

# Shell Scripting Basics

Tejas Parikh ([t.parikh@northeastern.edu](mailto:t.parikh@northeastern.edu))

CSYE 6225

Northeastern University

# What Are Shell Scripts?

- In the simplest terms, a shell script is a file containing a series of commands.
- The shell reads this file and carries out the commands as though they have been entered directly on the command line.
- Shell scripts and functions are both interpreted. This means they are not compiled.
- Shell scripts and functions are ASCII text that is read by shell command interpreter.

# Why Write Scripts?

- Good system administrators write scripts.
- Scripts standardize and automate the performance of administrative chores and free up admins' time for more important and more interesting tasks.
- In a sense, scripts are also a kind of low-rent documentation in that they act as an authoritative outline of the steps needed to complete a particular task.
- The shell is always available, so shell scripts are relatively portable and have few dependencies other than the commands they invoke.

# Case Sensitivity

- Linux is case sensitive.

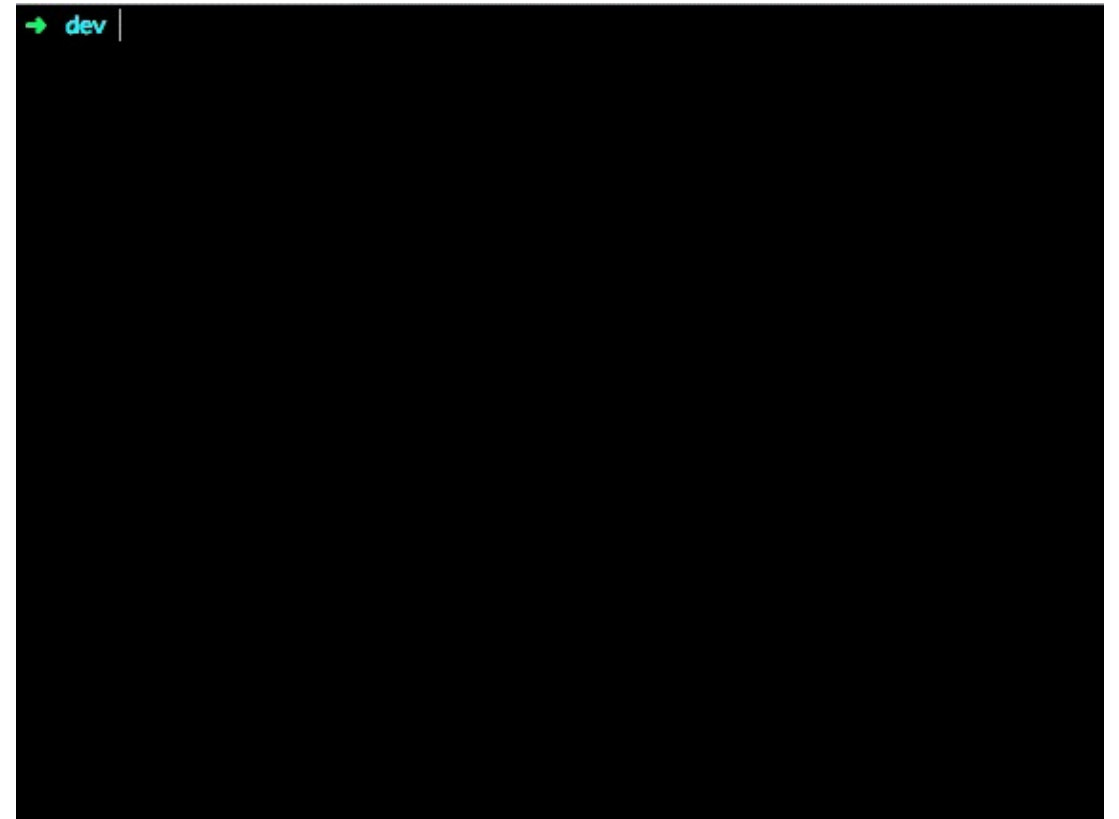
# Special Characters

- All of the following characters have a special meaning or function. If they are used in a way that their special meaning is not needed, they must be **escaped**.
- To escape, or remove its special function, the character must be immediately preceded with a backslash.
- `\ / ; , . ~ # $ ? & * ( ) [ ] ' ' " + - ! ^ = | < >`

# Writing a Shell Script

To create a shell script:

1. Use a text editor such as vi.  
Write required Linux commands and logic in the file.
2. Save and close the file (exit from vi).
3. Make the script executable.
4. Run the script.



# Declare the Shell in the Shell Script

- The `#!/` syntax used in scripts to indicate an interpreter for execution under UNIX / Linux operating systems.
- If no shell is declared, the script will execute in the *default* shell, defined by the system for the user executing the shell script.
- Shell declaration statement must appear on the *first line* of the shell script.
- Linux shell script that uses bash shell will start with the following line:

`#!/bin/bash`

# Comments in Shell Scripts

- Command readability and step-by-step comments are just the very basics of a well-written script.
- Using a lot of comments will make our life much easier when we have to come back to the code after not looking at it for six months, and believe me; we will look at the code again.
- Comment everything! This includes, but is not limited to, describing what our variables and files are used for, describing what loops are doing, describing each test, maybe including expected results and how we are manipulating the data and the many data fields.
- A hash mark, #, precedes each line of a comment.



# Variables

- A *variable* is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data. A variable is nothing more than a pointer to the actual data.
- Creating and setting variables within a script is fairly simple. Use the following syntax:

*varName=someValue*

**someValue** is assigned to given **varName** and **someValue** must be on right side of = (equal) sign. If **someValue** is not given, the variable is assigned the null string.

- You can display the value of a variable with **echo \$varName** or **echo \${varName}**

# Quoting

- Your bash shell understands special characters with special meanings. For example, `$var` is used to expand the variable value. Bash expands variables and [wildcards](#), for example:

```
echo "$PATH"  
echo "$PS1"  
echo /etc/*.conf
```

- Sometime you do not wish to use variables or wildcards. For example, do not print value of `$PATH`, but just print `$PATH` on screen as a word. You can enable or disable the meaning of a special character by enclosing them in single quotes. This is also useful to suppress warnings and error messages while writing the shell scripts.

```
echo "Path is $PATH" ## $PATH will be expanded
```

OR

```
echo 'I want to print $PATH' ## PATH will not be expanded
```

# Unset Variable

- Use unset command to delete the variables during program execution. It can remove both functions and shell variables.

```
vech=Bus  
echo $vech  
unset vech  
echo $vech
```

# Perform arithmetic operations

- Arithmetic expansion and evaluation is done by placing an integer expression using the following format:

```
$((expression))
```

```
$(( n1+n2 ))
```

```
$(( n1/n2 ))
```

```
$(( n1-n2 ))
```

- Add two numbers using x and y variable. Create a shell program called add.sh using a text editor:

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

```
x=5
```

```
y=10
```

```
ans=$(( x + y ))
```

```
echo "$x + $y = $ans"
```

# Bash variable existence check

In modern bash (version 4.2 and above):

```
if [[ -v name_of_var ]]
then
    echo "set"
else
    echo "not set"
```

*-v VAR, True if the shell variable VAR is set*

# If..else..fi

## if..then..else Syntax

---

```
if command
then
    command executed successfully
    execute all commands up to else statement
    or to fi if there is no else statement

else
    command failed so
    execute all commands up to fi

fi
```

# Nested ifs

You can put if command within if command and create the nested ifs as follows:

```
if condition
then
    if condition
    then
        .....
        ..
        do this
    else
        ....
        ..
        do this
    fi
else
    ...
    .....
    do this
fi
```

# for . . . in statement

```
for loop_variable in argument_list  
do  
  
    commands  
  
done
```



# while statement

```
while test_condition_is_true  
do  
  
    commands  
  
done
```

# case statement

```
case $variable in
    match_1)
        commands_to_execute_for_1
        ;;
    match_2)
        commands_to_execute_for_2
        ;;
    match_3)
        commands_to_execute_for_3
        ;;
    .
    .
    .

    *)      (Optional - any other value)
        commands_to_execute_for_no_match
        ;;
esac
```

# Command-Line Arguments

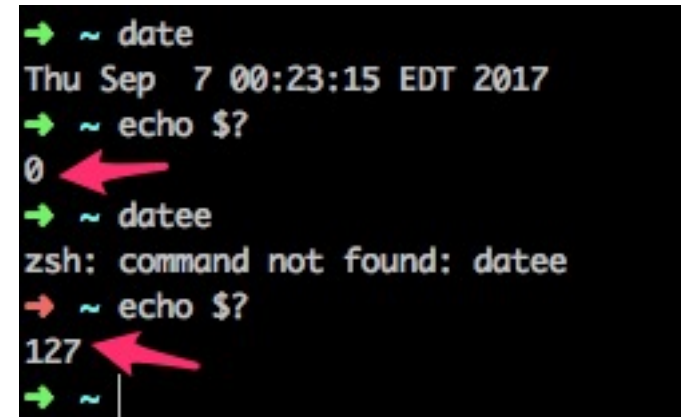
- The command-line arguments `$1`, `$2`, `$3`, . . . `$9` are positional parameters, with `$0` pointing to the actual command, program, shell script, or function and `$1`, `$2`, `$3`, . . . `$9` as the arguments to the command.
- The positional parameters, `$0`, `$2`, and so on in a function are for the function's use and may not be in the environment of the shell script that is calling the function.
- Where a variable is known in a function or shell script is called the scope of the variable.

# Special Parameters `$*` and `$@`

- There are special parameters that allow accessing all the command-line arguments at once.
- `$*` and `$@` both will act the same unless they are enclosed in double quotes, " ".
- The `$*` special parameter specifies all command-line arguments.
- The `$@` special parameter also specifies all command-line arguments.
- The `"$*"` special parameter takes the entire list as one argument with spaces between.
- The `"$@"` special parameter takes the entire list and separates it into separate arguments.

# The exit status of a command

- Each Linux command returns a status when it terminates normally or abnormally. You can use command exit status in the shell script to display an error message or take some sort of action.
- Exit Status
  - Every Linux command executed by the shell script or user, has an exit status.
  - The exit status is an integer number.
  - The Linux man pages state the exit statuses of each command.
  - 0 exit status means the command was successful without any errors.
  - A non-zero (1-255 values) exit status means command was failure.
  - You can use special shell variable called `?` to get the exit status of the previously executed command.



```
→ ~ date
Thu Sep  7 00:23:15 EDT 2017
→ ~ echo $?
0
→ ~ datee
zsh: command not found: datee
→ ~ echo $?
127
→ ~ |
```

A terminal window showing the execution of two commands. The first command is `date`, which outputs the current date and time. The second command is `datee`, which results in a "command not found" error. The exit status of the first command is shown as `0`, and the exit status of the second command is shown as `127`. Red arrows point to these exit status values.

# Check the Return Code (Script Example)

```
test    -d    /usr/local/bin
if [ "$?" -eq 0 ] # Check the return code
then      # The return code is zero

    echo '/usr/local/bin does exist'

else      # The return code is NOT zero

    echo '/usr/local/bin does NOT exist'

fi
```

# Conditional Execution

- You can link two commands under bash shell using conditional execution based on the exit status of the last command. This is useful to control the sequence of command execution. Also, you can do conditional execution using the if statement.
- The bash shell support the following two conditional executions:
  1. Logical AND && - Run second command only if first is successful.
  2. Logical OR || - Run second command only if first is not successful.

# Numeric Comparison

- [https://bash.cyberciti.biz/guide/Numeric\\_comparison](https://bash.cyberciti.biz/guide/Numeric_comparison)



# For loop and While loop

- Bash shell can repeat particular instruction again and again, until particular condition satisfies.
- A group of instruction that is executed repeatedly is called a loop.
- Bash supports:
  - The for loop
  - The while loop

# Linking Commands

Under bash you can create a sequence of one or more commands separated by one of the following operators

Operator	Syntax	Description	Example
;	command1; command2	Separates commands that are executed in sequence.	In this example, pwd is executed only after date command completes. date ; pwd
&	command arg &	The shell executes the command in the background in a subshell. The shell does not wait for the command to finish, and the return status is 0. The & operator runs the command in background while freeing up your terminal for other work.	In this example, find command is executed in background while freeing up your shell prompt. find / -iname "*.pdf" >/tmp/output.txt &
&&	command1 && command2	command2 is executed if, and only if, command1 returns an exit status of zero i.e. command2 only runs if first command1 run successfully.	[ ! -d /backup ] && mkdir -p /backup See <a href="#">Logical AND</a> section for examples.
	command1    command2	command2 is executed if and only if command1 returns a non-zero exit status i.e. command2 only runs if first command fails.	tar cvf /dev/st0 /home    mail -s 'Backup failed'you@example.com </dev/null See <a href="#">Logical OR</a> section for examples.
	command1   command2	Linux shell pipes join the standard output of command1 to the standard input of command2.	In this example, output of the <a href="#">ps command</a> is provided as the standard input to the <a href="#">grep command</a> ps aux   grep httpd

# Shell Functions

- Shell functions
- Sometime shell scripts get complicated.
- To avoid large and complicated scripts use functions.
- You divide large scripts into a small chunks/entities called **functions**.
- Functions makes shell script modular and easy to use.
- Function avoids repetitive code. For example, `is_root_user()` function can be reused by various shell scripts to determine whether logged on user is root or not.
- Function performs a specific task. For example, add or delete a user account.
- Function used like normal command.
- In other high level programming languages function is also known as procedure, method, subroutine, or routine.

```
hello() { echo 'Hello world!' ; }
```

# Trap Statement

- While running a script user may press Break or CTRL+C to terminate the process.
- User can also stop the process by pressing CTRL+Z.
- Error can occur do to bug in a shell script such as arithmetic overflow.
- This may result into errors or unpredictable output.
- Whenever user interrupts a signal is send to the command or the script.
- Signals force the script to exit.
- However, the trap command captures an interrupt.
- The trap command provides the script to captures an interrupt (signal) and then clean it up within the script.

# Additional Resources

See Lecture Page