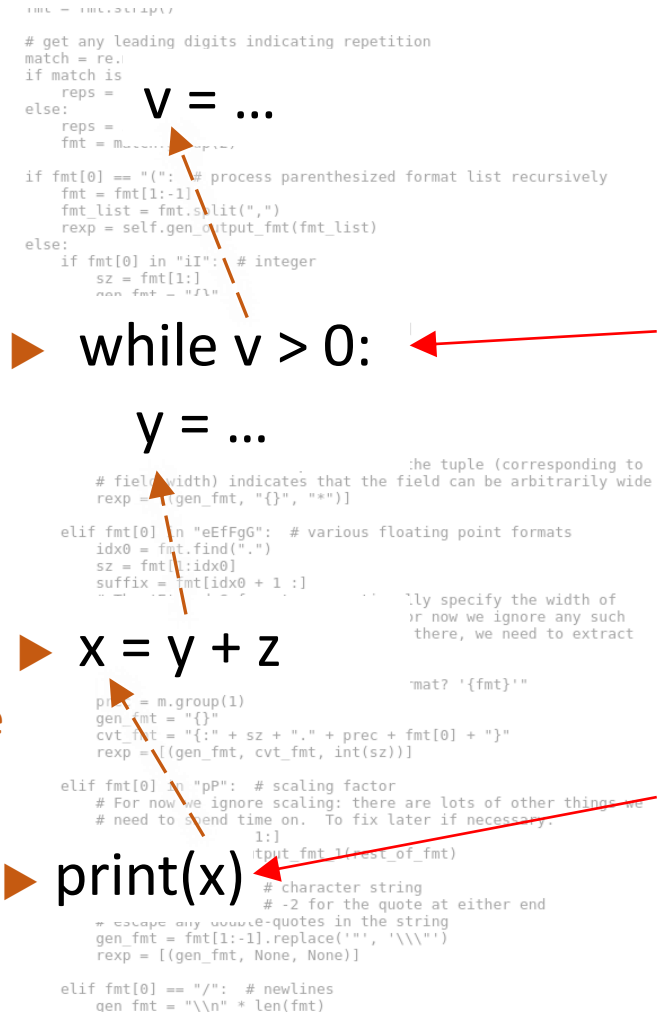
x = y + z

wrong value printed:
data-related bug

immediate cause: x's value

print(x)

Debugging: working backwards



root cause for y's value
immediate cause: v > 0

root cause for x's value

suppose that y's value is incorrect, then:
immediate cause: y's value

immediate cause: x's value

control-related bug

wrong value printed:
data-related bug

EXERCISE

Source code

```
# compute the absolute value of x
def abs(x):
    if x < 0:
        x = −x
        return x

# compute the sum of the absolute
# values of the numbers in a list L
def sumlist_abs(L):
    sum = 0
    i = 0
    while i < len(L):
        sum += abs(L[i])
        i += 1
    return sum
```

Execution behavior

```
>>> sumlist_abs([−2, −3])
5
>>> sumlist_abs([2, 3])
  File "sumlist.py", line 12
    sum += abs(L[i])
TypeError: unsupported operand
types for +=: 'int' and 'NoneType'
```

QUESTION:

- *What is the immediate cause?*
- *What should we print out to identify the root cause?*

SOLUTION

Source code

```
# compute the absolute value of x
def abs(x):
    if x < 0:
        x = −x
        return x

# compute the sum of the absolute
# values of the numbers in a list L
def sumlist_abs(L):
    sum = 0
    i = 0
    while i < len(L):
        sum += abs(L[i])
        i += 1
    return sum
```

Debugging process: step 0

File "sumlist.py", line 12

sum += abs(L[i])

TypeError: unsupported operand types for +=: 'int' and 'NoneType'

Immediate cause: one of the operands of += has the wrong type (NoneType)

∴ its value must be wrong as well

TODOs for the next debugging step:

1. which operand?
2. where did it get its value?

SOLUTION

Source code

```
# compute the absolute value of x
def abs(x):
    if x < 0:
        x = -x
        return x

# compute the sum of the absolute
# values of the numbers in a list L
def sumlist_abs(L):
    sum = 0
    i = 0
    while i < len(L):
        sum += abs(L[i])
        i += 1
    return sum
```

Debugging process: step 1

TODOs for the next debugging step:

1. which operand?
   - print out the operands of +=, i.e., sum and abs(L[i]), just before the += is evaluated
   - check to see if OK

SOLUTION

Source code

```
# compute the absolute value of x
def abs(x):
    if x < 0:
        x = -x
        return x

# compute the sum of the absolute
# values of the numbers in a list L
def sumlist_abs(L):
    sum = 0
    i = 0
    while i < len(L):
        sum += abs(L[i])
        i += 1
    return sum
```

Debugging process: step 1

TODOs for the next debugging step:

1. which operand?
   - print out the operands of +=, i.e., sum and abs(L[i]), just before the += is evaluated
   - check to see if OK

⇒ we find that abs(L[i]) is None

- this value is incorrect (*immediate cause*)

SOLUTION

Source code

```
# compute the absolute value of x
def abs(x):
    if x < 0:
        x = –x
        return x

# compute the sum of the absolute
# values of the numbers in a list L
def sumlist_abs(L):
    sum = 0
    i = 0
    while i < len(L):
        sum += abs(L[i])
        i += 1
    return sum
```

Debugging process: step 1

TODOs for the next debugging step:

1. which operand?
   - print out the operands of +=, i.e., sum and abs(L[i]), just before the += is evaluated
   - check to see if OK

⇒ we find that abs(L[i]) is None

- this value is incorrect (*immediate cause*)

TODOs for the next debugging step:

- identify where this value came from (root cause)

SOLUTION

Source code

```
# compute the absolute value of x
def abs(x):
    if x < 0:
        x = −x
        return x

# compute the sum of the absolute
# values of the numbers in a list L
def sumlist_abs(L):
    sum = 0
    i = 0
    while i < len(L):
        sum += abs(L[i])
        i += 1
    return sum
```

Debugging process: step 2

PROBLEM: abs(L[i]) is None

(immediate cause)

POSSIBLE CAUSES:

- the argument to abs() has the wrong value
- the argument to abs() is OK but the value returned is wrong

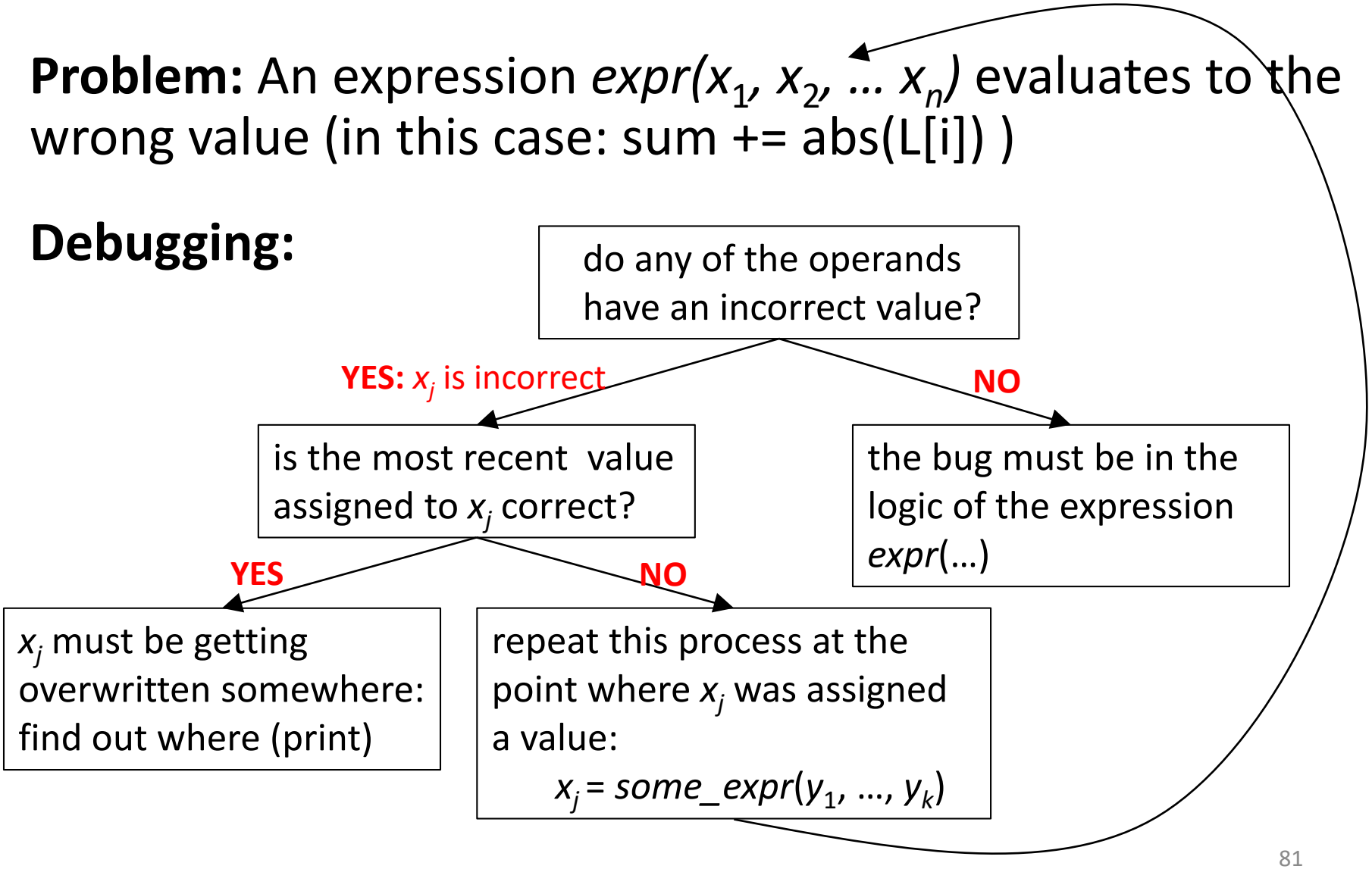
DEBUGGING ACTION:

- print out the argument L[i] and the return value abs(L[i]) at this point

Debugging process: Recap

Problem: An expression *expr*(x_1, x_2, … x_n) evaluates to the wrong value (in this case: sum += abs(L[i]))

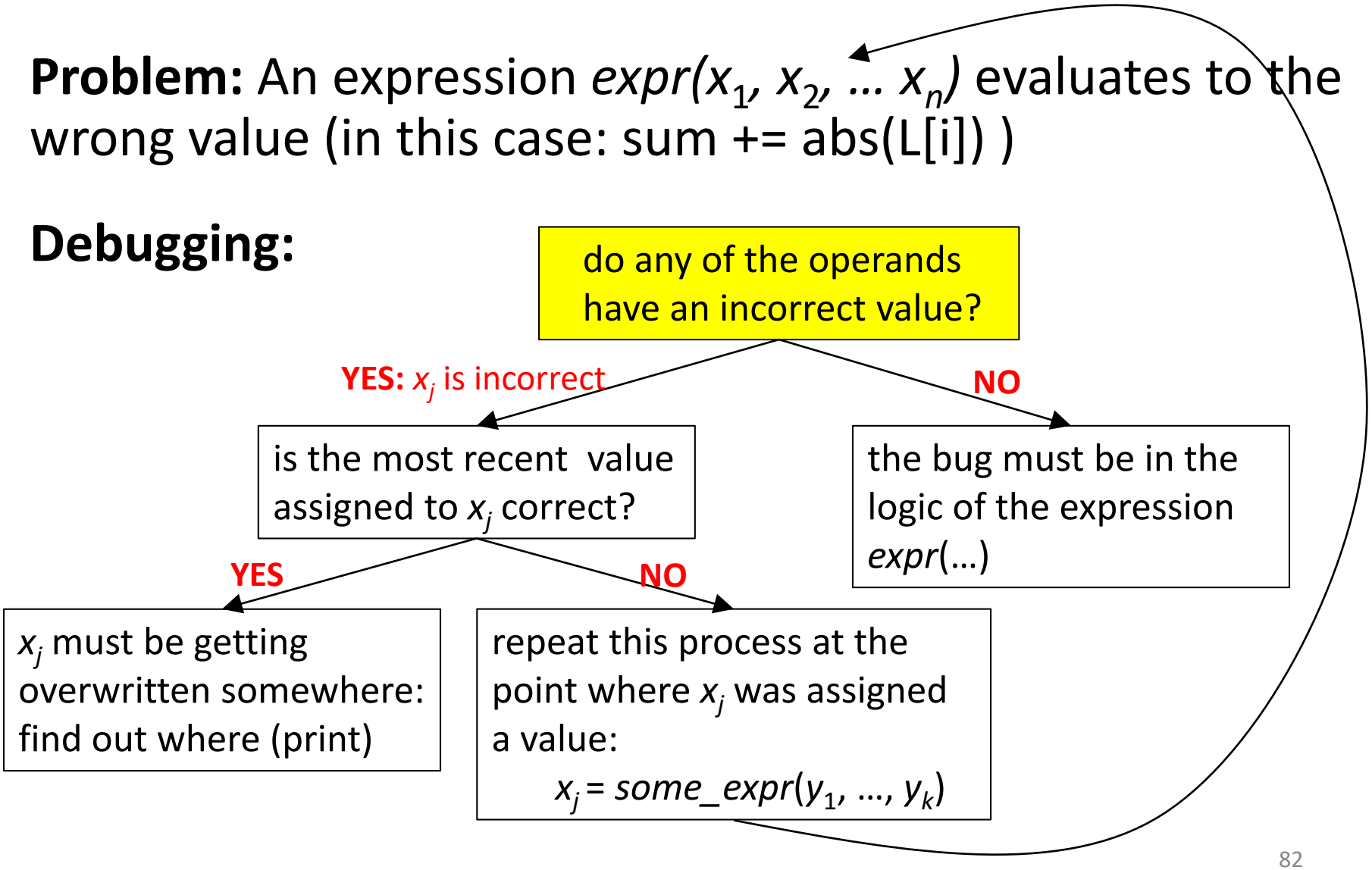
Debugging:



Debugging process: Recap

Problem: An expression *expr*(x_1, x_2, … x_n) evaluates to the wrong value (in this case: sum += abs(L[i]))

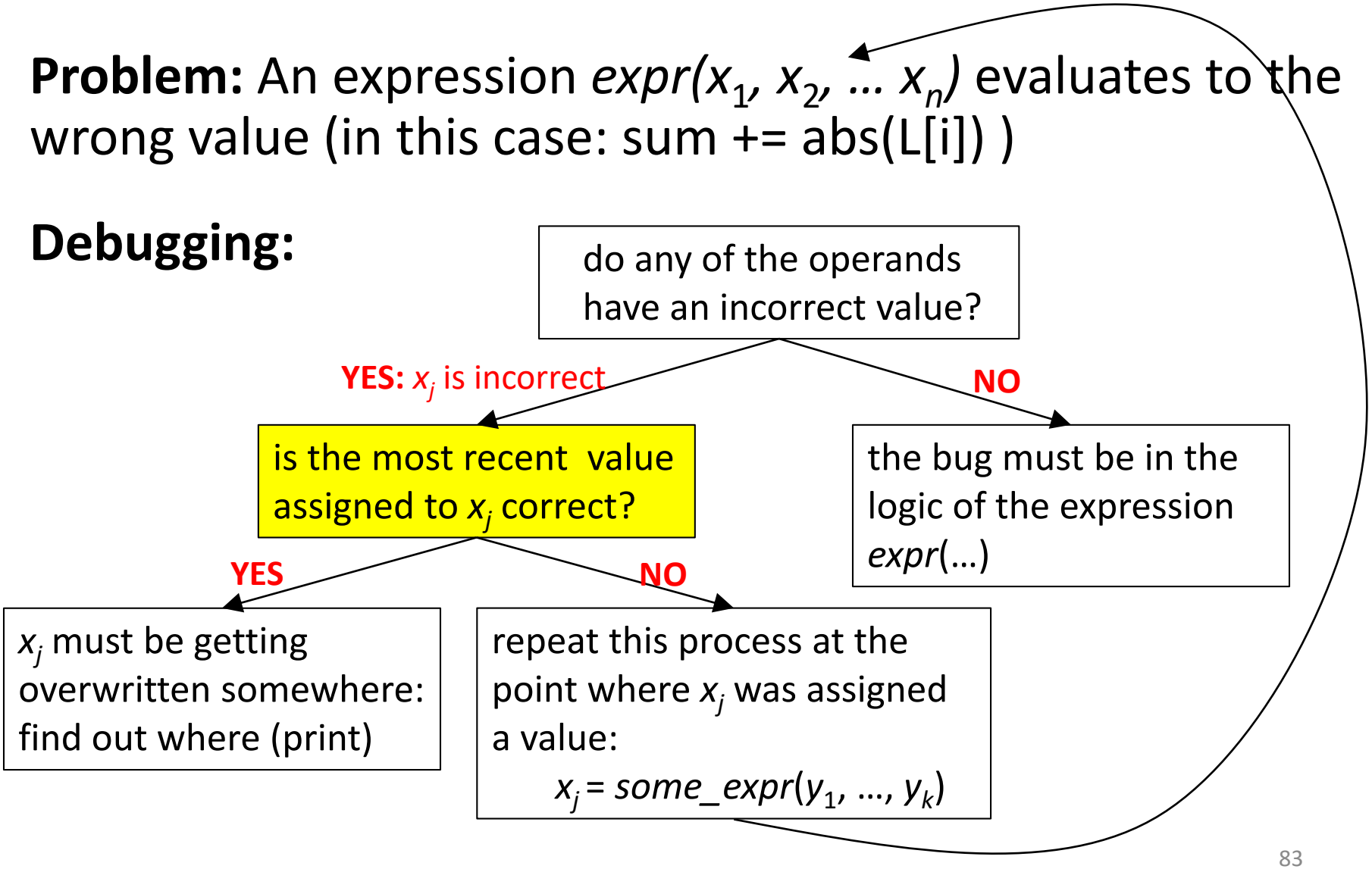
Debugging:



Debugging process: Recap

Problem: An expression *expr*(x_1, x_2, … x_n) evaluates to the wrong value (in this case: sum += abs(L[i]))

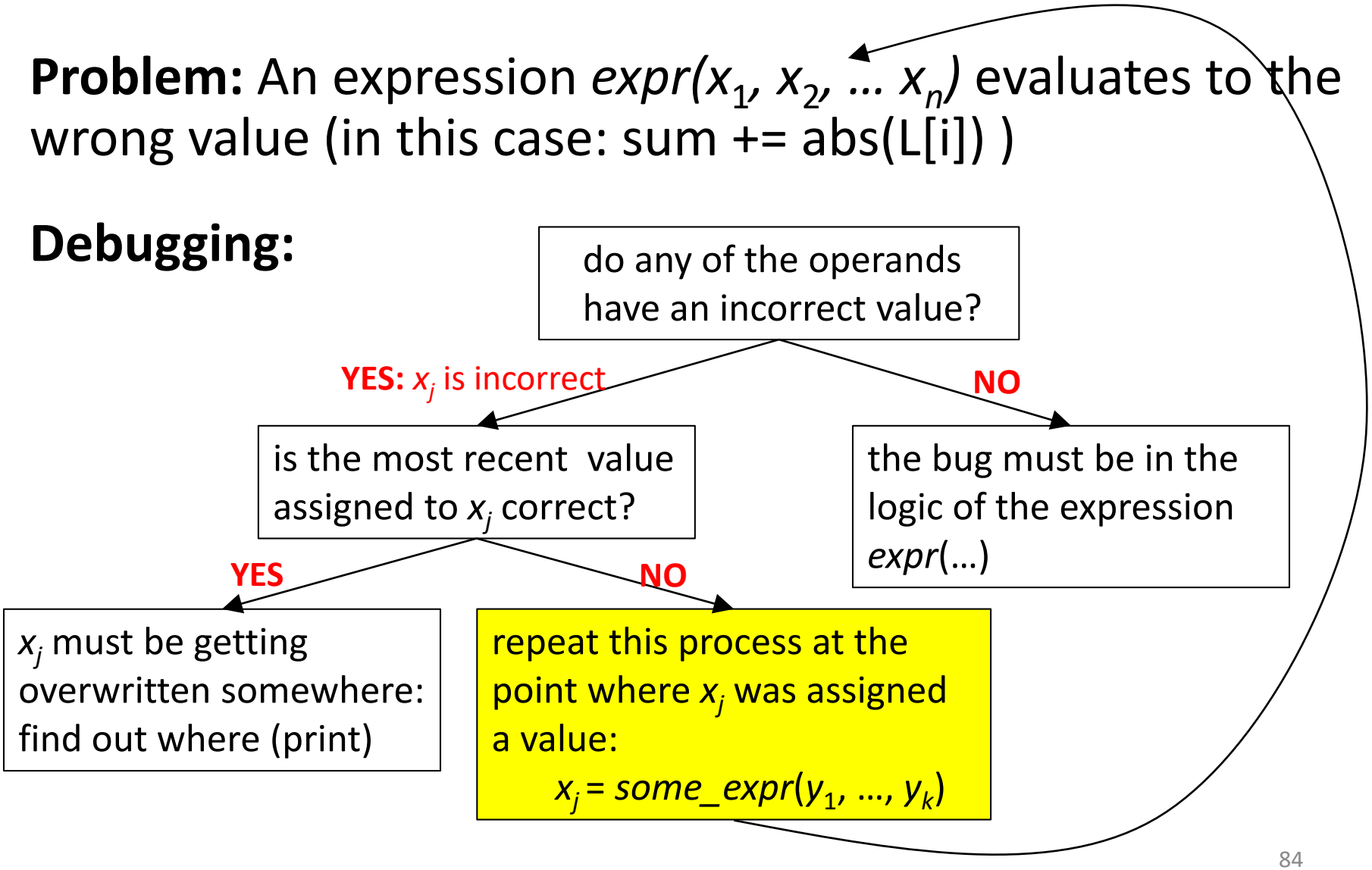
Debugging:



Debugging process: Recap

Problem: An expression *expr*(x_1, x_2, … x_n) evaluates to the wrong value (in this case: sum += abs(L[i]))

Debugging:



EXERCISE

Source code	Execution behavior

```
def myfun(u, v, L):
    payment = sum = 0
    i = 0
    while i < len(L):
        mdiff = max(L) – min(L)
        if mdiff > 5:
            p = u + v
        else:
            p = u – v
    ① if p < 0:
            p = 0
        payment += 2 * p + sum
        sum += abs(L[i])
        i += 1
    return payment
```

```
>>> myfun(6, 10, [2, 3])
File "myfun.py", line 13
    sum += abs(L[i])
TypeError: unsupported operand
types for +=: 'int' and 'NoneType'
```

QUESTION:

Suppose we find that the immediate cause is that abs(L[i]) *is* None

- *To determine the root cause: should we print the value of* p *at* ①*?*
- *Why or why not?*

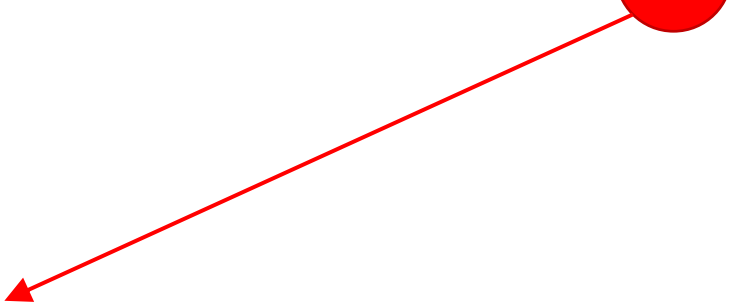
EXERCISE

Source code	Execution behavior

```
def process(data):
    assert data[0] is not None
    check_values(data)
    names = get_names(data)
    values = get_values(data)
    pn = proc_names(names)
    pv = proc_values(values)
    assert data[0] is not None
    print_output(pn, pv, data)

def main():
    data = read_data()
    process(data)
```

```
>>> main()
AssertionError
```

here



QUESTION:

- *What is the immediate cause?*
- *What should we do to identify the root cause?*

Bugs: control-related vs. data-related

Control-related vs. data-related bugs

- Control-related bugs: refers to incorrect execution of statements
  - e.g., infinite loop; statements being executed when they shouldn't; wrong order of statements;
- Data-related bugs: refers to incorrect computation of data values
  - e.g., incorrect expression; wrong values for operands
- *Sometimes, a bug can be treated as either control-related or data-related*

Example

```
# sum the elements of
# a list L
def sumlist(L):
    sum = 0
    i = 0
    while i < len(L):
        i += 1
        sum += L[i]
    return sum
```

```
>>> sumlist([1,2,3,4])
File "sumlist.py", line 8
    sum += L[i]
IndexError: list index out of range
```

Example

```
# sum the elements of
# a list L
def sumlist(L):
    sum = 0
    i = 0
    while i < len(L):
        i += 1
        sum += L[i]
    return sum
```

Control-related bug: statement order is wrong

Fix: *change statement order*

```
def sumlist(L):
    sum = 0
    i = 0
    while i < len(L):
        sum += L[i]
        i += 1
    return sum
```

```
>>> sumlist([1,2,3,4])
File "sumlist.py", line 8
    sum += L[i]
IndexError: list index out of range
```

Example

```
# sum the elements of
# a list L
def sumlist(L):
    sum = 0
    i = 0
    while i < len(L):
        i += 1
        sum += L[i]
    return sum
```

```
>>> sumlist([1,2,3,4])
File "sumlist.py", line 8
    sum += L[i]
IndexError: list index out of range
```

Control-related bug: statement order is wrong

Fix: *change statement order*

```
def sumlist(L):
    sum = 0
    i = 0
    while i < len(L):
        sum += L[i]
        i += 1
    return sum
```

Data-related bug: expression logic is wrong

Fix: *change expression logic*

```
def sumlist(L):
    sum = 0
    i = 0
    while i < len(L):
        i += 1
        sum += L[i−1]
    return sum
```

Summary

Summary

- "Immediate cause" vs. "root cause" of a bug:
  - immediate cause: the problem you observe
  - root cause: the problem you infer as giving rise to it
- Broadly speaking, bugs can be of two types:
  - *control related*: incorrect execution order of statements
  - *data related*: incorrect values computed for expressions
- Often,* control-related bugs arise from incorrectly computed values
- Locating a bug involves working back (iteratively) from the observed immediate cause to the ultimate root cause

* not always