**~Chapter 1: Bash Basics~**

#### ***$INTRODUCTION:-***

**#Background of UNIX and Bash**

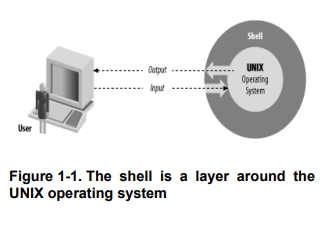
* UNIX has been around since the early 1970s and has evolved into many variants like Ultrix, AIX, Xenix, SunOS, Linux.
* Originally limited to mainframes and academics, UNIX now powers desktop, personal, and server systems globally.
* UNIX is foundational in computing education, businesses, and research.

**#Understanding What Happens When You Type a Command**

* Typing a command and hitting RETURN initiates a series of processes.
* This interaction is handled by a shell, a program that acts as a bridge between the user and the UNIX operating system.

**#The Shell Layer**

* The **shell** processes the input from the user, converts it into instructions, sends them to the OS, and displays the output back to the user.
* This is illustrated in **Figure 1-1**:  
   ➤ User → Shell → UNIX OS → Shell → User



#### 

**#Types of Interfaces**

* **Character-based interfaces (CLI)**:  
  + Bash is one.
  + Accepts typed text input and returns text output.
* **Graphical User Interfaces (GUI)**:
  + Common in desktops, supports mouse/touchscreen inputs and graphics.
* **CLI vs GUI**:
  + CLI is **lightweight, scriptable, and powerful** — especially important for repetitive or automated tasks.

| Feature | CLI (Command Line Interface) | GUI (Graphical User Interface) |
| --- | --- | --- |
| Interface Type | Text-based | Graphical (icons, windows, buttons) |
| Ease of Use | Requires memorization of commands | Intuitive and easier for beginners |
| Speed | Faster for experienced users | Slower due to visual navigation |
| Automation | Easily scriptable for repetitive tasks | Limited scripting capabilities |
| Resource Usage | Lightweight, consumes minimal system resources | Heavier, requires more memory and processing power |
| Flexibility | Highly flexible and customizable | Less flexible; depends on provided GUI options |
| Learning Curve | Steeper; needs command knowledge | Gentle; mostly point-and-click |
| Usage in Bioinformatics | Preferred for processing large datasets and running pipelines | Used mainly for visualization or simple tools |

**#Additional- Applications/Relevance of Bash to Computational Biology:-**

1. Automating Pipelines
   * Bioinformatics tasks like genome assembly, alignment (e.g., using Bowtie, BWA), and variant calling are often run via bash scripts.
2. Handling Large Datasets
   * Bash enables filtering, transforming, and analyzing large text-based data (like FASTA/FASTQ files) efficiently.
3. Reproducibility
   * Scripts ensure that analyses can be replicated by others — a core requirement in research.
4. Integration with Tools  
   * Bash can integrate tools written in Python, R, Perl, etc., making it a glue language for multi-tool workflows.
5. High-Performance Computing (HPC)
   * Most bioinformatics pipelines on HPC clusters rely on bash for job submission and resource management (e.g., via SLURM).

***$SHELL:-***

* The shell translates user command lines into instructions for the operating system.
* The shell processes commands through several logical steps:
  + Such as Parsing, interpreting, executing, and handling I/O.

**Characteristics of a Shell in detail**

1. Command Interpreter

Reads and interprets commands typed by the user.

1. Scriptable
   * Can execute a list of commands written in a file (shell script).
2. Supports Variables
   * Allows storing and using data via variables (e.g., MY\_NAME="Anita").
3. I/O Redirection
   * Can redirect input/output to and from files or other commands.
4. Pipeline Processing
   * Supports combining commands using pipes (|) to pass data between commands.
5. Interactive & Non-Interactive Modes
   * Can be used live in a terminal or run batch jobs from a script.
6. Environment Customization
   * Supports environment variables and aliases to customize user sessions.
7. Job Control
   * Allows background and foreground process handling (e.g., &, fg, bg).
8. User-Friendly Enhancements
   * Modern shells like bash support command history, tab completion, and command-line editing.
9. Portability
   * Shell scripts can run across different UNIX/Linux systems with little or no change.
10. Error Handling
    * Shells provide mechanisms like exit codes, trap, and set -e for managing errors.
11. Security Considerations
    * Can restrict operations via permissions and sandboxing (especially in multi-user systems).

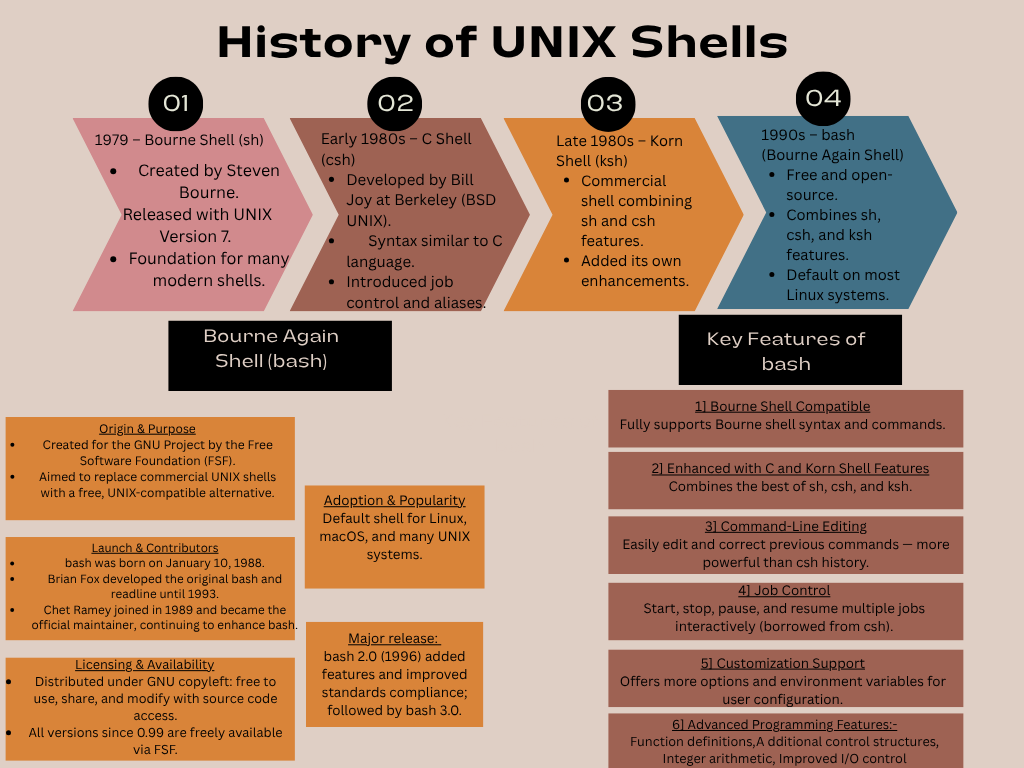
| **Step** | **Action** |
| --- | --- |
| 1 | **Breaks down** the command line into parts: sort, -n, phonelist, >, phonelist.sorted — called **words**. |
| 2 | **Identifies roles**: - `sort` = command  - `-n`, `phonelist` = arguments  - `>` and `phonelist.sorted` = \*\*I/O redirection\*\*. |

3 **Sets up I/O**: Redirects output to phonelist.sorted using >.

4 **Locates** the sort command and **executes** it with the given option and argument.

### **Concepts**

* Each shell action involves **substeps** handled by the operating system
* The **shell ≠ UNIX**:
  + Shell is the **interface**, not the OS itself
  + UNIX introduced the concept of separating the **user interface from the core OS** — a key software engineering principle.



**#Getting bash**

**~SAMPLE CODE~ (TERMINAL)**

**That different shells exist**

* A **shell** is a program that lets you interact with the operating system — you type commands, it runs them.
* Different types of shells were created over time, each with unique features:
  + **sh (Bourne shell)** → The original standard shell; basic scripting features.
  + **csh (C shell)** → Adds C-like syntax for scripting, with features like command history.
  + **ksh (Korn shell)** → Combines features of sh and csh; has more scripting power.
  + **bash (Bourne Again shell)** → The most popular today; powerful scripting, command-line editing, history, tab completion.

Why bash is better or significance of bash:

* bash is widely used on Linux and macOS systems → considered the standard shell for scripting and command-line work.
* It offers:
  + Command history (use ↑ ↓ to recall commands)
  + Auto-completion with the Tab key
  + Easy scripting with control structures (if, for, while)
  + Better error messages and debugging tools
  + Compatibility with most shell scripts designed for sh

**#Interactive shell use**

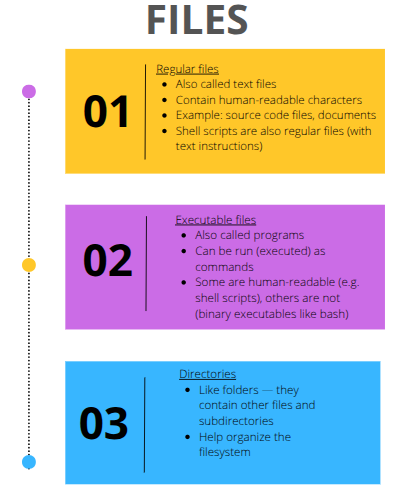
🔹 **Commands, Arguments, and Options**

A command line consists of:  
 1️⃣ The command (first word)  
 2️⃣ Arguments (optional, give the command more info — e.g., filenames, usernames)  
 3️⃣ Options (optional, usually start with a -, modify how the command works)

### **Examples:**

* lp myfile  
   → lp = command (print a file)  
   → myfile = argument (name of the file to print)
* lp -h myfile  
   → -h = option (do not print banner page)  
   → myfile = argument (file to print)
* lp -d lp1 -h myfile  
   → -d lp1 = option + its argument (-d means send to printer lp1)  
   → -h = option (no banner page)  
   → myfile = argument (file to print)

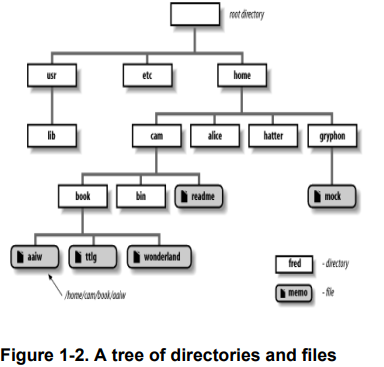
**#Files**

****

**1]Directories:**

### **Directories**

* Directories can contain other directories → This creates a hierarchical structure (tree).
* The tree starts at the root directory (/), which has no name.
* Rectangles in the diagram represent directories; ovals represent files.
* The full (absolute) pathname gives the location of a file/directory starting from root.



The working directory

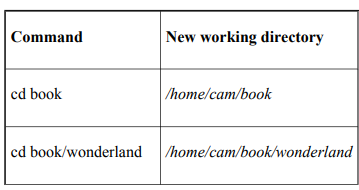
* The working directory is the directory you’re “in” at any moment.
* Instead of full pathnames, you can use relative pathnames (relative to working directory).
* When you log in:
  + Your working directory is usually your home directory (or login directory).

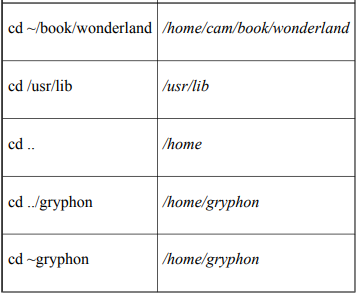
Tilde Notation

* The tilde ~ is used in Bash to abbreviate home directories.
* ~username → Refers to that user's home directory.
* ~ alone → Refers to your own home directory.  
   Example: ~/notes refers to notes file in your home directory.
* This is always an absolute pathname → works regardless of your current (working) directory.
* Useful when working outside your home directory (e.g., in /tmp).

🔹 Basic commands

* cd is used to change your working directory.
* pwd displays your current working directory.
* If cd is used without any argument → it changes to your home directory (same as cd ~).





**Filenames & ls command**

* ls lists file names in the working directory.
* By default, ls does not show hidden files (files starting with a dot .).
* ls can take filename arguments or options:
  + ls -l → long listing (shows file owner, size, modification time, etc.)
  + ls -a → lists all files, including hidden files.

Table 1-3. Using the \* wildcard

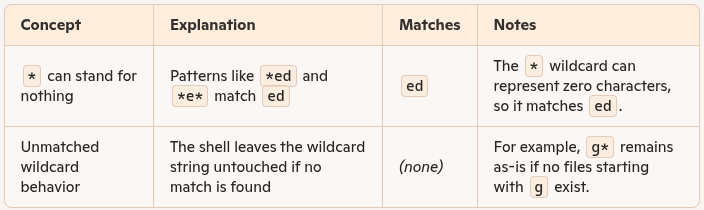


Table 1-2. Basic wildcards

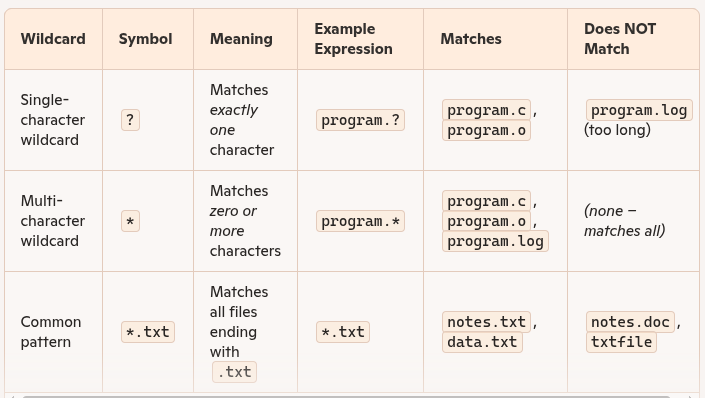
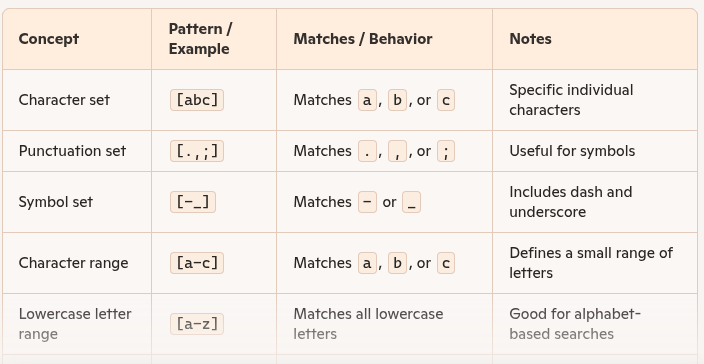
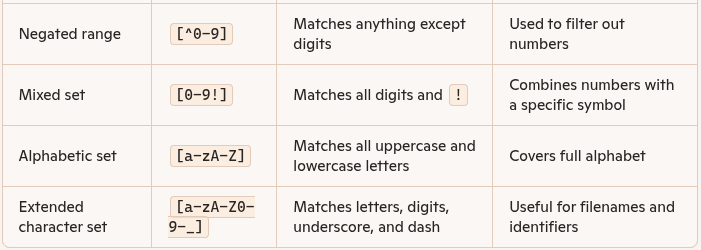


Table 1-4. Using the set construct wildcards





**Key Points on Wildcards and Shell Behavior**

| Concept | Explanation / Details |
| --- | --- |
| Wildcard Expansion Happens in Shell | – The shell expands wildcards before the command runs. The command only sees the final list of arguments. |
| Example: ls fr\* | - If files fred and frank exist, ls fr\* is expanded to ls fred frank. |
| No Match = Literal Argument | - If no match is found (e.g., ls g\*), the command gets the literal g\*, and an error is shown: g\*: No such file or directory. |
| Character Set Wildcard: [cho] | \*.[cho] matches files ending in .c, .h, or .o. Useful for grouping C source, header, and object files. |
| Wildcard Use = No Filename Knowledge Needed | Using wildcards removes the need to manually know or list specific filenames. |
| Pathname Expansion | Wildcards can be used not just on filenames, but as part of full directory paths. |
| Example: ls /usr\* | Lists files in /usr, /usr2, etc. — all directories starting with /usr. |
| Example: ls /usr\*/[be]\* | Lists files in /usr, /usr2, etc., that begin with b or e. |

Key Points: Brace Expansion in Bash:-

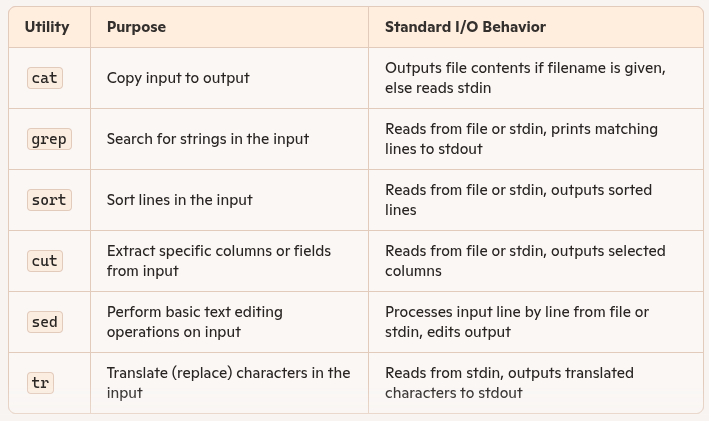
| **Topic** | **Key Details** |
| --- | --- |
| What is Brace Expansion? | A feature for generating arbitrary strings, not dependent on existing files or directories. |
| Basic Syntax | {item1,item2,...} used with optional prefix and suffix. |
| Example | echo b{ed,olt,ar}s expands to beds bolts bars. |
| Not File-Based | Unlike wildcards (\* or ?), brace expansion doesn't check for existing filenames—it just builds strings. |
| Nesting | Braces can be nested: b{ar{d,n,k},ed}s → bards barns barks beds. |
| Sequences with Braces | {2..5} → 2 3 4 5; {d..h} → d e f g h. |
| Combining with Wildcards | Can be used with wildcards: ls \*.{c,h,o} matches files ending in .c, .h, or .o. |
| Bash Version Requirement | Sequence-style brace expansion ({1..5}) is available only in Bash 3.0+. |
| Comma Requirement | Bash requires at least one unquoted comma for expansion to occur; otherwise, braces are treated as literal text. |
| Difference from Wildcards | Wildcards match existing files; brace expansion creates a list of static strings for the shell. |

**Key Points: UNIX Input and Output**

* UNIX I/O is a major innovation with a simple concept and wide impact.
* Data is handled as a sequence of characters (bytes), regardless of length.
* No complex structures like blocks, records, or card images are required.
* Everything is treated as a file—including hardware devices like terminals and disk drives.
* This unified I/O model simplifies programming and system design.
* Older systems used device-specific methods; UNIX uses a universal, consistent approach

**Key Points: Standard I/O in UNIX**

* Standard Input (stdin): Default input source, usually the keyboard.
* Standard Output (stdout): Default output destination, usually the screen/window.
* Standard Error (stderr): Used for error messages, also directed to the screen by default.
* Each UNIX program has one stdin, one stdout, and one stderr by default.
* Designed specifically for interactive terminal use, not old-style batch processing.
* The shell sets up standard I/O channels automatically when a program runs.
* Redirection allows input/output to come from or go to files instead of the terminal.
  + Example: mail < message.txt sends contents of a file instead of typing input manually.
* Pipelines (|) connect programs by sending one program’s output directly to another’s input.



**Important Key Points: I/O Redirection**

* Redirection changes the source of input or the destination of output for a command.

🔹 Input Redirection (<)

* Syntax: command < filename
* Redirects standard input from a file instead of the keyboard.
* Example: cat < cheshire prints contents of cheshire to screen.

🔹 Output Redirection (>)

* Syntax: command > filename
* Redirects standard output to a file instead of the screen.
* Example: date > now saves the current date and time to a file named now.

🔹 Combined Redirection

* You can use both input and output redirection together.
* Example: cat < file1 > file2 copies contents of file1 into file2 (similar to cp file1 file2).

🔹 Conceptual Understanding

* Treat commands as if they only read from stdin and write to stdout.
* Redirection is handled by the shell, not by the command itself.

**#piplines  
- TERMINAL [SAMPLE CODE]-**

**#bacground Jobs:**

-> Background jobs in UNIX are commands or processes that run without occupying the terminal, allowing the user to continue using the shell for other tasks. They're especially useful for long-running commands.

1. UNIX allows running multiple programs at once (multitasking).
2. Add & to run a command in the **background** (e.g., uncompress gcc.tar &).
3. Background jobs free up the terminal for other tasks.
4. Shell shows job ID and process ID when a background job starts (e.g., [1] 175).
5. Use jobs to check running background jobs.
6. When a background job finishes, a message like [1]+ Done appears.
7. Useful for long tasks that don’t need user input.

**Background Jobs and Priorities** – Background jobs save time by allowing multiple tasks to run concurrently.

1. Running many background jobs can slow down performance slightly due to shared system resources like CPU and memory.
2. All jobs have a priority, which determines how much CPU time they get.
   * Lower priority number = higher execution priority
   * Higher priority number = lower execution priority
3. User-launched jobs (foreground or background) typically run at the same standard priority.
4. System administrators (root users) can assign higher priority to certain jobs.
5. On multiuser systems, running many background jobs can hog resources, so:
   * Be mindful of other users.
   * Don't overuse system power.
6. Use the nice command to lower a job’s priority:  
   Syntax: nice command
   * Helps reduce impact on system performance and other users.
7. You can specify how "nice" a job is by adding a numeric value:
   * Example: nice -n 10 command
   * Check details using: man nice
8. Complex commands used with nice should be quoted to prevent misinterpretation by the shell
9. Only the root user can use nice to increase job priority (i.e., make it less nice).

**#quoting**

**-Terminal [sample code]-**

1. Quoting means treating special characters literally, without their shell-defined meanings.
2. Single quotes (') remove all special meaning from the characters inside them.
3. Use quoting when a command line includes special characters like \*, >, |, &, etc.

**Backslash-Escaping in UNIX Shell**

1. Backslash (\) escapes a character, meaning it removes its special meaning.
2. **Escaping a character** with \ is a form of quoting, similar to using single quotes.
3. **Backslash-escaping is useful for quoting individual characters** rather than whole strings.To print a **literal backslash**, either: i)Escape it: \\ ii)Or quote it: '\''

**Continuing Lines:- Two main methods to continue a line:**

1. End the line with a backslash (\) — no space or tab after it.
2. Use an open quote (' or ") without closing it — RETURN becomes part of the string.

🔹 What Are Control Keys?

* Typed by holding **CTRL** and pressing another key.
* Usually do **not display characters**, but perform special functions.
* **System-dependent**: behavior can vary across UNIX versions.

🔹 Commonly Used Control Keys

* CTRL-C (^C) – Interrupts and stops the current running command.
* CTRL-D (^D) – Signals end of input (EOF); used to finish input in programs like mail.
* CTRL-\ (^\\) – Forcefully stops a command (used if CTRL-C fails).
* CTRL-S (^S) – Pauses output to the terminal (flow control).
* CTRL-Q (^Q) – Resumes output after being paused by CTRL-S.
* DEL / CTRL-? – Deletes the last character (backspace).
* CTRL-U (^U) – Erases the entire command line.
* CTRL-Z (^Z) – Suspends the current job/process (can be resumed with fg).

🔹 What It Does

* help provides online documentation for bash built-in commands.
* Unique to bash — not found in other shells.

🔹 Usage

* help — Lists all built-in bash commands and their options.
* help <command> — Shows detailed usage info for a specific built-in command.
  + Example: help cd

🔹 Advanced Help Features

* Partial names allowed:
  + help re shows help for read, readonly, and return.
* Wildcards supported (must be quoted!):
* Use with more for paged output:
  + Example: help cd | more