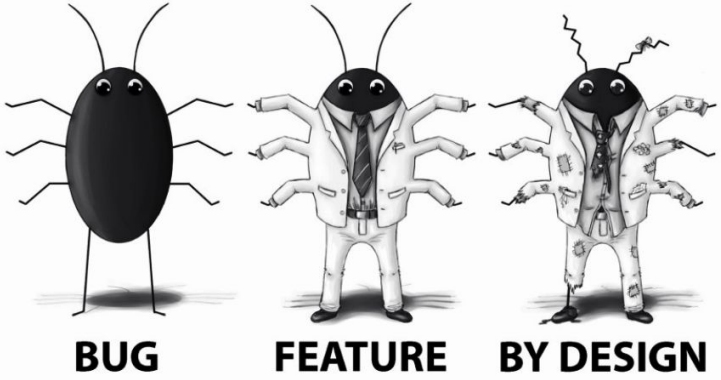


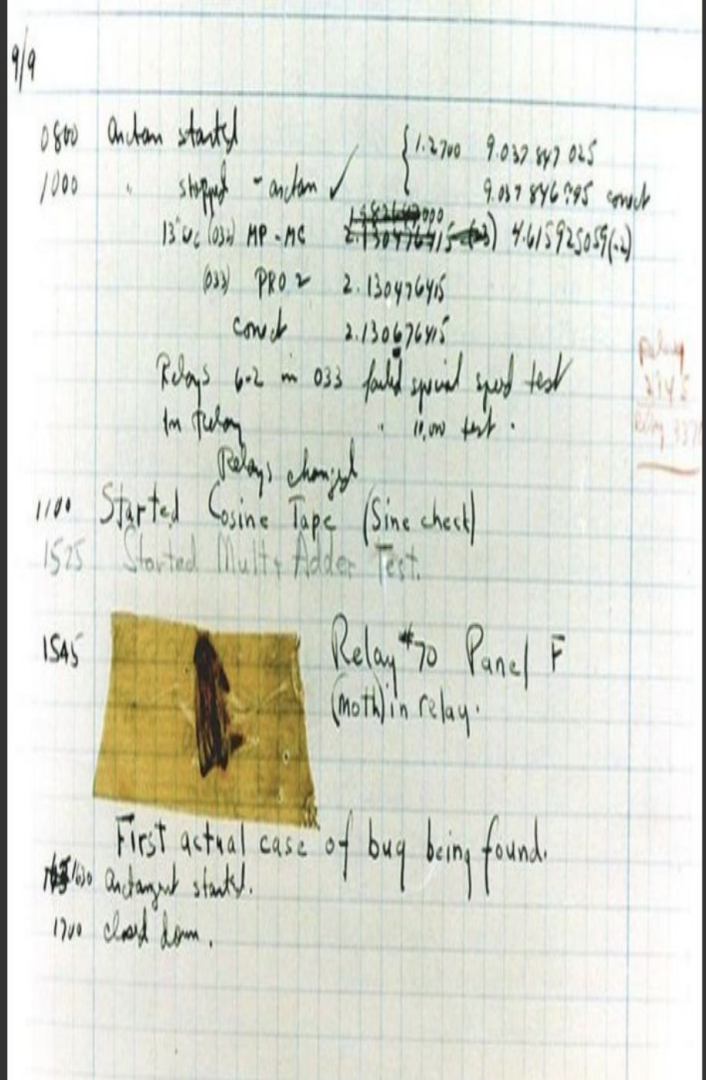
Debugging Lecture

Africa To Silicon Valley

A Bug’s Life

History's first documented bug

Grace Hopper was the one to document the bug on a notebook on the left. GHC is a big event now in celebration of her contribution to the field.



Pre-Debugging Steps

- Study program code
  - As how did I get the unexpected result
  - Is it related to another part of your code
- Study available data
- Make a hypothesis
- Pick simplest input to test with and fail fast
- Generalize and plan your changes (Tracking)

Note: Syntax errors are are easy to spot, while logic errors takes the most time

1. Think and Plan before writing the code
2. Draw pictures
3. Explain the code execution flow to someone else (interviewer) or a rubber ducky

Approaches

1. Brute Force method

    - Least economical and similar to spreading out your print statements strategically throughout your code

2. Backtracking

- Start from the statement at which an error symptom has been discovered, then derive backward till error is discovered. Also challenging as backward methods might increase.

3. Cause Elimination

- Listing causes that may presumably have contributed to the error symptom and tests are conducted to eliminate every error.

Approaches

4. Bisection method

- Find approximately the halfway point in your code and print out some relevant values in your code. If everything looks good, it’s code free, so the next search area would be in the ¾ area of the code print and try to pinpoint the place you are getting bad results.
- O(log n) time

Tools

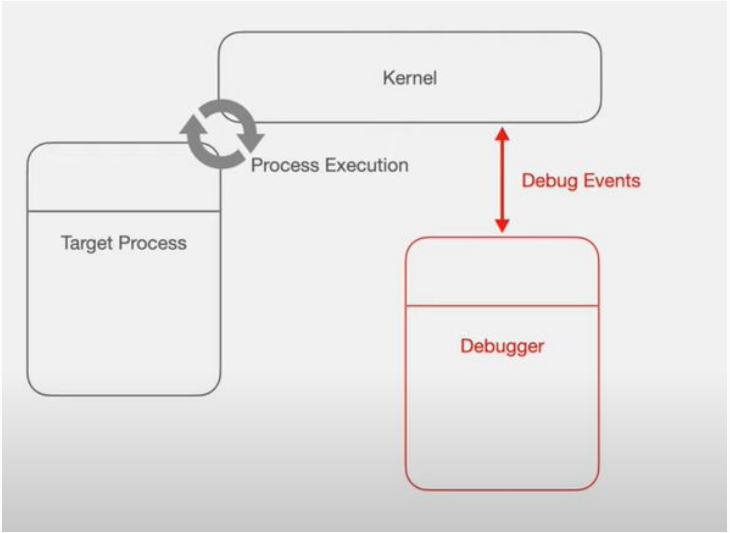
Modern Tools for debugging

Depending on your IDE and your setup there are plenty tools built-in to help you debug your code

1. **Breakpoints and Debuggers** [ref]
2. **Print statements and stdout**
3. **Visualizers and code tracers [ref]**
4. **Bug reporting solutions Eg. Extensions, Packages, ...**
5. **System logs**

Internals of a Built-in Debugger

- In general, debugger is utility that runs target program in controlled environment where you can control execution of program and see the state of program when program is paused.
- Once you set breakpoint, you can start program in debug mode, it will pause execution when program reaches the line where breakpoint is set.

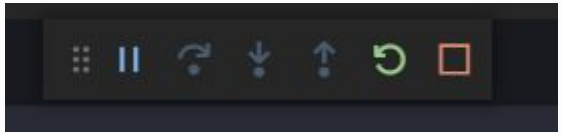


Using Built-in Debuggers

- **Breaking points:** Stops the execution of the program

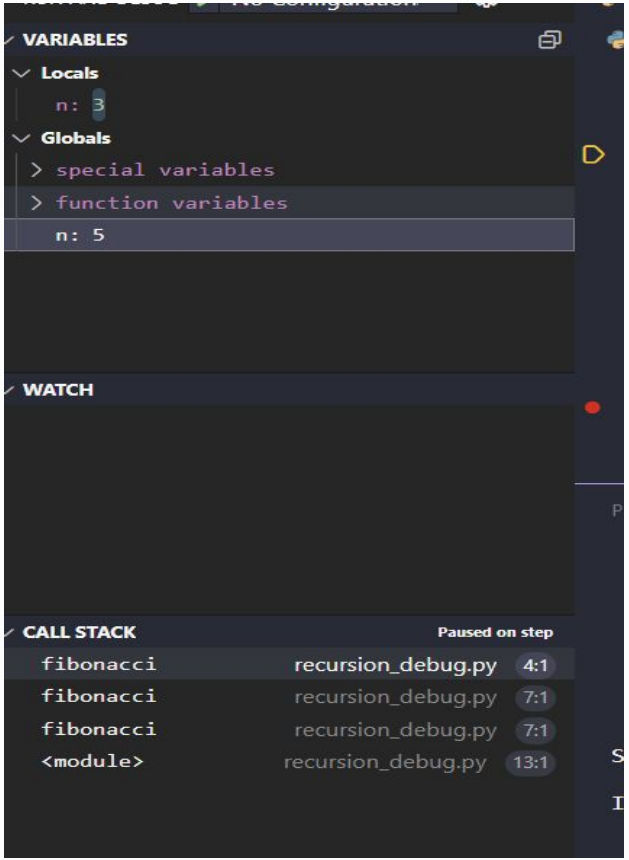
Now we can use stepping commands to execute program line by line.

1. **continue** – Resume program execution until next breakpoint is reached
2. **step into** – Execute program line by line stepping into function
3. **step over** – Execute program line by line but don't go inside function call
4. **step out** – Resume program execution until current function is finished



Pros of Debug mode

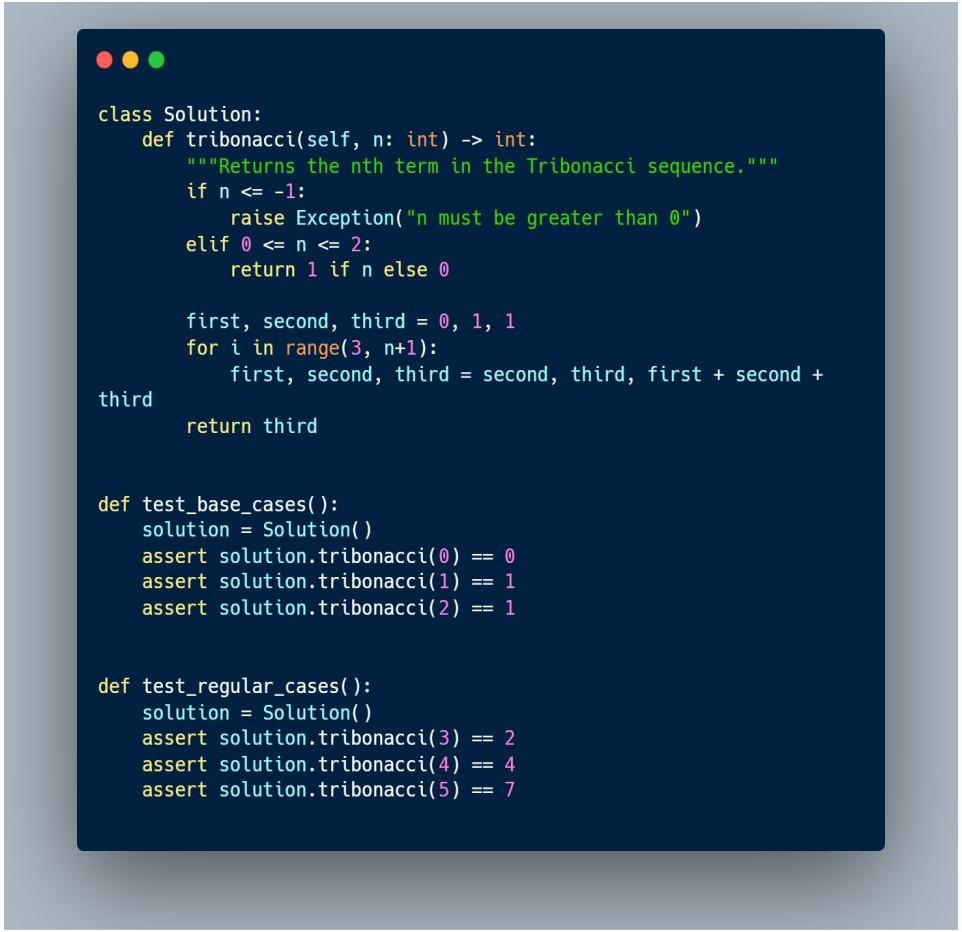
- No more prints in production code
- Visualization of stack trace along with values and variables
- Allows you to follow flow of execution, i.e. lines are printed along with the stack frames
- Track the changes on the fly
- In recursive functions, you can visualize your states and global variables easily



Assertions

- **Statements that you can use to set sanity checks during the development process**. Assertions allow you to test the correctness of your code by checking if some specific conditions remain true, which can come in handy while you're debugging code.

- This will make the program fail if conditions are not met before the online judge runs its test.



```python
class Solution:
    def tribonacci(self, n: int) -> int:
        """Returns the nth term in the Tribonacci sequence."""
        if n <= -1:
            raise Exception("n must be greater than 0")
        elif 0 <= n <= 2:
            return 1 if n else 0

        first, second, third = 0, 1, 1
        for i in range(3, n+1):
            first, second, third = second, third, first + second +
third
        return third


def test_base_cases():
    solution = Solution()
    assert solution.tribonacci(0) == 0
    assert solution.tribonacci(1) == 1
    assert solution.tribonacci(2) == 1


def test_regular_cases():
    solution = Solution()
    assert solution.tribonacci(3) == 2
    assert solution.tribonacci(4) == 4
    assert solution.tribonacci(5) == 7
```

Best Practices

Steps to follow

- Plan out your code ahead of time
- Decide what you are going to name the variables you will use over and over again
- Decide how you can best reuse code
- If your solution looks unnecessarily complicated, invest time in a simpler one before typing it out.

How to cut down your implementation/debugging time

Don’t

- Debug every line in your code
- Don’t step into every function
- Debug entire program

Change code without a note
Remember bug exists
Change and Test code
Forget where the bug was

Do

- Write a function / modularize
- Test the function, debug the function (Unit Test)
- Then do integration test / regression testing

Backup code
Write potential bug in a comment
Test code
Compare versions of your code

Quote of the day

Programming allows you to think about thinking, and while debugging you learn learning.

~ Nicholas Negroponte