

TESTING & DEBUGGING

(download slides and .py files to follow along)

6.1000 LECTURE 5

Tim Kraska, MIT EECS

“Debugging: Being the detective in a crime movie where you are also the murderer.”
— Unknown

Announcements

- Pset 1 is due at 10 pm, checkoffs start tomorrow (Thu) in office hours and go until next Wed
- No office hours this Friday
- No finger exercise today

Last Lecture:

Types of Problems with Code

- **Syntax:** program has **no meaning**, won't run

fix syntax error (line number given)

- **Crashes:** program has **meaning** but **invalid** at some point

- converting string '1' to an integer is valid, but converting string 'abc' to integer is an invalid operation

exceptions & assertions

- **Returns wrong answer:** valid **meaning** throughout, not what you meant

- we saw a lot of those examples in the mutability lecture

testing & debugging (today)

- **Runs forever:** (likely) ditto

testing & debugging (today)

TESTING

Black and Glass Box Testing

- **Black box testing**

- Based on the task **specification**:
(without looking at the code)
~Try to test all possible types of inputs



- **Glass box testing**

- Based on knowledge of the **code**:
~Try to test all parts of the code



Black and Glass Box Testing

- **Black box testing**
 - explore paths through **specification** (without looking at the code)
- **Glass box testing**
 - explore paths through **code**



Black Box Testing



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

- testing based on **specification** (as described by docstring)
- designed **without looking** at the code
- exploring **paths** through specification
 - build test cases that cover different parts of the specification
 - think about boundary conditions (empty lists, singleton list, large numbers, small numbers)
- + can be done by someone other than the programmer to avoid programmer **biases**
- + testing can be **reused** if implementation changes

Black Box Testing: Boundary Cases



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

CASE	x	eps
perfect square	25	0.0001
less than 1	0.05	0.0001
irrational square root	2	0.0001

} cases from
problem domain

Black Box Testing: Boundary Cases



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

CASE	x	eps
boundary	0	0.0001
perfect square	25	0.0001
less than 1	0.05	0.0001
irrational square root	2	0.0001

} edge case

} cases from
problem domain

Black Box Testing: Boundary Cases



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

CASE	x	eps	}	edge case
boundary	0	0.0001		
perfect square	25	0.0001		
less than 1	0.05	0.0001		
irrational square root	2	0.0001	}	cases from problem domain
extremes	2	1.0/2.0**64.0		
extremes	1.0/2.0**64.0	1.0/2.0**64.0	}	extreme values for parameters
extremes	2.0**64.0	1.0/2.0**64.0		
extremes	1.0/2.0**64.0	2.0**64.0		
extremes	2.0**64.0	2.0**64.0		

Glass Box Testing



- **Use code** directly to guide design of test cases
- **Path-complete** if every potential path through code is tested at least once
 - limitations:
 - **cannot test all paths** (loops and recursion)
 - doesn't show **missing paths** that should be included in your code but are not there

- Rules of thumb

- Branches

- For loops

- While loops

- Recursion (next lecture)

exercise all parts of a conditional

loop not entered

body of loop executed exactly once

body of loop executed more than once

same as for loops, cases that catch all ways to exit loop

similar to loops

Glass Box Testing



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

- **path-complete test suite** according to glass-box testing requirements:
 - negative number -2 (for if-branch)
 - positive number: 2 (for else-branch)

Glass Box Testing



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

- **path-complete test suite** according to glass-box testing requirements:
 - negative number -2 (for if-branch)
 - positive number: 2 (for else-branch)
- but **testing each code path was not sufficient** in this example since `abs(-1)` incorrectly returns -1
- therefore, **combine glass box testing with black box testing**

In the Unlikely Event Your Code Fails a Test



Debugging for Beginners

FRÉDO DURAND, MIT EECS & CSAIL



Who has ever had bugs?

Uplifting halftime coach speech

Even the best programmers create silly bugs

Everyone can debug

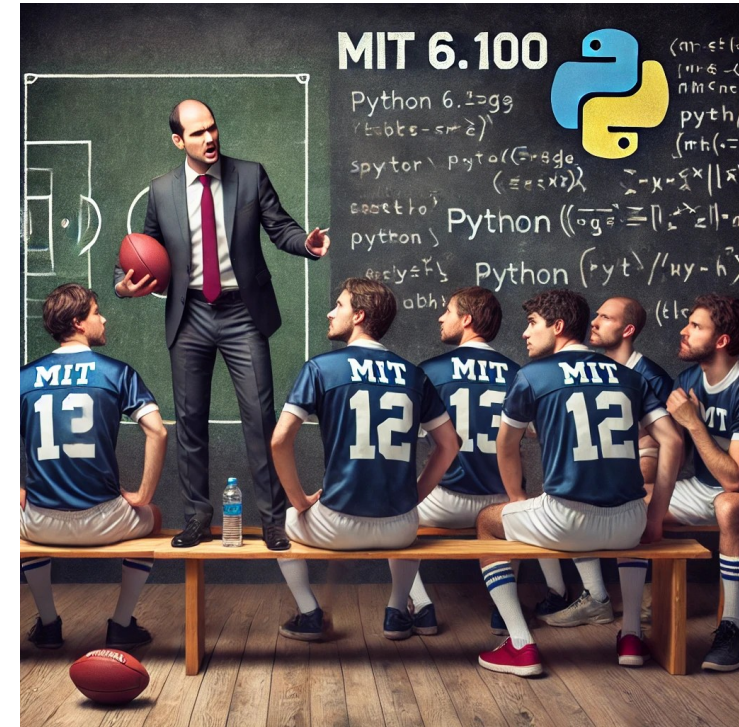
Exciting paradigmatic case of problem solving

Combination of process/tools and intuition/grit/smarts

But 97% of bugs can be fixed with 4 simple tools:

- **Read the error message**
- **PRINT**
- **Smart inputs**
- **Thinking out loud**

Most of it is simple, if a bit tedious



But remember that 93% of numbers are made up

Basic debugging

Leverage error messages

- read it, go where it says, google it

Understand your code's execution using PRINT

- Print info about both values and control flow (code location)
- Run method manually in parallel when possible
- Also for ideas/method: run simple examples w/ pen & paper

Find smart inputs

Think out loud

Why print debugging?

- Bugs could be due to problems at different levels of abstraction and understanding
 - error in understanding the goals, the logic of the solution, in translating the logic into code, having the high-level logic implemented correctly but having small details wrong, errors in understanding how a language or module works, etc.
- Abstract solution or code are hard to think about, and our mental models/understanding can be wrong
- Whereas one sequential execution of the code is concrete and can be followed step by step.
 - => That's what you should focus on
- We need a way to make the execution visible to us
 - Which parts of the code get executed in what order
 - What values are
- One simple solution: PRINT
- It's so useful that you should do some printing before you are aware of bugs

What to print?

Variable values

- Start with “important” ones, after you compute/update them

- But you may need to add trivial ones

- For mutable types, print even if you think they haven’t changed

Location in code

- Iteration number

- Branch of “if”

Reading the printouts

Printing is not enough, you need to read!

Not necessarily read everything, especially with loops

Sometimes just the beginning or end suffice

True, sometimes it's tedious

But tedious is better than daunting

Ideally verify values manually

Actually fixing the bug

Fix it (but keep track of change)

Test that it is actually fixed

Else check if changes were possibly useful

If not, undo them or comment them out

And keep debugging

Did it break something else?

(regression testing)

Are there similar bugs?

Optional: Remove / deactivate debugging code

Advanced: Version control (save current version)



If Lucky, a Helpful Error Message

- Trying to access beyond the limits of a list

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

Use type of error
to guide search

- Trying to convert an inappropriate type

```
int('test') → TypeError
```

- Referencing a non-existent variable

```
a → NameError
```

Python points to
location in code where
error occurred

- Mixing data types without appropriate coercion

```
'3' / 4 → TypeError
```

- Forgetting to close parenthesis, quotation, etc.

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

Print debugging

Square root with binary search

```
1  x = 4
2  epsilon = 0.01
3  low = 0
4  high = x
5  guess = (low + high) / 2
6
7  while error >= epsilon:
8
9      if guess**2 < x:
10         low = guess
11     else:
12         high = guess
13     guess = (low + high) / 2
14     error = guess**2 - x
15
16     print(f"{guess} is close to square root of {x:,}")
17
```


Leverage the error message

```
Traceback (most recent call last):  
  File "/Users/fredodurand/MIT Dropbox/Fredo Durand/6.100-lecture-materials/2025b-spring/debugging [fredo]/debug.py", line 7, in <module>  
    while error >= epsilon:  
          ^^^^^  
NameError: name 'error' is not defined. Did you mean: 'OSError'?
```

```
while error >= epsilon:  
      ^^^^^
```

line 7,

```
NameError: name 'error' is not defined.
```

```
1  x = 4
2  epsilon = 0.01
3  low = 0
4  high = x
5  guess = (low + high) / 2
6
7  while error >= epsilon:
8
9      if guess**2 < x:
10         low = guess
11     else:
12         high = guess
13     guess = (low + high) / 2
14     error = guess**2 - x
15
16     print(f"{guess} is close to square root of {x:,.}")
17
```

line 7,

```
while error >= epsilon:
    ^^^^^
```

NameError: name 'error' is not defined.

```
1  x = 4
2  epsilon = 0.01
3  low = 0
4  high = x
5  guess = (low + high) / 2
6  error = guess**2 - x
7
8  while error >= epsilon:
9
10     if guess**2 < x:
11         low = guess
12     else:
13         high = guess
14     guess = (low + high) / 2
15     error = guess**2 - x
16
17  print(f"{guess} is close to square root of {x:,}")
18
```

2.0 is close to square root of 4



Keep testing with more inputs

```
1  x = 400
2  epsilon = 0.01
3  low = 0
4  high = x
5  guess = (low + high) / 2
6  error = guess**2 - x
7
8  while error >= epsilon:
9
10     if guess**2 < x:
11         low = guess
12     else:
13         high = guess
14     guess = (low + high) / 2
15     error = guess**2 - x
16
17  print(f"{guess} is close to square root of {x:,}")
```

12.5 is close to square root of 400



Print debugging

```
1  x = 400
2  epsilon = 0.01
3  low = 0
4  high = x
5  guess = (low + high) / 2
6  error = guess**2 - x
7
8  while error >= epsilon:
9
10     if guess**2 < x:
11         low = guess
12     else:
13         high = guess
14     guess = (low + high) / 2
15     error = guess**2 - x
16
17  print(f"{guess} is close to square root of {x:,}")
```

Print debugging

Aka printf debugging

```
1  x = 400
2  epsilon = 0.01
3  low = 0
4  high = x
5  guess = (low + high) / 2
6  error = guess**2 - x
7  number_of_guesses = 0
8
9  while error >= epsilon:
10
11     if guess**2 < x:
12         low = guess
13     else:
14         high = guess
15         guess = (low + high) / 2
16         error = guess**2 - x
17         number_of_guesses += 1
18     print(f"iteration : {number_of_guesses} has guess: {guess} with error {error}")
19
20     if error >= epsilon:
21         print(f"Failed on square root of {x}")
22         print(f"The last guess for square root of {x} was {guess} with an error of {error:.4f} (acceptable error: {epsilon}) after {number_of_guesses}")
23     else:
24         print(f"{guess} is close to square root of {x}, with an error of {error:.4f} (acceptable error: {epsilon}) after {number_of_guesses}, guess")
25
```

*I could also print the square of guess
Maybe print low and high
Deciding what to print is part of the elusive art*

Where we are & control flow

Values


```
iteration : 1 has guess: 100.0 with error 9600.0
iteration : 2 has guess: 50.0 with error 2100.0
iteration : 3 has guess: 25.0 with error 225.0
iteration : 4 has guess: 12.5 with error -243.75
12.5 is close to square root of 400 with an error of -243.7500 (acceptable error: 0.01) after 4 guesses.
(base) frededurand@31-37-80 debugging [fredel %]
```

```
1  x = 400
2  epsilon = 0.01
3  low = 0
4  high = x
5  guess = (low + high) / 2
6  error = guess**2 - x
7  number_of_guesses = 0
8
9  while error >= epsilon:
10
11     if guess**2 < x:
12         low = guess
13     else:
14         high = guess
15     guess = (low + high) / 2
16     error = abs(guess**2 - x)
17     number_of_guesses += 1
18     print(f"iteration : {number_of_guesses} has guess: {guess} with error {error}")
19
20 if error >= epsilon:
21     print(f"Failed on square root of {x}")
22     print(f"The last guess for square root of {x} was {guess} with an error of {error:.4f} (acceptable error: {epsilon}) after {number_of_guesses}")
23 else:
24     print(f"{guess} is close to square root of {x}, with an error of {error:.4f} (acceptable error: {epsilon}) after {number_of_guesses:,} guesses")
25
```

20.000076293945312 is close to square root of 400 with an error of 0.0031 (acceptable error: 0.01) after 19 guesses.



Are there similar bugs?

```
1  x = 400
2  epsilon = 0.01
3  low = 0
4  high = x
5  guess = (low + high) / 2
6  error = guess**2 - x
7  number_of_guesses = 0
8
9  while error >= epsilon:
10
11     if guess**2 < x:
12         low = guess
13     else:
14         high = guess
15     guess = (low + high) / 2
16     error = abs(guess**2 - x)
17     number_of_guesses += 1
18     print(f"iteration : {number_of_guesses} has guess: {guess} with error {error}")
19
20 if error >= epsilon:
21     print(f"Failed on square root of {x}")
22     print(f"The last guess for square root of {x} was {guess} with an error of {error:.4f} (acceptable error: {epsilon}) after {number_of_guesses}")
23 else:
24     print(f"{guess} is close to square root of {x}, with an error of {error:.4f} (acceptable error: {epsilon}) after {number_of_guesses:,} guesses")
25
```

Actually fixing the bug

Fix it (but keep track of change)

Test that it is actually *fixed*

Else check if changes were possibly useful

If not, undo them or comment them out

And keep debugging

Did it break something else?

(regression testing)

Are there similar bugs?

Optional: Remove / deactivate debugging code

Advanced: Version control (save current version)




Are there similar bugs?

```
1  x = 400
2  epsilon = 0.01
3  low = 0
4  high = x
5  guess = (low + high) / 2
6  error = guess**2 - x
7  number_of_guesses = 0
8
9  while error >= epsilon:
10
11     if guess**2 < x:
12         low = guess
13     else:
14         high = guess
15     guess = (low + high) / 2
16     error = abs(guess**2 - x)
17     number_of_guesses += 1
18     print(f"iteration : {number_of_guesses} has guess: {guess} with error {error}")
19
20 if error >= epsilon:
21     print(f"Failed on square root of {x}")
22     print(f"The last guess for square root of {x} was {guess} with an error of {error:.4f} (acceptable error: {epsilon}) after {number_of_guesses}")
23 else:
24     print(f"{guess} is close to square root of {x}, with an error of {error:.4f} (acceptable error: {epsilon}) after {number_of_guesses:,} guesses")
25
```

Actually fixing the bug

- Fix it (but keep track of change)
- Test that it is actually *fixed*
 - Else check if changes were possibly useful
 - If not, undo them or comment them out
 - And keep debugging
- Did it break something else?
 - (regression testing)
- Are there similar bugs?
- Optional: Remove / deactivate debugging code
- Advanced: Version control (save current version)



```

1  x = 400
2  epsilon = 0.01
3  low = 0
4  high = x
5  guess = (low + high) / 2
6  error = abs(guess**2 - x)
7  number_of_guesses = 0
8
9  while error >= epsilon:
10
11     if guess**2 < x:
12         low = guess
13     else:
14         high = guess
15     guess = (low + high) / 2
16     error = abs(guess**2 - x)
17     number_of_guesses += 1
18     print(f"iteration : {number_of_guesses} has guess: {guess} with error {error}")
19
20 if error >= epsilon:
21     print(f"Failed on square root of {x}")
22     print(f"The last guess for square root of {x} was {guess} with an error of {error:.4f} (acceptable error: {epsilon}) after {number_of_guesses:,}")
23 else:
24     print(f"{guess} is close to square root of {x:,} with an error of {error:.4f} (acceptable error: {epsilon}) after {number_of_guesses:,} guesses."
25

```

Cleaning up

Remove excessive print (delete, comment, if debug_mode)

You can leave some just in case as long as they don't overwhelm you

```
1 x = 400
2 epsilon = 0.01
3 low = 0
4 high = x
5 guess = (low + high) / 2
6 error = abs(guess**2 - x)
7 number_of_guesses = 0
8
9 while error >= epsilon:
10
11     if guess**2 < x:
12         low = guess
13     else:
14         high = guess
15     guess = (low + high) / 2
16     error = abs(guess**2 - x)
17     number_of_guesses += 1
```

```
18 #print(f"iteration : {number_of_guesses} has guess: {guess} with error {error}")
19
```

```
20 if error >= epsilon:
21     print(f"Failed on square root of {x}")
22     print(f"The last guess for square root of {x} was {guess} with an error of {error:.4f} (acceptable error: {epsilon}) after {number_of_guesses:,}")
23 else:
24     print(f"{guess} is close to square root of {x:,} with an error of {error:.4f} (acceptable error: {epsilon}) after {number_of_guesses:,} guesses.")
25
```

Actually fixing the bug

Fix it (but keep track of change)

Test that it is actually fixed

Else check if changes were possibly useful

If not, undo them or comment them out

And keep debugging

Did it break something else?

(regression testing)

Are there similar bugs?

Optional: Remove / deactivate debugging code

Advanced: Version control (save current version)



Run more tests

... (not shown here)

Basic simple inputs: 0, 4

Some normal/average/random values

Adversarial edge cases: 0, 0.5, 1, -4

Edge cases can overlap with simple/sanity check

Five-minute break



Trying to fix my code

“Only half of programming is coding. The other 90% is debugging.”
anonymous

Another Example

```
def is_palindrome(x):  
    temp = x  
    temp.reverse  
    if temp == x:  
        return True  
    else:  
        return False
```

```
print(is_palindrome(list('abcba')))  
print(is_palindrome(list('palinnilap')))  
print(is_palindrome(list('ab')))
```

Palindrome: a sequence that reads the same forward and backwards.

Should test with examples of palindromes and non-palindromes

returns True

returns True

returns True

???

Bisection Search for Bug(s)

```
def is_palindrome(x):  
    temp = x  
    temp.reverse  
    print(temp, x)  
    if temp == x:  
        return True  
    else:  
        return False
```

find location to print intermediate values
after the bug has most likely occurred

Printed output:

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

Problem: both are the same
and temp is not reversed

```
print(is_palindrome(list('ab')))
```

test with example that
caused the bug.

Bisection Search, cont.

```
def is_palindrome(x):
```

```
    temp = x
```

```
    print('before reverse', temp, x)
```

```
    temp.reverse
```

```
    print('after reverse', temp, x)
```

```
    if temp == x:
```

```
        return True
```

```
    else:
```

```
        return False
```

```
print(is_palindrome(list('ab')))
```

Expect temp and x to have same value



We expect temp and x to be reverses of each other

Printed output:

```
before reverse ['a', 'b'] ['a', 'b']  
after reverse ['a', 'b'] ['a', 'b']  
_
```

Looks good before reverse
but reverse is not working.

Trying Again

```
def is_palindrome(x):
```

```
    temp = x
```

Expect temp and x to have same value

```
    print('before reverse', temp, x)
```

```
    temp.reverse()
```

```
    print('after reverse', temp, x)
```

```
    if temp == x:
```

```
        return True
```

```
    else:
```

```
        return False
```

We expect temp and x to be reverses of each other

Printed output:

```
before reverse ['a', 'b'] ['a', 'b']
after reverse ['b', 'a'] ['b', 'a']
```

```
print(is_palindrome(list('ab')))
```

Reversing the list is working now but both lists are reversed, which is not what we wanted.

Two Bugs Down

```
def is_palindrome(x):  
    temp = x.copy()  
    print('before reverse', temp, x)  
    temp.reverse()  
    print('after reverse', temp, x)  
    if temp == x:  
        return True  
    else:  
        return False
```



Printed output:

```
before reverse ['a', 'b'] ['a', 'b']  
after reverse ['b', 'a'] ['a', 'b']
```

This looks correct!



```
print(is_palindrome(list('ab')))
```

Some Pragmatic Advice

- look for (your) usual suspects
 - e.g., alias versus clone in list
- ask why the code is doing **what it is**, not why it is not doing **what you want**
- the bug is probably **not** where you think it is – eliminate locations – bisection search helps do this
- For mutations, print values even when you don't have changed them
- explain the problem to someone else
- don't believe the documentation
- take a break and come back to the bug later



Debugging as Search

- Want to **narrow down** space of possible sources of error
- Design experiments that **expose intermediate stages of computation (use print statements)**, and use results to further narrow search
- **Bisection search** can be a powerful tool for this

In short: don't be a deer in headlights!

1/ Leverage error messages

Read them, go to the location, and google if you don't understand

2/ Print

Print intermediate values

Here and there print where you are

(entering function XXX ; in else branch of YYY, etc.)

Print mutable types even if you think they haven't changed

Ideally compute solution in parallel on pen and paper

3/ Think of smart input values

4/ step back, think out loud

Talk to a rubber ducky

Do you have all the tools?

1/ Leverage error messages

Read them, go to the location, and go

2/ Print

Print intermediate values

Here and there print where you are

(entering function XXX ; in else)

Print mutable types even if you think they are not

Ideally compute solution in parallel on multiple cores

3/ Think of smart input values

4/ step back, think out loud

Is there more???



More advanced debugging tools

<https://pythontutor.com>

Step by step execution

Visualization of variables, environments

But only for small programs

Python Tutor: Visualize Code and Get AI Help for [Python](#), [JavaScript](#), [C](#), [C++](#), and [Java](#)

Python 3.11
[known limitations](#)

```
1 def is_palindrome(x):
2     temp = x
3     temp.reverse()
4     if temp == x:
5         return True
6     else:
7         return False
8
9 print(is_palindrome(list('abcba')))
10 print(is_palindrome(list('palinnilap')))
11 print(is_palindrome(list('ab')))
```

[Edit this code](#)

→ line that just executed
→ next line to execute

Print output (drag lower right corner to resize)

Frames

Global frame

is_palindrome

Objects

function is_palindrome(x)

list

0	1	2	3	4
"a"	"b"	"c"	"b"	"a"

is_palindrome

x

temp

<< First < Prev Next > Last >>

Step 7 of 22

Asserts

```
def binary_search_char(text: str, char: str) -> int:
```

```
    lower = 0
```

```
    upper = len(text) - 1
```

```
    while upper - lower > 1:
```

```
        assert lower <= upper, "Lower bound is greater than upper bound"
```

```
        assert 0 <= lower < len(text), "Lower index out of bounds"
```

```
        assert 0 <= upper < len(text), "Upper index out of bounds"
```

```
        mid = (lower + upper) // 2
```

```
        if text[mid] <= char:
```

```
            lower = lower + mid
```

```
        else:
```

```
            upper = upper - mid
```

```
    if text[lower] == char:
```

```
        return lower
```

```
    if text[upper] == char:
```

```
        return upper
```

```
    return -1
```

Declare
invariants which
must hold true

```
text = "abcdefffg"
```

```
char = "f"
```

```
index = binary_search_char(text, char)
```

```
print(index)
```

Logging

Setup a logging library

```
import logging
logger = logging.getLogger(__name__)
logging.basicConfig(
    filename="debug.log",  # this is the file where logs will be stored
    filemode="a",          # "a" = append, "w" = overwrite
    level=logging.DEBUG,    # log level
    format="%(asctime)s - %(levelname)s - %(filename)s:%(lineno)d - %(message)s"
)
```

Syntax not important for now. We cover objects later

Using the logging function

```
def is_palindrome(x):
    temp = x
    logging.debug("before reverse: temp=%s, x=%s", temp, x)
    temp.reverse()
    logging.debug("after reverse: temp=%s, x=%s", temp, x)
    if temp == x:
        return True
    else:
        return False
```

Called print substitution.
You can also do it with strings

```
print("before reverse: temp=%s, x=%s" % (temp, x))
```

Basic cheat sheet for debugging

Actionable steps

Leverage error message when possible,

- read it, go where it says, google it

Understand your code's execution using print

- Print info about both values and control flow (code location)
- Read what you have printed (tedious but easy)
- Be smart about what to print but err on the side of more info
- Run method manually in parallel when possible
- Think backward: How did we get there/this value?
- Understand built-in & external functions. Test them.
- Alternative: use Online PythonTutor

Find smart inputs

- Simple sanity checks easy to run manually
- Average case
- Adversarial edge cases
- Random
- Keep a systematic list of tests you run systematically

Think out loud

To a friend, to a rubber ducky, write it down

High level advice

You can do it!

- Avoid deer in headlights syndrome!

Keep it concrete.

- Focus on execution & examples, not code or ideas

Step back

- Take a break
- Question your focus and assumptions

Test and debug early & often

Keep track of what you are doing

- take note
- scopy files/code
- make changes removable

Fix/change one thing at a time.

(Slightly) More advanced approaches

Narrow down where errors may be:

- Binary search:
see if value in middle of code is correct to see if bug is before or after
keep splitting in two

Pattern matching & adaptive focus for common bugs and behaviors

- E.g. code runs forever -> focus on loops
- Array bound issues, off by one, first last case
- Your bank of patterns and intuition will sharpen as you learn

Defensive programming and assert

- add asserts so code crashes when a condition is wrong

Modify code

- Start from last version that worked
- Create minimal version of problem
- Call/test parts in isolation

More advanced: Use debugger (I rarely do)

Scientific method perspective:

Exploratory/observational experiments

- Mostly previous slide

Form Hypotheses about causes for bug

Come up with & Run Verification experiments

- Smart inputs
- smart measurements/print

Take notes about everything

Kuhn's structure of scientific revolutions

- Ordinary science, verify current paradigm (testing before bug occurs)
- Crisis: a bug occurs
- New paradigm at first may lead to worse predictions
- Eventually angles get smoothed and it all restarts

Links about debugging

- <https://blog.hartleybrody.com/debugging-code-beginner/>
- <https://andypi.co.uk/2024/01/26/concise-guide-to-debugging-anything-cheat-sheet/>
- <https://jvns.ca/blog/2019/06/23/a-few-debugging-resources/>
- <https://blog.regehr.org/archives/199>
- <https://wizardzines.com/zines/debugging-guide/>
- <https://greenteapress.com/thinkpython2/html/thinkpython2021.html>
- <https://www.freecodecamp.org/news/what-is-debugging-how-to-debug-code/>
- <https://blog.stackademic.com/the-ultimate-cheat-sheet-for-debugging-like-a-pro-a4488f7cacee>
- <https://medium.com/swlh/a-beginners-guide-to-debugging-for-beginners-21eb119a8445>

Repeat after me with conviction

I am not a deer in headlights



Extra material

Jokes

From <https://blog.stackademic.com/the-ultimate-cheat-sheet-for-debugging-like-a-pro-a4488f7cacee>

To lighten the mood, here are some funny debugging quotes:

“The first rule of programming is: If it works, it’s not done yet.” — Unknown

“I have not failed. I’ve just found 10,000 ways that won’t work.” – Thomas Edison

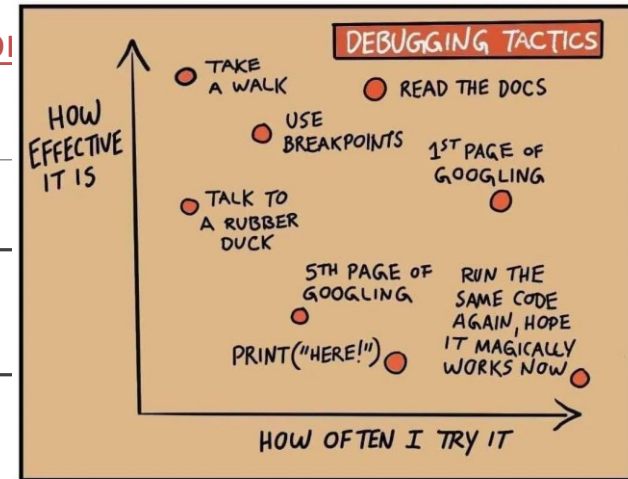
From <https://thepythoncodingbook.com/errors-and-bugs/>


Only half of programming is coding. The other 90% is debugging.

anonymous

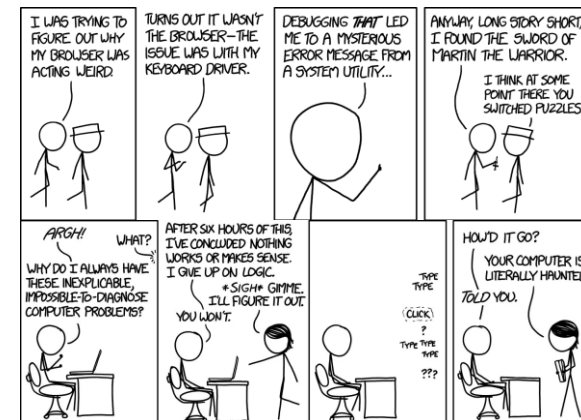
Sometimes it pays to stay in bed on Monday rather than spending the rest of the week debugging Monday's code.

Dan Solomon/Christopher Thompson



Posted in [Programmer Humor](#) by [U/DemoBeast](#)  [reddit](#)

[simplest-debugging-strategy-that-many-beginners-ignore-ce14e98edb2e](#)



Decent links

<https://blog.hartleybrody.com/debugging-code-beginner/>

<https://andypi.co.uk/2024/01/26/concise-guide-to-debugging-anything-cheat-sheet/>

<https://jvns.ca/blog/2019/06/23/a-few-debugging-resources/>

<https://blog.regehr.org/archives/199>

<https://wizardzines.com/zines/debugging-guide/>

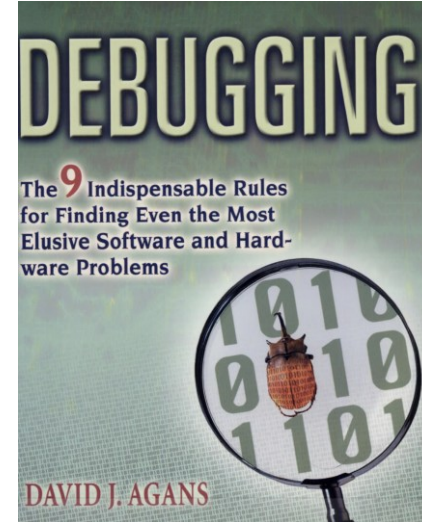
<https://greenteapress.com/thinkpython2/html/thinkpython2021.html>

<https://www.freecodecamp.org/news/what-is-debugging-how-to-debug-code/>

<https://blog.stackademic.com/the-ultimate-cheat-sheet-for-debugging-like-a-pro-a4488f7cacee>

<https://medium.com/swlh/a-beginners-guide-to-debugging-for-beginners-21eb119a8445>

1. Understand the system
2. Make it fail
3. Quit thinking and look
4. Divide and conquer
5. Change one thing at a time
6. Keep an audit trail
7. Check the plug
8. Get a fresh view
9. If you didn't fix it, it ain't fixed.



<https://www.amazon.com/Debugging-Indispensable-Software-Hardware-Problems/dp/0814474578>

① first steps

preserve the crime scene.....	9
read the error message.....	10
reread the error message.....	11
reproduce the bug.....	12
inspect unreproducible bugs.....	13
identify one small question.....	14
retrace the code's steps.....	15
write a failing test.....	16

② get organized

brainstorm some suspects.....	18
rule things out.....	19
keep a log book.....	20
draw a diagram.....	21

③ investigate

add lots of print statements.....	23
use a debugger.....	24
jump into a REPL.....	25
find a version that works.....	26
look at recent changes.....	27
add assertions everywhere.....	28
comment out code.....	29
analyze the logs.....	30

④ research

read the docs.....	32
find the type of bug.....	33
learn one small thing.....	34
read the library's code.....	35
find a new source of info.....	36

⑤ simplify

write a tiny program.....	38
one thing at a time.....	39
tidy up your code.....	40
delete the buggy code.....	41
reduce randomness.....	42

⑥ get unstuck

take a break.....	44
investigate the bug together.....	45
timebox your investigation.....	46
write a message asking for help.....	47
explain the bug out loud.....	48
make sure your code is running.....	49
do the annoying thing.....	50

⑦ improve your toolkit

try out a new tool.....	52
types of debugging tools.....	53
shorten your feedback loop.....	54
add pretty printing.....	55
colours, graphs, and sounds.....	56

⑧ after it's fixed

do a victory lap.....	58
tell a friend what you learned.....	59
find related bugs.....	60
add a comment.....	61
document your quest.....	62

<https://wizardzines.com/zines/debugging-guide/>

Concise Guide to Debugging Anything (1)



Example: A Python script that processes user data from a CSV file and stores results in a database.



1. Understand the System (Read the code, documentation or ask)

- Thoroughly read the system's documentation and consult with experts to understand its intended functionality.
- Carefully step through the code, understanding what each part is supposed to do. When in doubt, don't guess – refer back to the documentation or ask for clarification.

Read your script line by line, understand how Python's CSV module works, and know how the script interacts with the database. If unsure about a function, you look it up in the Python documentation.



2. Replicate the Failure (Observe and record the bug occurring again)

- Replicate the failure under the exact conditions it occurred, not just similar ones.
- For intermittent failures, vary the conditions until the issue can be consistently replicated. Document every detail, including any seemingly irrelevant ones.

Your script fails to process certain rows in the CSV file. To replicate this, you ensure the testing environment mirrors production with the same CSV file, Python version, and database setup. You run the script and observe it failing on the same rows, confirming that you've replicated the issue.



3. Search and Identify the Bug (Don't theorize without evidence)

- Engage in a thorough search to identify the exact cause of the bug. Rely on debug logging to make the bug and its cause visible.
- Avoid theorizing about potential causes without evidence - do a practical search.
- Remember the debug process might inadvertently modify the conditions and hide the failure.

You add print() statements or use Python's logging module to log the data processed at each step. You notice that the script fails when encountering special characters. By logging the exact input processed at the failure point, you identify that the script doesn't handle Unicode characters correctly.



4. Narrow the Search (Divide & Conquer or Successive Approximation)

- Apply the divide & conquer algorithm to narrow down the search area. Understand the range of the search and determine if the bug is upstream or downstream from the current point.
- Ideally, start at a known problematic point and work back up through the system, checking each branch until finding the source of the bug.

Your script is a multi-step process, and you're unsure where it's failing. You add checkpoints at the halfway point of each major section. If the script fails before reaching the midpoint of a particular section, you know the issue is upstream. This helps you isolate the problematic code block quickly.



5. Change One Thing at a Time (Control all other variables)

- When attempting to replicate the failure or identify the bug, change only one variable at a time, changing back any variables to the original condition before the next test.
- Keep a forensic mindset, analyzing what has changed since the last time the system worked correctly.

You suspect the failure might be due to the Python version or the CSV file format. First, you change only the Python version while keeping the same file to see if the issue persists. Then, you revert to the original version and try a different CSV file format. This controlled approach helps identify the exact cause.

Concise Guide to Debugging Anything (2)



6. Keep an Audit Trail (Write down details of the debugging process)

- Keep a detailed audit trail of all actions taken, the order in which they were done, and the results of each action.
- This record is invaluable in ensuring that all areas have been checked, in providing a clear account of your debugging process to others, and serving as a reference for future issues

You maintain a detailed log file using Python's logging module. Each action, such as opening a file, processing a row, or making a database entry, is logged with a timestamp. When the script fails, you have a comprehensive record of what happened immediately before the failure. You explain the bug fix in a git commit message.



7. Check Obvious Assumptions (that are fast to verify)

- Always verify the most basic assumptions first, such as whether the system is powered on, the service is running or expired data is cached.
- Before narrowing your search, confirm that the entire scope of the system is being checked, including all tools, dependencies and platforms you're working with.

Before diving deep into debugging, you check the basics: Is the CSV file present in the expected folder? Is the network connection to the database dropping? Is the database server running? You add checks in your script like os.path.isfile() to verify the file's existence before proceeding.



8. Ask for Help (from online resources and then experts)

- When you've hit a wall, a fresh perspective from others can shed new light on the problem.
- Start with async online resources like StackOverflow, ChatGPT before asking experts (e.g. Github issues) which introduces delay
- Always provide a bug description, logs, errors and what has been checked so far, etc.

If you fail to solve the issue independently, you write a detailed question on ChatGPT. You include the Python version, a snippet of your code, the exact error message, and what you've tried so far. ChatGPT lists a number of possible reasons for the failure, including one you haven't thought of, which gives you a new point to test.



9. Confirm the Fix Works (By testing with it applied and removed)

- Rigorously test the fix to confirm it addresses the issue. Then, remove the fix and retest to ensure that the issue reoccurs, confirming that your fix is directly resolving the problem.
- For particularly elusive bugs, add logging to capture details of the failure so it can be traced if it occurs again in production.

After adjusting your script to handle Unicode characters, you re-run it with the same problematic CSV file. It processes all rows successfully. Then, you remove the fix and confirm the script fails again, validating that your fix directly addresses the issue.



10. Fix the Underlying Process (Find the design or systemic problem)

- Reflect on the debugging process and identify any systemic improvements that could prevent similar issues in the future.
- For example, standardizing error and debug logs, enhancing test coverage, automating testing and deployment to minimize human error, and thoroughly understanding dependencies during the design phase.

To prevent similar issues, you decide to add more logging items and robust error handling to your script. You also incorporate a unit test that runs automatically on every git commit, testing a range of CSV formats and special cases to ensure the script is resilient to common data issues.

- **If they don't know how to get started, ask them to describe the problem in detail:**

- What are the goals of the problem?
- What are the inputs?
- What are the outputs?
- What is their relationship?
- Can we solve a small example by hand?
- Is there a part of the problem that they could write code for?
 - (and worry about the rest later?)
- Can you describe the algorithm in words?

<https://www.csteachingtips.org/tips-tutors>

- **If they have a syntax error, ask them:**

- What line is the syntax error is on?
- What does the text of the error mean?
- What does the internet suggest about how to fix this error?
- What have they tried to fix this error?



- **If their code doesn't work, ask them:**

- What evidence do we have that the code doesn't work?
- What test case doesn't work and what incorrect behavior or output results?
- Could we come up with a simpler example that demonstrates the error?
- What lines of code might be producing the bug?
- Why hypotheses do we have for what might be causing the problem?
- How can we test these hypotheses?
 - (e.g. writing new test cases, adding print statements, using a debugger)
- Could we walk through an example that doesn't work: by hand? with a debugger?

<https://blog.hartleybrody.com/debugging-code-beginner/>

- #1. Print things a lot
- #2. Start with code that already works
- #3. Run your code every time you make a small change
- #4. Read the error message
- #5. Google the error message
- #6. Guess and Check
- #7. Comment-out code
- #8. If you're not sure where the problem is, do a binary search
- #9. Take a break and walk away from the keyboard
- #10. How to ask for help

reproduce your bug (but how do you do that?)

reproduce your bug quickly

accept that it's probably your code's fault

start doing experiments

change one thing at a time

check your assumptions

weird methods to get information

write your code so it's easier to debug

A Scientific Approach to Debugging

1. Verify the Bug and Determine Correct Behavior
2. Stabilize, Isolate, and Minimize
3. Estimate a Probability Distribution for the Bug
4. Devise and Run an Experiment
5. Iterate Until the Bug is Found
6. Fix the Bug and Verify the Fix
7. Undo Changes
8. Create a Regression Test
9. Find the Bug's Friends and Relatives

What If You Get Stuck?

From Andrew Adams

Obviously you need to test your code to see if it works or not by running it on some inputs. What's less obvious is that you shouldn't just do this manually. You should write down those tests in a separate program so that you can rerun them quickly and easily, so that you don't break something that was already working without realizing when you make a change. Tests represent the ground you have gained and held in your war against the problem. Many students will already appreciate this. However, a second-order effect that I've come to appreciate more and more is that when you write a test you're often just writing down the cases you already had in mind while writing the code, so once you get good at coding they're likely to work already. That doesn't mean your code is correct though. Your code is still probably broken in cases you didn't think to test, and possibly wouldn't ever think to test. There are unknown unknowns. There is a trick though: maybe you can't think of the right test cases, but a random number generator can stumble upon them accidentally. So if at all possible, test your program on millions of random inputs. It will find bugs in code you thought was perfect. You don't need to use fancy fuzz-testing tools or frameworks. Just use a seeded random number generator to construct some random input, and make sure to log the seed (or the random input) so that you can reproduce any failure it finds. Sometimes I leave them running overnight. Highly recommended.