A SMALL HANDBOOK

BASH SCRIPTING

All Necessary Commands From Basics to Advance

VIEH GROUP

Chapter 1
Bash Basics

Bash, short for "Bourne Again SHell," is a widely-used command-line interpreter for Unix and Linux systems. It provides a powerful scripting environment for automating tasks, managing files, and interacting with the system.

Section 1: Getting Started

1.1 Installation

Most Unix-based systems come with Bash pre-installed. However, if you need to install it on your own, you can use package managers like apt or yum on Debian/Ubuntu or Red Hat based systems, respectively.

```
# For Debian/Ubuntu
sudo apt-get update
sudo apt-get install bash
```

1.2 Running Bash

You can access Bash by opening a terminal. Type `**bash**` and press Enter to start an interactive Bash shell. You can exit the shell using the `**exit**` command.

Section 2: Basic Commands

2.1 Command Structure

Bash commands follow a simple structure:

```
command [options] [arguments]
```

For example:

```
ls -l /path/to/directory
```

Here, `**ls**` is the command, `**-l**` is an option, and `**/path/to/directory**` is the argument.

2.2 Working with Files and Directories

`**ls**`: List files and directories.
`**cd**`: Change directory.
`**cp**`: Copy files or directories.
`**mv**`: Move or rename files and directories.
`**rm**`: Remove files or directories.

Section 3: Variables and Data Types

3.1 Variables

In Bash, variables are created without explicit declaration:

```
name="John"
```

To use the variable:

```
echo "Hello, $name!"
```

3.2 Data Types

Bash primarily deals with strings, but you can perform arithmetic operations:

```
num1=5
num2=3
sum=$((num1 + num2))
echo "The sum is: $sum"
```

Section 4: Control Structures

4.1 Conditionals

```
if [ condition ]; then
    # Code to execute if condition is true
else
    # Code to execute if condition is false
fi
```

4.2 Loops

For Loop

```
for item in "${array[@]}"; do
    # Code to execute for each item in the array
done
```

While Loop

```
while [ condition ]; do
    # Code to execute while the condition is true
done
```

Section 5: Functions

Functions allow you to group code for reuse:

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

Section 6: Input and Output

6.1 Standard Input, Output, and Error

In Bash, each process has three standard file descriptors:

`0` (stdin): Standard input.
`1` (stdout): Standard output.
`2` (stderr): Standard error.

Redirecting output can be done using **`>`** for stdout and **`2>`** for stderr:

```
echo "This goes to file.txt" > file.txt
```

6.2 Pipes

Pipes **(`|`)** allow you to combine commands, sending the output of one command as input to another:

```
ls -l | grep "file"
```

Section 7: Environment Variables

7.1 Predefined Variables

Bash provides several built-in variables, such as:

```
`$HOME`: User's home directory.
`$PATH`: Search path for executables.
`$USER`: Current username.
```

7.2 Custom Variables

You can set your own variables:

```
export MY_VARIABLE="some value"
```

And access them in other scripts or sessions.

Section 8: Scripting

8.1 Shebang

The shebang **(`#!`)** at the beginning of a script specifies the interpreter:

```
#!/bin/bash
```

8.2 Execution Permissions

Make your script executable:

```
chmod +x script.sh
```

And run it:

```
./script.sh
```

Section 9: Conditional Expressions

Bash supports various conditional expressions:

- String comparisons: `==`, `!=`.
- Numeric comparisons: `-eq`, `-ne`, `-lt`, `-le`, `-gt`, `-ge`.
- File tests: `-e` (exists), `-f` (is a regular file), `-d` (is a directory).

Section 10: Arrays

Arrays in Bash are versatile:

```
my_array=("apple" "banana" "cherry")

# Access elements
echo ${my_array[0]}

# Iterate through elements
for fruit in "${my_array[@]}"; do
    echo $fruit
done
```

Section 11: Advanced Scripting Techniques

11.1 Command Substitution

Command substitution allows you to capture the output of a command and use it as part of another command or assignment:

```
current_date=$(date)
echo "Today is $current_date"
```

11.2 Arithmetic Operations

Bash supports arithmetic operations directly:

```
num1=5
num2=3
result=$((num1 * num2))
echo "The result is: $result"
```

11.3 String Manipulation

Manipulating strings is a common task in scripting:

```
string="Hello, World!"
substring=${string:0:5}  # Extracts "Hello"
length=${#string}        # Gets the length of the string
```

Section 12: Debugging

12.1 Debugging Mode

Activate debugging mode to trace script execution:

```
bash -x script.sh
```

12.2 `set` Command

The set command allows fine-grained control over script behavior, including options like -e to exit on error and -u to treat unset variables as an error.

Section 13: Best Practices

13.1 Code Organization

Keep your scripts organized with functions, comments, and clear indentation. This enhances readability and maintainability.

13.2 Error Handling

Implement robust error handling by checking command return codes and responding accordingly:

```
if [ $? -eq 0 ]; then
    echo "Command executed successfully."
else
    echo "Command failed with an error."
fi
```

Chapter 2 Advanced Bash Concepts

Section 1: Functions and Modularity

1.1 Creating Functions

Functions allow you to modularize your scripts for better organization and reusability:

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

greet "Alice"
```

1.2 Return Values

While Bash functions do not return values conventionally, you can use global variables or command substitution:

```
function add() {
    result=$(( $1 + $2 ))
}

add 5 3
echo "The sum is: $result"
```

Section 2: File I/O

2.1 Reading from Files

Reading from files is essential. Use ***`while read`*** to iterate through lines:

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

2.2 Writing to Files

Appending or overwriting files can be done with >> and >:

```
echo "New content" >> file.txt
```

Section 3: Regular Expressions

3.1 Pattern Matching

Bash supports pattern matching with the `=~`operator:

```
if [[ $string =~ .*pattern.* ]]; then
    echo "Pattern found in the string."
fi
```

3.2 Extracting Matches

Capture groups can be used to extract specific parts of a matched pattern:

```
if [[ $string =~ ([0-9]+) ]]; then
    echo "Number found: ${BASH_REMATCH[1]}"
fi
```

Section 4: Advanced Control Structures

4.1 Case Statement

A ***`case`*** statement is useful for multi-way branching:

```
case $option in
    "start")
        echo "Starting..."
        ;;
    "stop")
        echo "Stopping..."
        ;;
    *)
        echo "Unknown option."
        ;;
esac
```

4.2 Select Loop

The select loop simplifies interactive menus:

```
PS3="Choose an option: "
options=("Option 1" "Option 2" "Quit")
select choice in "${options[@]}"; do
    case $choice in
        "Option 1")
            echo "You chose Option 1."
            ;;
        "Option 2")
            echo "You chose Option 2."
            ;;
        "Quit")
            break
            ;;
        *)
            echo "Invalid choice."
            ;;
    esac
done
```

Section 5: Advanced Scripting Techniques

5.1 Process Substitution

Process substitution allows you to use the output of a command as input for another, providing a concise syntax:

```
diff <(command1) <(command2)
```

5.2 Array Manipulation

Arrays in Bash support powerful operations:

```
# Concatenation
array1=("apple" "banana")
array2=("cherry" "date")
concatenated=("${array1[@]}" "${array2[@]}")

# Slicing
sliced=("${array1[@]:1:2}")
```

Section 6: Signals and Traps

6.1 Signals

Bash scripts can respond to signals, allowing graceful termination:

```
trap 'cleanup_function' EXIT
trap 'handle_interrupt' INT
```

6.2 Traps

Traps are used to catch signals and execute specific actions:

```
trap 'echo "Script interrupted."' INT
```

Section 7: Advanced Environment Variables

7.1 `*$PS1*` - Customizing the Prompt

Customize your shell prompt for a personalized command-line experience:

```
PS1="\u@\h \w $ "
```

7.2 `*$PATH*` Manipulation

Extend the `***$PATH***` variable to include additional directories for executable files:

```
export PATH=$PATH:/path/to/your/directory
```

Section 8: Security Best Practices

8.1 Input Validation

Always validate user input to prevent injection attacks:

```
read -p "Enter a number: " userInput
if ! [[ $userInput =~ ^[0-9]+$ ]]; then
    echo "Invalid input. Please enter a number."
fi
```

8.2 Avoiding Hardcoded Passwords

Avoid storing passwords directly in scripts. Use environment variables or external secure mechanisms.