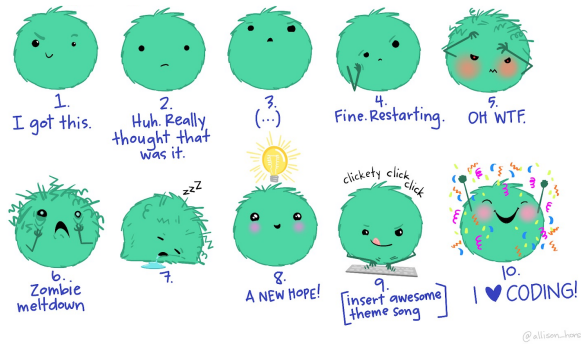


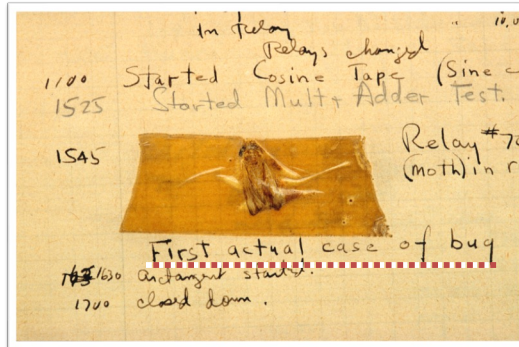
## Introduction to Debugging



University of Colorado  
Boulder

Artwork by [@allison\\_horst](#)

## The First Programming “Bug”



- Bugs == Errors
- The phrases (bug and debugging) are often attributed to Grace Hopper and her team – who found a literal bug (a moth) stuck in their computational machine (Mark II) – but the term is actually an age-old engineering term dating back over a century.
- In this workshop, we will discuss the process of debugging and the different techniques that you can use to better find and fix the bugs.

### Description (Smithsonian):

‘American engineers have been calling small flaws in machines “bugs” for over a century. Thomas Edison talked about bugs in electrical circuits in the 1870s. When the first computers were built during the early 1940s, people working on them found bugs in both the hardware of the machines and in the programs that ran them.

In 1947, engineers working on the Mark II computer at Harvard University found a moth stuck in one of the components. They taped the insect in their logbook and labeled it “first actual case of bug being found.” The words “bug” and “debug” soon became a standard part of the language of computer programmers.

Among those working on the Mark II in 1947 was mathematician and computer programmer Grace Hopper, who later became a Navy rear admiral. This log book was probably not Hopper's, but she and the rest of the Mark II team helped popularize the use of the term computer bug and the related phrase "debug."

Image & Description Source:

[https://americanhistory.si.edu/collections/nmah\\_334663](https://americanhistory.si.edu/collections/nmah_334663)

## Debugging Process



Observe a Bug

Before we can start debugging, we must first recognize the need for debugging – i.e. observe a bug.

In the context of software programming, a bug refers to a moment when a system behaves in an unexpected or unintended manner.

This “unexpected behavior” can run the gambit of incorrect output to the full-blown system crashes!

## Debugging Process



Observe a Bug



Find the Cause

After observing a bug, we must start searching for its underlying cause – which could be any one of a variety of potential errors.

It is important to note that the error, or errors, can exist across three spaces – the project's code, the system running the code, and then engineer's mind.

Today we'll be focusing on the first space, code, but it is important to always remember to check for issues in the system's hardware and your own understanding of the system and the code.

## Debugging Process



Observe a Bug



Find the Cause



Fix it!

Once the error, or errors, has been identified it must be fixed.

This is the final, and often easiest, step of the debugging process.

## Debugging Process



Very Challenging!!

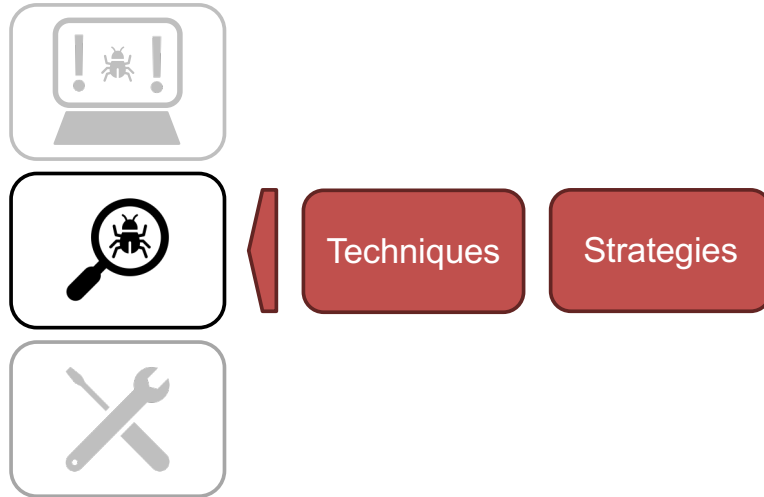


Research Computing  
UNIVERSITY OF COLORADO BOULDER

6

The most challenging part of debugging is this second step – finding the error.

## Debugging Process



In this workshop, we will focus on different debugging strategies and techniques that can make it easier for you to find software errors.

While this workshop will be taught with Python, it is important to know that these strategies and techniques can be easily transferred to other programming languages and engineering domains.



## Example Program

calculator.py

```
1 def calculate(operation, valueA, valueB):
2     result = 0
3
4     if operation == "multiply":
5         result = valueA * valueB
6
7     elif operation == "divide"
8         result = valueA / valueB
9
10    elif operation == "add":
11        result = valueA + valueA
12
13    elif operation == "subtract":
14        result = valueA - valueB
15
16    return result
```



Research Computing  
UNIVERSITY OF COLORADO BOULDER

8

The example program for this workshop is called “calculator.py” and contains a simple method for calculating basic arithmetic operation.

This method contains three errors which we will work on finding and fixing together.

You can follow along by copy+pasting the calculator.py’s code into a new file on your personal workstation, VS Code (OnDemand), or use the free online IDE Trinket for Python:

<https://trinket.io/python/cd7747ea1eaa>

This file is also provided in this presentations Github Repo:

[https://github.com/ResearchComputing/introduction\\_to\\_debugging\\_shortcours](https://github.com/ResearchComputing/introduction_to_debugging_shortcours)

e

## Error 1 – Syntax Error

calculator.py

```
6
7     elif operation == "divide"
8         result = valueA / valueB
9
```

Terminal

```
File "<file_path>/calculator.py", line 7
    elif operation == "divide"
                                ^
SyntaxError: expected ':'
```



Research Computing  
UNIVERSITY OF COLORADO BOULDER

9

## Error Message Breakdown

**Software Program**

**Line No.**

File "<file\_path>/calculator.py",

line 7

```
elif operation == "divide"
```

^

SyntaxError: expected ':'



Research Computing  
UNIVERSITY OF COLORADO BOULDER

10

## Error Message Breakdown

### Code Snippet

File "<file\_path>/calculator.py", line 7

```
elif operation == "divide"
```

```
SyntaxError: expected ':'
```

Hint



Research Computing  
UNIVERSITY OF COLORADO BOULDER

11

How to google Errors



## Internet Search

What to include:

1. Error Message - `SyntaxError: expected ':'`
2. Programming Language
3. Framework / Software Library





## Internet Search

What to include:

1. Error Message - `SyntaxError: expected ':'`
2. Programming Language
3. Framework / Software Library

Where to look:

1. StackOverflow
2. Github – Issues page for Framework / Software Library

## Error 2 – Wrong Value

calculator.py

```
16  
17     print(calculate("add", 3, 2))  
18
```

Terminal

6

**Expected Value: 5**



Research Computing  
UNIVERSITY OF COLORADO **BOULDER**

14

## Error 2 – Wrong Value

calculator.py

```
10 elif operation == "add":  
11     result = valueA + valueA  
...  
17 print(calculate("add", 3, 2))  
18
```

Terminal

6

**Expected Value: 5**




Research Computing  
UNIVERSITY OF COLORADO BOULDER

15



## Error 2 – Wrong Value

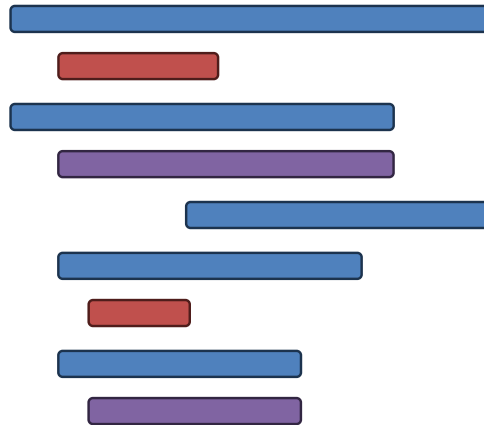
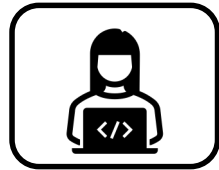
```
calculator.py
10 elif operation == "add":
11     result = valueA + valueA
...
17 print(calculate("add", 3, 2))
18
```



```
Terminal
6
```

Expected Value: 5

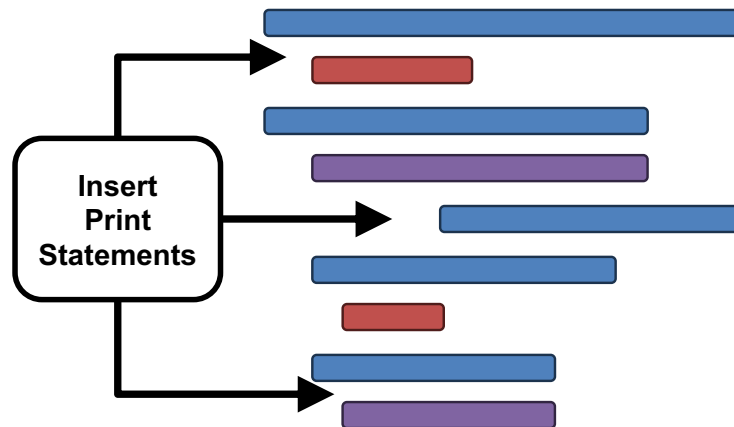
## Code Tracing



Mental Tracing – Reading the code line by line, helpful to read out-loud and/or add comments to explain what the system is doing.

Code Stepping w/ Print Statements – Use carefully placed print statements to observe the state of variables.

## Code Tracing



Mental Tracing – Reading the code line by line, helpful to read out-loud and/or add comments to explain what the system is doing.

Code Stepping w/ Print Statements – Use carefully placed print statements to observe the state of variables.

## Strategic Print Statements (1)

calculator.py

```
1 def calculate(operation, valueA, valueB):
2     print("calculate(",
3         "operation=",operation,
4         ",valueA=",valueA,
5         ",valueB=",valueB,")")
6
7
8     result = 0
9
10 ...
```

## Strategic Print Statements (2)

calculator.py

```
7  if operation == "multiply":
8      print("Selected multiply operation")
9      result = valueA * valueB
10
11  elif operation == "divide":
12      print("Selected divide operation")
13      result = valueA / valueB
14
15  elif operation == "add":
16      print("Selected add operation")
17      result = valueA + valueB
18
19  elif operation == "subtract" :
20      print("Selected subtract operation")
21      result = valueB - valueA
22  ...
```

## Strategic Print Statements (3)

calculator.py

```
22 ...  
23     print("Result of operation:", result)  
24     return result  
25  
26  
27 print(calculate("add", 3, 2))
```

Terminal

```
calculate(operation= add ,valueA= 3 ,valueB= 2 )  
Selected add operation  
Result of operation: 6
```



## Conditional Print Statements

calculator.py

```
1 def calculate(operation, valueA, valueB):
2     if DEBUG:
3         print("calculate(",
4             "operation=", operation,
5             ", valueA=", valueA,
6             ", valueB=", valueB, ")")
7
8     result = 0
9
10    ...
11
12    DEBUG = True
13
14    print(calculate("add", 3, 2))
15
```

22



Research Computing  
UNIVERSITY OF COLORADO BOULDER

Discuss benefits of different types of print statements/approaches

Show how we can make a DEBUG Boolean, which can turn debug statements on/off – great for long term coding projects.

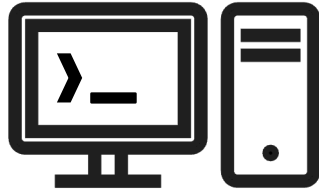
Provide a general format for debug statements and log statements.

- Make sure to provide enough but not too much information
- If dealing with confidential or sensitive data – Be very careful! Discuss with domain experts and the Secure Research Computing team for guidance to ensure you are not leaking provide information!

Format:

TimeStamp [CATEGORY/TAG] : ERROR MESSAGE

## Printing vs Logging



- Terminal (temporary)
- Active debugging
- Remove or “mute”



- File (permanent)
- Active/Passive debugging
- Always on



Research Computing  
UNIVERSITY OF COLORADO BOULDER

Create a debugging method

Create 3-4 potential categories for your log messages

Write them down in a readme file or as comment in the method header.



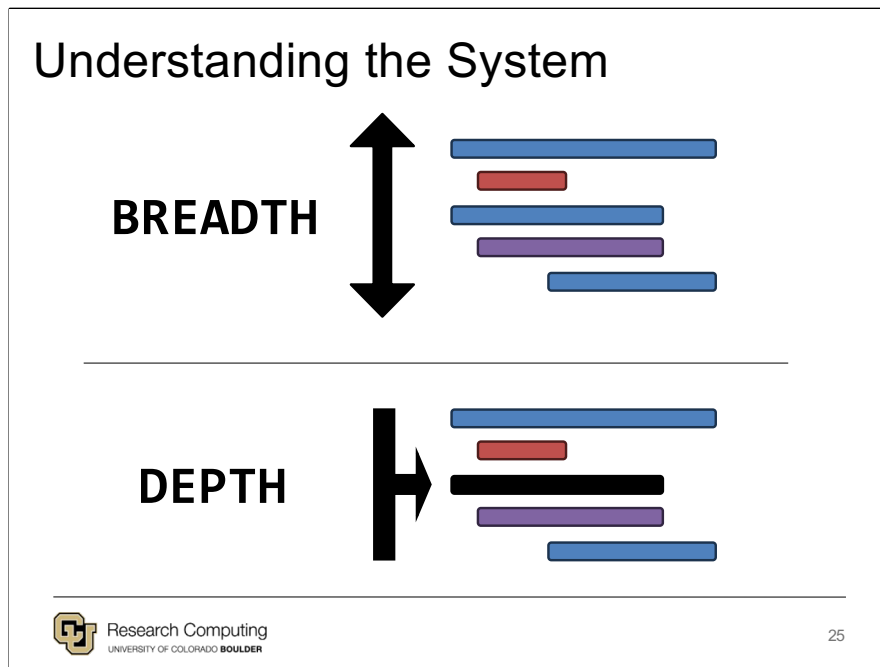
## Custom Logger

logger.py

```
1 def log_msg(type, msg, file="log.txt"):
2     log_file = open(file, "a")
3     match type:
4         case "LOG":
5             log_file.write(msg)
6
7         case "DEBUG":
8             if DEBUG:
9                 print("DEBUG: ", msg)
10                log_file.write("DEBUG: ", msg)
11
12     log_file.close()
13
```

24

Documentation on Python Logger:  
<https://docs.python.org/3/howto/logging.html>



Update this note to highlight the pitfall of diving into debugging (i.e. depth search). Try to always first familiarize yourself with the system with a breadth search where you consider the organization and structure of your project's different components.

Combine in discussion of forward/backward reasoning.

## Isolated Testing – Commenting Out

```
calculator.py
1 def calculate(operation, valueA, valueB):
2     result = 0
3
4     if operation == "multiply":
5         result = valueA * valueB
6
7     elif operation == "divide":
8         result = valueA / valueB
9
10    # elif operation == "add": # Single Line
11    # result = valueA + valueA
12
13    '''
14    elif operation == "subtrct": Block Comment
15    result = valueA - valueB
16    '''
```

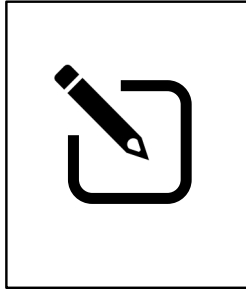


Research Computing  
UNIVERSITY OF COLORADO BOULDER

Note: Be mindful of indentation when using triple quotes as a “block comment” - this is creating a String literal which need to follow the same indentation rules of the surrounding code chunk.

# Debugging Tips

## Incremental Edits



### **Incremental Edits**

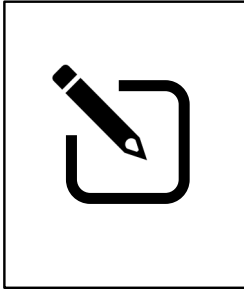
Tracking changes can be challenging – especially if you make a large number within a short period of time (remember there are limits to “undo, ctrl+Z”!)

So, whenever possible, only make one change at a time, test it, and then decide if you want to roll it back or keep it.

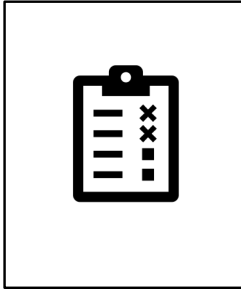
Software like Github can help, but isn’t really intended for tracking the constant back-and-forth updates often made while debugging. So it’s best not to rely on a tool, but rather focus on having a methodical process for keeping your debugging changes in check.

## Debugging Tips

### Incremental Edits



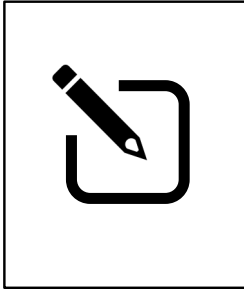
### Test Cases



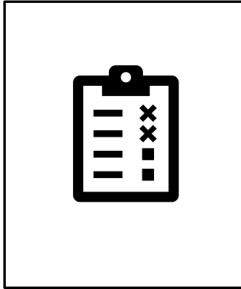
**Test Cases: \*\*add notes\*\***

## Debugging Tips

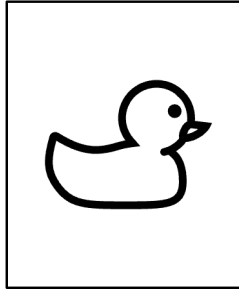
Incremental Edits



Test Cases



Rubber Duck



Research Computing  
UNIVERSITY OF COLORADO BOULDER

### **Rubber Duck Debugging:**

Wikipedia Article: [https://en.wikipedia.org/wiki/Rubber\\_duck\\_debugging](https://en.wikipedia.org/wiki/Rubber_duck_debugging)

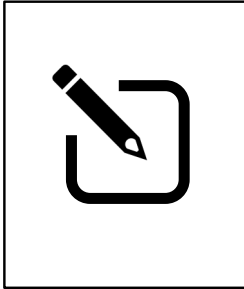
Explain, out loud, to your rubber duck friend what your program is intended to do and the bug you are experiencing.

In detail, explain what each part and/or line-of-code in your program does to your duck.

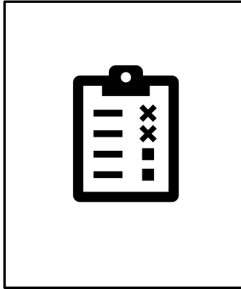
By discussing your work out-loud, you may reach a “Eureka!” moment, where the issue you were overlooking before become crystal clear.

## Debugging Tips

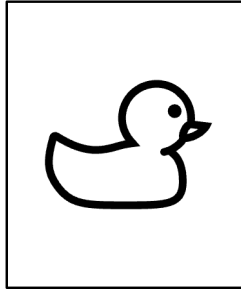
### Incremental Edits



### Test Cases



### Rubber Duck



**Delete & Start Over**



Research Computing  
UNIVERSITY OF COLORADO BOULDER

### **Delete & Start Over:**

Last Resort – helpful but can be dangerous, since you never identified the true cause of the bug. This means the bug could still pop-up after re-writing your code or arise later-on leading to a frustrating cycle of write-rewrite

## Asking for Help

When asking for debugging support, make sure to :

- Specify your goal – what are you trying to do?
- SLURM Job ID's (if applicable)
- Include descriptions and screenshots of errors you receive
- Describe the environment and/or software you are using
- *The more information you can provide, the better!*

**Help Ticket:** <https://www.colorado.edu/rc/userservices/contact>

**Email:** [rc-help@colorado.edu](mailto:rc-help@colorado.edu)



How to ask for help from RC Support Staff – We are experts in HPC/Alpine/Blanca, but not necessarily in your specific workflow.

- List types of requests we are best suited to answer.

Lab members and campus groups

Also, make sure to review online resources for more nuanced help. Some key areas to check are StackOverflow and Github Repos related to your research projects/tools.