



# SHELL SCRIPTING

---

## Notes

---

### Introduction to Shell Scripting Basics

#### ◆ What is Shell Scripting?

Shell scripting is a powerful way to automate and streamline tasks within a Linux environment by writing and executing commands. It's invaluable for system administration, task automation, and enhancing productivity.

🔑 **Types of Shells:** Some popular shells include:

- **bash**(BourneAgainShell)-mostwidelyused
- **sh**(BourneShell)
- **ksh**(KornShell)
- **zsh**(ZShell)

💡 **Check Your Shell Type:** Run the following command to identify your current shell:

```
echo $0
```

---

### ◆ **How to Create a File and Use It?**

Creating and editing files using the `vi` command is a core skill for working in Unix and Linux environments. Here's a step-by-step guide on creating a file, entering content, and saving it with `vi`, as well as how to run your script and view its output.

#### 1. **Opening `vi` to Create a New File**

- Open your terminal.

Type the following command to create or open a file in `vi`:

```
vi my_script.sh
```

- *Note: If `my_script.sh` doesn't exist, `vi` will create a new file with that name.*

#### 2. **Basic `vi` Editing Commands**

When you open `vi`, you're in command mode, where you can issue commands to `vi`. Here's how to enter insert mode to start writing:

- Press `i` to enter insert mode. You can now type content into the file.

#### 3. **Writing Content in `vi`**

Let's create a simple script that prints "Hello, World!" and displays the current date and time. With `vi` in insert mode, type the following script:

```
#!/bin/bash
```

```
# This script prints a greeting and the current date  
and time
```

```
echo "Hello, World!"
```

```
echo "The current date and time is:"
```

```
date
```

#### 4. ExitInsertMode:

Press `Esc` to return to command mode.

#### 5. SavingandExitinginvi

To save the file and exit `vi`:

- Type `wq` and press Enter.

`:wq` means "write (save) and quit."

Alternatively:

- To save without exiting, type `w` and press Enter.
- To quit without saving changes, type `q!` and press Enter.

#### 6. MakingtheScriptExecutable

Before running your script, you need to make it executable. In the terminal, type:

```
chmod +x my_script.sh
```

#### 7. RunningtheScriptandViewingOutput

Run the script by typing:

```
./my_script.sh
```

#### 8. RuntheScriptWithoutMakingItExecutable:

You can directly run it with `bash` by typing:

```
bash my_script.sh
```

### View the Output:

When you execute the script, you should see output similar to this:

```
Hello, World!
```

```
The current date and time is:
```

```
Tue Nov 6 12:34:56 UTC 2023
```

---

### ◆ Process Control Shortcuts

- **Ctrl + C**: Terminates a running process immediately. Useful for stopping commands or scripts that are running.
  - **Ctrl + Z**: Pauses (stops) a process and sends it to the background. Resume with **fg** (foreground) or **bg** (background).
- 

### ◆ Comments in Shell Scripting

#### Single-Line Comments:

Use the **#** symbol at the beginning of a line to create a single-line comment.

```
# This is a single-line comment
```

```
echo "Hello, World!" # This comment is on the same line as  
a command
```

#### Multi-Line Comments:

Bash does not have a direct syntax for multi-line comments, but you can simulate it using a here-document with **<<**.

```
: <<'COMMENT'
```

```
This is a multi-line comment.
```

```
You can add multiple lines here.
```

```
COMMENT
```

Alternatively, use the `<<` syntax directly:

```
<<comment
```

```
This is a multi-line comment.
```

```
It won't be executed.
```

```
comment
```

---

## Writing Your First Shell Script

### Getting Started with Scripting

A script is a series of commands executed sequentially. Here's a simple example that prints a message to the screen:

```
#!/bin/bash # Shebang to specify the interpreter
```

```
echo "Hello World!"
```

#### Output:

```
Hello World!
```

## How to Run Your Script:

### Make it Executable:

```
chmod +x script.sh
```

### Execute the Script:

```
./script.sh
```

### Why Shebang( `#!/bin/bash`) Matters

This line tells the system to use Bash to run the script, ensuring compatibility across Linux environments.

---

## Working with Variables and Arrays

Variables and arrays allow data storage and manipulation in scripts, making automation more flexible.

- ◆ **Variables:** Define a variable and use `$` to access its value.

```
NAME="Linux"
```

```
echo "Welcome to $NAME Scripting"
```

- ◆ **Constants:** Make variables read-only with `readonly`.

```
readonly VERSION="1.0"
```

- ◆ **Arrays:** Store multiple values.

```
myArray=(1 2 3 "Hello" "World")
```

```
echo "${myArray[1]}"
```

Outputs: 2

- **Get the length of an array:**

```
echo "${#myArray[@]}"          or echo "${#myArray[*]}"
```

This returns the total number of elements in the array.

- **Get specific values from an array:**

To get values starting from a specific index:

```
echo "${myArray[*]:1}"
```

This fetches all elements starting from the second element (index 1).



- **To get a specific range of values:**

```
echo "${myArray[*]:1:2}"
```

This fetches 2 elements starting from index 1.

- **Update an array (Add new elements):**

```
myArray+=(5 6 8)
```

- **Working with Associative Arrays**

**Declare and Initialize:**

```
declare -A myArray
```

```
myArray=( [name]=Paul [age]=20 )
```

**Access Values:**

```
echo "${myArray[name]}"
```

**Get Array Length:**

```
echo "${#myArray[@]}"
```

**Update Array:**

```
myArray+=( [city]=NewYork )
```

💡 Arrays are handy for managing lists of values, such as filenames or configurations.

---

## ✳ String and Arithmetic Operations

### String Manipulation Examples:

```
str="Shell Scripting"
```

#### Length:

```
echo ${#str}
```

Outputs: 15

#### Replace:

```
echo ${str/Scripting/Programming} Outputs: Shell Programming
```

#### Extract Substring:

```
echo ${str:6:9}
```

Outputs: Scripting

### Get the length of a string:

```
myVar="Hello World!"
```

```
length=${#myVar}
```

```
echo $length Output:12
```

### Convert to uppercase:

```
upper=${myVar^^}
```

```
echo $upper Output:HELLOWORLD!
```

### Convert to lowercase:

```
lower=${myVar,,}
```

```
echo $lower Output:helloworld!
```

### Replace a substring:

```
replace=${myVar/World/Buddy}
```

```
echo $replace Output: Hello Buddy!
```

### Extract a substring (slice):

```
slice=${myVar:6:5}
```

```
echo $slice Output:World
```



## ◆ USER INTERACTIONS

Taking Input from the User in Shell Scripting

### ● BasicInput:

```
read var_name  
echo "You entered: $var_name"
```

#### Example Output:

(User types "John")  
You entered: John

### ● Input with Prompt:

```
read -p "Your name: " NAME  
echo "Hello, $NAME!"
```

#### Example Output:

(Your name: User types "John")  
Your name: John  
Hello, John!

Key Difference:

- **BasicInput:** Takes input without a prompt.
- **Input with Prompt:** Displays a prompt to guide the user.

## ◆ Arithmetic Operations

### ● Using the **let** command:

Increment:

```
let a++
```

This increments the value of **a** by 1.

Assignment with multiplication:

```
let a=5*10
echo$a    Output:50
```

- Using `(( ))` for arithmetic operations:

Increment:

```
((a++))
```

Assignment with multiplication:

```
((a=5*10))
echo$a    Output:50
```

**Key Difference:**

- `let` is more traditional, while `(( ))` is more modern and allows for more complex arithmetic expressions.

Use `$( (expression) )` for complex calculations:

```
echo $((5 * (3 + 2)))    Output: 25
```

---

## 📌 Conditional Statements in Shell Scripting

**if Statement:**

```
if [ $a -gt $b ]; then
    echo "a is greater than b"
fi
```

### if-else Statement:

```
if [ $a -gt $b ]; then
    echo "a is greater than b"
else
    echo "a is not greater than b"
fi
```

### elif (else if) Statement:

```
if [ $a -gt $b ]; then
    echo "a is greater than b"
elif [ $a -eq $b ]; then
    echo "a is equal to b"
else
    echo "a is less than b"
fi
```

### Case Statement:

```
case $a in
    a) echo "a is 1" ;; b) echo "a is
    2" ;; *) echo "a is neither 1 nor
    2" ;;
esac
```

## Key Notes:

- Always put spaces around operators in conditions.
  - **elif** and **else** are optional but useful for handling multiple conditions.
- 

## Comparison Operators

**Equal to:** **-eq** or **==**: Checks if two values are equal.

```
[ $a -eq $b ]
```

**Greater Than or Equal to:** **-ge**: Checks if the left operand is greater than or equal to the right.

```
[ $a -ge $b ]
```

**Less Than or Equal to:** **-le**: Checks if the left operand is less than or equal to the right.

```
[ $a -le $b ]
```

**Not Equal to:** **-ne** or **!=**: Checks if two values are not equal.

```
[ $a -ne $b ]
```

**Greater Than:** **-gt**: Checks if the left operand is greater than the right.

```
[ $a -gt $b ]
```

**Less Than:** **-lt**: Checks if the left operand is less than the right.

```
[ $a -lt $b ]
```

---

## ◆ Logical Operators

Using **&&** (AND) Operator:

```
a=10
b=5
if [ $a -gt 5 ] && [ $b -lt 10 ]; then
    echo "Both conditions are true"
else
    echo "One or both conditions are false"
fi
```

Using **||** (OR) Operator:

```
a=10 b=15 if [ $a -gt 5 ] || [ $b -lt
10 ]; then

    echo "At least one condition is true"
else
    echo "Neither condition is true"
fi
```

Combining **&&** and **||** Operators:

```
a=10
b=5
c=15
if [ $a -gt 5 ] && [ $b -lt 10 ] || [ $c -eq 15 ]; then
    echo "Condition met"
else
    echo "Condition not met"
fi
```

Explanation:

- The **&&** ensures both conditions must be true.
- The **||** checks the second condition if the first fails.

### ◆ Ternary Operator (One-liner If-Else)

A simple way to write an **if-else** statement in one line:

```
a=10  
[ $a -gt 5 ] && echo "Greater" || echo "Not Greater"
```

**Explanation:**

- If `$a -gt 5` is true, it prints "Greater".
  - Otherwise, it prints "Not Greater".
- 

### 🔄 For Loop

The **for** loop iterates over a list or a range of values and performs actions for each item.

**Syntax:**

```
for item in list; do  
    # Commands to execute for each item  
done
```

**Example:**

```
for i in 1 2 3; do  
    echo "Number: $i"  
done
```

**Output:**

```
Number: 1  
Number: 2  
Number: 3
```



### Range Example:

```
for i in {1..3}; do
    echo "Count: $i"
done
```

### Output:

```
Count: 1
Count: 2
Count: 3
```

---

## While Loop

The **while** loop runs as long as the specified condition is true.

### Syntax:

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

### Example:

```
count=1
while [ $count -le 3 ]; do
    echo "Count is: $count"
    ((count++)) # Increment count
done
```

### Output:

```
Count is: 1
Count is: 2
Count is: 3
```

---

## Until Loop

The **until** loop continues to execute until the condition becomes true.

### Syntax:

```
until [ condition ]; do  
# Commands to execute  
done
```

### Example:

```
count=1  
until [ $count -gt 3 ]; do  
    echo "Count is: $count"  
    ((count++))  
done
```

### Output:

```
Count is: 1  
Count is: 2  
Count is: 3
```

---

## Infinite Loop

An infinite loop continues running indefinitely until it is manually stopped (e.g., using Ctrl+C).

### For Loop Infinite Example:

```
for (( ; ; )); do  
  
    echo "This is an infinite loop"  
  
done
```

### While Infinite Example:

```
while ;; do

    echo "Infinite loop with while"

done
```

### Until Infinite Example:

```
until false; do

    echo "Infinite loop with until"

done
```

### Output (repeats indefinitely):

```
This is an infinite loop
This is an infinite loop
...
```

---

### Select Loop

The `select` loop creates a simple menu system, which allows users to select an option from a list. It's useful when you need a user-driven selection process.

#### Syntax:

```
select option in list; do

    # Commands based on user choice

done
```

**Example:**

PS3="Choose a fruit: "

```
select fruit in Apple Banana Orange Exit; do  
  case $fruit in  
    Apple) echo "You chose Apple";;  
    Banana) echo "You chose Banana";;  
    Orange) echo "You chose Orange";;  
    Exit) break;; *) echo "Invalid  
    option";;  
  esac  
done
```

**Example Output:**

```
1)   Apple   2)  
Banana 3) Orange  
4) Exit Choose a  
fruit:   2   You  
chose Banana
```

**Explanation:**

- PS3 sets the prompt message.
- The **select** loop displays options, and each selection runs the corresponding case statement.
- The **break** statement exits the loop when the user selects "Exit."

---

**Summary of Loop Types**

- **ForLoop**: Iterates over a list or range.
  - **WhileLoop**: Continues as long as a condition is true.
  - **Until Loop**: Continues until a condition becomes true.
  - **InfiniteLoop**: Runs indefinitely until interrupted.
  - **SelectLoop**: Displays a menu for user selection.
-

### Tip:

Loops are powerful tools for automating repetitive tasks. You can use them for various purposes like iterating over files, arrays, or ranges. For example, you can rename all files in a directory using a loop.

---

## Functions

### 1. Defining Functions:

You can define a function using either of these two syntaxes:

```
function function_name { ... }  
or  
function_name() { ... }
```

### 2. Basic Function:

Functions are used to encapsulate reusable blocks of code.

```
greet() {  
    echo "Hello, welcome to the shell script!"  
}  
greet # Calling the function
```

### 3. Functions with Parameters:

Functions can accept arguments, which are accessed via `$1`, `$2` etc.

```
greet_user() {  
    echo "Hello, $1!"  
}  
greet_user "Adhyansh"
```