Introduction to Shell Scripting

I. Prerequisites

Variables in Bash

Variables in Bash are used to store data, which can be used and manipulated throughout a script. They can hold various types of information, such as strings, numbers, and command outputs.

Declaring Variables

In Bash, variables are created by assigning a value to a name without spaces:

```
variable=Hello
```

Variable Naming Rules

- Variable names can contain letters, numbers, and underscores (`_`).
- They **must not** start with a number.
- Avoid using special characters or spaces.
- By convention, uppercase names are used for environment variables, while lowercase names are used for user-defined variables.

Accessing a Variable in Bash

The `$` symbol is used in Bash to reference (expand) the value of a variable.
Variations of `$` Usage

1. **Basic Variable Expansion**

```
variable=Hello
echo $variable    # Outputs: Hello
```

2. **Using `${}` to Avoid Ambiguity**
   When a variable is followed by a string, `${}` prevents misinterpretation.

```
file="report"
echo "$file2024"   # Incorrect: Tries to expand $file2024 (which doesn't exist) echo "${file}2024"      # Correct: Outputs "report2024"
```

Use `""`, `''`, or Nothing with Variables
1. **Double Quotes (`""`)**

- Used when the variable may contain spaces or special characters.

```
greeting="Hello World"
echo "$greeting"       # Outputs: Hello World
```

2. **Single Quotes (`''`)**

- Prevents variable expansion (treats everything as plain text).

```
greeting='Hello $USER'
echo '$greeting'       # Outputs: $greeting (not Hello username)
```

3. **No Quotes**

- Works fine for simple variables without spaces.

```
name="John Doe"
echo $name  # Outputs: John (only first word)
echo "$name"  # Correct: Outputs John Doe
```

Best Practice: Always use `"$variable"` unless you specifically need to prevent expansion (`''`) or know the value is safe without quotes.

Types of Variables

1. **Local Variables**

Local variables exist only in the current shell session:

```
greeting="Hello"
echo $greeting  # Available only in this session
```

2. **Environment Variables**

Environment variables are available across all shell sessions and subprocesses:
$**HOME** - home directory for the current user;
$**PATH** - list of directory directory paths where the interpreter searches for the rest of the path when executable files are run by providing their name only;
$**PS1** - primary prompt variable displayed in the terminal;
$**PS2** - secondary prompt variable;
$**TERM** - the type of the terminal.

3. **Special Variables**

Bash provides several built-in special variables:

- `$0` → Name of the script
- `$1, $2, ...` → Positional parameters
- `$#` → Number of arguments passed to the script
- `$@` → All arguments as separate words
- `$*` → All arguments as a single string
- `$$` → Process ID of the script
- `$?` → Exit status of the last executed command

Command Chaining

In UNIX, commands can be chained together, allowing the output of one command to serve as the input for another. This is useful for processing and filtering data efficiently.
Piping Commands (`|` Operator)

1. The **pipe (`|`) operator** is used to pass the output of one command as input to another.

```
ls | more
```

- The `ls` command lists directory contents.
- The `more` command paginates the output, displaying it page by page.

Commands can be chained multiple times:

```
ls | sort | more
```

- `ls` lists the files.
- `sort` orders them alphabetically.
- `more` displays them page by page.

🔹 **Nota Bene:** When commands are chained with `|`, they are executed in **parallel**. The left command generates data (`stdout`), while the right command processes it (`stdin`).

2. Executing Commands Sequentially (`;` Operator)

The **semicolon (`;`) operator** allows executing multiple commands sequentially. Each command runs **regardless of the success or failure** of the previous one.

```
cd example; ls -al
```

- Changes the directory to `example`.
- Lists all files (including hidden ones) with details.

3. Conditional Execution

- **Logical AND (`&&`)**

The `&&` operator ensures that the second command runs **only if the first one succeeds** (exit code `0`).

```
ls -l dir && echo "Success!"
```

→ If `ls -l dir` executes successfully (e.g., the user has permissions), the message **"Success!"** is printed.

→ If `ls -l dir` fails (e.g., directory does not exist), the `echo` command is skipped.

- **Logical OR (`||`)**

The `||` operator executes the second command **only if the first one fails** (exit code ≠ 0).

Example:

```
ls -l dir || echo "Directory not found!"
```

- If `ls -l dir` **fails**, `"Directory not found!"` is printed.
- If `ls -l dir` **succeeds**, the second command is ignored.

- **Complex Chaining (`&&` and `||` Together)**

Example:

`false || echo "Step 1" && echo "Step 2"`

Step-by-Step Execution:

1. `false` **fails** (exit code ≠ 0), so we move to `||` (OR condition).
2. Since `false` failed, `echo "Step 1"` **executes**.
3. `echo "Step 1"` **succeeds** (exit code 0), so we move to `&&` (AND condition).
4. Because `echo "Step 1"` succeeded, `echo "Step 2"` **executes**.

Final Output:

`Step 1`

`Step 2`

Redirections

Redirection in Bash allows changing the **source** of standard input (`stdin`) and the **destination** of standard output (`stdout`) and standard error (`stderr`). This mechanism enables commands to read input from files and save outputs or errors to files.

1. **Standard Input (`<` Operator)**

The `<` operator redirects input from a file instead of the keyboard.

```
sort < data.txt
```

- The `sort` command reads from `data.txt` and outputs sorted lines to the terminal.

2. **Standard Output (`>` Operator)**

The `>` operator redirects the output of a command to a file, **overwriting** existing content.

```
sort > ordered.txt
```

- The `sort` command takes input from the keyboard and writes the sorted output to `ordered.txt`.
- If `ordered.txt` already exists, it will be **overwritten**.

3. **Redirecting Both Input and Output**
   Both input and output redirections can be used simultaneously.

```
sort < data.txt > ordered.txt
```

- Reads input from `data.txt`.
- Writes the sorted output to `ordered.txt` (overwriting it).

4. **Redirecting Standard Error (`2>` Operator)**

Errors are typically displayed in the terminal. To redirect error messages to a file, use `2>`.

```
sort 2> errors.txt
```

- Redirects **error messages** from `sort` to `errors.txt`.
- Standard output remains visible in the terminal.

5. **Appending Output (`>>` Operator)**

To **append** output to a file without overwriting its contents, use `>>`.

```
sort >> ordered.txt
```

- The output of `sort` is **added** to the end of `ordered.txt`, preserving previous content.

II. Decision statements

1. IF Statement

a) Simple If Statement

```
if [ condition ]; then
  # commands
fi


# Example
if [ $num -gt 5 ]; then
  echo "Number is greater than 5"
fi
```

```
if test condition; then
  # commands to execute if the
  condition is true
fi

# Example
if test $num -gt 5; then
  echo "Number is greater than 5"
fi
```

b) If-Else Statement

```
if [ condition ]; then
  # commands to execute if condition is true
else
  # commands to execute if condition is false
fi
```

c) If-Elif-Else Statement

```
if [ condition1 ]; then
  # commands to execute if condition1 is true
elif [ condition2 ]; then
  # commands to execute if condition1 is false and condition2 is true
else # commands to execute if condition1 and condition2 are false
fi
```

d) Nested If Statements

```
if [ condition1 ]; then
  if [ condition2 ]; then
    # commands to execute if both condition1 and condition2 are true
  fi
fi
```

e) Using `&&` and `||` for Compound Conditions

```
if [ condition1 ] && [ condition2 ]; then
  # commands to execute if both condition1 and condition2 are true
fi

if [ condition1 ] || [ condition2 ]; then
  # commands to execute if either condition1 or condition2 is true
fi
```

f) Using `[[ ]]` for Advanced Conditions

`[[ ]]` allows for more advanced condition testing, pattern matching, improved string comparisons etc.

```
if [[ condition ]]; then
  # commands
fi
```

g) Arithmetic Conditions

For arithmetic comparisons, you can use `(( ))`.

```
if (( expression )); then
  # commands
fi
```

2. Statement used for testing

a) Using `test`

```
if test condition; then
  # commands to execute if the condition is true
fi
```

Examples

- **File existence check:**
  We can retrieve *file* related attributes using: **-e** (exists), **-f** (regular file), **-d** (directory), **-h** or **-L** (symbolic link), **-r** (the current user has *read* permissions on the file), **-w** (the current user has *write* permissions on the file), **-x** (the current user has *execute* permissions on the file).

```
if test -f "/path/to/file"; then
    echo "File exists."
fi
```

```
if [ -f "/path/to/file" ]; then
    echo "File exists."
fi
```

- **String comparison:**
  When working with *strings* we can use: **=** (equal), **!=** (not equal), **\<** and **\>** for lexicographical comparisons, **-z** (zero - empty string) and **-n** (non-zero - not empty string).

```
bash code
```

```
if test "$string1" = "$string2";
then
    echo "Strings are equal."
fi
```

```
if [ "$string1" = "$string2" ];
then
    echo "Strings are equal."
fi
```

```
if [[ $string1 == $string2 ]]; then
    echo "Strings are equal."
fi
```

- **Arithmetic comparison:**
  The following *arithmetic* operators are provided: **-eq** (equal), **-ne** (not equal), **-lt** (less than), **-gt** (greater than), **-le** (less or equal) and **-ge** (greater or equal).

```
bash code
```

```
if test $num1 -eq $num2; then
    echo "Numbers are equal."
fi
```

```
if [ $num1 -eq $num2 ]; then
    echo "Numbers are equal."
fi
```

3. WHILE statement

The `while` loop in Bash scripting is used to repeatedly execute a block of commands as long as a given condition is true. The syntax for a `while` loop is as follows:

```
while [ condition ]; do
    # commands to execute
done
```

```
while test condition; do
    # commands to execute
done
```

4. FOR statement

The `for` loop in Bash scripting is used to iterate over a set of values and execute a block of commands for each value.

It's a fundamental construct for looping in Bash, with several syntax variations to accommodate different scenarios. Here are the main forms of the `for` loop in Bash:

```
for variable in list_of_values; do
    # commands to execute
done
```

Example 1: Looping Through a List of Strings

```
for name in Alice Bob Charlie; do
  echo "Hello, $name!"
done
```

Example 2: Looping Through a Range of Numbers

In Bash, you can use brace expansion to generate a range of numbers.

```
for number in {1..5}; do
  echo "Number: $number"
done
```

Example 3: Looping Through Files in a Directory

```
for file in /path/to/directory/*; do
  echo "Processing $file"
done
```

C-Style Syntax

Bash also supports a C-style syntax for `for` loops, which is useful for more complex iteration scenarios:

```
for (( i=0; i<5; i++ )); do
   echo "Counter: $i"
done
```

This form is reminiscent of the `for` loop in the C programming language, allowing for an initialization expression, a condition, and an increment expression.

Iterating Over Command Output

You can iterate over the output of a command by combining the `for` loop with command substitution:

```
for user in $(cat users.txt); do
  echo "Welcome, $user"
done
```

5. Other useful statements

- **break** - allows escape from a loop before the exit condition is fulfilled;
- **continue** - skips the rest of the current iteration of a loop;
- **expr** *expression* - evaluates an expression; this command requires 3 arguments: the first operand, followed by an operator and then the second operand (the arguments must be separated by space).

```
while [ $countdown -ge 0 ]; do
    echo "Countdown: $countdown"

    # If countdown reaches 2, exit the loop early
    if [ $countdown -eq 2 ]; then
        echo "Breaking out of the while loop at countdown=$countdown"
        break
    fi

    # Decrement countdown using expr
    countdown=$(expr $countdown - 1)
done

for number in {1..5}; do
    # Check if the number is even
    if [ $(expr $number % 2) -eq 0 ]; then
        echo "Skipping number $number because it's even."
        continue
    fi

    # This command will be skipped for even numbers
    echo "Processing number $number"
done
```

III. Substitutions

1. Command Substitution

- **Syntax:** Enclose commands within backticks (`` `...` ``) or `$(...)`.
- **Purpose:** Executes the command and substitutes its output.

Example with Backticks:

```
directory=`pwd`
```

Example with `$(...)`:

```
directory=$(pwd)
```

2. Arithmetic Substitution

- **Syntax:** For basic arithmetic operations, use `expr` with operands and operators separated by spaces. For more advanced arithmetic, use double parentheses `((...))` or `$((...))`.
- **Purpose:** Evaluates arithmetic expressions.

Basic Arithmetic with `expr`:

```
counter=1
counter=`expr $counter + 1`
```

Advanced Arithmetic with `((...))`:

```
var=1
(( var++ ))
echo "Results: $var"
```

Retrieving Results with `$((...))`:

```
var=$((1+2))
echo "Results: $var"
```

3. Subshell Execution

- **Syntax:** Enclose commands within simple parentheses `( ... )`.
- **Purpose:** Executes commands in a subshell. Useful for limiting side effects.

Example:

```
echo "Today: $(date) is a beautiful day!"
```

4. Group Commands

- **Syntax:** Enclose a series of commands within curly braces `{ ... }`. If preceded by `$`, it returns the value of a variable contained within.
- **Purpose:** Groups commands to be executed together.

Example without `$`:

```
true && { var=5; echo "bla"; echo "$var"; exit 1; }
```

Example with `${...}`:

```
var="./dir/dir2"
echo ${var}/file.txt
```

5. Quoting Mechanisms

- **Double Quotes `"..."`:** Preserves literal value except for `$` (variable expansion).
- **Single Quotes `'...'`:** Preserves the literal value of all characters within the quotes.
- **Backticks `` `...` ``:** For command substitution; considered outdated in favor of `$(...)`.

Examples:

```
var=5
echo "variable var: $var"
# Output: variable var: 5

echo 'variable var: $var'
# Output: variable var: $var
```

6. Handling Spaces in Strings

To prevent the shell from interpreting strings with spaces as multiple arguments, enclose them in quotes.

Example:

```
var="ana has apples"
if test -n "$var" then
  echo "$var" # Correctly handles spaces within the string.
fi
```

This summary covers the key aspects of substitutions, arithmetic operations, subshell execution, command grouping, quoting mechanisms, and handling spaces in Bash scripting.

IV. Functions

Functions in Bash allow reusing code by defining blocks that can be called multiple times within a script.

1. Defining a Function

Functions are defined using the following syntax:

```
function function_name() {
    # Commands
}
```

Or without `function` keyword:

```
function_name() {
    # Commands
}
```

2. Calling a Function

To execute a function, simply write its name:

```
my_function
```

3. Function with Arguments

Arguments passed to a function are accessed using positional parameters `$1`, `$2`, etc.

```
hello() {
    echo "Hello, $1!"
}
hello "Alice"  # Output: Hello, Alice!
```

4. Function with Return Value

A function can return an exit status (`0` for success, non-zero for failure). In Bash, the `return` statement does **not** return a value—it **sets the exit status of the function**.

```
check_number() {
    if [ "$1" -gt 10 ]; then
        return 0  # Success
    else
        return 1  # Failure
    fi
}
check_number 15 && echo "Valid" || echo "Invalid"
```

5. Function with Output

Instead of returning an exit code, functions can output data using `echo`:

```
get_date() {
    echo $(date +%Y-%m-%d)
}
current_date=$(get_date)
echo "Today's date is: $current_date"
```

6. Local Variables in Functions

Use `local` to declare variables within a function to prevent them from affecting the global scope:

```
calculate() {
    local num=10  # Local variable
    echo $((num * 2))
}
echo "Result: $(calculate)"
```

7. Recursive Functions

Functions can call themselves for recursion:

```
countdown() {
    if [ "$1" -le 0 ]; then
        echo "Done!"
    else
        echo "$1"
        countdown $(( $1 - 1 ))
    fi
}
countdown 5
```

8. Summary

Feature	Description
Definition	`function_name() { commands; }`
Calling	`function_name`
Arguments	`$1, $2, ...`
Return Values	`return 0` (success), `return 1` (failure)
Output	`echo` to return values
Local Variables	`local variable_name=value`
Recursion	Functions can call themselves

Using functions improves code structure, reusability, and maintainability in Bash scripts.

V. Assignments

1. Create a script that can be called in the form: ./script_name output.txt n1 n2 ...
2. If there are fewer than 2 arguments, display an appropriate message, and the program will end with code 1.
3. If "file" represents the name of a regular file, an appropriate message will be displayed; if it is not a regular file, the program will end with code 2.
4. It is assumed that n1, n2, ... are numbers; there is no need to verify this. A message will be displayed showing a list of all the received arguments.
5. Display how many numbers in the list are even numbers.
6. Calculate the sum of all the odd numbers.
7. Display how many digits this sum has.
8. Write in a file called "output.txt" the sum of the numbers and how many digits this sum has, along with an appropriate message.
9. Check if the directory `/tmp` exists inside the current working directory (use a string comparison). If it exists, print "Temporary directory exists". If it does not exist, print "Temporary directory does not exist" and exit with code 3.
10. Refactor the code so that each functionality mentioned above has its dedicated function. Pay attention to the return conditions!
11. Refactor the code so that the result of the functions will be printed in the file "output.txt".

Example:

```
./lab2.sh output.txt 1 2 3 4 5 6 7 8 9
```

Expected Output:

```
List of received arguments: output.txt 1 2 3 4 5 6 7 8 9
No. of odd numbers in the list: 5
Sum of odd numbers: 25
Number of digits of the sum of odd numbers: 2
Temporary directory exists
The sum of odd numbers and the number of digits of the sum have been written in the file 'output.txt'.
```

Contents of `output.txt`:

```
Sum of odd numbers: 25
Number of digits of the sum: 2
```