🪄 Shell-Scripting 👨🏻‍💻👩🏻‍💻🚀

A shell script is a text file that contains a sequence of commands for a UNIX-based operating system. It is called a shell script because it combines a sequence of commands, that would otherwise have to be typed into the keyboard one at a time, into a single script. The shell is the operating system's command-line interface (CLI) and interpreter for the set of commands that are used to communicate with the system.

Types of shell:

- Korn shell (ksh): It Was written by David Korn at AT&T Bell Labs. It is a superset of the Bourne shell. So it supports everything in the Bourne shell. It has interactive features. It includes features like built-in arithmetic and C-like arrays, functions, and string manipulation facilities. It is faster than the C shell. It is compatible with the script written for C shell.

- Bourne shell (sh): It was written by Steve Bourne at AT&T Bell Labs. It is the original UNIX shell. It is faster and more preferred. It lacks features for interactive use like the ability to recall previous commands. It also lacks built-in arithmetic and logical expression handling. It is the default shell for Solaris OS

- C-shell (csh): Bill Joy created it at the University of California at Berkeley. It incorporated features such as aliases and command history. It includes helpful programming features like built-in arithmetic and C-like expression syntax.

- Bourne again shell (Bash):The Bourne-Again Shell (BASH) is a widely-used command-line shell for Unix and Unix-like operating systems. It was created by Brian Fox for the GNU Project as a free software replacement for the original Bourne shell (sh). BASH is often used for system administration tasks and automation, as well as for writing scripts and programs. It includes many built-in commands and features, such as job control, command history, shell functions, and the ability to execute commands from a file or script. BASH also supports advanced scripting features such as loops, conditionals, and variables.

Steps to the writing shell script:

- Create a file using a text editor such as the vi or any other editor. Name script file with extension .sh
- Start the script with #! /bin/sh
- Write some code.
- Save the script file as the filename.sh
- For executing the script type bash filename.sh

what is a shebang line? #!/bin/bash?

A shebang line is a special line of text that tells the operating system which program to use to interpret the rest of the script. It is always the first line of a shell script and starts with the character #!. The shebang line is also known as a hashbang or a bang line.
We use shebang lines because they make it easy to run shell scripts. Without a shebang line, the operating system would not know which program to use to interpret the script. This would require the user to manually specify the program, which can be inconvenient.

Sample Shell script:

```
###########
#Author: XXXX
#date: xxx
#############
#This script gives the information xxxxxx
#!bin/bash
mkdir test
echo "Directory is created"
cd test
echo "directory as changed"
touch file.txt
echo "file is created"
```

- **Comments:**

In shell scripting, comments are used to add explanatory or descriptive text to the script that is ignored by the interpreter. They are helpful for documenting the purpose, functionality, or usage of the script, making it easier for others (including yourself) to understand and maintain the code.

```
# This is a single-line comment
echo "Hello, World!"  # This is another comment
```

Mult- line Comments:

```
# This is the first part of a multi-line comment
# More comment lines...
# Final comment line
echo "Hello, World!"
```

- **Variables:**

Variables are used to store data in shell scripts. They can be used to store anything from simple strings to complex objects. Variables are declared using the varname=value syntax.

```
#!/bin/bash
name="John"
age=25
city="New York"
echo "Name: $name"
echo "Age: $age"
echo "City: $city"
age=$((age + 1))
city="San Francisco"
echo "Updated Age: $age"
echo "Updated City: $city"
```

In shell scripts, there are two ways to define the variables such as system-defined & user-defined variables

```
#!/bin/bash

# System-defined variables
echo "System-defined variables:"
echo "Home directory: $HOME"
echo "User name: $USER"
echo "Current working directory: $PWD"
echo "Shell: $SHELL"
echo "Script name: $0"
echo

# User-defined variables
name="Alice"
```

```
age=30

echo "User-defined variables:"
echo "Name: $name"
echo "Age: $age"
echo

# Modifying user-defined variables
name="Bob"
age=$((age + 5))

echo "Updated user-defined variables:"
echo "Name: $name"
echo "Age: $age"
```

Using Read command in Shell- script

The `read` command is used in shell scripts to read input from the user and assign it to a variable. It prompts the user for input and waits until the user provides a response.

```
#!/bin/bash

# Prompt the user for their name
echo "Enter your name:"
read name

# Prompt the user for their age
echo "Enter your age:"
read age
```

```
# Display the user's name and age
echo "Hello, $name! You are $age years old."
```

- **Loops**

Loops are used to repeat a block of code a certain number of times or until a certain condition is met. There are two types of loops in shell scripts: for loops and while loops.
For loops are used to iterate over a collection of data. The collection can be a list of strings, a list of numbers, or even a list of objects. For loops are declared using `for the` keyword, a variable, and a list of values

```
#!/bin/bash

# Example using if statement
echo "Enter a number:"
read num

if [ $num -gt 10 ]; then
    echo "The number is greater than 10."
fi

# Example using if-else statement
echo "Enter your age:"
read age

if [ $age -ge 18 ]; then
    echo "You are eligible to vote."
else
    echo "You are not eligible to vote."
fi
```

```
# Example using while loop
count=1

while [ $count -le 5 ]; do
    echo "Count: $count"
    count=$((count + 1))
done

# Example using for loop
fruits=("apple" "banana" "orange" "grape")

for fruit in "${fruits[@]}"; do
    echo "Fruit: $fruit"
done
```

In this script, we have the following examples:

1. `if` statement: The script prompts the user to enter a number, and if the number is greater than 10, it prints "The number is greater than 10."
2. `if-else` statement: The script prompts the user to enter their age and checks if the age is greater than or equal to 18. If true, it prints "You are eligible to vote." Otherwise, it prints "You are not eligible to vote."
3. `while` loop: The script initializes a variable `count` to 1 and executes a loop that prints the value of `count` and increments it until it reaches 5.
4. `for` loop: The script creates an array of fruits and uses a `for` loop to iterate through each element of the array, printing the name of each fruit.

Advance Text processing commands

1. AWK is a versatile text-processing language commonly used in Linux and Unix environments. It is designed for data extraction and manipulation based on

patterns. AWK operates on a line-by-line basis, where it scans each line, applies patterns, and performs actions accordingly.

```
#!/bin/bash


# Example using AWK
awk '/error/ {print $0}' logfile.txt
```

2. GREP (Global Regular Expression Print) is a command-line tool used for searching text files or output for patterns and printing matching lines. It is widely used for pattern matching and filtering in Linux.

```
#!/bin/bash


# Example using GREP
grep "error" logfile.txt
```

3. SED (Stream Editor) is a powerful command-line tool used for text manipulation. It operates by reading the input line by line, applying specified commands, and modifying the output accordingly. SED can perform operations like find and replace, insert or delete lines, and perform more complex text transformations.

```
#!/bin/bash


# Example using SED
sed 's/foo/bar/' config.txt > updated_config.txt
```

https://github.com/Munikanth5/Shell-Scripts

GitHub - Munikanth5/Shell-Scripts: Shell-Scripts • github.com