HZBRL

Let's spend some quality time so that you won't have to pursue farming. . .

Kazi Rifat Morshed

Undergraduate Student (3rd Year, 2nd Term)
Joint Secretary (CLUSTER Exeecutive Committee 2026)
Computer Science and Engineering Discipline
Khulna University

February 20, 2026

Why are we here???

Figure: what and why...??? [2]

Table of contents I

Lets Start: Why Ho Zo Bo Ro Lo (Khichuri) I

- No specific goals
- Includes many things that you should know
- Academia/Teachers won't teach you
- Will try to practice
- stop me AND ASK QUESTIONS
- I AM HERE TO ASSIST YOU
- Needs TIME! (almost 3-4 hours)
- Attention Please...
- Avoid sufferings in the long run

Roadmap: How to become a/an ???

- https://roadmap.sh/
- https://roadmap.sh/frontend
- https://roadmap.sh/frontend/projects
- https://roadmap.sh/backend
- https://roadmap.sh/backend/projects
- https://roadmap.sh/full-stack

How Things Start (Bootloader & OS): The First Spark I

- **BIOS/UEFI:** The very first code that runs when you press the power button.
- **Bootloader (GRUB/Windows Boot Manager):**
  - Small program that loads the OS into memory.
  - **GRUB:** Common in Linux, allows you to choose between different Kernels or OSs.
- **The Kernel:** Once loaded, it takes over the hardware.

Files & Hierarchies: Text vs Binary I

- **Text Files:** Readable by humans (C++ source, Python scripts, .txt). Encoded in ASCII/UTF-8.
- **Binary Files:** Compiled programs, images, videos. Looks like "garbage" if opened in Notepad.
- **The "Extension" Lie:**
  - Extensions (.exe, .png, .cpp) are just hints for the OS.
  - **Demo:** Rename `script.py` to `script.txt`. It still contains the same data!
  - Changing a `.docx` to `.zip` and opening it (it's actually a collection of XMLs!).

Files & Hierarchies: Directory Hierarchy I

Windows (Drive-based)

- `C:\` (Root)
- `C:\Windows` (System)
- `C:\Users\Name` (Home)
- Backslashes \

Linux (Root-based)

- / (Root)
- `/bin`, `/etc`, `/home`
- Everything is a file!
- Forward slashes /

Under the Hood (Compilation & Execution): How Code Actually Runs

- **The Translation Pipeline:**

The Path

Source Code (text file) → Assembly → Machine Code (saved as Binary in SSD)

- **How Compiled Code Works:**

Simple Exeecution Path

SSD (Secondary Memory) → RAM (Primary Memory) → CPU

- **Analogy:** Python is like a high-level recipe; Assembly is the specific muscle movements the chef makes.
- **Assembly Transmission:** How C or C++ looks like when stripped down to ‘MOV‘, ‘ADD‘, and ‘PUSH‘ instructions.

Under the Hood (Compilation & Execution): How Code Actually Runs

- **The Translation Pipeline:**

The Path

Source Code (text file) → Assembly → Machine Code (saved as Binary in SSD)

- **How Compiled Code Works:**

Simple Exeecution Path

SSD (Secondary Memory) → RAM (Primary Memory) → CPU

- **Analogy:** Python is like a high-level recipe; Assembly is the specific muscle movements the chef makes.
- **Assembly Transmission:** How C or C++ looks like when stripped down to ‘MOV‘, ‘ADD‘, and ‘PUSH‘ instructions.

Under the Hood (Compilation & Execution): How Code Actually Runs

- **The Translation Pipeline:**

The Path

Source Code (text file) $\rightarrow$ Assembly $\rightarrow$ Machine Code (saved as Binary in SSD)

- **How Compiled Code Works:**

Simple Exeecution Path

SSD (Secondary Memory) $\rightarrow$ RAM (Primary Memory) $\rightarrow$ CPU

- **Analogy:** Python is like a high-level recipe; Assembly is the specific muscle movements the chef makes.
- **Assembly Transmission:** How C or C++ looks like when stripped down to ‘MOV‘, ‘ADD‘, and ‘PUSH‘ instructions.

Under the Hood (Compilation & Execution): How Code Actually Runs

- **The Translation Pipeline:**

The Path

Source Code (text file) → Assembly → Machine Code (saved as Binary in SSD)

- **How Compiled Code Works:**

Simple Exeecution Path

SSD (Secondary Memory) → RAM (Primary Memory) → CPU

- **Analogy:** Python is like a high-level recipe; Assembly is the specific muscle movements the chef makes.
- **Assembly Transmission:** How C or C++ looks like when stripped down to ‘MOV‘, ‘ADD‘, and ‘PUSH‘ instructions.

1. The High-Level Concept (Python)

```python
a = 5
b = 4
c = a + b
print(c) # Simple and readable

```

Listing 1: Python code for sum and print

- **Abstraction:** We don't care about memory addresses.
- **Interpreter:** Python handles the "translation" on the fly.

AMD64 Architectire: Addition + Print (Linux Syscall) I

```
section .bss
    digit resb 2       ; Reserve 2 bytes for the character + newline
section .text
    global _start
_start:
    ; 1. Perform Math
    mov rax, 5
    mov rbx, 4
    add rax, rbx       ; RAX = 9

    ; 2. Convert Integer to ASCII
    ; ASCII '0' is 48. To get '9', we add 48 to the integer 9.
    add rax, 48
    mov [digit], al    ; Store the lower 8 bits (the char) into memory
    mov byte [digit+1], 10 ; Add a newline character (\n)

```

AMD64 Architectire: Addition + Print (Linux Syscall) II

```
    ; 3. Syscall: sys_write (Print to Screen)
    mov rax, 1          ; Syscall number for write
    mov rdi, 1          ; File descriptor (stdout)
    mov rsi, digit      ; Address of our buffer
    mov rdx, 2          ; Number of bytes to print
    syscall

    ; 4. Syscall: sys_exit (Clean Exit)
    mov rax, 60         ; Syscall for exit
    xor rdi, rdi        ; Return code 0
    syscall

```

Listing 2: AMD64 Architecture Assembly Code (what is actually happening inside your laptop)

Intel 8086: Addition + Print (16-bit)

```
MOV AX, 5
MOV BX, 4
ADD AX, BX      ; AX = 9

; --- Printing Logic (Single Character) ---
ADD AX, 48      ; Convert integer 9 to ASCII '9' (48 is '0')
MOV DL, AL      ; Move result to DL (Data Register)
MOV AH, 02h     ; Function 02h of INT 21h is "Display Character"
INT 21h         ; Call DOS interrupt to print

```

Listing 3: Assembly code for Intel-8086 (Technolgy of 1978)

The Final Layer: Assembly vs. Machine Code

Human-Readable (ASM)

```
MOV AX, 5
MOV BX, 4
ADD AX, BX

; Printing Logic
ADD AX, 48
MOV DL, AL
MOV AH, 02h
INT 21h
```

Computer-Readable (Binary)

```
10111000 00000101 00000000
10111011 00000100 00000000
00000001 11011000


00000101 00110000 00000000
10001000 11000010
10110100 00000010
11001101 00100001
```

- Why CPU Architecture matters?
- Why OS Matters?

The Final Layer: Assembly vs. Machine Code

Human-Readable (ASM)

```
MOV AX, 5
MOV BX, 4
ADD AX, BX

; Printing Logic
ADD AX, 48
MOV DL, AL
MOV AH, 02h
INT 21h
```

Computer-Readable (Binary)

```
10111000 00000101 00000000
10111011 00000100 00000000
00000001 11011000


00000101 00110000 00000000
10001000 11000010
10110100 00000010
11001101 00100001
```

- Why CPU Architecture matters?
- Why OS Matters?

Under the Hood (Compilation & Execution): The Life of a Process (Live Demo Idea) I

- **The Infinite Loop:** Write a 3-line Python script that does `while True: pass`.
- **Observation:** Run it and show the CPU spike in Task Manager or `htop`.
- **The "Kill" Command:**
    - Show how to find the **PID** (Process ID).
    - Demonstrate killing it via CLI: `kill -9 <PID>` (Linux/Mac) or `taskkill /F /PID <PID>` (Windows).
- **Lesson:** Code isn't just text; once it runs, it's a living process managed by the OS.

The Gateway (CLI & Shells): Navigation 101 I

- **Where am I?** `pwd` (Linux) or `cd` (Windows - no args).
- **Moving around:**
  - `.` → Current directory.
  - `..` → Parent directory.
  - `cd ..` → Go up one level.
  - `cd ../../` → Go up two levels.
  - `./myscript.sh` → Run something *here*.
- **Paths:**
  - **Absolute:** From the root (`/home/user/doc` or `C:\Users\Doc`).
  - **Relative:** From where you are right now (`../images/photo.jpg`).

The Gateway (CLI & Shells): CLI Parameters & Execution I

- **Arguments/Flags:** Programs aren't just names; they take instructions.
  - `ping 8.8.8.8` (Argument: IP)
  - `ping -h` or `ping --help` (Flag: show help)
- **CLI vs. Double-Click:**
  - Double-click: OS opens a "shell-less" window. If a program just prints and exits, the window disappears instantly!
  - CLI: The shell stays open, so you can see the `stdout` (output) and `stderr` (errors).

The Gateway (CLI & Shells): How GUI Actually Works I

- **The Event Loop:** GUI programs don't just "run and end"; they "listen."
- *"Is the mouse clicked?"* → *"No"* → *"Wait"* → *"Is the mouse clicked?"*
- **Layers:** Hardware → Kernel → Window Manager (X11/Wayland/DWM) → Toolkit (GTK/Qt/Win32) → Your App.

The Gateway (CLI & Shells): The "Cherno" Style Practice I

- **The OS as a Middleman:** Explain that the CLI is just a way to talk to the OS Kernel without a "pretty" UI.
- **Shell vs. Terminal:**
  - **The Shell:** The logic (Bash, Zsh, PowerShell).
  - **The Terminal:** The window that displays it.
- Move through the file system using *only* the keyboard.
- `cd`, `ls` / `dir`, `mkdir`, `rm -rf` (with a warning!).
- **Challenge:** Create a project folder, a virtual env, and a python file without touching the mouse once.

The Gateway (CLI & Shells): Choosing Your Flavor (Multiple CLIs) I

- **Bash:** The industry standard (Linux/macOS).
- **Zsh:** Bash but "prettier" (Default on modern Macs).
- **Fish:** The "friendly" shell with auto-suggestions.
- **PowerShell:** The powerhouse for Windows (Object-oriented).

C/C++: Deep Dive: Compilers I

- **What is a Compiler?** It translates high-level C++ to Machine Code.
- **The "Big Three":**
  - **GCC (GNU):** The Linux standard.
  - **Clang (LLVM):** Great error messages, used by Apple/Google.
  - **MSVC:** The Windows (Visual Studio) standard.
- **CLI Compilation:**
  - `g++ main.cpp -o myapp`
  - `./myapp`

C/C++: Deep Dive: Headers & Preprocessors I

- **What is a Header (.h/.hpp)?**
  - A promise to the compiler: "I will define this function later, just trust me for now."
  - **The Cherno's Guide:** `https://youtu.be/9RJTQmK0YPI` (Watch this!)
- **#include:** Literally "Copy and Paste" the content of that file here.
- **#define:** A search-and-replace tool.

```cpp
#define PI 3.1415
#define LOG(x) std::cout << x << std::endl;

```

C/C++: Deep Dive: Communication with Code (Argc/Argv) I

```c
int main(int argc, char* argv[]) {
    // argc = number of arguments
    // argv = the list of strings
    printf("Program Name: %s\n", argv[0]);
    if (argc > 1) {
        printf("First Argument: %s\n", argv[1]);
    }
    return 0;
}

```

Listing 4: How code sees CLI arguments

C/C++: Deep Dive: Build Tools (Why CMake?) I

- **The Problem:** Compiling 1000 files manually with g++ is impossible.
- **The Solution: CMake / QMake**.
- They don't compile code; they *generate* instructions (Makefiles/Project Files) on how to compile.

CMake Example

```
add_executable(MyApp main.cpp utils.cpp)
```

C/C++: Deep Dive: GUI in C (GTK Example) I

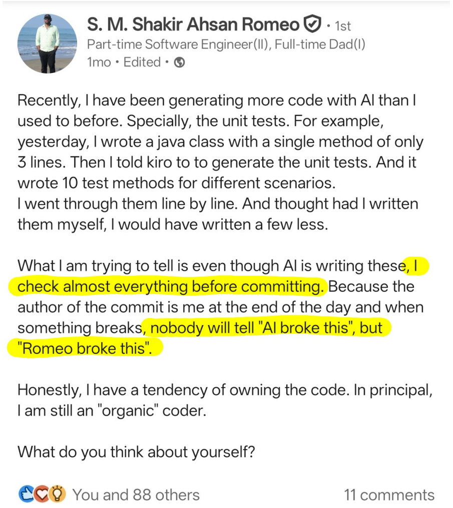
```c
#include <gtk/gtk.h>

int main(int argc, char *argv[]) {
    gtk_init(&argc, &argv);
    GtkWidget *window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "Hello HZBRL!");
    g_signal_connect(window, "destroy", G_CALLBACK(gtk_main_quit),
   NULL);
    gtk_widget_show(window);
    gtk_main();
    return 0;
}

```

Listing 5: Simple GTK Window

C/C++: Deep Dive: GUI in C (GTK Example) II

- **Docs:** `https://docs.gtk.org/`
- Notice how it uses `argc/argv` and an "Event Loop" (`gtk_main()`).

Copilot vs Slave: Myself vs AI I



S. M. Shakir Ahsan Romeo · 1st
Part-time Software Engineer(II), Full-time Dad(I)
1mo • Edited •

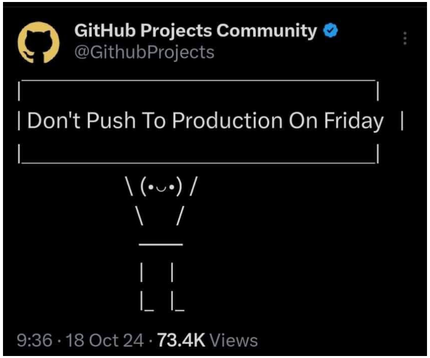
Recently, I have been generating more code with AI than I used to before. Specially, the unit tests. For example, yesterday, I wrote a java class with a single method of only 3 lines. Then I told kiro to to generate the unit tests. And it wrote 10 test methods for different scenarios.
I went through them line by line. And thought had I written them myself, I would have written a few less.

What I am trying to tell is even though AI is writing these, I check almost everything before committing. Because the author of the commit is me at the end of the day and when something breaks, nobody will tell "AI broke this", but "Romeo broke this".

Honestly, I have a tendency of owning the code. In principal, I am still an "organic" coder.

What do you think about yourself?

You and 88 others 11 comments



GitHub Projects Community
@GithubProjects

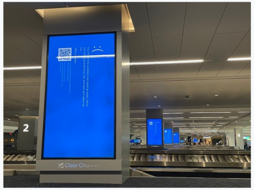
```
 ________________________________
|                                |
| Don't Push To Production On Friday |
|________________________________|
        \ (•◡•) /
         \     /
          ——
          |   |
          |_  |_
```

9:36 · 18 Oct 24 · 73.4K Views

Copilot vs Slave: Myself vs AI II

The "don't push to production on Friday'' rule is a long-standing industry meme, but it gained massive mainstream attention due to the CrowdStrike outage on Friday, July 19, 2024. While the rule existed for years to prevent weekend-ruining bugs, the CrowdStrike incident became the ultimate "cautionary tale'' because:
The Timing: **CrowdStrike pushed a faulty "content update"** early Friday morning (UTC). The Impact: It caused **approximately 8.5 million Windows machines to crash (Blue Screen of Death), disrupting airlines, banks, and hospitals globally.**
The Irony: It reinforced the meme so perfectly that it's now often cited as the primary reason why companies should avoid Friday deployments.

2024 CrowdStrike-related IT outages



Multiple blue screens of death caused by a faulty software update on baggage carousels at LaGuardia Airport, New York City

[1]

The Engine Room (Fundamentals): Structuring Code I

- **What is an Interpreter?** Explain how Python reads code line-by-line vs. all at once.
- **Compiler vs. Interpreter:** Use a simple analogy (Translating a book beforehand vs. having a live translator at a meeting).
- **Python Bytecode (.pyc):** Briefly mention that Python actually compiles to bytecode before interpreting (e.g., `__pycache__`).
- **Built-in Libraries:** What comes "out of the box" (e.g., `math`, `os`, `random`).
- **Packages & External Libraries:** Code written by others (e.g., `requests`, `numpy`).
- **The `import` Statement:**
  - **Import at Top:** Standard practice (readability, dependency checking).
  - **Import inside functions:** When to use it (preventing circular imports or saving memory).

The Ecosystem (Pip & Venv): Virtual Environments (The "Silo" Concept) I

- **What is Pip?** The "App Store" for Python.
- **Requirements.txt:** Crucial for sharing project dependencies with others.
- **Why use them?** Explain "Dependency Hell" (e.g., Project A needs v1.0, Project B needs v2.0).
- **Creating & Activating:**
  - **CLI:** `python -m venv .venv` and `source bin/activate`.
  - **PyCharm UI:** How to point the "Project Interpreter" to a specific environment.
- **Environment Variables & PATH:** Why typing `python` in the terminal sometimes results in "command not found."

Distributions & Specialized Tools: Deployment I

- **Anaconda vs. Miniconda:**
  - **Anaconda:** The "Everything included" (3GB+) version.
  - **Miniconda:** The "DIY" version (Preferred by most pros).
- **Why Conda?** It manages non-python dependencies (like C++ libraries or CUDA for AI) better than Pip.
- **Compiling to Executable:** Mention tools like `PyInstaller` or `Nuitka`.
- **Bundling:** Explain that this "bundles" the interpreter so the user doesn't need to install Python.

The Data Science & AI Workflow: Hardware for AI (The GPU Question) I

- **Jupyter Notebooks:** Explain the `.ipynb` format—running code in "cells" instead of one big script.
- **Google Colab:** "Jupyter in the Cloud." Explain that it's a VM (Virtual Machine) provided by Google.
- **Why GPU is important?**
  - **CPU:** A few smart people doing complex math (Serial).
  - **GPU:** Thousands of average people doing simple math simultaneously (Parallel).
- **Why are they expensive?** High demand for AI, specialized VRAM, and manufacturing complexity.

Infrastructure (Virtual Terminals & Cloud): Virtual Terminals (The "Azure Experience") I

- **What is it?** It's not a physical computer; it's a terminal window connected to a server thousands of miles away.
- **The "Gist":** You are sending text commands over the internet, the remote OS executes them, and sends the text output back to you.
- **Why it matters:** This is how 99% of professional Python code is deployed (Servers, Cloud Functions, Bots).

Environment Management (The Practical Side): Python Environment Deep Dive I

- **Import Logic:** Why `import` goes at the top (readability/PEP8) vs. inside a function (lazy loading/avoiding circular dependencies).
- **The Path Variable:** When you type `python`, the CLI looks through a list of folders (the PATH) to find the executable.
- **Package Managers:** `Pip` (Python specific) vs. `Conda` (System-wide library management).

Image Sources I

Smishra1 — Wikipedia contributors.
Crowdstrike bsod at lga, 2024.
URL: https://en.wikipedia.org/wiki/File:CrowdStrike_BSOD_at_LGA.jpg.

PlanetF1.
George russell leads leaving pits hungary, 2021.
URL: https://www.planetf1.com/news/george-russell-surprised-hungary-tears.