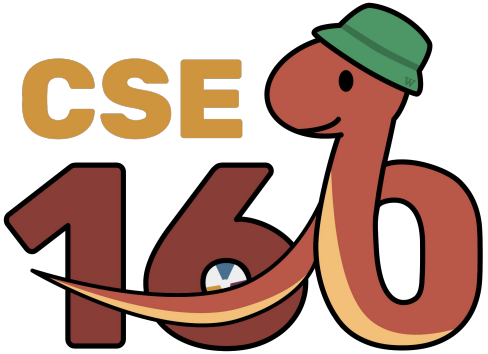


# Debugging

CSE 160 Summer 2026

Questions?

Sli.do #cse160

# Announcements

- **Homework Assignment 5 due Monday, August 17 at 11:59PM**
- **Programming Practice 7 due Sunday, August 16 at 11:59PM**
- **Resubmission Cycle 4 available later today!**
  - Homeworks 1-4 eligible
- **Final Exam information and Practice Finals** now available on course website!
  - Final Exam on Friday, August 21 at 9:40am in HRC 155

# Today's Roadmap

1. Three Types of Errors
2. Debugging Tools
3. Debugging Approaches
    a. The Scientific Method
    b. Partitioning

**Jupyter Hub**

# The Problem

- You want your program to do one thing
- Your program is doing something else!

How did you discover this problem?

- The code will not even run: Python complained about something
- One of my test cases failed

# Three Types of Errors

Jupyter Hub

- Syntax errors:
  - The program contains invalid python syntax and will not run
  - Maybe you forgot what the correct syntax is?
  - Example: `b0.py`
- Runtime errors:
  - The program runs, but it stops at some point due to an error.
  - Examples: `b1.py` & `b2.py`
- Logical errors:
  - The program runs, but does not give the results you want.
  - Examples: `b3.py` & `b4.py`

# Debugging Tools

- `assert` statements:  to confirm things you think are true in your program
- `print` statements: to see what values are/where you are in your program
- Understanding Python Error Messages
- Python Tutor:  for visualizing data structures
- Python Debuggers: Useful tools, but can have a steep learning curve.
- The most important tool:
  - Your Brain!

# Think Pair Share

Is there an error in this code? If so, what type of error is it?

A. Syntax Error

B. Runtime Error

C. Logical Error

D. There are no errors

```
# Sums the integers 0-9
total = 0
for i in range(10):
    total + i
print(total)
```

Sli.do

# Today's Roadmap

1. ~~Three Types of Errors~~
2. ~~Debugging Tools~~
3. Debugging Approaches
    a. The Scientific Method
    b. Partitioning

**Jupyter Hub**

# The Scientific Method

1. Create a hypothesis
2. Design an experiment to test that hypothesis
3. Understand the result of your experiment

Tips:

- Be systematic
- Never do anything if you don't have a reason
- Don’t just flail
    - Random guessing is likely to dig you into a deeper hole
- Don’t make assumptions (verify them)
- Take notes! Document what you have  tried and what the results were.

# Example experiments

- An alternate implementation of a function
  - Run all your test cases afterward
- A new, simpler test case
  - Examples:  smaller input, or test a function in isolation
  - Can help you understand the reason for a failure

# How to Read a Traceback

```
Traceback (most recent call last):
  File "/lec24/animals.py", line 19, in <module>
    print(dog(5, 6))
  File "/lec24/animals.py", line 16, in dog
    z = cat(x, y)
  File "/lec24/animals.py", line 12, in cat
    dub = 2 * bird(x, y)
  File "/lec24/animals.py", line 8, in bird
    five_more = 5 + fish(y, x)
  File "/lec24/animals.py", line 3, in fish
    for i in x:
TypeError: 'int' object is not iterable
```

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

Call stack or traceback

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

The error message

**Jupyter Hub**

# Common Python Error Types

- **AssertionError**
  - Raised when an assert statement fails.
- **IndexError**
  - Raised when a sequence (e.g. list, string, tuple) index is out of range.
- **KeyError**
  - Raised when a dictionary key is not found in the set of existing keys.
- **NameError**
  - Raised when a local or global name is not found.
- **SyntaxError**
  - Raised when the parser encounters a syntax error. (e.g. missing closing parenthesis)
- **IndentationError**
  - Syntax errors related to incorrect indentation.
- **TypeError**

# Today's Roadmap

1. ~~Three Types of Errors~~
2. ~~Debugging Tools~~
3. Debugging Approaches
    a. ~~The Scientific Method~~
        - ~~Tracebacks & Common Python Errors~~
    b. Partitioning

**Jupyter Hub**

# Partitioning

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

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

- 4 ways to Partition
  - In the program code
  - In test cases
  - During the program execution
  - During the development history

# A. Partition the program code

- Localize the defect to **part of the program**
  - e.g., one function, or one part of a function
- Code that isn't executed cannot contain the defect
- Probably the defect is not in a Python library or CSE160 provided code

3 approaches:

- Test one function at a time
- Add assertions or print statements
  - The defect is executed before the failing assertion
- Split complex expressions into simpler ones
  - Example: Failure in:

```
result = find_friends(calc(graph.neighbors(user)))
```

Change it to:

```
nbors = graph.neighbors(user)
calc_nbors = calc(nbors)
result = find_friends(calc_nbors)
```

# B. Partition the Test Cases

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

# C. Partition the Execution Time

- A sequence of `print` statements is a record of the execution of your program
- The `print` statements let you see and search multiple moments in time
- `print` statements are a useful technique, in moderation
- Be disciplined
  - Too much output is overwhelming rather than informative
  - Remember the scientific method:  have a reason (a hypothesis to be tested) for each `print` statement

# D. Partition your Development History

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

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

# A metaphor about debugging

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

- You're lost in the woods.

Don't press on into unexplored territory -- go back the way you came!

- (and leave breadcrumbs!)