Lab 3: Debugging

2025-09-19

Table of contents

Debugging

Today we're going to explore the concept of debugging and some of the tooling that allows for debugging python code.

It's widely recognized and accepted that any sizeable code base will have a non-trivial number of bugs (flaws in the design of a system; term used as early as 1878 by Thomas Edison). The main goal of testing is to make sure the main expected cases are behaving correctly.

Sometimes your code doesn't do what you expect or want it to do, and it's not clear just by reading through it, what the problem is.

In interpreted languages (especially those with interactive shells like python) one can sometimes run the code piece by piece and inspect the state of the variables.

If the code involves a loop or a function, a common practice is to judiciously place a few print statements that dump the state of some variables to the terminal so that you can spot the problem by tracing through it.

When the code involves multiple functions and complex state, this strategy starts to break down. This is where you start rolling up your sleeves and invoking a debugger!

Debugging can be a slow process, so you typically start a debugging session by deciding which line in your code you would like to start tracing the behavior from, and you place a breakpoint. Then you can have the debugger run the program up to that point and stop at it, allowing you to:

1- inspect the current state of variables 2- step through the code line by line 3- step over or into functions as they are called 4- resume program execution

Advanced debugging

Some bugs are really tricky to catch. Those are typically the bugs that happen very rarely, and in unclear circumstances. In statistical computing and data analysis settings these might be conditions that happen in iterative algorithms sometimes, but not very often, and can be hard to reproduce.

One way to deal with these bugs is to start the debugging session with placing one or more **conditional** breakpoints. These are similar to regular breakpoints but are not going to cause the debugger to stop the execution of the program and hand you the controls unless a specific condition (that you specify) evaluates to true as the program is executing that particular line of code. You may use some conditions that are similar to what you would place in an assert statement (conditions that shouldn't happen) or any other conditions that you think may be associated with the occurrence of the anomalous behavior your are debugging.

Integrated GUI debugger (with VS Code)

Today we will experiment with the visual debugging tools integrated with IDEs. We will do that in VS Code (unless you have another IDE with debugger integration). We will load a piece of code, go through it to understand what it does, then try to discover the problem with it and fix it. Our main function is `get_commits`, which makes use of the GitHub API to retrieve the recent commits to a repository. The fact that there are nested functions makes debugging less straightforward, but we will see how the bug could nevertheless be uncovered. Although the focus is on visual debuggers today, the ideas apply directly to command-line debuggers.

```
import json
import pandas as pd
import requests
import warnings

def get_login(item):
    if not isinstance(item, dict):
        raise TypeError(f"`item` argument must be a dictionary")
    return item['author']['login']

def process(data):
    if not isinstance(data, dict) or "commit" not in data.keys() or \
       not isinstance(data['commit'], dict) or \
       "author" not in data['commit'].keys() or \
       not isinstance(data['commit']['author'], dict) or \
       "name" not in data['commit']['author'].keys() or \
       "date" not in data['commit']['author'].keys():
        raise RuntimeError(f"Unexpected json structure returned for commit: {data}")
    return {"user": data['commit']['author']['name'],
            "date": data['commit']['author']['date'],
            "login": get_login(data)}

def get_commits(org, repo, extended=False):
    if not isinstance(org, str):
```

```
        raise TypeError(f"`org` argument must be a string")
    if not isinstance(org, str):
        raise TypeError(f"`repo` argument must be a string")
    main_url = 'https://api.github.com/repos'
    perpage = 100
    # Search parameters
    request = f"{main_url}/{org}/{repo}/commits?per_page={perpage}"
    try:
        response = requests.get(request)
    except Exception as err:
        print("Problem making GitHub commits API request over the internet")
        raise

    if response.status_code == 200:  # Or `response.reason == 'OK':`
        content = response.content
    else:
        raise RuntimeError(f"Error in API request: {response.status_code}.")

    data = json.loads(content)
    data_cleaned = [process(item) for item in data]
    df = pd.DataFrame(data_cleaned)
    return df
```

Post-mortem debugging

A usual workflow to debug a piece of code that is crashing is to run it, investigate the exact location where the crash happens, add a breakpoint statement and rerun the code. Oftentimes, the bug is not deterministic or only happens at certain iterations inside a loop, so a conditional breakpoint will have to be engineered. Post-mortem debugging simplifies such a workflow, sometimes drastically, by running the program until it crashes and automatically placing the user into a debugging session immediately before the crash. To enter post-mortem debugging automatically, a script that exits abnormally can be called via `python -m pdb myscript.py`. Alternatively, one may import `pdb` and call `pdb.pm()` after a function call crashes.

Acknowledgements

This lab was originally authored by Ahmed Eldeeb and adapted for the Fall 2025 semester.