LAB NOTEBOOK

SWASTI KUMARI

Summer internship

Beginning 26 May 26, 2025

Monday, May 26, 2025

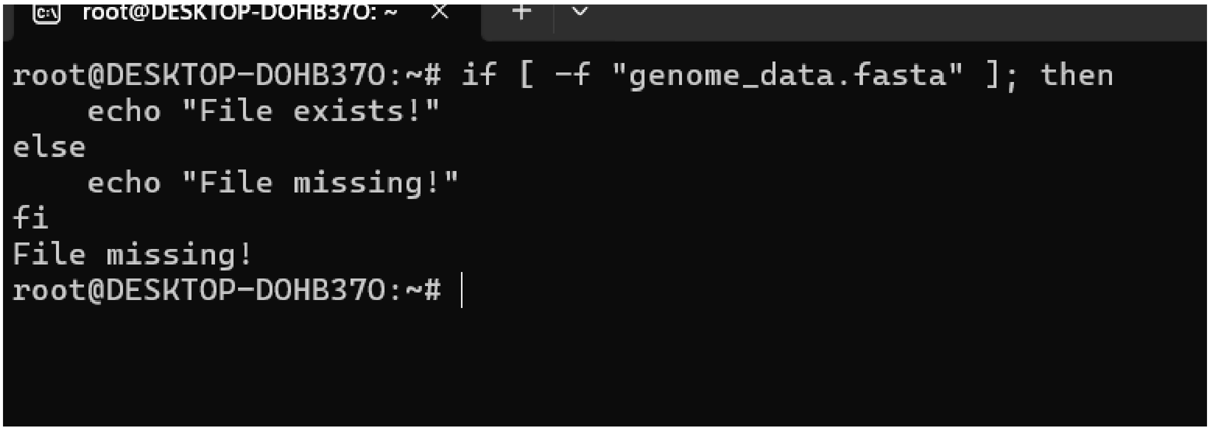
Chapter 5. Flow Control

Topic: Flow Control- if-else, return, and exit

Today, I explored key flow control concepts in Bash scripting that help automate tasks and manage execution behaviour efficiently.

1. if-else Statements

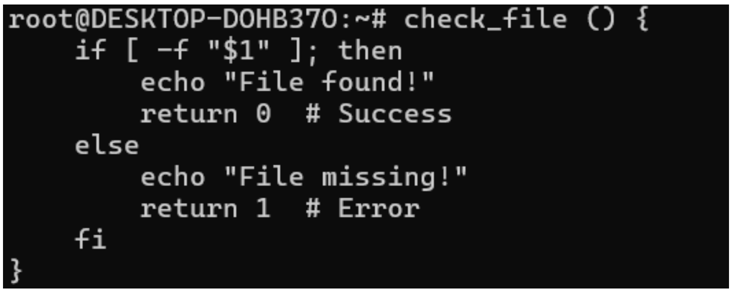
- The if statement allows conditional execution based on whether an expression is **true (exit status 0)** or **false (exit status ≠ 0)**.
- **Example: Checking if a file exists**



```
root@DESKTOP-DOHB37O:~# if [ -f "genome_data.fasta" ]; then
    echo "File exists!"
else
    echo "File missing!"
fi
File missing!
root@DESKTOP-DOHB37O:~#
```

2. return Statement in Functions

- Used **only inside functions** to exit and pass an exit status.
- Syntax: return N (where N is an integer between 0-255).
- Example:



```
root@DESKTOP-DOHB37O:~# check_file () {
    if [ -f "$1" ]; then
        echo "File found!"
        return 0  # Success
    else
        echo "File missing!"
        return 1  # Error
    fi
}
```

3. exit Statement

- Terminates **the entire script**, unlike return, which only exits a function.
- Syntax: exit N (similar to return).

The `exit` command in Bash is used with the syntax, `exit [n]`. Its function is to terminate a script and return a value to the parent script or shell. It's a way to signal the end of a script's execution and optionally return a status code to the calling process.

- **#!/bin/bash**
- **echo 'Hello, World!'**
- **exit 0**
- In this example, we've created a simple Bash script that prints 'Hello, World!' to the console and then terminates with an exit status of 0. The `exit 0` command signals successful execution of the script
.

Tuesday, 27 May 2025

1. Retrieval and Preparation of FASTA Sequence

- Downloaded a **FASTA file** from **NCBI** for genomic analysis.
- Verified the file location using ls in the terminal.
- Navigated to the **Downloads** directory to ensure correct execution.

2. Development and Execution of Bash Script

1. Created a Bash script named **fasta_analysis.sh** to automate sequence analysis.
2. Implemented **loops** (while read -r line) for efficient line-by-line processing.
3. Extracted sequence identifiers (headers) using conditional statements.
4. Computed the **total sequence length** using ${#sequence}.
5. Calculated the **GC percentage** by counting occurrences of G and C bases:

3. Explored exit statuses ($?), understand and performed how commands signal success or failure.

4. Practiced condition testing (-eq, -ne, -lt, -gt) for numerical comparisons.

5. Used string comparisons (=, !=, -z, -n) to validate file extensions and sequence headers.

Wednesday, 28 May 2025

1. File Attribute Checks:

- Practiced using file test operators:
    - -f → check if a file exists.
    - if [-f "$file"] → conditional used to verify file presence before proceeding with analysis.

2. Integer Conditionals:

- Used -gt, -lt, -eq etc. for numeric comparisons.
- Example:

```
if [ "$length" -gt 100 ]; then

  echo "Sequence is longer than 100 bp"
```

3. String Conditionals:

- Learned how to compare strings:

    - = for equality, != for inequality.
    - Lexicographical comparisons using <, > (escaped as \< and \> in [ ] brackets).
- Example:

```
if [ "$gene" = "WNT2" ]; then
echo "Gene of interest: WNT2"
```

4. FASTA File Handling:

- Checked FASTA format using head -n 1 and pattern match >.
- Removed FASTA headers using: grep -v "^>"
- Used tr -d '\n' to convert multi-line sequence into a single line.

5. Mini Projects & Scripts:

Created and executed the following bash scripts:

- **check_fasta.sh –** Validates if a file is in FASTA format.
- **check_length.sh –** Checks if sequence length exceeds 100 base pairs.
- **gc_content.sh –** Calculates GC content of a sequence.

Each script used:

- Conditional logic (if-else)
- Commands like grep, wc, tr, echo, and head.

6. Project Contribution:

- Contributed to a **WNT2 gene** bioinformatics project.

Thursday, 29 May 2025

1. Completed **Chapter 5**, which covered control flow structures including:
    - for loop
    - case statements
    - while loop
    - until loop
2. Practiced and implemented the above loops using **FASTA sequences** for hands-on application.
3. Wrote and executed scripts to process sequence data and understand the use of different loop types in bioinformatics tasks.

Friday, 30 May 2025

1. **Revised** Chapters 1 to 5, which included:

- **Chapter 1:** Introduction to Shell Scripting
- **Chapter 2:** Variables and Data Types
- **Chapter 3:** Operators
- **Chapter 4:** Conditional Statements
- **Chapter 5:** Loops (for, while, until, case)

2. **Shot and recorded explanatory videos for:**
   Chapter 1 and Chapter 2.

Saturday, 31 May 2025

1. **Recorded chapter 3 to 5**

Monday, 2 June 2025

1. **Started reading chapter 6**

   Topic Covered:

   1. **Command-line options**
   2. **Shift command**
   3. **Options with arguments**
   4. **getopts command**
   5. **Typed variables**

1. Command-line Options

- Learned how shell scripts can accept **positional parameters** like $1, $2, etc.
- Understood how to pass arguments while running a script:
- ./myscript.sh arg1 arg2
- Used $@ and $# to get all arguments and their count.

2. Shift Command

- Learned how shift is used to shift positional parameters to the left.
- Helpful in looping through all arguments without manually referencing $1, $2, etc.

3. Options with Arguments

- Understood the difference between options (like -a, -b) and arguments (like filenames or values).
- Implemented scripts that check options using conditions, for example:

4. getopts Command

- Studied getopts as a more professional way to parse options.
- Handles both short options and their arguments.

5. Typed Variables

- Explored how to declare typed variables using declare or typeset.

Tuesday, 3 June 2025

Topics Covered:

1. **Typed Variables**
   - Studied how to declare variables with specific types using declare and typeset.
   - Learned about read-only and integer typed variables.
2. **Integer Variables and Arithmetic**
   - Covered how to perform basic arithmetic operations in shell scripting.
   - Explored different methods for integer calculations.
3. **Arithmetic Conditionals**
   - Learned how to use arithmetic expressions within if and conditional statements.
   - Covered comparison operators used in arithmetic conditions.
4. **Arithmetic Variables and Assignment**
   - Studied assigning arithmetic results to variables.
   - Covered updating variables using arithmetic operations.
5. **Arithmetic for Loops**
   - Learned how to use arithmetic expressions in for loops.
   - Studied the structure and usage of C-style arithmetic loops in Bash.

.

Wednesday, 4 June 2025

Topics Covered:

1. **Arrays**
    - Learned how to declare and use arrays in Bash.
    - Studied how to access individual elements, loop through arrays, and modify them.
    - Covered both indexed and associative arrays.
2. **Started Working on FASTQ Analyser**
    - Began working on a script/tool to analyze FASTQ files (commonly used in bioinformatics).
    - Gained understanding of FASTQ file structure (sequence identifier, raw sequence, optional line, and quality scores).
    - Started initial planning and implementation of data reading and basic statistics.

Thursday, 5 June 2025

1. **Worked on FASTQ Analyser**
    - Continued development of the FASTQ file analyzer script.
    - Implemented logic for reading and parsing FASTQ files.
    - Extracted and processed sequence and quality data.
2. **Completed FASTQ Analyser**
    - Finalized the script with complete functionality.
    - Script now calculates basic statistics such as total reads, sequence lengths, and average quality scores.
    - Tested the script on sample FASTQ files for validation.
3. **Started Chapter 7**
    - Began reading and understanding the concepts introduced in Chapter 7.
    - Covered the initial sections and noted key points for further exploration.

Friday, 6 June 2025

Topics Covered:

1. **Chapter 7: Input/Output and Command-Line Processing**
    - Studied the basics of how Bash handles input and output streams.
    - Understood standard input (stdin), standard output (stdout), and standard error (stderr).
2. **I/O Redirectors**
    - Learned how to redirect input and output using operators like >, >>, <, 2>, and &>.
    - Practiced redirecting output to files and combining standard output and error.
3. **Here-documents**
    - Explored how to use here-documents (<<) to provide multiline input directly within scripts.
    - Useful for feeding blocks of text into commands or files.
4. **File Descriptors**
    - Studied file descriptor numbers: 0 (stdin), 1 (stdout), and 2 (stderr).
    - Learned how to manage file descriptors for advanced redirection and custom input/output handling.

Saturday, 7 June 2025

1. Recorded video of chapter 6

Monday, 9 June 2025

1. **String I/O**
   - Learned how Bash handles input and output of strings.
   - Practiced using various commands to manipulate and display string data.
2. **echo Command**
   - Used echo to display messages and variables to the terminal.
   - Simple and commonly used for output in scripts.
3. **Options to echo**
   - Studied commonly used options like -n (no newline) and -e (enable interpretation of escape characters).
4. **Echo Escape Sequences**
   - Learned to use escape sequences such as \n, \t, \\, \" for formatting output.
   - Used in combination with -e option.
5. **printf Command**
   - Explored printf for more controlled and formatted output than echo.
   - Studied syntax and format specifiers like %s, %d, %f.
6. **Additional Bash printf Specifiers**
   - Learned extra specifiers such as %x (hex), %o (octal), and how to control field width and precision.
   - Useful for creating neatly formatted reports or data output.
7. **read Command**
   - Practiced using read to take user input from the terminal or from files.
   - Used options like -p for prompting and -a for reading into arrays.
8. **I/O Redirection and Multiple Commands**
   - Learned how to redirect input/output across multiple commands using pipes (|) and file descriptors.
   - Practiced combining read, echo, and printf with redirection techniques.

Tuesday, 10 June 2025

Topics Covered:

1. **Command Blocks**
   - Learned to group multiple commands using curly braces {} or parentheses ().
   - {} runs in the current shell, () runs in a subshell.
   - Useful for redirection, function definitions, and logical grouping.
2. **Reading User Input**
   - Practiced different ways to take input from users during script execution using read.
   - Covered handling multiple variables and default values.
3. **Command-Line Processing**
   - Studied how Bash processes commands in multiple steps: tokenizing, parsing, expansion, and execution.
   - Understood how quoting and variable expansion affect command behavior.
4. **Quoting**
   - Learned the difference between single quotes ' ', double quotes " ", and backticks ` `.
   - Used quotes to prevent unwanted expansion or preserve whitespace.
5. **command, builtin, and enable**

    - command: Runs an external command even if a function or alias exists with the same name.
    - builtin: Forces Bash to run a built-in version of a command.
    - enable: Used to enable or disable Bash built-ins.
6. **eval Command**
    - Studied how eval takes a string and evaluates it as a command.
    - Useful for dynamically building and executing commands at runtime.
    - Also learned the risks of using eval due to security concerns if input is not sanitized.

Wednesday, 11 June 2025

Topics Covered:

1. **Started Chapter 8: Process Handling and Job Control**
    - Began exploring how Bash handles processes and background jobs.
    - Focus on understanding PIDs, job numbers, and job management commands.
2. **Process IDs and Job Numbers**
    - Learned how each running process has a unique Process ID (PID).
    - Understood that background jobs are assigned Job Numbers (e.g., [1], [2]).
    - Used ps, echo $$, and jobs to view current shell PID and job statuses.
3. **Job Control**
    - Studied how to run commands in the background using &.
    - Used fg, bg, and jobs to manage foreground and background processes.
    - Understood how Bash tracks and displays job states (Running, Stopped, Done).
4. **Suspending a Job**
    - Learned how to suspend a foreground job using Ctrl+Z.
    - Observed how the shell moves the job to a stopped state.
    - Practiced resuming suspended jobs in background with bg or bringing them back to foreground with fg.

Thursday, 12 June 2025

Topics Covered:

1. **Signals**
    - Learned what signals are in UNIX/Linux (software interrupts sent to processes).
    - Common signals include SIGINT, SIGTERM, SIGKILL, SIGHUP, etc.
    - Understood how signals can terminate, pause, or restart processes.
2. **Control-Key Signals**
    - Practiced using key combinations to send signals from the keyboard:
        - Ctrl+C: Sends SIGINT (Interrupt)
        - Ctrl+Z: Sends SIGTSTP (Suspend)
        - Ctrl+\: Sends SIGQUIT (Quit with core dump)
    - Learned how these signals affect foreground processes.
3. **kill Command**
    - Used kill to send signals to processes using their PID.
    - Example: kill -9 PID to forcefully terminate a process.
    - Also used kill %jobnumber to signal background/suspended jobs.
4. **ps Command**
    - Studied how to list active processes using ps.
    - Learned about various options to format output and see details like PID, TTY, status, etc.
5. **System V Format**
    - Used ps -e or ps -f (System V-style options).

    - Showed full listing with PID, PPID, UID, command name, and more.

6. **BSD Format**
    - Used ps ax or ps aux (BSD-style options).
    - Provided a detailed snapshot of all processes including those without terminals.
    - Compared output differences with System V format.

Friday, 13 June 2025

Topics Covered:

1. **trap Command**
    - Learned how to use trap to catch and respond to signals in shell scripts.
    - Used for cleanup tasks or custom handling of signals like EXIT, INT, TERM.
2. **Traps and Functions**
    - Explored how trap interacts with functions.
    - Understood that traps set outside a function apply to it, but can also be customized inside functions.
    - Studied behavior with nested functions and signal inheritance.
3. **Process ID Variables and Temporary Files**
    - Used special variables like $$ (current script PID) and $! (last background PID).
    - Created and safely cleaned up temporary files using trap to ensure deletion on exit or interrupt.
4. **Ignoring Signals**
    - Learned how to ignore specific signals using trap '' SIGNAL.
    - Example: trap '' INT to prevent Ctrl+C from interrupting the script.
5. **disown Command**
    - Used disown to remove jobs from the shell's job table.
    - Prevents background jobs from receiving SIGHUP when the shell exits.
6. **Resetting Traps**
    - Reset traps to default behavior using trap - SIGNAL.
    - Example: trap - INT re-enables default handling for Ctrl+C.

Saturday, 14 June 2025

Topics Covered:

1. **Coroutines**
    - Learned how coroutines allow concurrent execution by using background processes and synchronization.
    - Used pipelines and FIFOs to coordinate between multiple running functions.
    - Useful for building cooperative multitasking systems in shell.
2. **wait Command**
    - Studied how wait pauses the script until background processes finish.
    - Used wait with or without a specific PID to synchronize background tasks.
3. **Advantages and Disadvantages of Coroutines**
    - **Advantages:** Efficient multitasking, better CPU usage, improved performance for I/O-heavy scripts.
    - **Disadvantages:** Harder to debug, risk of race conditions, more complex code structure.
4. **Parallelization**
    - Learned how to run multiple commands or functions in parallel using &.

   - Combined with wait to synchronize and optimize time-heavy tasks like data processing.
5. **Subshells**
   - Studied how () creates a subshell—an independent environment from the current shell.
   - Changes inside do not affect the parent shell.
6. **Nested Subshells**
   - Practiced using subshells within subshells.
   - Useful for isolating blocks of code or running temporary environments without affecting the outer shell.
7. **Process Substitution**
   - Used <(command) and >(command) to pass the output/input of a command as a file-like argument.
   - Very useful when working with tools that expect file names but we want to use live command output.

Sunday, 15 June 2025

Topics Covered:

1. **Chapter 11: Shell Scripting**
   - Learned how to write effective and maintainable shell scripts.
   - Focused on proper structure with script headers, comments, and indentation.
   - Studied the importance of naming conventions, consistent formatting, and modular code using functions.
   - Covered the use of #!/bin/bash shebang, setting executable permissions, and documentation within scripts.
2. **Chapter 12: Bash for Your System**
   - Understood how to obtain, configure, build, and install a custom version of Bash from source code.
   - Explored the contents of the Bash source archive.
   - Studied how to use the configure script to enable or disable specific features during installation.
   - Learned the full compilation and installation process using make and make install.
   - Covered verifying the Bash version and switching to the newly installed version if needed.

Monday, 16 June 2025

Topics Covered:

1. **Started Chapter 9: Debugging Shell Programs**
   - Began learning techniques to identify and fix bugs in Bash scripts.
   - Focused on tools and features provided by Bash to aid in debugging.
2. **Basic Debugging Aids**
   - Learned how to enable shell debugging using set -x and set -v.
   - set -x: Displays commands and their arguments as they are executed.
   - set -v: Shows the script content line by line before execution.
   - Useful for tracing flow and identifying unexpected behavior.
3. **Fake Signals**
   - Studied Bash's fake signals like EXIT, ERR, DEBUG, and RETURN.
   - These are not real UNIX signals but can be trapped using the trap command for custom actions.
   - Example: trap 'echo "Exiting..."' EXIT — runs when the script exits.
4. **ERR Trap**

- Learned how the ERR trap is triggered when a command returns a non-zero exit status.
- Used to catch and handle errors during script execution.
- Helpful in debugging scripts without exiting on each error.
- Accessed exit status using $? inside the trap function.