


CS 45, Lecture 11

Debugging and Profiling

Winter 2023

Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

Administrivia

- Assignment 4 is due tonight! It covers all things Git. Reach out if you need more time.
- Assignment 5 will come out sometime tonight or tomorrow!
- Thank you for the feedback 

What we will cover today

In today's lecture, we will learn about :

- Basic debugging techniques such as printing and logging
- Debugging tools
- Profiling your code for memory leaks, resource management, timing

This lecture may feel like a bunch of tools and demos. You don't need to become an expert in these now, but it's worth knowing they are out there!

Installations

Throughout this lecture, we will be looking at a number of different tools.

Some of these won't be installed on your machine but feel free to install as we go. For Python tools, use:

```
pip3 install <name-of-tool>
```

Introduction to Debugging

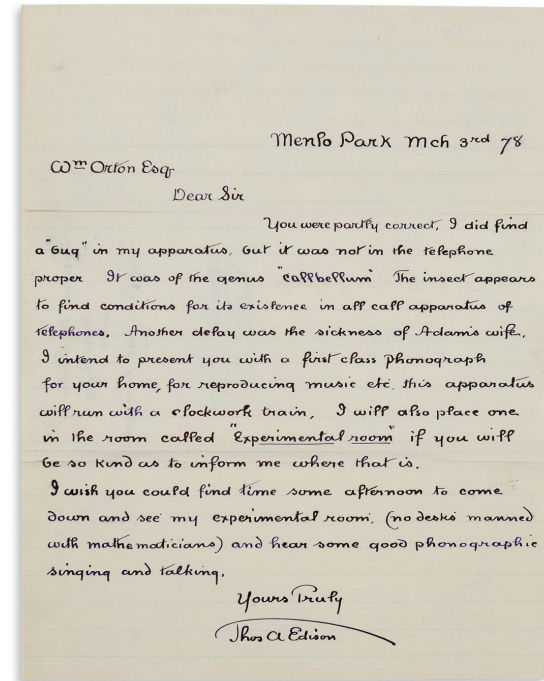
What's the #1 issue I saw student having when working as a CA?

Not knowing how to debug.



History of Debugging

The terminology of the term debugging has a fun history to it.

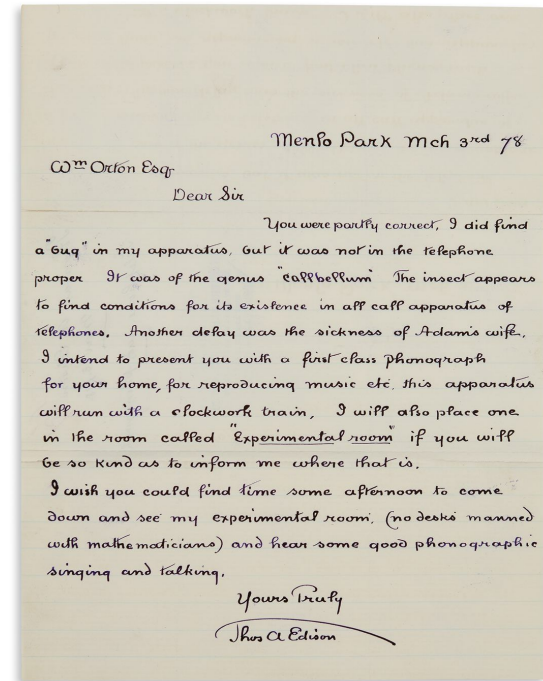


Note by Thomas Edison where he used the term bug to describe a technical error.

History of Debugging

The terminology of the term debugging has a fun history to it.

In 1878, Thomas Edison was the first to use the term "bug" to describe a technical error.



Note by Thomas Edison where he used the term bug to describe a technical error.

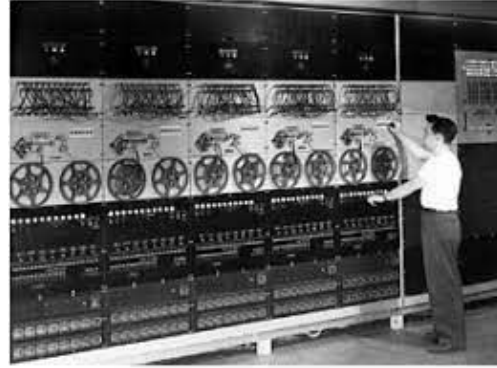
History of Debugging

In 1945, the first computers were being built.

History of Debugging

In 1945, the first computers were being built.

Mark III was one of these computers that was being built at Harvard. Grace Hopper was developing Mark III when she encountered a problem with its functionality.



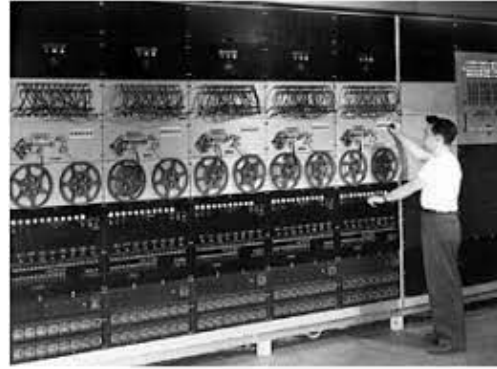
Mark III

History of Debugging

In 1945, the first computers were being built.

Mark III was one of these computers that was being built at Harvard. Grace Hopper was developing Mark III when she encountered a problem with its functionality.

After running some tests, she decided to look inside and found an actual moth, which led to the usage of the word debugging in programming.



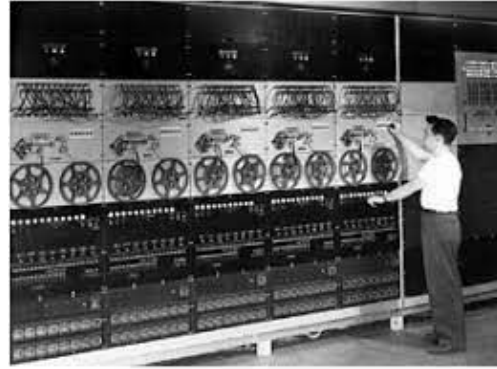
Mark III

History of Debugging

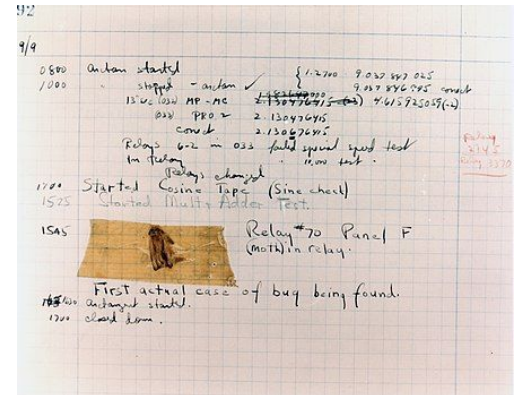
In 1945, the first computers were being built.

Mark III was one of these computers that was being built at Harvard. Grace Hopper was developing Mark III when she encountered a problem with its functionality.

After running some tests, she decided to look inside and found an actual moth, which led to the usage of the word debugging in programming.



Mark III



Note by Grace Hopper with moth attached. 11

Print Debugging

The simplest approach to debugging is to add print statements to figure out where your issue is.

Print Debugging

The simplest approach to debugging is to add print statements to figure out where your issue is.

This approach is known as `printf()` debugging (so called after the C function by the same name).

Print Debugging

The simplest approach to debugging is to add print statements to figure out where your issue is.

This approach is known as `printf()` debugging (so called after the C function by the same name).

“The most effective debugging tool is careful thought, coupled with judiciously placed print statements.”

– Brian Kernighan, *Unix for Beginners*

Print Debugging

Each language has a standard printing function that can be used to print to standard output or standard error:

Python

Standard Output

```
print("Inside if-statement")
```

Standard Error

```
print("Inside if-statement", file=sys.stderr)
```

C++

Standard Output

```
std::cout << "Inside if-statement" << std::endl;
```

Standard Error

```
std::cerr << "Inside if-statement" << std::endl;
```

Print Debugging

The placement and structure of your print statements is important.

Print Debugging

The placement and structure of your print statements is important.

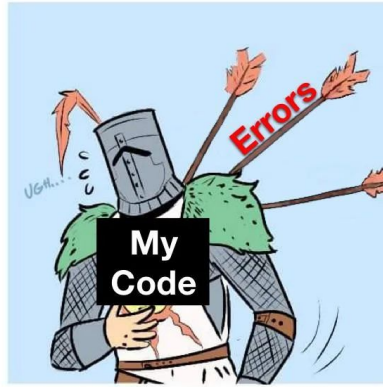
1. Wording matters: “MADE IT HEEEEERRRRREEE” is less helpful than “Inside else case to check if getData succeeded”

Print Debugging

The placement and structure of your print statements is important.

1. Wording matters: “MADE IT HEEEEERRRRREEE” is less helpful than “Inside else case to check if getData succeeded”
2. Print statements are useful inside of if-else statements to see which branch of code execution is taken. They are also useful after a loop or a function call to see that the loop / function exits.

Print Debugging



Logging

A more complex version of print debugging is **logging**. Logging is used to capture information about a system run. You can think of logging as a more structured and systematic framework to add print statements.

Logging

A more complex version of print debugging is **logging**. Logging is used to capture information about a system run. You can think of logging as a more structured and systematic framework to add print statements.

It is used as part of industry standard for implementation of larger systems.

Logging

A more complex version of print debugging is **logging**. Logging is used to capture information about a system run. You can think of logging as a more structured and systematic framework to add print statements.

It is used as part of industry standard for implementation of larger systems.

At a bare minimum, logging should allow you to do everything that print statements do: print messages to standard output and standard error.

Logging

Logging is normally done by designating different “levels” for each log message.

Different log levels have different levels of importance. A log message of type `ERROR` requires immediate attention while a log message of type `TRACE` might just be a “nice to have” confirmation that a given piece of code is executing.

Logging

ERROR

High severity, requires immediate attention (abort)

WARNING

Moderate severity, detected an unexpected problem

INFO

Moderate severity, reporting important information

DEBUG

Used for debugging purposes

TRACE

Used for tracing execution of code

Logging

Log levels allow a developer to toggle between different levels and filter based on these levels.

A developer might only be interested in `WARNING` or `ERROR` messages for a certain run

In general, the default level of logging for production level code is `INFO`

Logging

Some languages have a built-in logging library such as Python. Others, such as C++, require you to implement a logging library.

Here is logging in Python:

```
import logging

logging.debug("We're debugging. Something happened!")
logging.info("For your info, something happened.")
logging.warning("A warning occurred. Beware!")
logging.error("Something is in error. Go fix it.")
logging.critical("Critical condition. Go seek shelter. NOW.")
```

Logging: Python

FATAL

Extremely high severity, application will abort

ERROR

High severity, requires immediate attention

WARNING

Moderate severity, detected an unexpected problem

INFO

Moderate severity, reporting important information

DEBUG

Used for debugging purposes

Logging: Python

By default, the logging level is set to `WARNING` which means only the last three lines will get printed:

```
import logging

logging.debug("We're debugging. Something happened!")
logging.info("For your info, something happened.")
logging.warning("A warning occurred. Beware!")
logging.error("Something is in error. Go fix it.")
logging.critical("Critical condition. Go seek shelter. NOW.")
```

Logging: Python

We can change the logging level to increase or decrease the number of logging messages we see:

```
import logging

logging.basicConfig(level = logging.DEBUG)

logging.debug("We're debugging. Something happened!")
logging.info("For your info, something happened.")
logging.warning("A warning occurred. Beware!")
logging.error("Something is in error. Go fix it.")
logging.critical("Critical condition. Go seek shelter. NOW.")
```

Logging: Python

Logging allows you to send the output to a variety of different places, not just standard output and standard error.

You can send your log messages to a file, a remote log server, a window event log, or a database.

```
import logging

logging.basicConfig(filename='example.log', level=logging.DEBUG)

logging.debug("We're debugging. Something happened!")
logging.info("For your info, something happened.")
```

Advanced Logging

Let's take a look at how to implement logging in Python, including some fancy features with formatting and customization!

[Python Logging Demo]

Logging

Third party logs are useful when you use external libraries or dependencies.

In UNIX, most programs write their logs in `/var/log`

Example: if you have an issue where all of your apps freeze, you might find the `/var/log/system.log` file (on a Mac), the `/var/log/journal` file (on Linux), or the Event Viewer (on Windows) which will give you more information about why your apps are crashing

Debuggers

When print debugging and logging is not enough, you should use a **debugger**. A debugger is a program that allows you to examine another program in order to detect errors in that other program.

With a debugger, you can:

- Halt execution of the program when it reaches a certain line
- Step through the program one line at a time
- Inspect values of variables after the program crashes

Debuggers

Many programming languages come with some sort of default debugger:

Python → `pdb` debugger

C/C++ → `gdb` and `lldb` are both C/C++ debuggers

Go → Delve is a GoLang debugger

Java → `jdb` is a Java debugger

In general, when choosing a debugger, you simply want to find one that is compatible with the language you are coding in.

Debuggers: pdb

Some common debugging commands:

list - displays some (around 11) lines of the program

step - execute a single line, and step into called function (if necessary)

next - execute a single line, do not step into called function

print - prints a variable or symbol

break - set a breakpoint

Debuggers: pdb

Let's deep dive into the `pdb` debugger!

To load a program with `pdb`:

```
python3 -m pdb mbi.py
```

Debuggers: pdb

Let's deep dive into the `pdb` debugger!

To load a program with `pdb`:

```
python3 -m pdb mbi.py
```

Imports `pdb` as a module
to be run on `mbi.py`

Debuggers: pdb

Let's deep dive into the `pdb` debugger!

To load a program with `pdb`:

```
python3 -m pdb mbi.py
```

Name of program we are running

Imports `pdb` as a module
to be run on `mbi.py`

Let's practice!

I've uploaded a buggy program called `area_of_rectangle.py`.

```
curl -Lo area_of_rectangle.py  
http://stanford-cs45.github.io/res/lec11/area_of_rectangle.py
```

Let's try to use `pdb` to debug it!

```
python3 -m pdb area_of_rectangle.py
```

Try adding logging statements using the `logging` library.

Debuggers: Life Hack (for C/C++)

Using `gdb` or `lldb` to find where your program is crashing!

Debuggers: Life Hack (for C/C++)

Using `gdb` or `lldb` to find where your program is crashing!

We need to compile our code:

```
g++ -std=c++11 -g -o weather_report weather_report.cc
```

Debuggers: Life Hack (for C/C++)

Using `gdb` or `lldb` to find where your program is crashing!

We need to compile our code:

```
g++ -std=c++11 -g -o weather_report weather_report.cc
```

Version of C++
we want to use

Debuggers: Life Hack (for C/C++)

Using `gdb` or `lldb` to find where your program is crashing!

We need to compile our code:

```
g++ -std=c++11 -g -o weather_report weather_report.cc
```

Version of C++
we want to use

Create debugging
symbols

Debuggers: Life Hack (for C/C++)

Using `gdb` or `lldb` to find where your program is crashing!

We need to compile our code:

```
g++ -std=c++11 -g -o weather_report weather_report.cc
```

To run the program, we can use:

```
./weather_report
```

Debuggers: Life Hack (for C/C++)

Using `gdb` or `lldb` to find where your program is crashing!

We need to compile our code:

```
g++ -std=c++11 -g -o weather_report weather_report.cc
```

To run the program, we can use:

```
./weather_report
```

To run the program under `lldb`, we can use:

```
lldb weather_report
```

Debuggers: Life Hack (for C/C++)

Once your program is in `gdb` or `lldb`, you need to run it:

Debuggers: Life Hack (for C/C++)

Once your program is in `gdb` or `lldb`, you need to run it:

We need to compile our code:

```
(lldb) run
```

Debuggers: Life Hack (for C/C++)

Once your program is in `gdb` or `lldb`, you need to run it:

We need to compile our code:

```
(lldb) run
```

Once it crashes, you can run backtrace (or `bt`) to find where it crashed:

```
(lldb) bt
```


Web Debugging

Most modern browsers support built-in debugging tools.

You can enter developer mode by pressing `F12` or hitting `Cmd + Option + I`

You can navigate and examine the files, add breakpoints, trace execution, and add logging statements.

Compiler Errors

Compiler errors are your friends 😁

Always look at the line number and where the error occurred.

Look up compiler errors online on sites like StackOverflow. If you're running into an error, it's most likely someone else has run into that same error before.

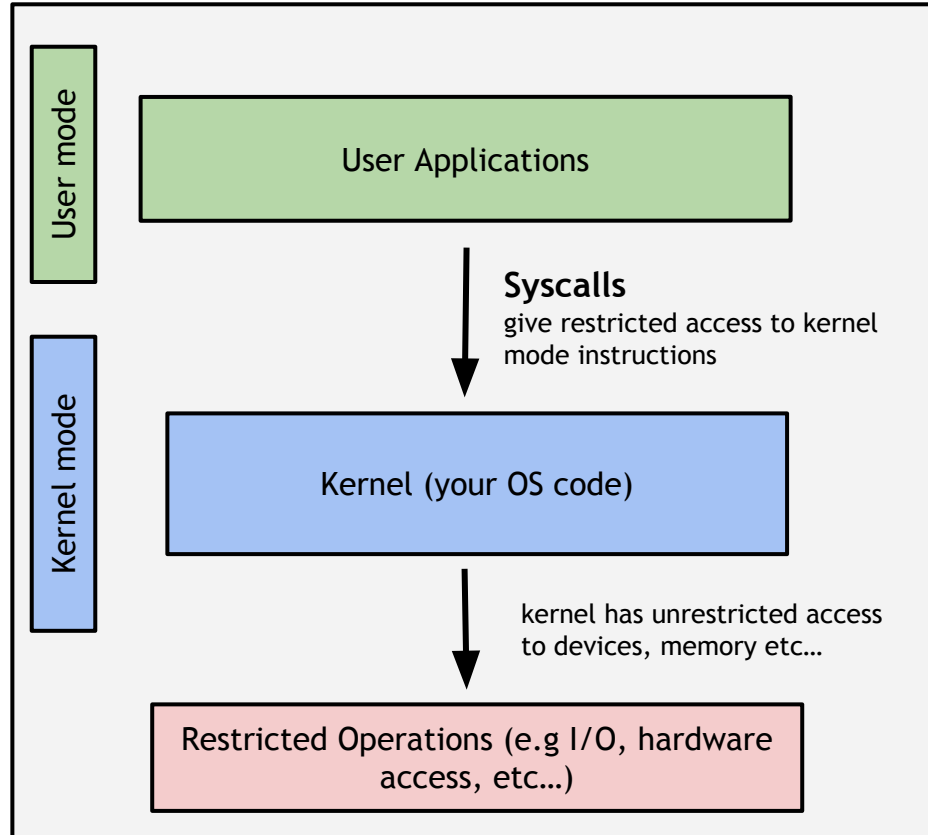
Specialized Tools

We can debug code even if we don't have the source code. One example is debugging **system calls**

System calls are functions that are executed by the kernel (which is the computer program at the core of a computer's operating system and generally has complete control over everything in the system).

We don't have access to the internal implementations of system calls which means we can't look inside to see what's going on in the code.

Specialized Tools



Specialized Tools

We can use a tool called **strace** in order to trace system calls.

strace allows us to observe the execution of a system call.

We don't have access to the internal implementations of system calls which means we can't look inside to see what's going on in the code.

Testing Your Code

A key part of debugging is testing your code :)

You can choose a testing framework to use in order to implement your tests.

There are also tools that will report the test coverage for your tests. This is part of being a good programmer: ensuring that you commit code in small chunks and test each chunk.

Testing Your Code

coverage is a testing coverage tool for Python:

Code Profiling

Profilers are good for when your code runs as expected (yay!) but is inefficient...

Profilers help you understand which parts of your program take up the most time and resources so you can focus on optimizing those parts.

"Premature optimization is the root of all evil." - Donald Knuth

Code Profiling

Profilers come in two flavors:

Tracing Profilers: keep a record of every function call your program makes.

Advantages: more accurate analysis

Disadvantages: add a lot of overhead to the program

Examples: gprof, VTune,

Sampling Profilers: periodically probe program to record the program's stack.

Advantages: does not disturb application at run time

Disadvantages: provides approximations

Examples: OProfile, perf, AMD uProf

Code Profiling

Python has a built in code profiler called `cProfile` that will allow us to identify bottlenecks.

```
python3 -m cProfile -s tottime site_scraper.py
```

Code Profiling

Sometimes we want to do line by line analysis of a specific function. Is there a single line in this function that is taking the most time?

```
kernprof -l -v site_scraper.py
```

Let's practice!

Try using `cProfile` and `line_profiler` on your `area_of_rectangle.py`

```
python3 -m cProfile area_of_rectangle.py
```

```
kernprof -l -v area_of_rectangle.py
```

Timing Your Code

If your code takes a really long time to run, this could be an indication of an issue.

Ideally, you need to figure out how much time the specific program took to run. (There are other things running on your computer that may be running in the background and slowing things down.)

Timing Your Code

`time` is a command that is used to execute a program and print a real time analysis of how long the program took to execute.

In a zsh shell, there is a `time` keyword. If you want to use the `time` command, type: `command time <name-of-program>`

Example:

```
time ./memory_leak      or      time python3 site_scraper.py
```

Timing Your Code

The `time` command will report statistics on three different "types" of times: `real`, `user`, and `sys`:

```
adrazen$: time ./program
real    0m0.193s
user    0m0.012s
sys     0m0.056s
```

`real` is wall clock time (from start to finish)

`user` is time spent in user mode (for this program)

`sys` is time spent in kernel mode (for this program)

To get the actual CPU time your program used, add `user` + `sys`

Note that `real` will include time waiting for I/O, or time used by other processes.

Memory Access Tools

Memory access tools allow us to identify memory leaks and inefficient memory usage.

`valgrind` is a memory access and memory leak detection tool for GNU/Linux systems.

`leaks` is a similar tool for macOS systems.

Memory Access Tools

Memory access tools allow us to identify memory leaks and inefficient memory usage.

[Valgrind / Leaks Demo]

Memory Access Tools



Memory Access Tools

Posted by u/Blubbpaule 7 days ago

Has this game a memory leak?

Question



Ruhart · 7 days ago

I'm running 32gb and I crashed on the intro all over. I was so excited I

5



MarbhIasc · 7 days ago

I have 32gb and suffer almost cont

3



darkanthony3 · 7 days ago

This exact thing is happening to me.

2



deletable666 · 4 days ago

I have noticed that if I go into the settings menu like I am about to s shoots back up when I exit back to the game. I assume going into th something in the game. It works every time. I am waiting for a patch untenable to play doing. For reference I am on ultra everything with to 8-20 after a bit of time or something loading. 32gb of memory, 12

2



mudermarshmallows · 3 mo. ago

DURAGON CULAW

Games should be complete on release. If it's a very common bug it should have been found and fixed before launch.

11



Comment deleted by user · 3 mo. ago



Krazytre · 3 mo. ago

In this day and age, most games have issues that are fixed in the most commonly released "Day 1 patch", or shortly after release they'll release a fix for some, if not most, of the bugs and glitches that were in the game, maybe a few days to a week.

Yes, it sucks that we're always relying on patches to fix a game that shouldn't have been broken on release, but that's the way it is for most game companies now.

5

how I got better at debugging

Remember: the bug is happening for a logical reason.

It's never magic. Really. Even when it makes no sense.

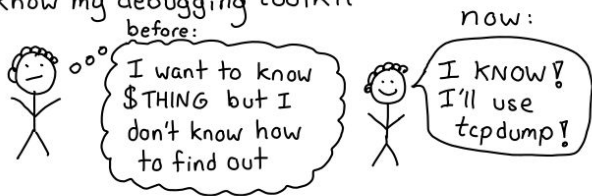
Be confident I can fix it



Talk to my coworkers



know my debugging toolkit



most importantly: I learned to like it

