**Shell Scripting**

**Shell scripting** is the process of writing a series of commands for the shell (command-line interpreter) to execute. These scripts automate tasks by grouping multiple commands together into a single executable file, which can be run like a program.

**Common uses of shell scripting:**

* Automating system maintenance tasks.
* Managing backups.
* Batch file processing.
* Monitoring system performance.
* **which bash -** it shows the path of bash or shell
* **echo $shell -** displays the path of the user's default shell.

**Script.sh -** file name with extension **.sh**

#!/bin/bash - shebang

echo “Hello ! I am Danish” - command

cp script.bash /mnt - copying file from current path to mnt

* **ls -l** - check the file permission

( The file must have execute permission for user )

* **chmod 744** - Permission read, write, execute for user and read for group and other
* **echo :-** Echo is used to display the text and variable

**echo** “This is shell scripting day 1”

* **Comments :-**

**# This is comment** - single line comment

**<< comment -** multiline comment

This is

Multiline cmmt

**comment -** multiline comment

* **Variable :-** It used to store the value

echo “This is used to print” - print command

name = “Danish” - declaring a variable

echo “My name is **$name**” - calling a variable by **$variable\_name**

1. **User defined variables :-** These are variables that you create within your shell script. They can store any type of data, such as strings, numbers, or paths.
2. **Predefined (System-defined) Variables :** These are variables that are automatically set by the shell or the system. They store information such as the shell environment, user details, or script parameters.

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | $0 | The name of the script. | |
| |  |  | | --- | --- | | $1, $2, … | Positional parameters for arguments passed to the script (e.g., $1 for the first argument). | |
| |  |  | | --- | --- | | $# | The number of positional parameters passed to the script. | |
| |  |  | | --- | --- | | $@ | All positional parameters as separate words. | |
| |  |  | | --- | --- | | $\* | All positional parameters as a single word. | |
| |  |  | | --- | --- | | $$ | The process ID (PID) of the current script. | |
| |  |  | | --- | --- | | $? | The exit status of the last command executed. | |
| |  |  | | --- | --- | | $\_ | The last argument of the last command executed. | |
| |  |  | | --- | --- | | $HOME | The current user's home directory. | |
| |  |  | | --- | --- | | $USER | The current logged-in user's name. | |
| |  |  | | --- | --- | | $PATH | The list of directories searched for executable commands. | |

* **Input function : -** Taking input from user by a prompt.

**echo “Enter username : ”**

**read username -** taking input

**echo “you entered $username”**

**read -p “Enter username : ” username** - taking username by prompt

**sudo useradd -m $username** - Add the user and create their home directory

(**-m**: This flag ensures that a home directory is created for the new user)

# Read two values and store them in variables 'name' and 'age'

**read name age** - you can read multiple values, separated by space

# Print the values

**echo "Name: $name, Age: $age"**

* **Argument :** In shell scripting, arguments are inputs provided to a script when it is executed. These arguments can be accessed within the script using special variables.

**./script.sh argument1 argument2 -** passing the arguments, separated by space

$0: Refers to the name of the script itself.

$1, $2, $3, etc.: Refers to the first, second, third, etc., arguments passed to the script.

**# Accessing the arguments**

echo "Script name: $0" - Refers to the name of the script

echo "First argument: $1" - This prints the argument1

echo "Second argument: $2" **-** This prints the argument2

* **Operators in Shell Scripting**

1. **Arithmetic Operators :** Used for mathematical calculations.

+ # Addition

- # Subtraction

\* # Multiplication

/ # Division

% # Modulus (Remainder)

\*\* # Exponentiation

+=, -=, \*=, /=, %= # Assignment operators

**Example :**

a=10

b=5

echo $((a + b)) # Output: 15

echo $(expr $a - $b) # Output: 5

**2. Comparison Operators :** Used for comparing values (mostly in conditional statements).

-eq # Equal to

-ne # Not equal to

-gt # Greater than

-lt # Less than

-ge # Greater than or equal to

-le # Less than or equal to

**Example :**

a=10

b=20

if [ $a -gt $b ]; then

echo "a is greater"

else

echo "b is greater"

fi

**3. Logical Operators :** Used to evaluate multiple conditions.

&& or -a # Logical AND

|| or -o # Logical OR

! # Logical NOT

**Example :**

a=10

b=20

if [ $a -gt 5 ] && [ $b -gt 15 ]; then

echo "Both conditions are true"

fi

**4. String Operators :** Used for string comparisons.

= # Equal to

!= # Not equal to

-z # String is empty

-n # String is not empty

**Example :**

str1="hello"

str2="world"

if [ "$str1" = "$str2" ]; then

echo "Strings are equal"

else

echo "Strings are not equal" fi

**5. File Test Operators :** Used to check file properties.

-e # File exists

-f # File exists and is a regular file

-d # Directory exists

-r # File is readable

-w # File is writable

-x # File is executable

-s # File is not empty

**Example :**

file="test.txt"

if [ -f "$file" ]; then

echo "File exists"

else

echo "File does not exist"

fi

* **Conditional statement :-** In shell scripting, if-else statements are used to perform conditional execution of commands based on whether a condition evaluates to true or false.

**Syntax :**

if [ condition ]; then

# Commands to execute if the condition is true

else

# Commands to execute if the condition is false

fi

**1) IF-else**

#!/bin/bash

read -p "Enter a number: " num # Prompt user to enter a number

if [ $num -gt 10 ]; then # Check if the number is greater than 10

echo "The number is greater than 10."

else

echo "The number is 10 or less."

fi

**2) Multiple Conditions with elif**

#!/bin/bash

read -p "Enter a number: " num # Prompt user to enter a number

if [ $num -gt 0 ]; then # Check if the number is positive, negative, or zero

echo "The number is positive."

elif [ $num -lt 0 ]; then

echo "The number is negative."

else

echo "The number is zero."

fi

**3) String Comparison**

#!/bin/bas

read -p "Enter a word: " word

if [ "$word" = "hello" ]; then # # Check if the word matches "hello"

echo "You entered hello!"

else

echo "You did not enter hello."

Fi

* **Array** : An array in shell scripting (especially Bash) is a data structure that allows you to store multiple values in a single variable.

**Syntax** :

myArray=(1 3 4 Danish "HELLO Khan")

echo ${myArray[3]}      # prints 3rd element

echo ${myArray[\*]}      # print all elements

1. **Associative Array (Key-Value Pair)**

#/bin/bash

declare -A mymap

mymap[fruit]="apple"

mymap[color]="red"

**# Accessing element**

echo ${mymap[fruit]} # apple

1. **String operation on array**

echo ${myarr}

echo "The upper case of version is -----> ${myarr^^} "

echo "The lower case of version is -----> ${myarr,,} "

newstr=${myarr/Danish/Khan}     # to replace word form string

echo ${newstr}

sliced=${myarr:4:6}     # for slicing the word (first no. represent the index no. and second Word length (eg.Danish=6))

echo $sliced

* **Case** : The case statement is a powerful tool that allows you to execute code based on the value of a variable.

**Syntax** :

case $variable in

pattern1)

commands

;;

pattern2)

commands

;;

\*)

default commands

;;

esac

**Eg.** In vs code file – 14\_cash.sh

* **For-Loop :** The for loop in shell scripting is used to iterate over a list of items, such as numbers, strings, files, or the output of a command.

**Syntax :**

**for variable in list**

**do**

**# Commands to execute**

**done**

**1**) **Iterating Over a List of Strings**

# List of items

for fruit in apple banana orange

do

echo "I like $fruit"

done

**2) C-Style for Loop (Numeric Loop)**

for ((i = 1; i <= 5; i++))

do

echo "Iteration $i"

done

* **While Loop :** The while loop in shell scripting executes a block of commands **repeatedly as long as a condition is true**.

**Syntax :**

while [ condition ]

do

# commands

done

* + 1. **While loop**

#!/bin/bas

count=1

while [ $count -le 5 ]

do

echo "Count is: $count"

((count++)) # increment

Done

* + 1. **Reading file**

#!/bin/bash

while read names

do

    echo "names from file $names"

done < names.txt

* **Until loop :** The until loop is the opposite of a while loop.  
  It keeps executing the block of code until the condition becomes true.

**Syntax :**

until [ condition ]

do

# commands

done

**eg.**

num=10

until [ $num -le 0 ]

do

    echo "num is $num"

    let num--

done