Table of Contents

1. Basic-bash-script
    - Inputs
    - Variables
    - Variable Operations
    - Variable Types
    - Arithmetic Operations
    - Condition Checking
    - Numbers
    - Strings
    - Files and Directories
    - Outputs
    - Source Command
    - IF-Else Use Case
    - Functions in Bash
    - Advanced-Bash-script
2. Advanced-Bash-script
    - Variable Operations
    - String Operations
    - Flow Control
    - File Processing

Inputs

Positional Parameters

Positional parameters allow you to pass arguments to the script when it is executed.

- `$0`: The name of the script.
- `$1`, `$2`, ... : The first, second, etc., arguments passed to the script.
- `$#`: The number of arguments passed to the script.

Input after Script Run

You can read input from the user after the script has started using the `read` command.

```
read $NAME
echo $NAME
```

Variables

Variable Operations

- **Declare a variable:** `declare VAR_NAME=value`
- **Access a variable:** `$VAR_NAME`
- **Unset a variable:** `unset VAR_NAME`
- **Read-only variable:** `readonly VAR_NAME` or `declare -r VAR_NAME`

Example:

```
declare Variable="AHMED"
echo ${Variable}

Variable="WAGDY"
echo ${Variable}

unset Variable
echo ${Variable}

declare -r VAR_NAME="MOHY"
echo ${VAR_NAME}
```

Variable Types

- **Local Variables:** Declared inside a function.
- **Global Variables:** Declared outside a function.
- **Environment Variables:** System variables accessible across shell sessions.

Example of an environment variable:

```
export environment_var="Bash"

echo ${environment_var}
```

Arithmetic Operations

Bash supports basic arithmetic operations. You can use the `declare -i` flag to define integer variables.

```
declare -i num1=20
declare -i num2=50

declare -i result=$(($num1+$num2))
echo $result

declare -i result=$(($num1-$num2))
echo $result
```

Condition Checking

Numbers

```
if (($num1 > $num2)); then
    echo "num1 is greater than num2"

else
    echo "num1 is less than num2"
fi
```

Strings

```
declare KERNEL_TYPE="Linux"
declare CURRENT_KERNEL_TYPE=`uname -s`

if [[ $KERNEL_TYPE == $CURRENT_KERNEL_TYPE ]]; then
    echo "Both are same"
else
    echo "Both are different"
fi

if [[ -z $KERNEL_TYPE ]]; then
    echo "Kernel Type is empty"
    #-z Flag: This flag checks if the length of the string is zero
fi

if [[ -n $KERNEL_TYPE ]]; then
    echo "Kernel Type is not empty"
    #-n Flag: This flag checks if the length of the string is non-zero.
fi
```

Files and Directories

Check if a file or directory exists and perform operations accordingly.

```
if [[ -s $PWD/test.txt ]]
then
    #-s Flag: Checks if a file exists and is not empty.
    echo "File exists and not empty"
else
    if [[ -e $PWD/test.txt ]]
    then
        #-e Flag: Checks if a file or directory exists.But it can be empty
        echo "File exists and empty"


    else
        #-d Flag: Checks if a directory exists.
        echo "File does not exist"
    fi
fi

if [[ ! -d $PWD ]]
then
    echo "It is a directory"
fi
```

Another Example :

```
# Make sure path ends with /
for directory in '/home/wagdy/Desktop/embedded/'*; do
    if [[ -d "${directory}" && ! -L "${directory}" ]]; then
        #the -d flag Checks if ${directory} exists and is a directory
        #-L "${directory}": Checks if ${directory} exists and is a symbolic
link.
        # and we make sure that we don't have a symbolic link

        echo "${directory}"
    fi
done
```

Magic Variables

The magic variables inside the functions are used to print some important information about the fun.

```
    echo "$#" #print the number of arguments
    echo "$@" #print all the arguments
    echo "$*" #print all the arguments
    echo "$0" #print the name of the script
    echo "$?" #print the exit status of the last command
    echo "$$" #print the process id of the current script
    #tha hash carries integer datatype
```

Outputs

The script demonstrates two types of outputs:

1. **Exit Status:** Indicates the success or failure of the last command (`0`-`255`).
2. **Standard Output:** Displayed using commands like `echo`, `printf`, etc.

```
echo "Exit Status: $?"
echo $$
```

Source-Command

When you use the `source` command, the commands in the script are executed within the current shell session. This means that any variables set, functions defined, or environment changes made by the script will persist in the current shell after the script has finished running.

```
source script_name.sh
```

Use-case-on-IF-ELSE

> make a small program that takes two number inputs and pick the operation from the user and apply it

```
#Use Case
#!/bin/bash

echo "Enter first number"
read FIRSTNUM
echo "Enter second number"
read SECONDNUM
echo "Enter operator choose with +,-,*,/"
read OPERATOR

if [ "$OPERATOR" == "+" ]; then
    echo "THE sum is $((FIRSTNUM + SECONDNUM))"
elif [ "$OPERATOR" == "-" ]; then
    echo "The sub is $((FIRSTNUM + SECONDNUM))"

fi


echo "THE sum is $((FIRSTNUM + SECONDNUM))"
```

Functions-in-BASH

> Any variable whether it is written inside a function or outside a function by default is a global variable To make it Local use the `local` before the varaible

```
function print(){
    echo "$1" #make the function takes the fist argument from the user

}

#Calling the function
print "HHHHHHI"
```

The output will be like :

```
HHHHHHI
```

Example of how to reviece a return value from a funciton

> [!NOTE]
> the fucntion only Returns the last echo , If you want to receive multiple values you can pass it to the echo Echo here doesn't print in the console output but it acts as a return for the function

```
function sum(){
    declare -i RESULT=$(($1 + $2))
    echo "$RESULT"
    return 0
}
#We take the function return which's echo and store it in the variable
RESULT=$(sum 5 6)
echo "${RESULT}"
```

Advanced-Bash-script

Bash Script: Advanced Variable and String Operations

This script demonstrates various advanced operations in Bash, including variable operations, string manipulations, flow control, and file processing. It serves as a reference for different syntax and usage patterns in Bash scripting.

Variable-Operations-2

Default Value Assignment

1. **`${VARIABLE:-default}`**
   If `VARIABLE` is unset or empty, use `default`.
   Example:

   ```
   declare NAME=""
   echo "Name: ${NAME:-WAGDY}" # Outputs "WAGDY" since NAME is empty
   ```

2. **`${VARIABLE:=default}`**
   If `VARIABLE` is unset, set it to `default`.
   Example:

```
echo "Name: ${NAME:=WAGDY}" # Sets NAME to "WAGDY" if it wasn't
defined
```

3. `${VARIABLE:?default}`
   If `VARIABLE` is unset, raise an error with `default` message.
   Example:

   ```
   echo "Name: ${NAME2:?WAGDY}" # Raises an error if NAME2 is not
   declared
   ```

String Operations

Substring Extraction

1. `${string:position:length}`
   Extract a substring from `string` starting at `position` for `length` characters.
   Example:

   ```
   declare str="Hello World"
   echo "String: ${str:3}" # Outputs "lo World"
   echo "String: ${str: -3}" # Outputs "rld"
   ```

Pattern Matching

1. `${string%substring}`
   Remove the shortest match from the end.
   Example:

   ```
   declare filename="file.txt"
   declare basic_name=${filename%.*}
   echo "Basic Name: $basic_name" # Outputs "file"
   ```

2. `${string##substring}`
   Remove the longest match from the beginning.
   Example:

   ```
   declare extension=${filename##*.}
   echo "Extension: $extension" # Outputs "txt"
   ```

3. **Search Pattern Inside String**
   Check if a string contains a specific substring.

Example:

```
declare string="Linux is fun, Hello World"
if [[ "${string}" == *World* ]]; then
    echo "Found" # Outputs "Found"
fi
```

Trimming Spaces

1. **Trim Left and Right Spaces**
   Example:

```
declare str2="  Hello World  "
trimmed=$(echo -e "${str2}" | sed -e 's/^[[:space:]]*//' | sed -e
's/[[:space:]]*$//')
echo "Trimmed: $trimmed" # Outputs "Hello World"
```

Flow Control

Loops

1. **While Loop**
   Example:

```
declare -i a=0
while (( $a<5 )); do
    echo "${a}"
    ((a++))
done
```

2. **For Loop**
   Example:

```
for i in {1..5}; do
    echo "${i}"
done
```

Switch Case

1. **Case Statement**
   Example:

```bash
declare os=$1
case "${os}" in
    linux) echo "case: ${os}" ;;
    windows) echo "windows" ;;
    *) echo "Undefined Operating System" ;;
esac
```

Select Statement

1. **Create User Menu**

   Example:

   ```bash
   select os in "Linux" "Windows"; do
        echo ${os}
   done
   ```

File Processing

Reading Files

1. **Read File Line by Line**

   Example:

   ```bash
   cat $PWD/test.txt | while read line; do
       echo "${line}"
   done

   while IFS= read -r line; do
       echo "$line"
   done < test.txt
   ```

2. **Conditional Line Processing**

   Example:

   ```bash
   cat $PWD/test.txt | while read line; do
       if [[ "${line}" == "linux" ]]; then
           echo "Linux Kernel"
       fi
   done
   ```

Writing to Files

1. **Write with Redirection**

   Example:

```
echo "operating-system:GNU/Linux" > ./file.txt
```

2. **Append to File**
   Example:

```
echo "Appended Data" >> ./file.txt
```