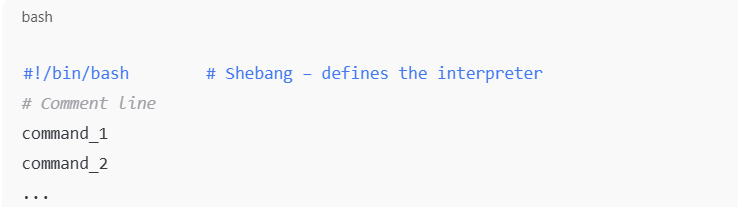
***CHAPTER 4 : BASIC SHELL PROGRAMMING***

**4.1 SHELL SCRIPT**

A shell script is a program written for the shell, a command-line interpreter that provides a user interface for Unix/Linux. Shell scripts are typically used to:

* **Automate repetitive tasks** Tasks that are performed frequently can be written once in a script and reused multiple times, improving efficiency.
* **Perform system administration (backups, monitoring)** Shell scripts are essential for maintaining and monitoring systems, handling log files, checking system health, and performing backups.
* **Combine and execute multiple shell commands** Rather than executing each command one at a time, a shell script allows users to bundle them and run as a single process.
* **Handle conditional logic and looping over files or inputs** Shell scripts support decision-making (if-else), loops (for, while), and input handling, making them powerful tools for dynamic task execution.

**Structure of a Shell Script:**



#### Shebang (#!)

* First line of a script, tells the OS which interpreter to use.
* Example: #!/bin/bash for Bash shell.

Example: Hello Script

#!/bin/bash

# A simple script

echo "Hello Anusha , WSL Bash script is working!"

**Example: Script with Variables and Input  
#!/bin/bash**

**echo "Enter your name:"**

**read name**

**echo "Good morning, $name!"**

Key Points: Working with Bash Functions

1. Two Ways to Define Functions
   * There is no functional difference between the two syntax styles.
   * Both forms are valid and used interchangeably.
2. Deleting Functions

Use the command:  
unset -f function\_name

* + This removes the function definition from memory.

1. Function Execution
   * After defining a function, run it by typing its name followed by arguments, just like a script or command.
2. Viewing Defined Functions

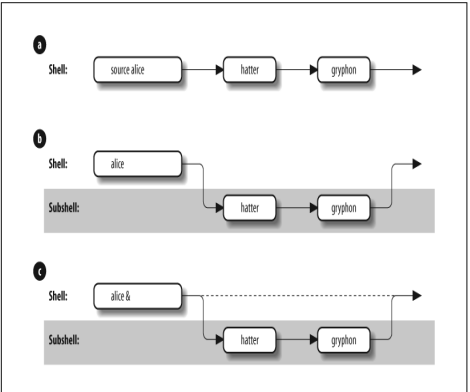
To see all functions with their full definitions:  
declare -f

To view just the function names:  
declare -F

1. Output Handling

Output of declare -f may be long—use more, less, or redirect to a file:  
declare -f | more

* + declare -f > functions.txt



## 

## **4.1.1 Functions in Shell Scripts**

Shell functions allow code reuse, modularity, and better readability.

Functions are essential because they allow you to:

* Organize code better
* Avoid repeating code (DRY principle)
* Improve readability and debugging
* Handle modular tasks like parsing FASTA/FASTQ files, filtering sequences, formatting output, etc.

SYNTAX :

**Example 1:Count DNA Sequences in a FASTA File**

**#!/bin/bash**

**count\_sequences() {**

**file=$1**

**count=$(grep -c "^>" "$file")**

**echo "Number of sequences in $file: $count"**

**}**

**# Usage:**

**count\_sequences "example.fasta"**

* grep -c "^>" counts lines starting with > (each represents a sequence header in FASTA).
* This is useful for validating inputs in sequence analysis pipelines.

**EXAMPLE 2:Get GC Content of a DNA Sequence File**

**#!/bin/bash**

**calculate\_gc\_content() {**

**file=$1**

**seq=$(grep -v "^>" "$file" | tr -d '\n')**

**gc=$(echo "$seq" | grep -o "[GCgc]" | wc -l)**

**total=$(echo "$seq" | wc -c)**

**gc\_content=$(echo "scale=2; $gc\*100/$total" | bc)**

**echo "GC Content: $gc\_content%"**

**}**

**# Usage:**

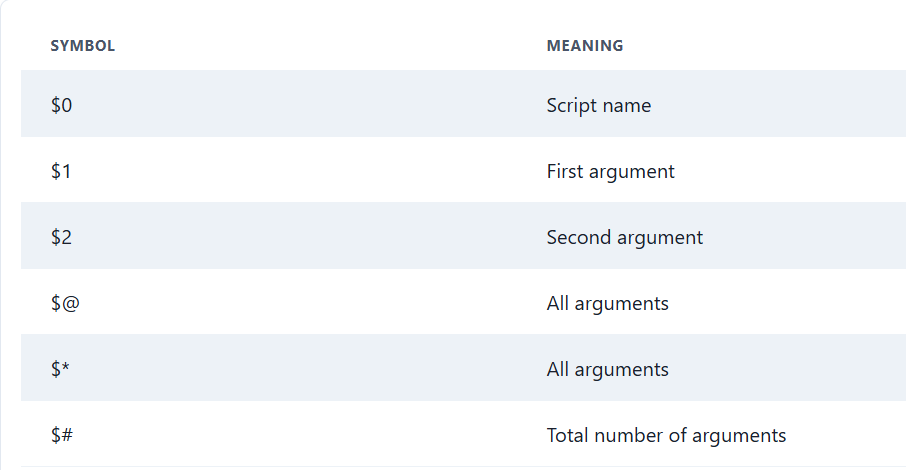
**calculate\_gc\_content "example.fasta"**

**4.2 SHELL VARIABLES**

Shell variables are fundamental to any shell script or interactive Bash session. They are used to store, retrieve, and manipulate data such as strings, numbers, file names, paths, and user input.

* No spaces around =
* Variables are accessed using $ before their name.

**4.2.1 Positional Parameters**

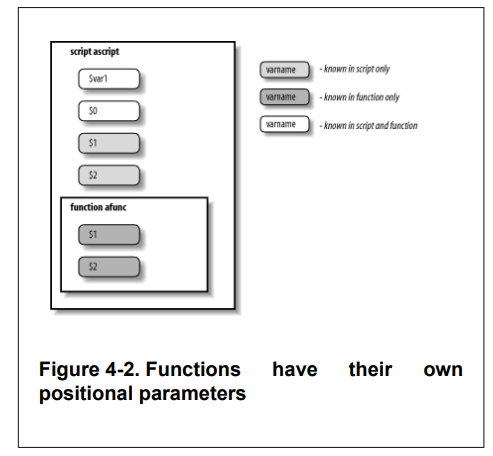
When a shell script is executed, the arguments passed to it are automatically stored in positional.

**EXAMPLE:**

**#!/bin/bash**

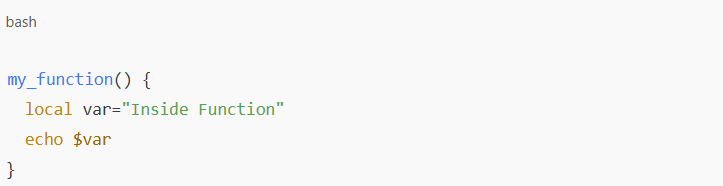
**echo "First argument: $1"**

**echo "Total arguments: $#"**



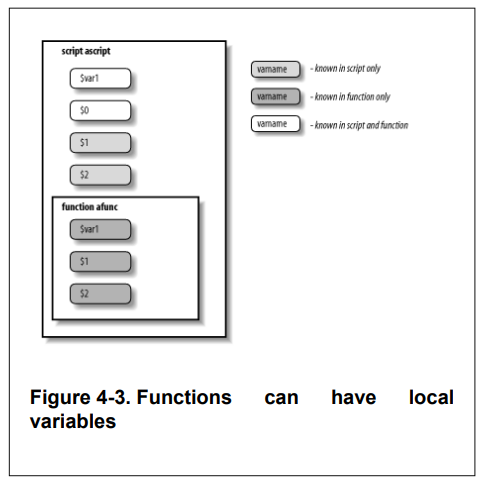
**4.2.2 LOCAL VARIABLES IN FUNCTIONS**

A local statement inside a function definition makes the variables involved all become local to that function. The ability to define variables that are local to "subprogram" units (procedures, functions, subroutines, etc.) is necessary for writing large programs, because it helps keep subprograms independent of the main program and of each other.



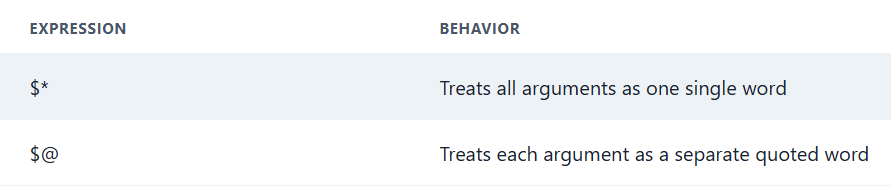
This avoids conflicts with other parts of the script.

### **Key Characteristics:**

* Declared using the local keyword
* Scope is limited to the function where they are defined
* Prevent unintended changes to variables with the same name in other parts of the script

Both $@ and $\* represent all positional arguments, but:

IFS=, echo "$\*"



This matters when arguments contain spaces.

### **4.2.4 More on Variable Syntax**

Bash provides **advanced forms of variable usage**, such as:

* **Default values:** ${var:-default} → Use default if var is unset
* **Substring operations:** ${var:0:4} → Extract substring from position 0 of length 4
* **Length of a variable:** ${#var} → Returns the length of the value

***4.3 STRING OPERATORS***

The curly-bracket syntax allows for the shell's string operators. String operators allow you to manipulate values of variables in various useful ways without having to write full-blown programs or resort to external UNIX utilities.

In particular, string operators let us do the following:

• Ensure that variables exist (i.e., are defined and have non-null values)

• Set default values for variables

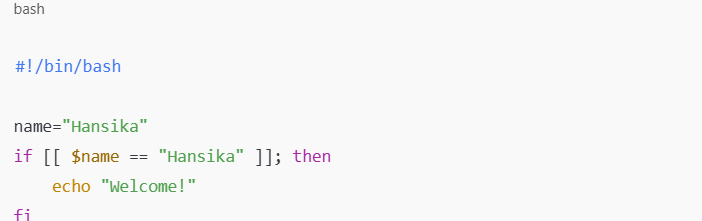
• Catch errors that result from variables not being set • Remove portions of variables' values that match patterns.

## **4.3.1 Syntax of String Operators**

## Bash uses both [[ ... ]] conditional expressions and parameter expansion (${...}) for string operations.

Examples of string operations:



Example: String Comparison

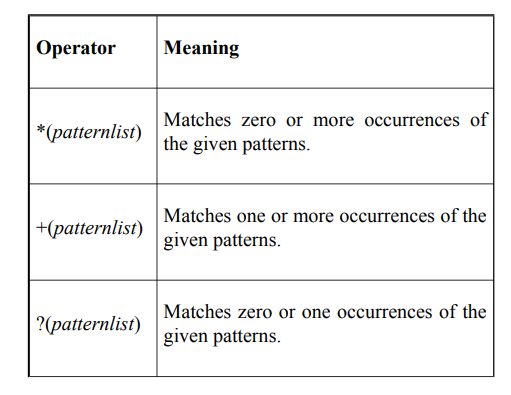
**4.3.2 Patterns and Pattern Matching**

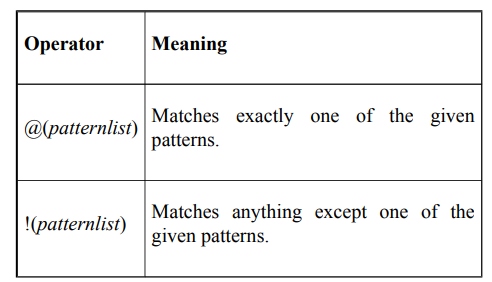
Pattern matching is a technique that allows Bash to compare strings not by exact match, but by checking if a string fits a certain pattern or wildcard expression.

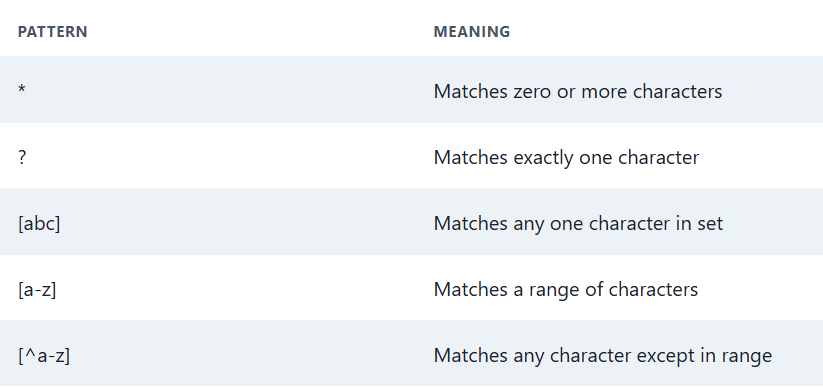
It is commonly used to:

* Match file names (like \*.fasta)
* Detect string prefixes/suffixes (e.g., ATG\*)
* Filter or extract specific text data
* Validate sequences, headers, gene IDs, etc.

Pattern matching does not use regular expressions by default; instead, Bash uses glob patterns (like \*, ?, [ ]) for matching.







Some examples of these include:

• \*(alice|hatter|hare) would match zero or more occurrences of alice, hatter, and hare. So it would match the null string, alice, alicehatter, etc.

• +(alice|hatter|hare) would do the same except not match the null string.

• ?(alice|hatter|hare) would only match the null string, alice, hatter, or hare.

• @(alice|hatter|hare) would only match alice, hatter, or hare.

• !(alice|hatter|hare) matches everything except alice, hatter, and hare.

### **Real-World Bioinformatics Relevance**

In **bioinformatics**, pattern matching is especially useful when:

* Filtering FASTA/FASTQ headers (e.g., >chr\*)
* Matching gene or protein IDs (e.g., BRCA\*)
* Validating DNA/RNA sequence start codons (e.g., ATG\*)
* Finding file types in bulk processing (e.g., \*.fasta, \*.vcf)

sequence="ATGCGTACG"

if [[ $sequence == ATG\* ]]; then

echo "Starts with ATG (start codon)"

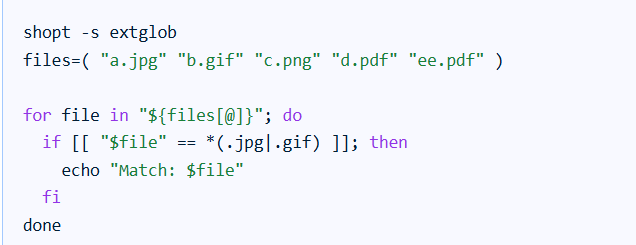
fi

## **4.3.3 Length Operator**

Get the **length of a string** using ${#var}:

## **4.3.4 Extended Pattern Matching in Bash**

Extended pattern matching in Bash allows you to perform advanced string pattern comparisons using logical expressions and repetition rules — similar to regex but simpler and native to Bash.

It enhances the basic wildcard system (\*, ?, [abc]) and is useful in file filtering, data validation, and string logic inside scripts.

This code snippet first enables extended globbing. Then, it iterates through an array of filenames. The if statement uses the ?(pattern) construct, which in this case matches files ending with either .jpg or .gif. If a match is found, it prints the filename. The . in \*(.jpg|.gif) is not a part of the globbing pattern, it is to match a literal dot.

**4.4 COMMAND SUBSTITUTION**

* Command substitution allows capturing the output of a command and using it as a value in a script.
* It enables dynamic assignment of values based on system state, file content, or processing result.
* It is essential for automation, script flexibility, and dynamic data handling.
* The preferred syntax for command substitution is $(command) because it is cleaner and supports nesting.
* The older syntax is `command`, which is harder to read and maintain.
* Command substitution can be used in variable assignment, command arguments, or string construction.
* It is evaluated before the main command runs, replacing the entire expression with the command’s output.
* It is widely used in bioinformatics scripts for tasks such as counting sequences or capturing filenames.
* For example:  
  #!/bin/bash
* file="example.fasta"
* if [[ ! -f "$file" ]]; then
* echo "Error: File '$file' not found!"
* exit 1
* fi
* count=$(grep -c "^>" "$file")
* echo "Total sequences: $count"

## **4.5 Directory Stack in Bash (pushd and popd)**

### **Concept of a Stack (LIFO)**

* A **stack** is a Last-In-First-Out (LIFO) structure, like a stack of plates.
* **push** adds a new item on top.
* **pop** removes the item from the top.
* In directory navigation, this lets us **remember and return to previous folders** in order.

### **🔷 Why Use pushd and popd in Bash?**

* Bash has cd - to switch to the **previous directory**, but it remembers **only one**.
* For **multi-level navigation**, we need a **custom stack**.
* Using pushd/popd, we can **push directories onto a stack** and **pop back through history** easily.

### **🔷 Stack Setup**

* Declare a global stack variable in .bash\_profile:



* This ensures:  
  + The stack is **initialized once per login**
  + It is **available to all child shells**

### **🔧 Custom pushd Implementation**

pushd() {

dirname=$1

DIR\_STACK="$dirname ${DIR\_STACK:-$PWD' '}"

cd "${dirname:?"missing directory name."}"

echo "$DIR\_STACK"

}

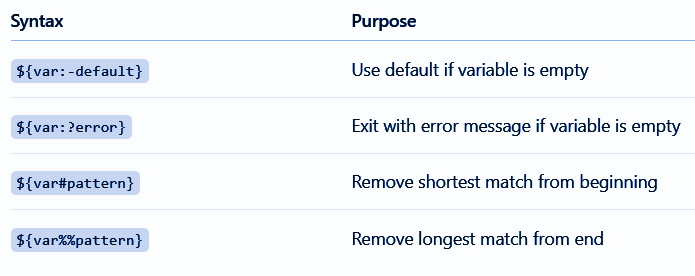
#### **Key Parts:**

* $1: Directory passed to the function
* ${DIR\_STACK:-$PWD' '}: If DIR\_STACK is empty, use current directory
* cd "${dirname:?}": If no directory is given, throw error
* Appends to the front of the stack and changes directory
* The stack preserves navigation history, allowing multi-level return using popd.
* Helps navigate deeply nested directories efficiently in custom Bash setups

**🔧 Custom popd Implementation**



#### **Key Parts:**

* ${DIR\_STACK#\* }: Removes the first item (i.e., top of stack)
* ${DIR\_STACK%% \*}: Extracts the new top item
* Changes to that directory and prints it

EXAMPLE SESSION :

$ pwd

/home/Anusha

$ pushd projects

/home/Anusha-Vostro/projects /home/Anusha

$ pushd /etc

/etc /home/Anusha/projects /home/Anusha

$ popd

/home/Anusha/projects /home/Anusha

$ pwd

/home/Anusha/projects

$ popd

/home/Anusha

$ pwd

/home/Anusha

$ popd

bash: cd: too few arguments

EXPLANATION:

1.Each pushd command adds the new directory to the top of the stack and switches to it, preserving the previous directories in DIR\_STACK.

2.Each popd command removes the top directory from the stack and switches back to the next one stored.

3.When all entries are popped, further popd calls result in an error (cd: too few arguments) because the stack is empty.

4.This simulates **multi-level backtracking** through directories, much more powerful than cd - which remembers only one level.

5.The directory stack behaves like **LIFO (Last-In-First-Out)** — the last pushed directory is the first one popped.

6.This approach is useful in scripts or manual navigation when working across **multiple nested project folders** without losing track of the path history.

***SUMMARY***

