# The Missing Semester: Debugging and Profiling


Sri Venkateswara
College of Engineering

# Why Debugging and Profiling?

- A golden rule in programming is that **code does not do what you expect it to do, but what you tell it to do**.
- Bridging that gap can sometimes be a quite difficult feat.
- In this lecture we are going to cover useful techniques for dealing with **buggy and resource hungry code**: debugging and profiling.

# Debugging Techniques - Printf debugging and Logging

*"The most effective debugging tool is still careful thought, coupled with judiciously placed print statements"* — Brian Kernighan, *Unix for Beginners*.

- **Add print statements** around where you have detected the problem, and keep iterating until you have extracted enough information to understand what is responsible for the issue.
- **use logging in your program**, instead of ad hoc print statements
- A tip for making logs more readable is to **color code** them.
- Programs like ls or grep are using **ANSI escape codes**, which are special sequences of characters to indicate your shell to change the color of the output.
- Check out python package colorama to print coloured output in python

# Debugging Techniques - Printf debugging and Logging

Download test file here

```
$ python logger.py
# Raw output as with just prints
$ python logger.py log
# Log formatted output
$ python logger.py log ERROR
# Print only ERROR levels and above
$ python logger.py color
# Color formatted output
```

# Debugging Techniques -Debuggers

Debuggers are programs that let you **interact with the execution of a program**.

- Halt execution of the program when it reaches a certain line.
- Step through the program one instruction at a time.
- Inspect values of variables after the program crashed.
- Conditionally halt the execution when a given condition is met.
- And many more advanced features

# Debugging Techniques – Python Debugging

Many programming languages come with some form of debugger. In Python this is the Python Debugger pdb.

Here is a brief description of some of the commands pdb supports:
- **l**(ist) - Displays 11 lines around the current line or continue the previous listing.
- **s**(tep) - Execute the current line, stop at the first possible occasion.
- **n**(ext) - Continue execution until the next line in the current function is reached or it returns.
- **b**(reak) - Set a breakpoint (depending on the argument provided).
- **p**(rint) - Evaluate the expression in the current context and print its value. There's also **pp** to display using pprint instead.
- **r**(eturn) - Continue execution until the current function returns.
- **q**(uit) - Quit the debugger.

# Debugging Techniques – Python Debugging

Copy this code into bubble.py

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n):
            if arr[j] > arr[j+1]:
                arr[j] = arr[j+1]
                arr[j+1] = arr[j]
    return arr

print(bubble_sort([4, 2, 1, 8, 7, 6]))
```

python –m pdb bubble.py

# Debugging Techniques – Web Development

For web development, the Chrome/Firefox developer tools are quite handy. They feature a large number of tools, including:

- **Source code** - Inspect the HTML/CSS/JS source code of any website.
- **Live HTML, CSS, JS modification** - Change the website content, styles and behavior to test (you can see for yourself that website screenshots are not valid proofs).
- **Javascript shell** - Execute commands in the JS REPL.
- **Network** - Analyze the requests timeline.
- **Storage** - Look into the Cookies and local application storage.

# Profiling

- Even if your code functionally behaves as you would expect, that might not be good enough if it **takes all your CPU or memory in the process.**
- Algorithms classes often teach **big *O* notation** but not how to find hot spots in your programs.
- Profilers and monitoring tools will help you understand which parts of your program are taking most of the time and/or resources so you can **focus on optimizing those parts**.

# Profiling - Time

Copy this code into time.py

```
import time, random
n = random.randint(1, 10) * 100

# Get current time
start = time.time()

# Do some work
print("Sleeping for {} ms".format(n))
time.sleep(n/1000)

# Compute time between start and now
print(time.time() - start)
```

python time.py

# Profiling - Time

Wall clock time can be misleading since your computer might be running other processes at the same time or waiting for events to happen. It is common for tools to make a distinction between *Real*, *User* and *Sys* time. In general, *User* + *Sys* tells you how much time your process actually spent in the CPU.

- ***Real* -** Wall clock elapsed time from start to finish of the program, including the time taken by other processes and time taken while blocked (e.g. waiting for I/O or network)
- ***User* -** Amount of time spent in the CPU running user code
- ***Sys* -** Amount of time spent in the CPU running kernel code

# Profiling -CPU

- Most of the time when people refer to *profilers* they actually mean *CPU profilers*, which are the most common.
- There are two main types of CPU profilers: *tracing* and *sampling* profilers.
- Tracing profilers keep a record of **every function cal**l your program makes.
- Sampling profilers **probe your program periodically** (commonly every millisecond) and **record the program's stack**. They use these records to present aggregate statistics of what your program spent the most time doing.

# Profiling - CPU

```
#!/usr/bin/env python

import sys, re

def grep(pattern, file):
    with open(file, 'r') as f:
        print(file)
        for i, line in enumerate(f.readlines()):
            pattern = re.compile(pattern)
            match = pattern.search(line)
            if match is not None:
                print("{}: {}".format(i, line), end="")

if __name__ == '__main__':
    times = int(sys.argv[1])
    pattern = sys.argv[2]
    for i in range(times):
        for file in sys.argv[3:]:
            grep(pattern, file)
```

Copy this code into grep.py

python -m cProfile -s tottime grep.py 1000 '^(import|\s*def)[^,]*$' *.py

# Profiling - Memory

- In languages like C or C++ memory leaks can cause your program to never release memory that it doesn’t need anymore.
- In garbage collected languages like Python it is still useful to use a memory profiler because as long as you have pointers to objects in memory they won’t be garbage collected.

# Profiling -Memory

pip install memory-profiler

Copy this code into example.py

```
@profile
def my_func():
    a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a

if __name__ == '__main__':
    my_func()
```

python -m memory_profiler example.py

# Resource Monitoring

The first step towards analyzing the performance of your program is to understand what its actual resource consumption is. Programs often run slowly when they are resource constrained, e.g. without enough memory or on a slow network connection. There are a myriad of command line tools for probing and displaying different system resources like CPU usage, memory usage, network, disk usage and so on.

- **General Monitoring** - Probably the most popular is **htop**, which is an improved version of top. htop presents various statistics for the currently running processes on the system. htop has a myriad of options and keybinds, some useful ones are: <F6> to sort processes, t to show tree hierarchy and h to toggle threads.
- **I/O operations** - **iotop** displays live I/O usage information and is handy to check if a process is doing heavy I/O disk operations

# Resource Monitoring

- **Disk Usage** - **df** displays metrics per partitions and **du** displays **d**isk **u**sage per file for the current directory.
- **Memory Usage** - **free** displays the total amount of free and used memory in the system. Memory is also displayed in tools like htop.
- **Open Files** - **lsof** lists file information about files opened by processes. It can be quite useful for checking which process has opened a specific file.
- **Network Connections and Config** - **ss** lets you monitor incoming and outgoing network packets statistics as well as interface statistics. A common use case of ss is figuring out what process is using a given port in a machine. For displaying routing, network devices and interfaces you can use ip. Note that netstat and ifconfig have been deprecated in favor of the former tools respectively.
- **Network Usage** - **nethogs** and **iftop** are good interactive CLI tools for monitoring network usage.