SHELL:

shell is a program that provides an interface for users to interact with the operating system. It allows users to input commands to perform tasks such as managing files, executing programs, and controlling system processes. The shell processes these commands, sends them to the kernel for execution, and displays the result

Functions of a Shell:

Command Execution: Users can type commands to interact with the system, like listing files (ls), changing directories (cd), or managing processes.

Scripting: Shells support scripting languages that allow users to automate tasks by writing shell scripts—a series of commands executed in sequence.

Environment Management: The shell provides access to environment variables (such as PATH, USER, HOME) that configure the system environment.

Types of Shells:

Bash (Bourne Again Shell): The most common shell, widely used because of its scripting capabilities and compatibility with many Linux distributions.

Zsh: Known for additional features like enhanced tab completion and better customization.

Fish: A user-friendly shell with features like syntax highlighting and autosuggestions.

Tcsh: A shell with C-style syntax, typically used in legacy systems or academic environments

CHECH YOUR DEFAULT SHELL USING:

echo $0

echo $SHELL

shell scripting:

Shell scripting is a type of programming that is used to automate tasks.

It involves writing a series of commands in a text file, which the shell executes to perform tasks.

Key points about shell scripting:

Shell: The "shell" is a command-line interface in an operating system, like Bash (Bourne Again Shell) in Linux or macOS, or PowerShell in Windows.

Scripting: Involves writing a sequence of commands that the shell can execute. These commands might include file operations, program execution, or control structures (like loops and conditionals).

Text File: The script is typically saved with a .sh extension (for Unix-like systems) and can be run directly in the shell.

Automation: Shell scripts are commonly used to automate repetitive tasks such as backups, system monitoring, software installation, or configuration changes.

Execution: Shell scripts are executed by running the script file from the shell, often with the ./ notation (e.g., ./script.sh).

Common file extensions for shell scripts include .sh, .bash, .zsh, .ksh, etc.

It is generally preferred to omit the file extension and rely on the shebang (#!) to identify the interpreter for execution.

The shebang (#!) in a script specifies which interpreter should be used to run the script.

simple shell script:

#!/bin/bash

# This is a simple shell script

echo "Hello, World!"

different ways to execute script file:

file with x permission:

./script.sh: Direct execution (after making it executable).

/full/path/to/script.sh: Execute using path references.

file without x permission:

bash script.sh or sh script.sh: Execute via shell.

source script.sh or . script.sh: Run in the current shell environment.

/bin/bash script.sh: Execute without changing file permissions.

Comments:

Single-line Comments:

# This is a comment in Linux shell scripting

Multi-line Comments:

<<str

....

content

....

str

variables:

variables are used to store data that can be referenced and manipulated throughout the script. These variables can hold various types of data like strings, integers, or even the output of commands.

Defining Variables:

my\_variable="Hello, World!"

Correct variable names : Incorrect variable names:

variable 2\_variable

variable1 2variable

variable\_2 variable$

\_variable3 variable\*^

Accessing Variable Values:

echo $my\_variable # Outputs: Hello, World!

echo "value of variable: $my\_variable"

setting value to variable:

var\_name="value"

PI=3.142

NAME=SOFTILITY

book = "Linux Shell Scripting" # Wrong, spaces around = operator

Reading User Input into Variables:

Use the read command to take user input and store it in a variable.

Example:

echo "Enter your name: "

read name

echo "Hello, $name!"

The syntax of read is as follows:

read [options] var1 var2 … varN

We can read multiple values from user input as follows:

$ read name usn marks

Foo 345 78

$ echo $name $usn $marks

Foo 345 78

We can read only the n characters and don't wait for the user to input a complete line as follows:

$ read -n 5 # option -n number takes only 5 characters from user

Input: Hello

$ echo $REPLY

Hello

If no variable in an argument is specified, the input value by a user will be stored in

the inbuilt variable REPLY and can be accessed further using $REPLY.

Command Output in Variables:

current\_date=$(date) # Stores the current date and time in variable

echo "Today's date is: $current\_date"

Exporting Variables (Making them Global):

To make a variable available to other scripts or processes, use the export command.

Example:

export my\_var="This is an exported variable"

Unset Variables:

To delete a variable and free up memory, use the unset command.

Example:

unset my\_var

* Defining: variable\_name="value"
* Accessing: $variable\_name
* Reading user input: read variable\_name
* Environment variables: $HOME, $PATH, etc.
* Functions: Variables inside functions are local by default.
* Command output: variable\_name=$(command)
* Arithmetic: (( )) for calculations.
* Exporting: export variable\_name

readonly command:

readonly command is used to make a variable immutable, meaning that its value cannot be changed after it is set.

Syntax:

readonly variable\_name="value"

If you try to assign a new value to a readonly variable, the shell will produce an error.

Example:

readonly PI=3.14159

echo "Value of PI: $PI"

PI="New value" # This will produce an error: "PI: readonly variable"

We can also use declare -r followed by a variable name to make it constant

declare -r number=27

Troubleshooting a shell script:

bash -x script.sh

bash command with -x, shell allows us to see commands that the shell is executing.

Shell Parameters:

|  |  |
| --- | --- |
| **Parameter** | **Function** |
| $1-$9 | Represent positional parameters for arguments one to nine. |
| ${10}-${n} | Represent positional parameters for arguments after nine. |
| $0 | Represent the name of the script. |
| $\* | Represent all the arguments as a single string. |
| $@ | Same as $\*, but differs when enclosed in quotes ("$@"). |
| $# | Represent the total number of arguments passed to the script. |
| $$ | The process ID (PID) of the script. |
| $? | Represent the last return code (exit status) of the last command. |

Array Concept in Shell Scripting:

an array is a data structure that allows you to store multiple values (or elements) in a single variable. Unlike other programming languages, arrays in shell scripts are not strongly typed, and they don't have predefined sizes. You can store values of any type (strings, integers) in an array, and the array can dynamically grow or shrink as you add or remove elements.

Creating Arrays:

fruits=("Apple" "Banana" "Orange")

This is equivalent to:

fruits[0]="Apple"

fruits[1]="Banana"

fruits[2]="Orange"

Accessing Array Elements:

echo ${array\_name[index]}

Example:

echo ${fruits[0]} # Output: Apple

echo ${fruits[1]} # Output: Banana

To access the entire array (all elements), use the following syntax

Example:

echo ${fruits[@]} # Output: Apple Banana Orange

echo ${fruits[\*]} # Output: Apple Banana Orange

Array Length:

echo ${#fruits[@]} # Output: 3

get range of values from array:

from 2nd+2values(2-4):

echo ${fruits[@]:2:2}

from 3rd to all(3-\*):

echo ${fruits[@]:3}

add new valus to existing array:

fruits+=(str1 str2 str3)

update new valus in existing array:

fruits[x]=strx

Removing Array Element:

unset fruits[1] # Removes the element at index 1 (Grapes)

Key-Value Arrays in Shell Scripting (Associative Arrays):

key-value pairs are a way to store data where each item is identified by a unique key, rather than by an index number.

Creating and Using Associative Arrays in Bash:

declare -A array\_name

Example:

declare -A capitals

capitals["USA"]="Washington, D.C."

capitals["India"]="New Delhi"

capitals["Germany"]="Berlin"

---or---

declare -A capitals

capitals=(["USA"]="Washington, D.C." ["India"]="New Delhi" ["Germany"]="Berlin")

Accessing Values in Associative Arrays:

${array\_name[key]}

Example:

echo ${capitals["USA"]} # Output: Washington, D.C.

echo ${capitals["India"]} # Output: New Delhi

Modifying Values in Associative Arrays:

capitals["USA"]="New York"

echo ${capitals["USA"]} # Output: New York

Deleting Key-Value Pairs:

unset capitals["Germany"]

echo ${capitals["Germany"]} # Output: (empty)

Getting All Keys and All Values:

To get all keys in an associative array, use ${!array\_name[@]}.

To get all values, use ${array\_name[@]}.

Example:

echo "Keys: ${!capitals[@]}" # Output: USA India Germany

echo "Values: ${capitals[@]}" # Output: Washington, D.C. New Delhi Berlin:

Comparison operators:

* -eq (is equal to) - [ $a -eq $b ]
* -ne (is not equal to) - [ $a -ne $b ]
* -gt (is greater than) - [ $a -gt $b ]
* -ge or >= (is greater than or equal to) - [ $a -ge $b ]
* -lt (is less than) - [ $a -lt $b ]
* -le (is less than or equal to) - [ $a -le $b ]
* < (is less than) - (($a < $b))
* <= (is less than or equal to) - (($a <= $b))
* > (is greater than) - (($a > $b))
* >= (is greater than or equal to) - (($a >= $b))

Comparison operators that are valid for string variables:

(consider a and b as two string variables; for example, a="Hello" b="World") are as follows:

* = (is equal to); for example, [ $a = $b ]
* != (is not equal to); for example, [ $a != $b ]
* < (is less than); for example, [ $a \< $b ] or [[ $a \< $b ]] or (( $a \< $b ))
* >(is greater than); for example,[ $a \> $b ] or [[ $a > $b ]] or (( $a \> $b ))
* -n (string is non-empty); for example,[ -n $a ]
* -z (string has zero length or null); for example,[ -z $a ]

Shell uses the < and > operators for redirection, so it should be used with an escape

(\) if used under [ … ]. Double parentheses, (( ... )) or [[ … ]], doesn't need an escape

sequence. Using [[ … ]] also supports pattern matching.

We will see the usage and examples of operators in more detail in Chapter 3, Effective

Script Writing

conditional statements with if and else:

Simple if and else:

The syntax of the if condition is as follows:

if [ conditional\_expression ]

then

statements

fi

The syntax of if and else is as follows:

if [ conditional\_expression ]

then

statements

else

statements

fi

The if, elif, and else statements:

if [ conditional\_expression1 ]

then

statements

elif [ conditional\_expression2 ]

then

statements

elif [ conditional\_expression3 ]

then

statements

# More elif conditions

else

statements

fi

Nested if:

if [ conditional\_expression1 ]

then

if [ conditional\_expression2 ]

then

if [conditional\_expression3 ]

then

statements

fi

fi

fi

example:

#!/bin/bash

# Input three numbers

echo "Enter three numbers:"

read -p "Enter first number: " num1

read -p "Enter second number: " num2

read -p "Enter third number: " num3

# Compare the numbers to find the greatest

if [ $num1 -ge $num2 ] && [ $num1 -ge $num3 ]; then

greatest=$num1

elif [ $num2 -ge $num1 ] && [ $num2 -ge $num3 ]; then

greatest=$num2

else

greatest=$num3

fi

# Output the greatest number

echo "The greatest number is: $greatest"

case statement:

The case statement is used for pattern matching. It checks a variable or expression against a set of patterns and executes the corresponding block of code.

case $variable in

pattern1)

# Code to execute if pattern1 matches

;;

pattern2)

# Code to execute if pattern2 matches

;;

\*)

# Code to execute if no patterns match

;;

esac

example:

#!/bin/bash

# Prompt user for input

read -p "Enter first number: " n1

read -p "Enter second number: " n2

read -p "Enter operator (+, -, \*, /): " op

# Perform the calculation and display the result

case $op in

+)

echo "$n1 $op $n2: `expr $n1 + $n2`"

;;

-)

echo "$n1 $op $n2: `expr $n1 - $n2`"

;;

\\*)

echo "$n1 $op $n2: `expr $n1 \\* $n2`"

;;

/)

# Check if second number is zero before division

if [ $n2 -eq 0 ]; then

echo "Error: Division by zero is not allowed."

else

echo "$n1 $op $n2: `expr $n1 / $n2`"

fi

;;

\*)

echo "Invalid operator! Please use one of the following: +, -, \*, /"

;;

esac

Test Command ([ ]) and Conditional Operators

In shell scripting, the condition inside [] is tested using operators. Some common operators include:

Comparison Operators for Numbers:

-eq: Equal to

-ne: Not equal to

-lt: Less than

-le: Less than or equal to

-gt: Greater than

-ge: Greater than or equal to

String Operators:

=, ==: Equal to

!=: Not equal to

-z: String is empty

-n: String is not empty

File Test Operators:

-e: File exists

-f: File is a regular file

-d: File is a directory

-r: File is readable

-w: File is writable

-x: File is executable

Example of file testing:

#!/bin/bash

filename="myfile.txt"

if [ -f $filename ]; then

echo "$filename exists and is a regular file."

else

echo "$filename does not exist or is not a regular file."

fi

6. Using Logical Operators

You can combine conditions using logical operators:

&& (AND): Both conditions must be true.

|| (OR): At least one condition must be true

Common String Operators:

=: Equal to

!=: Not equal to

-z: String is empty (i.e., length is 0)

-n: String is not empty (i.e., length is greater than 0)

Example 1: String Equality Check (=)

#!/bin/bash

# Read two strings

read -p "Enter first string: " str1

read -p "Enter second string: " str2

# Compare strings

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

echo "The strings are equal."

else

echo "The strings are not equal."

fi

Check if a String is Empty (-z):

#!/bin/bash

# Read a string

read -p "Enter a string: " str

# Check if the string is empty

if [ -z "$str" ]; then

echo "The string is empty."

else

echo "The string is not empty."

fi

Check if a String is Not Empty (-n):

#!/bin/bash

# Read a string

read -p "Enter a string: " str

# Check if the string is not empty

if [ -n "$str" ]; then

echo "The string is not empty."

else

echo "The string is empty."

fi

ternary operator:

exp?true:flase -- in c

[[ $age -ge 18 ]] && echo "adult" || echo "minor" -- in shell

looping constructs:

used to repeatedly execute a block of code based on certain conditions or a set of iterations. There are primarily three types of looping statements in shell scripts:

while loop

until loop

for loop

The select loop

while loop:

The while loop repeatedly executes a block of commands as long as the given condition evaluates to true.

syntax:

while [ condition ]

do

# commands to execute

done

Example 1: Loop until a condition is true:

count=1

while [ $count -le 5 ]

do

echo "Count: $count"

((count++))

done

until Loop:

The until loop executes a block of commands until a given condition is true. It is the opposite of the while loop.

Syntax:

until [ condition ]

do

# commands to execute

done

Example 1: Loop until a condition becomes true:

count=1

until [ $count -gt 5 ]

do

echo "Count: $count"

((count++)) -or- count=`expr $count - 1` --or-- let count++

done

for Loop

The for loop is used when you know in advance how many times you want to execute a statement or a block of statements.

Syntax:

for variable in list

do

# commands to execute

done

Example 1: Iterating over a list of values:

for i in 1 2 3 4 5

do

echo "Number: $i"

done

Example 2: Iterating a range of numbers:

for i in {1..5}

do

echo "Number: $i"

done

Example 3: Using C-style syntax for for loop:

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

do

echo "Iteration: $i"

done

Controlling Loops

break: Exits the loop entirely.

continue: Skips the rest of the current iteration and moves to the next iteration.

ls > names

reads content of fil, line by line:

while read l

do

echo "content: $l"

done < names

echo "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"

reads content of file, word by word:

file=/home/hr384/scripts/names

for line in $(cat $file)

do

echo "content: $line"

done

combined:

#!/bin/bash

# File to read

filename="sample.txt"

# Read each line from the file

while read line

do

# Process each word in the line

for word in $line

do

echo "Word: $word"

done

done < "$filename"

Using IFS to Handle Special Characters:

(Internal Field Separator) to split lines differently.

#!/bin/bash

# File to read

filename="sample.txt"

# Set Internal Field Separator to comma (for CSV-like files)

IFS=","

# Read each line from the file

while read line

do

# Process the comma-separated values

for word in $line

do

echo "Word: $word"

done

done < "$filename"

IFS to any of the following types of symbols or characters:

Space (' ')

Tab ($'\t')

Newline ($'\n')

Comma (,)

Colon (:)

Semicolon (;)

Pipe (|)

Slash (/)

Hyphen (-)

Equal sign (=)

Period (.)

Combinations of multiple delimiters (,:;/\)

Function:

functions allow you to organize your code into reusable blocks that can perform specific tasks. Functions are particularly useful for avoiding code repetition and improving the readability of your script. Here's a basic overview of how to define and use functions in shell scripting:

A function in a shell script can be defined in two main ways:

Using the function keyword:

function function\_name {

# function body

}

Without using the function keyword:

function\_name() {

# function body

}

calling function:

once a function is defined in the script, you can call it by using its name followed by with / without parentheses () if it takes arguments, or just the function name if it doesn't.

function\_name arg1 arg2 argN

#!/bin/bash

# Function definition

greet() {

echo "Hello, $1!" # $1 is the first argument passed to the function

}

# Function call

greet "Alice"

Parameters

Functions can take arguments (also called parameters).

$1, $2, $3, etc., represent the first, second, third, etc., arguments passed to the function.

$@ or $\* can be used to refer to all the arguments passed to a function.

$# gives the number of arguments passed to the function.

Returning Values:

Returning a Value with echo:

#!/bin/bash

# Function to multiply two numbers

multiply() {

local result=$(( $1 \* $2 ))

echo $result # return result by echoing it

}

# Call the function and store the result

result=$(multiply 4 3)

echo "Result: $result"

Returning a Value with return:

fun0(){

echo "fun0 val:0"

return 0

}

6 fun1(){

echo "fun1 val:1"

return 1

}

fun2(){

echo "fun2 val:2"

return 2

}

fun3(){

echo "fun3 val:3"

return 3

}

fun4(){

echo "fun4 val:4"

re=irshaq

echo $re

}

fun0

echo "$?"

fun1

echo "$?"

fun2

echo "$?"

fun3

echo "$?"

result=$(fun4)

echo "$result"

Arguments:

Types of Arguments:

Script Arguments: These are passed to the script when you run it from the command line.

Function Arguments: These are passed to a function when it is called within the script.

Let's explore both types in more detail.

1. Script Arguments

When you execute a shell script, you can pass arguments to the script on the command line. These arguments are then accessed within the script using special variables.

Special Variables for Script Arguments:

$0: The name of the script.

$1, $2, $3, ..., $n: These are the positional arguments passed to the script (up to the nth argument).

$#: The total number of arguments passed to the script.

$@ or $\*: All the arguments passed to the script as a single string. $@ is more commonly used as it treats each argument as a separate entity when enclosed in quotes.

$?: The exit status of the last executed command or function.

$$: The process ID of the script itself.

Using shift with Arguments

shift moves all arguments to the left, so each time the loop runs, the script processes the next argument ($1).

The loop continues as long as the number of arguments ($#) is greater than 0.

Example:

#!/bin/bash

echo "all args: $\*"

echo "total: $#"

for i in $\*

do

echo "$1"

shift

done

If you run the script as:

./myscript.sh Hello World 123

The output will be:

Argument: Hello

Argument: World

Argument: 123

useful concepts:

basename: Extracts the file name from a given path, removing the directory portion.

Example: basename /home/user/file.txt outputs: file.txt.

dirname: Extracts the directory path from a full file path, removing the file name.

Example: dirname /home/user/file.txt outputs: /home/user.

realpath: Resolves and prints the absolute, canonical path of a file, following symlinks and resolving relative paths.

Example: realpath myfile.txt outputs the full absolute path: /home/user/myfile.txt.

difference b/w @ and \* in arrays:

10 20 30 40 50

arr=("$\*")

arr2=("$@")

echo "\\*: $\*"

echo "\#: $@"

echo "arr[0]: ${arr[0]}" : 10 20 30 40 50

echo "arr2[0]: ${arr2[0]}" : 10

echo "#arr[0]: ${#arr[0]}" : 14

echo "#arr2[0]: ${#arr2[0]}" : 2

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

substring extraction from a string concept:

1. Using Parameter Expansion (Bash)

Bash provides a powerful feature called parameter expansion, which allows you to extract substrings directly from a variable.

Syntax:

${string:start:length}

string: The variable that holds the string.

start: The starting index (0-based) from where the extraction begins.

length: The number of characters to extract. This is optional, and if not provided, it extracts everything from the start position to the end of the string.

Examples:

Extract from position 3, length 4:

mystring="Hello, World!"

substring="${mystring:3:4}" # Extracts "lo, "

echo $substring

Extract from position 7 to the end:

mystring="Hello, World!"

substring="${mystring:7}" # Extracts "World!"

echo $substring

Extract starting from the end (negative index):

mystring="Hello, World!"

substring="${mystring: -6:5}" # Extracts "World"

echo $substring

RANDOM:

Random Number Generation: The RANDOM variable generates a random integer each time it is accessed. The numbers are pseudo-random, meaning they are generated using an algorithm based on a seed value, which is usually derived from the system's current time. As a result, RANDOM produces numbers that appear random, but are not truly random.

echo $RANDOM

Generating a Random Number Within a Specific Range:

To generate a random number within a specific range (for example, between 0 and 99), you can use the modulo operator (%).

echo $((RANDOM % 100))

to include 100 also

echo $((RANDOM % 100 + 1))

simple toss script:

if [ $((RANDOM % 2)) -eq 0 ]; then

echo "Heads"

else

echo "Tails"

fi

if you only 3 random numbers:

echo "${RANDOM:0:3}"

for i in {1..100}

do

echo -e"$RANDOM"

done

shuf command:

shuf -i 100-500

generates 400 random numbers.

shuf -i 1-50 -n x[any number b/w 1-50] genersates those many outputs

debugging command:

set -xue:

set -x:

gives detailed output of echo line of script

set -u:

gives information realted to usage of undeclaired variables

set -e:

when command fail exits from sccript.