Bash Scripting

Detailed Lesson on Bash Scripting

Bash scripting allows for automation of tasks and efficient handling of repetitive commands. This lesson covers the basics of creating and running Bash scripts, variables, control structures, functions, and useful examples.

---

1. Introduction to Bash Scripting

Bash (Bourne Again SHell) is a command-line interpreter used primarily in Linux and Unix systems. A **Bash script** is essentially a series of commands written in a file that can be executed to automate tasks.

1.1 Creating and Running a Bash Script

1. **Create a script**:
   You can create a Bash script using any text editor (e.g., `nano`, `vim`, or `gedit`).

   ```
   nano myscript.sh
   ```

2. **Write the script**:
   Every Bash script starts with a shebang (`#!`) followed by the path to the Bash interpreter:

   ```
   #!/bin/bash
   echo "Hello, World!"
   ```

3. **Make the script executable**:
   Change the file's permissions to make it executable:

   ```
   chmod +x myscript.sh
   ```

4. **Run the script**:
   To run the script, use:

   ```
   ./myscript.sh
   ```

---

2. Bash Scripting Basics

2.1 Variables

In Bash, variables store data. There's no need to declare a variable type (string, integer, etc.).

```
#!/bin/bash
name="John"
age=25
echo "My name is $name and I am $age years old."
```

- **Note**: No spaces around the `=` sign while assigning values.
- Variables are referenced using a `$` prefix.

2.2 User Input

You can prompt the user for input using the `read` command.

```
#!/bin/bash
echo "Enter your name:"
read name
echo "Hello, $name!"
```

2.3 Command Substitution

You can store the output of a command into a variable using command substitution (`$(command)`).

```
#!/bin/bash
current_date=$(date)
echo "Today's date is $current_date"
```

---

3. Control Structures

Control structures allow you to direct the flow of the script based on conditions and loops.

3.1 Conditional Statements (`if-else`)

The `if-else` statement allows you to execute code based on conditions.

```
#!/bin/bash
read -p "Enter a number: " number

if [ $number -gt 10 ]; then
  echo "The number is greater than 10."
else
  echo "The number is 10 or less."
fi
```

- **Note**: Square brackets `[ ]` are used for conditions. Use `-gt` (greater than), `-lt` (less than), `-eq` (equal), etc., for comparisons.

3.2 Loops

For Loop

A `for` loop is used to iterate over a list of items.

```
#!/bin/bash
for i in {1..5}; do
  echo "Number $i"
done
```

While Loop

A `while` loop runs as long as the condition is true.

```
#!/bin/bash
counter=1
while [ $counter -le 5 ]; do
  echo "Counter is $counter"
  ((counter++))  # Increment counter
done
```

---

4. Functions

Functions allow you to group code into reusable blocks.

```
#!/bin/bash

greet() {
  echo "Hello, $1!"
}

greet "Alice"  # Calling the function with an argument
```

In this example, the function `greet` is called with the argument "Alice". The `$1` represents the first argument passed to the function.

---

5. Handling Arguments

Bash scripts can accept arguments passed during execution. These arguments are accessed using special variables:

- `$0`: The name of the script.
- `$1, $2, ...`: Positional arguments.
- `$#`: The number of arguments passed.
- `$@`: All arguments passed to the script.

```
#!/bin/bash
echo "Script name: $0"
echo "First argument: $1"
echo "Second argument: $2"
```

To run the script:

```
./myscript.sh arg1 arg2
```

---

6. File Operations

6.1 Reading a File

You can read the contents of a file line by line using a `while` loop.

```
#!/bin/bash
filename="sample.txt"
while IFS= read -r line; do
  echo "$line"
done < "$filename"
```

- **IFS** is the internal field separator, set to handle spaces properly.

6.2 Writing to a File

Use redirection (`>`, `>>`) to write output to a file.

```
#!/bin/bash
echo "This is a new line" > output.txt   # Overwrite file
echo "This is another line" >> output.txt  # Append to file
```

---

7. Error Handling

7.1 Exit Status

Every command returns an exit status. `0` indicates success, while any non-zero value indicates an error.

```
#!/bin/bash
mkdir mydir
if [ $? -eq 0 ]; then
  echo "Directory created successfully."
else
  echo "Failed to create directory."
fi
```

- `$?`: Contains the exit status of the last command.

7.2 Using `set -e` for Error Handling

`set -e` tells Bash to exit the script if any command fails (non-zero exit status).

```
#!/bin/bash
set -e
mkdir mydir
cd mydir
```

---

8. Script Debugging

You can debug your Bash scripts by adding the `-x` option in the shebang line.

```
#!/bin/bash -x
```

This will show the commands being executed line by line, which helps in debugging.

---

9. Practical Bash Script Examples

9.1 Backup Script

```
#!/bin/bash
backup_dir="/backup"
source_dir="/home/user/documents"

mkdir -p "$backup_dir"
cp -r "$source_dir"/* "$backup_dir"
echo "Backup completed!"
```

9.2 Disk Usage Monitor

```
#!/bin/bash
threshold=80
disk_usage=$(df / | grep / | awk '{print $5}' | sed 's/%//')

if [ $disk_usage -gt $threshold ]; then
  echo "Warning: Disk usage is above $threshold%!"
else
  echo "Disk usage is under control."
fi
```

10. Best Practices

1. **Use comments**: Add comments to explain sections of your script.

```
# This is a comment
```

2. **Test scripts in small pieces**: Especially when dealing with complex logic.
3. **Error checking**: Always check for errors and handle them gracefully.
4. **Use meaningful variable and function names**: Make your scripts readable and maintainable.
5. **Use shebang at the top**: Always include `#!/bin/bash` to ensure your script runs in the correct shell.

Summary

- Bash scripting is a powerful tool for automating tasks and managing repetitive operations.
- Variables, control structures (if-else, loops), functions, and file handling are key components.
- You can use Bash scripting for tasks like backups, monitoring, file manipulation, and more.