Shell Scripting

15 February 2025 10:37 AM

What is Shell Script

A shell script is a list of commands in a computer program that is run by the Unix shell which is a command-line interpreter. A shell script usually has comments that describe the steps.

- Allow you to run if-else statements and loops.
- A file that contains a series of commands.
- A plain text file.
- It executes the command on each line, one line at a time.
- The terminal usually allows just one command at a time.
- Shell script allows you to combine and run multiple commands together

When do we need a shell script?

- You need to enter multiple shell commands and you will need to do it again in the future.
- If you know how a piece of work need to perform.. you can put that knowledge in a shell script
- ShellScript eliminates repetitive tasks through automation.
- Using a good script can reduce the change of error.
- A shell script can perform a task faster than a human can.
- You have to do a task more than once, but it's something that you rarely do.

Why use Shell

- Speed of deployment
- No worrying about low-level programming objects.
- Ease and speed of learning.
- Performance and efficiency.
- Anything you can do on the command line can be automated by writing a shell script.
- Can automate tedious or repetitive tasks.
- allow you to hand off work to others.
- Act as a form of documentation.
- Fairly quick and easy to write.

Create Your First Script

Shell scripting is one of the most efficient ways to automate tasks on Unix-based systems. In this tutorial, we'll guide you through creating your **first shell script**, understanding basic commands, and executing them step-by-step. Let's get started!

What is Shell Scripting?

A **shell script** is a plain text file containing a series of commands that the shell executes sequentially. It helps automate complex or repetitive tasks.

Key Features

- Automates repetitive tasks.
- Executes multiple commands with a single script.
- Acts as a form of documentation for workflows.
- Supports decision-making with **if-else statements**, loops, and variables.

Your First Shell Script

Let's create a simple script to understand how shell scripting works.

Step 1: Create the Script

1. Open your terminal and create a new file named `first.sh`.
2. Add the following lines to the file:

```
echo "Hello World"
echo "This is our first shell script."
pwd
date
```

Step 2: Change File Permissions

By default, the script doesn't have execute permissions. Use the `chmod` command to grant execute permissions:

```
chmod +x first.sh
```

Step 3: Run the Script

Execute the script using the `./` command:

```
./first.sh
```

Expected Output

```
Hello World
This is our first shell script.
/home/kali/shellscript-youtube
Wed May 4 06:36:07 AM EDT 2022
```

Explaining the Commands

1. **`echo`**:
   - Prints the text to the terminal.
   - Example: `echo "Hello World"`.
2. **`pwd`**:
   - Displays the current working directory.
3. **`date`**:
   - Prints the current date and time.
4. **`chmod +x`**:
   - Grants execute permissions to the script.
5. **`./first.sh`**:
   - Executes the shell script.

Tips

1. **Add Comments**:
   - Use `#` to include notes and explanations in your script for better understanding and maintenance.
2. **Test Small Blocks of Code**:
   - Test individual commands before adding them to the script.
3. **Use Meaningful Names**:
   - Give your script files clear and descriptive names.

Understanding Shebang

Shell scripting allows you to automate tasks and simplify workflows on Unix-based systems. In this blog, we'll explore the **shebang (`#!`)**, its purpose, and how to execute shell scripts with different interpreters.

What is a Shebang?

The **shebang (`#!`)** is a special syntax used in scripts to specify the interpreter that should execute the script. It must be the **first line** in a script and always starts with `#!`.

Why is it Called Shebang?

- `#` represents the **sharp sign**.
- `!` represents the **bang sign**.
- Together, they form "shebang" (or **sharpbang**).

Shebang Syntax

The syntax is straightforward:

```
#!/path/to/interpreter
```

For example:

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

Why Use a Shebang?

1. **Defines the Interpreter**:
   - Specifies which interpreter (e.g., bash, python) should execute the script.
2. **Improves Compatibility**:
   - Ensures the script runs the same way across different environments.
3. **Exec Support**:
   - Makes the script behave like an actual executable file.
4. **Enhanced Process Identification**:
   - Running a script with `ps` shows the script name instead of the default shell.

Shebang in Action

Example 1: Bash Script

1. Create a script `first.sh`:

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

```
echo "Hello from Bash!"
echo "This is a Bash script."
```
2. Grant execute permission:
```
chmod +x first.sh
```
3. Execute the script:
```
./first.sh
```
4. **Output**:
```
Hello from Bash!
This is a Bash script.
```

Example 2: Python Script

1. Create a script `second.sh`:
```
#!/usr/bin/python
print("Hello, this is a Python script!")
```
2. Grant execute permission:
```
chmod +x second.sh
```
3. Execute the script:
```
./second.sh
```
4. **Output**:
```
Hello, this is a Python script!
```

Checking Available Shells

In Linux, you can view all available shells using the `/etc/shells` file:

```
cat /etc/shells
```

Example Output:

```
/bin/bash
/bin/python
/bin/sh
/usr/bin/perl
/usr/bin/tcl
/bin/sed -f
/bin/awk -f
```

What Happens Without a Shebang?

If a script doesn't include a shebang:

- It runs using the default shell of the system.
- You can check your default shell using the `echo $SHELL` command.

Impact:

- Scripts may behave differently depending on the default shell.

Practical Notes

1. Always provide the full path to the interpreter in the shebang (e.g., `/bin/bash`).
2. Use `chmod +x` to ensure the script is executable.
3. Test scripts with different interpreters to explore compatibility.

Understanding Shell Built-ins and Command Execution

Shell commands play a critical role in scripting and system management. In this blog, we'll explore **shell built-ins**, their importance, and how the shell processes commands using a specific sequence. Let's dive in!

What are Shell Built-ins?

A **shell built-in** is a command that is executed directly by the shell itself rather than requiring an external program.

Advantages of Built-ins

1. **Faster Execution**:
   - No need to load an external program, saving time.
2. **Efficiency**:
   - Especially useful for frequently used commands where the overhead of loading an external program is unnecessary.

How Shell Executes Commands

When you run a command in the shell, the shell follows a specific sequence to determine how to execute it:

1. **Check for a Function**:
   - If a function with the same name exists, it is executed.
2. **Check for Built-ins**:
   - If no function is found, the shell checks its list of built-ins.
3. **Search in the PATH**:
   - If no built-in matches, the shell searches for an executable in the directories specified in the `PATH` environment variable.

Practical Examples

Example 1: Checking Command Type

The `type` command helps identify whether a command is a built-in, a function, or an external program.

Using `uptime`

```
┌──(gaurav㉿learning-ocean)-[~]
└─$ uptime
01:37:53 up 2 min, 2 users, load average: 0.60, 0.59, 0.26
```

Identify the type of `uptime`:

```
┌──(gaurav㉿learning-ocean)-[~]
└─$ type -a uptime
uptime is /usr/bin/uptime
uptime is /bin/uptime
```

Here, `uptime` is an **external command** located in `/usr/bin` and `/bin`.

Using `echo`

```
┌──(gaurav㉿learning-ocean)-[~]
└─$ echo "Hello from Built-in"
Hello from Built-in
```

Check the type of `echo`:

```
┌──(gaurav㉿learning-ocean)-[~]
└─$ type -a echo
echo is a shell builtin
echo is /usr/bin/echo
echo is /bin/echo
```

In this case, `echo` is both a **shell built-in** and an **external command**. The shell prioritizes the built-in for faster execution. Both versions provide the same functionality.

Example 2: PATH Variable

If a command isn't a built-in, the shell searches for its executable in the directories listed in the `PATH` environment variable.

Check the `PATH` value:

```
┌──(gaurav㉿learning-ocean)-[~]
└─$ echo $PATH
/usr/local/sbin:/usr/sbin:/sbin:/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games:/home/kali/.dotnet/tools
```

Built-ins vs. External Commands

The execution priority of commands depends on whether they are built-in or external. Built-ins are prioritized for faster execution.

Example: `echo`

Built-in:

```
┌──(gaurav㉿learning-ocean)-[~]
└─$ echo "This is a shell built-in"
This is a shell built-in
```

External:

```
┌──(gaurav㉿learning-ocean)-[~]
└─$ /usr/bin/echo "This is an external command"
This is an external command
```

Reserved Keywords

Keywords are reserved words in the shell that cannot be used as variable names because they have predefined meanings. For example:

```
┌──(gaurav㉿learning-ocean)-[~]
└─$ type -a if
if is a reserved word
```

Examples of reserved keywords:

- `if`
- `then`
- `else`
- `fi`

Printing Messages in Different Colors

In this tutorial, we'll learn how to use the **echo** command to print messages in different colors using shell scripts. This is particularly useful for visually distinguishing between **success**, **failure**, and **warning** messages in your scripts.

Introduction to the echo Command

The `echo` command is used to print messages to the screen. With the `-e` option, we can enable the interpretation of backslash-

escaped characters and use ANSI color codes to format the output.

Hello World Script with Colors

Let's write a simple script to demonstrate how to use the `echo` command for printing messages in different colors.

Script

```
#!/bin/bash
# Print basic messages
echo "This is Gaurav Sharma"
echo 'This is our first                shell script'
# Print colored messages
echo -e "\033[0;31m fail message here"
echo -e "\033[0;32m success message here"
echo -e "\033[0;33m warning message here"
```

Explanation

1. **`echo`**:
   - Prints messages to the terminal.
   - Multiple spaces between words are ignored unless enclosed in quotes.
2. **`\033`**:
   - Escape sequence for initiating color codes.
3. **Color Codes**:
   - **Red**: `\033[0;31m`
   - **Green**: `\033[0;32m`
   - **Yellow**: `\033[0;33m`

Running the Script

Save the script as `echo.sh` and make it executable:

```
chmod +x echo.sh
```

Run the script:

```
┌──(gaurav㉿learning-ocean)-[~/shellscript-youtube]
└─$ ./echo.sh
This is Gaurav Sharma
This is our first              shell script
fail message here
success message here
warning message here
```

Output

1. **Fail Message**: Displays in **red**.
2. **Success Message**: Displays in **green**.
3. **Warning Message**: Displays in **yellow**.

Comments and Escape Sequences

Comments and **escape sequences** are fundamental to writing clear and maintainable shell scripts. Comments help developers document their scripts, and escape sequences allow for advanced formatting and multi-line commands.

Comments in Shell Scripts

Comments are notes added to the script to explain what the code does. They are ignored during execution.

Types of Comments

1. **Single-line Comments**: Start with `#`. These comments are used for short explanations or to temporarily disable a line of code.
   Example:
   ```
   # This is a single-line comment
   echo "Hello, World!"  # Inline comment
   ```
2. **Multi-line Comments**: Multi-line comments can be created using creative tricks, as there is no direct syntax for them in shell scripts.
   Examples:

- Using `: ' ... '`:
  ```
  : '
  This is a multi-line comment.
  It spans across multiple lines.
  '
  ```
- Using `: <<'END_COMMENT' ... END_COMMENT`:
  ```
  : <<'END_COMMENT'
  ```

```
This is another way to create
multi-line comments in shell scripts.
END_COMMENT
```

Escape Sequences

Escape sequences modify the behavior of characters in strings, enabling multi-line commands or special character usage.

Common Escape Sequences

1. **Newline (`\n`)**: Adds a new line.
2. **Tab (`\t`)**: Adds a horizontal tab.
3. **Vertical Tab (`\v`)**: Adds a vertical tab.
4. **Backslash (`\\`)**: Escapes the next character.

Example Script

```
#!/bin/bash
# Example to demonstrate comments and escape sequences
# Single-line comment example
echo "Hello, World!" # This is an inline comment
# Multi-line comment
: '
The following code demonstrates
the use of escape sequences in shell scripts.
'
# Escape sequences
echo "This is\na new line."
echo -e "This\tis\ta\ttabbed\toutput."
echo -e "Vertical\vtab\vexample."
echo -e "Multi-line with escape:\nLine 1\nLine 2\nLine 3"
# Strong quotes example
echo 'This is single-quoted, so \n will not be interpreted.'
# Combining comments and echo
echo "
########## Script Starting ########
Purpose: Install NGINX
####################################
"
```

Output

When you run the script, you will see:

```
Hello, World!
This is
a new line.
This    is      a       tabbed  output.
Vertical
tab
example.
Multi-line with escape:
Line 1
Line 2
Line 3
This is single-quoted, so \n will not be interpreted.
########## Script Starting ########
Purpose: Install NGINX
####################################
```

User-Defined Variables in Shell Scripting

Variables in shell scripting are an essential part of managing dynamic values within your scripts. They act as pointers to data, enabling you to manipulate and utilize values efficiently.

What is a Variable?

- A **variable** is a character string that represents a value.
- It is a pointer to the actual data stored in memory.
- Shell allows you to create, assign, and delete variables dynamically.

Rules for Variable Names

1. Variable names can include:
   - Letters (a-z, A-Z),
   - Numbers (0-9), and
   - Underscores (_).

2. **Invalid Characters**:
   - No spaces or special characters (e.g., -, !, *).
3. Variable names cannot begin with a number.
4. Variable names cannot be reserved shell keywords.
5. By convention, variable names are often written in **UPPERCASE**.

Examples of Valid Variable Names:

```
_VARIABLE_NAME
VARIABLE_NAME
VARIABLE_1_NAME
```

Examples of Invalid Variable Names:

```
2_VARNAME   # Starts with a number
-VARIABLE   # Contains a hyphen
VARIABLE!   # Contains an exclamation mark
```

Defining and Using Variables

Syntax:

```
variable_name=value
```

Note: No spaces around the = sign.

Example:

```
MY_MESSAGE="Hello World"
```

Examples and Output

Example Script:

```
#!/bin/bash
# variable_example.sh
name="Saurav"
age="20"
work="programm"
var="ing"
# Printing values
echo "Name: ${name}"
echo "Age: ${age}"
echo "I am ${work}${var}."
# Incorrect variable reference
echo "I am $working"  # 'working' not defined
```

Output:

```
┌──(gaurav㉿learning-ocean)-[~/shellscript-youtube]
└─$ ./variable_example.sh
Name: Saurav
Age: 20
I am programming.
I am
```

Common Errors

1. Missing quotes or incorrect use of spaces:
   ```
   MY VARIABLE=Hello World  # Incorrect (spaces not allowed)
   MY_VARIABLE="Hello World"  # Correct
   ```
2. Using undefined variables:
   ```
   echo $undefined_var  # Prints nothing
   ```

Advanced Examples

Variable Scope:

```
#!/bin/bash
# variable_scope.sh
# Variables with different cases
name="Gaurav"
NAME="Saurav"
nAmE="Amit"
echo "Different Variables: ${name}, ${NAME}, ${nAmE}"
# Using underscores in variable names
_variableName="First variable"
echo "${_variableName}"
```

Output:

```
┌──(gaurav㉿learning-ocean)-[~/shellscript-youtube]
└─$ ./variable_scope.sh
Different Variables: Gaurav, Saurav, Amit
First variable
```

Invalid Variable Names:

```
#!/bin/bash
# invalid_variable.sh
3variable="Invalid"
variable-name="Invalid"
```

Output:

```
┌──(gaurav㉿learning-ocean)-[~/shellscript-youtube]
└─$ ./invalid_variable.sh
./invalid_variable.sh: line 2: 3variable=Invalid: not found
./invalid_variable.sh: line 3: variable-name=Invalid: not found
```

Best Practices

- Always use meaningful variable names.
- Use UPPERCASE for constant values.
- Ensure proper quoting to avoid unexpected behavior.
- Follow naming conventions to improve readability and maintainability.

System Variables in ShellScript

System variables in ShellScript are predefined by the Linux Bash shell and play a vital role in managing environment configurations, script execution, and debugging processes. These variables are defined in **uppercase letters** and can be used to extract valuable information about the shell environment.

What are System Variables?

System variables are created and maintained by the Linux shell itself. These variables configure aspects of the shell's behav ior and environment, such as:

- User details
- File paths
- System locales
- Shell configurations

Key Commands for Managing Variables

- **env**: Displays the environment variables of the current session.
- **printenv**: Prints all or specific environment variables.
- **set**: Displays all shell variables, including functions.
- **unset**: Removes a specified variable.
- **export**: Makes a variable available for sub-shells.

Common System Variables and Use Cases

Below is a table of frequently used system variables along with their descriptions and practical use cases:

Variable	Description	Use Case
USER	Current logged-in user	Identify the user running the script: `echo "Script executed by $USER"`
HOME	Home directory of the current user	Use as a base path for file operations: `cd $HOME/Documents`
SHELL	Path of the user's current shell	Check if Bash is being used: `if [[ $SHELL == "/bin/bash" ]]; then ...`
HOSTNAME	Name of the machine	Display machine-specific messages: `echo "Running on $HOSTNAME"`
PWD	Present working directory	Log the current directory for debugging: `echo "Current directory: $PWD"`
UID	User ID of the current user	Ensure script runs as root: `if [[ $UID -ne 0 ]]; then echo "Run as root"; fi`
PATH	Directories searched for commands	Add custom directory for scripts: `export PATH=$PATH:/my/custom/scripts`
OSTYPE	Type of the operating system	Check OS compatibility: `if [[ $OSTYPE == "linux-gnu" ]]; then ...`
PPID	Parent process ID	Debug process hierarchy: `echo "Parent process ID: $PPID"`
SECONDS	Number of seconds since the script started	Time script execution: `echo "Script ran for $SECONDS seconds"`
RANDOM	Generate a random number	Simulate dice roll: `echo $(( $RANDOM % 6 + 1 ))`
LINES	Number of lines on the terminal	Optimize display based on terminal size: `echo "Terminal lines: $LINES"`
COLUMNS	Number of columns on the terminal	Adjust script output width: `echo "Terminal width: $COLUMNS"`
PS1	Primary shell prompt string	Customize shell prompt: `PS1="MyShell> "`
LANG	Current locale settings	Switch language dynamically: `export LANG=fr_FR.UTF-8`
LOGNAME	Name of the current user	Display the user performing critical actions: `echo "User: $LOGNAME"`
TMPDIR	Path to the temporary directory	Store temporary files: `temp_file="$TMPDIR/my_temp.txt"`

Using System Variables in Shell Scripts

Below is an example of a script that demonstrates the use of system variables:

```
#!/bin/bash
# Script: system_variables.sh
# Purpose: Demonstrate system variables usage.
echo "User: $USER"
echo "Home Directory: $HOME"
echo "Shell: $SHELL"
echo "Machine Name: $HOSTNAME"
echo "Working Directory: $PWD"
echo "User ID: $UID"
echo "System Path: $PATH"
echo "Operating System: $OSTYPE"
echo "Parent Process ID: $PPID"
echo "Script Runtime: $SECONDS seconds"
echo "Random Number: $RANDOM"
echo "Terminal Dimensions: $LINES lines x $COLUMNS columns"
echo "Temporary Directory: $TMPDIR"
```

Output Example

```
User: gaurav
Home Directory: /home/gaurav
Shell: /bin/bash
Machine Name: learning-ocean
Working Directory: /home/gaurav/scripts
User ID: 1000
System Path: /usr/local/bin:/usr/bin:/bin
Operating System: linux-gnu
Parent Process ID: 12345
Script Runtime: 5 seconds
Random Number: 27659
Terminal Dimensions: 24 lines x 80 columns
Temporary Directory: /tmp
```

Best Practices

- **Use descriptive names** for user-defined variables to avoid conflicts with system variables.
- **Validate paths** (e.g., `$HOME`, `$PATH`) to ensure scripts run reliably across environments.
- **Avoid overwriting** critical system variables unless absolutely necessary.