Writing Scripts

OBJECTIVES:

- ✓105.2 Customize or write simple scripts
- ✓107.2 Automate system administration tasks by scheduling jobs

SHELL VARIABLES

Before talking about writing scripts, it's a good idea to see how the Bash shell stores data for us to use in our scripts. The Bash shell uses a feature called environment variables to store information about the shell session and the working environment (thus the name environment variables). This feature stores the data in memory so that any program or script running from the shell can easily access it. This is a handy way to store persistent data that identifies features of the user account, system, shell, or anything else you need to store. There are two types of environment variables in the Bash shell:

- ✓ Global variables
- ✓ Local variables

This section describes each type of environment variable and shows how to use them.

Global Environment Variables

- Global environment variables are visible from the shell session and from any child processes that the shell spawns.
- Local variables are available only in the shell that creates them.
- This makes global environment variables useful in applications that spawn child processes requiring information from the parent process.
- The Linux system sets several global environment variables when you start your Bash session.
- The system environment variables always use all capital letters to differentiate them from normal user environment variables.
- To view the global environment variables, use the `printenv` command.

```
$ printenv
HOSTNAME=testbox.localdomain
TERM=xterm
SHELL=/bin/bash
$ echo $HOME
/home/rich
```

Local Environment Variables

- Local environment variables, as their name implies, can be seen only in the local process in which they are defined.
- The Linux system also defines standard local environment variables for you by default.
- Unfortunately, there isn't a command that displays only local environment variables.
- The `set` command displays all environment variables set for a specific process.
  - ✓ However, this also includes the global environment variables.

```
$ set
BASH=/bin/bash
BASH_ARGC=()
BASH_ARGV=()
BASH_LINENO=()
BASH_SOURCE=()
...
```

Setting Local Environment Variables

- When you start a Bash shell (or spawn a shell script), you're allowed to create local variables that are visible within your shell process.
- You can assign either a numeric or a string value to an environment variable by assigning the variable to a value using the equal sign:

```
$ test=testing
```

- Any time you need to reference the value of the `test` environment variable, just reference it by the name `$test`.
- If you need to assign a string value that contains spaces, you'll need to use a quotation mark to delineate the beginning and the end of the string:

```
$ test=testing a long string
-bash: a: command not found
$ test='testing a long string'
```

Setting Local Environment Variables

- For the local environment variable we defined, we used lowercase letters, whereas the system environment variables we've seen so far have all used uppercase letters.
  - ✓ This is a standard convention in the Bash shell.
- If you create new environment variables, it is recommended (but not required) that you use lowercase letters.
- This helps distinguish your personal environment variables from the scores of system environment variables.

Setting Global Environment Variables

- Global environment variables are visible from any child processes created by the process that sets the global environment variable.
- The method used to create a global environment variable is to create a local environment variable and then export it to the global environment.
- This is done by using the `export` command:

```
$ echo $test
testing a long string
$ export test
$ bash
$ echo $test
testing a long string
```

Locating System Environment Variables

- The Linux system uses environment variables to identify itself in programs and scripts.
- This provides a convenient way to obtain system information for your programs.
- When you start a Bash shell by logging into the Linux system, by default Bash checks several files for commands.
  - ✓ These files are called startup files.
- The startup files Bash processes depend on the method you use to start the Bash shell.
- There are three ways of starting a Bash shell:
  - ✓ As a default login shell at login time
  - ✓ As an interactive shell that is not the login shell
  - ✓ As a noninteractive shell to run a script

Login Shell

- When you log into the Linux system, the Bash shell starts as a login shell.
- The login shell looks for four different startup files to process commands from.
- The `/etc/profile` file is the main default startup file for the Bash shell.
  - ✓ Whenever you log into the Linux system, Bash executes the commands in the `/etc/profile` startup file.
- There are three startup files are all used for the same function, to provide a user specific startup file for defining user-specific environment variables.
- Most Linux distributions use only one of these three startup files:
  - ✓ `$HOME/.bash_profile`
  - ✓ `$HOME/.bash_login`
  - ✓ `$HOME/.profile`

Interactive Shell

- **If you start a Bash shell without logging into a system (such as if you just type bash at a CLI prompt or open a GUI terminal), you start what's called an interactive shell.**
  - ✓ The interactive shell doesn't act like the login shell, but it still provides a CLI prompt for you to enter commands.
- **If Bash is started as an interactive shell, it doesn't process the `/etc/profile` file.**
- **Instead, it checks for the .bashrc file in the user's HOME directory (often referred to as `~/.bashrc`).**
- **The `.bashrc` file does two things.**
  - ✓ First, it checks for a common `/etc/bash.bashrc` file.
    - The common `bash.babshrc` file provides a way for you to set scripts and variables used by all users who start an interactive shell.
  - ✓ Second, it provides a place for the user to enter personal aliases and private script functions.

Noninteractive Shell

- This is the shell that the system starts to execute a shell script.
  - ✓ This is different in that there isn't a CLI prompt to worry about.
- However, there may still be specific startup commands you want to run each time you start a script on your system.
- To accommodate that situation, the Bash shell provides the BASH_ENV environment variable.
- When the shell starts a noninteractive shell process, it checks this environment variable for the name of a startup file to execute.
  - ✓ If one is present, the shell executes the commands in the file.
- On most Linux distributions, this environment value is not set by default.

Using Command Aliases

- A command alias allows you to create an alias name for common commands (along with their parameters) to help keep your typing to a minimum.
- Most likely your Linux distribution has already set some common command aliases for you.
- To see a list of the active aliases, use the alias command with the -p parameter:

```
$ alias -p
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
alias vi='vim'
alias which='alias | /usr/bin/which --tty-only --readalias--show-dot --show-tilde'
```

- You can create your own aliases by using the alias command:

```
$ alias li='ls -il'
$ li
total 52
4508 drwxr-xr-x 2 rich rich 4096 Jun 12 11:21 Desktop
4512 drwxr-xr-x 2 rich rich 4096 Jun 12 11:21 Documents
...
```

Using Command Aliases

- After you define an alias value, you can use it at any time in your shell, including in shell scripts.
  - ✓ Command aliases act like local environment variables.
- They're valid only for the shell process in which they're defined:

```
$ alias li='ls -il'
$ bash
$ li
bash: li: command not found
```

- Since the Bash shell always reads the `$HOME/.bashrc` startup file when starting a new interactive shell, just put command alias statements there, and they will be valid for all shells you start.

THE BASICS OF SHELL SCRIPTING

Shell scripting allows you to write small programs that automate activities on your Linux system. Shell scripts can save you time by giving you the flexibility to quickly process data and generate reports that would be cumbersome to do by manually entering multiple commands at the command prompt. You can automate just about anything you do at the command prompt using shell scripts.

This section walks through the basics of what shell scripts are and how to get started writing them.

Running Multiple Commands

- One exciting feature of the Linux command line is that you can enter multiple commands on the same command line and Linux will process them all.
  - ✓ Just place a semicolon (;) between each command you enter.

```
$ date ; who
Thu Feb 20 19:20:06 EST 2019
rich :0 2019-02-20 19:15 (:0)
```

- Though this may seem trivial, it is the basis of how shell scripts work.

Redirecting Output

- **Often when you run a command, you'd like to save the output for future reference.**
  - ✓ To help with this, the Bash shell provides output redirection.
- **Output redirection allows us to redirect the output of a command from the monitor to another device, such as a file.**
  - ✓ This feature comes in handy when you need to log data from a shell script that runs after business hours, so you can see what the shell script did when it ran.
- **To redirect the output from a command, you use the greater-than symbol (>) after the command and then specify the name of the file that you want to use to capture the redirected output.**

```
$ date > today.txt
$ cat today.txt
Thu Feb 20 19:21:12 EST 2019
```

```
$ who >> today.txt
$ cat today.txt
Thu Feb 20 19:21:12 EST 2019
anisa :0 2019-02-20 19:15 (:0)
```

Piping Data

- Whereas output redirection allows us to redirect command output to a file, piping allows us to redirect the output to another command.
  - ✓ The second command uses the redirected output from the first command as input data.
  - ✓ This feature comes in handy when using commands that process data, such as the sort command.
- The piping symbol is the bar (|) symbol.

```
$ ls | sort
Desktop
Documents
Downloads
Music
Pictures
Public
...
```

The Shell Script Format

- **Placing multiple commands on a single line, either by using the semicolon or piping, is a great way to process data, but it can still get rather tedious.**
  - ✓ Each time you want to run the set of commands, you need to type them all at the command prompt.
- **However, Linux allows us to place multiple commands in a text file and then run the text file as a program from the command line.**
  - ✓ This is called a shell script because we're scripting out commands for the Linux shell to run.
  - ✓ Shell script files are plain text files.
- **First, for your shell script to work you'll need to follow a specific format for the shell script file.**
- **The first line in the file must specify the Linux shell required to run the script.**

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

  - ✓ The Linux world calls the combination of the pound sign and the exclamation symbol (#!) the shebang.
- **It signals to the operating system which shell to use to run the shell script.**
  - ✓ Most Linux distributions support multiple Linux shells, but the most common is the Bash shell.
- **You can run shell scripts written for other shells, as long as that shell is installed on the Linux distribution.**
- **After you specify the shell, you're ready to start listing the commands in your script.**

The Shell Script Format

- You don’t need to enter all of the commands on a single line; Linux allows you to place them on separate lines.
  - ✓ Also, the Linux shell assumes each line is a new command in the shell script, so you don’t need to use semicolons to separate the commands.
- A simple shell script file:

```
$ cat test1.sh
#!/bin/bash
# This script displays the date and who's logged in
date
who
```

- Lines that start with a pound sign are called comment lines.
  - ✓ They allow you to embed comments into the shell script program to help you remember what the code is doing.
  - ✓ The shell skips comment lines when processing the shell script.
  - ✓ You can place comment lines anywhere in your shell script fi le after the opening shebang line.

Running the Shell Script

- If you just enter a shell script file at the command prompt to run it, you may be a bit disappointed:

```
$ test1.sh
test1.sh: command not found
```

- Unfortunately, the shell doesn't know where to find the test1.sh command in the virtual directory.
  - ✓ The reason for this is the shell uses a special environment variable called `PATH` to list directories where it looks for commands.
  - ✓ If your local `HOME` folder is not included in the `PATH` environment variable list of directories, you can't run the shell script file directly.
- Instead, you must use either a relative or an absolute path name to point to the shell script file.
- The easiest way to do that is by adding the ./ relative path shortcut to the file:

```
$ ./test1.sh
bash: ./test1.sh: Permission denied
```

- Now the shell can find the program file, but there's still an error message.

Running the Shell Script

- This time the error is telling us that we don't have permissions to run the shell script file.
  - ✓ A quick look at the shell script file using the `ls` command with the `-l` option shows the permissions set for the file:

```
$ ls -l test1.sh
-rw-r--r-- 1 rich rich 73 Feb 20 19:37 test1.sh
```

- By default, the Linux system didn't give anyone execute permissions to run the file.
- You can use the `chmod` command to add that permission for the file owner:

```
$ chmod u+x test1.sh
$ ls -l test1.sh
-rwxr--r-- 1 rich rich 73 Feb 20 19:37 test1.sh
```

- You should now be able to run the shell script file and see the output:

```
$ ./test1.sh
Thu Feb 20 19:48:27 EST 2019
rich :0 2019-02-20 19:15 (:0)
```

Runs The Script In The Current Shell

- `source` or `.` execute a shell script within the context of the current shell.

```
$ cat test_source.sh
cd /usr/games
pwd
echo hi
$ cd /home/anisa
$ pwd
/home/user/sample/script
$ source test_source.sh or $ . test_source.sh
/usr/games
hi
$ pwd
/usr/games
```

`. script ≠ ./script`

Runs The Script In The Current Shell

- Running the script directly from the command prompt spawns a new subshell for the script, making any local environment variables in the shell not available to the script.
- If you want to make any local environment variables available for use in the shell script, launch the script using the `exec` command:

```
$ exec ./test1.sh
Thu Feb 20 19:51:27 EST 2019
rich :0 2019-02-20 19:15 (:0)
```

- The `exec` command runs the script in the current shell.
- The `exec` command will execute a command in place of the current shell, that is, it terminates the current shell and starts a new process in its place.

ADVANCED SHELL SCRIPTING

The previous section walked through the basics of how to group normal command-line commands together in a shell script file to run in the Linux shell. This section adds to that by showing more features available in shell scripts to make them look and act more like real programs.

Displaying Messages

- When you string commands together in a shell script file, the output may be somewhat confusing to look at.
- It would help to be able to customize the output by separating the output and adding our own text between the output from the listed commands.
- The echo command allows you to display text messages from the command line.
- When used at the command line, it's not too exciting:

```
$ echo This is a test
This is a test
```

- But with echo, you can now insert messages anywhere in the output from the shell script file.

Displaying messages from shell scripts

```
$ cat test1.sh
#!/bin/bash
# This script displays the date and who's logged in
echo The current date and time is:
date
echo
echo "Let's see who's logged into the system:"
who

$ ./test1.sh
The current date and time is:
Thu Feb 20 19:55:44 EST 2019

Let's see who's logged into the system:
anisa :0 2019-02-20 19:15 (:0)
```

Using Variables in Scripts

- Part of programming is the ability to temporarily store data to use later on in the program.
- As mentioned earlier, you do that in the Bash shell by using either global or local variables.
- This method also works when working with shell scripts.

Using Global Environment Variables

- **Global environment variables** are often used to track specific system information, such as the name of the system, the name of the user logged into the shell, the user's user ID (UID), the default home directory for the user, and the search path the shell uses to find executable programs.
- You can tap into these environment variables from within your scripts by using the environment variable name, preceded by a dollar sign.

Shell Script File To Display Environment Variables

```
$ cat test2.sh
#!/bin/bash
# display user information from the system.
echo User info for userid: $USER
echo UID: $UID
echo HOME: $HOME

$ chmod u+x test2.sh
$ ./test2.sh
User info for userid: anisa
UID: 1000
HOME: /home/anisa
```

Defining Local Variables

- Most variables used in shell scripts are local variables used for storing your own data within your shell scripts.
- You assign values to local variables using the equal sign.
  - ✓ Spaces **must not** appear between the **variable name**, the **equal sign**, and **the value**.

```
var1=10
var2=23.45
var3=testing
var4="Still more testing"
```

- The shell script automatically determines the data type used for the variable value.
- local variables defined within the shell script are accessible only from within the shell script by default.
  - ✓ If you want the variables within a shell script to be visible from the parent shell that launched the shell script, use the `export` command.

Using Local Variables In A Shell Script

```
$ cat test3.sh
#!/bin/bash
# testing variables
days=10
guest=Katie
echo ${guest} checked in $days days ago

$ chmod u+x test3.sh
$ ./test3.sh
Katie checked in 10 days ago
```

Command-Line Arguments

- One of the most versatile features of shell scripts is the ability to pass data into the script when you run it.
  - ✓ This allows you to customize the script with new data each time you run it.
- One method of passing data into a shell script is to use command-line arguments.
- Command-line arguments are data you include on the command line when you run the command.
- Just start listing them after the command, separating each data value with a space, in this format:

```
command argument1 argument2 ...
```

- You retrieve the values in your shell script code using special numeric positional variables.
  - ✓ Use the variable $1 to retrieve the first command-line argument, $2 the second argument, and so on.

Using Command-line Arguments In A Shell Script

```
$ cat test4.sh
#!/bin/bash
# Testing command line arguments
echo $1 checked in $2 days ago

$ chmod u+x test4.sh
$ ./test4.sh Anisa 4
Barbara checked in 4 days ago
$ ./test4.sh Mohsen 5
Jessica checked in 5 days ago
$ ./test4.sh Mohammad
rich checked in days ago
```

Getting User Input

- Providing command-line options and parameters is a great way to get data from your script users, but sometimes your script needs to be more interactive.
- There are times when you have to ask a question while the script is running and wait for a response from the person running your script.
- The Bash shell provides the read command just for this purpose.

Basic Reading

- The `read` command accepts input from the standard input (the keyboard) or from another file descriptor.
- After receiving the input, the read command places the data into a standard variable
- Accepting user input in your scripts:

```
$ cat test5.sh
#!/bin/bash
# testing the read command
echo -n "Enter your name: "
read name
echo "Hello $name, welcome to my program."

$ ./test5.sh
Enter your name: Moshen Mohammad Amini
Hello Moshen Mohammad Amini, welcome to my program.
```

Basic Reading

- the `read` command includes the `-p` option, which allows you to specify a prompt directly in the read command line.

```
$ cat test6.sh
#!/bin/bash
# testing the read -p option
read -p "Please enter your age:" age
days=$[ $age * 365 ]
echo "That makes you over $days days old!"

$ ./test6.sh
Please enter your age:23
That makes you over 8395 days old!
```

Basic Reading

- you can specify multiple variables using `read` command.
- Each data value entered is assigned to the next variable in the list.
  - ✓ If the list of variables runs out before the data does, the remaining data is assigned to the last variable.

```
$ cat test7.sh
#!/bin/bash
# entering multiple variables
read -p "Enter your name: " first last
echo "Checking data for $last, $first…"
$ ./test7.sh
Enter your name: Rich Blum
Checking data for Blum, Rich…
```

Basic Reading

- You can also specify no variables on the read command line.
  - ✓ If you do that, the read command places any data it receives in the special environment variable REPLY.

```
$ cat test8.sh
#!/bin/bash
# testing the REPLY environment variable
read -p "Enter a number: "
factorial=1
for (( count=1; count <= $REPLY; count++ ))
do
   factorial=$[ $factorial * $count ]
done
echo "The factorial of $REPLY is $factorial"
$ ./test8.sh
Enter a number: 5
The factorial of 5 is 120
```

Timing Out and Counting Input Characters

- **There's a danger when using the read command.**
  - ✓ It's quite possible that your script will get stuck waiting for the script user to enter data.
- **If the script must go on regardless of whether any data was entered, you can use the -t option specify a timer.**
  - ✓ The **-t** option specifies the number of seconds for the read command to wait for input.
  - ✓ When the timer expires, the read command returns a non-zero exit status.
- **Instead of timing the input, you can also set the read command to count the input characters with -n option.**
  - ✓ When a preset number of characters has been entered, it automatically exits, assigning the entered data to the variable.
  - ✓ As soon as you press the single character to answer, the read command accepts the input and passes it to the variable.
  - ✓ There's no need to press the Enter key.

Timing Out and Counting Input Characters

```
$ cat test9.sh
#!/bin/bash
# timing the data entry
if read -t 5 -p "Please enter your name: " name
then
echo "Hello $name, welcome to my script"
else
echo
echo "Sorry, too slow!"
fi
$ ./test9.sh
Please enter your name: Mohsen
Hello Mohsen, welcome to my script
$ ./test9.sh
Please enter your name:
Sorry, too slow!
```

```
$ cat test10.sh
#!/bin/bash
# getting just one character of input
read -n1 -p "Do you want to continue [Y/N]? " answer
case $answer in
Y | y)  echo
        echo "fine, continue on…";;
N | n)  echo
        echo OK, goodbye
exit;;
esac
echo "This is the end of the script"
$ ./test10.sh
Do you want to continue [Y/N]? Y
fine, continue on…
This is the end of the script
$ ./test10.sh
Do you want to continue [Y/N]? n
OK, goodbye
```

Silent Reading

- There are times when you need input from the script user but you don't want that input to display on the monitor.
  - ✓ The classic example of this is when entering passwords, but there are plenty of other types of data that you will need to hide.
- The **-s** option prevents the data entered in the **read** command from being displayed on the monitor
  - ✓ actually, the data is displayed, but the read command sets the text color to the same as the background color.

```
$ cat test11.sh
#!/bin/bash
# hiding input data from the monitor
read -s -p "Enter your password: " pass
echo
echo "Is your password really $pass?"
$ ./test11.sh
Enter your password:
Is your password really T3st1ng?
```

The Exit Status

- When a shell script ends, it returns an exit status back to the parent shell that launched it.
- The exit status tells us whether the shell script completed successfully.
- Linux provides us with the special `$?` variable, which holds the exit status value from the last command that executed.
- To check the exit status of a command, you must view the `$?` variable immediately after the command ends.
  - ✓ It changes values according to the exit status of the last command executed by the shell.

```
$ who
rich :0 2019-02-20 23:16 (:0)
$ echo $?
0
```

- By convention, the exit status of a command that successfully completes is 0.
  - ✓ If a command completes with an error, then a positive integer value appears as the exit status.

The Exit Status

- You can change the exit status of your shell scripts by using the exit command.
- Just specify the exit status value you want in the exit command:

```
$ /bin/bash
$ exit 120
exit
$ echo $?
120
```

- As you write more complicated scripts, you can indicate errors by changing the exit status value.
  - ✓That way, by checking the exit status you can easily debug your shell scripts.

WRITING SCRIPT PROGRAMS

So far we've explored how to combine regular command-line commands within a shell script to automate common tasks that you may perform as the system administrator. But shell scripts allow us to do much more than just that. The Bash shell provides more programming-like commands that allow us to write full-fledged programs within our shell scripts, such as capturing command output, performing mathematical operations, checking variable and file conditions, and looping through commands.

This section walks through some of the advanced programming features available to us from the Bash shell.

Command Substitution

- Quite possibly one of the most useful features of shell scripts is the ability to store and process data.
  - ✓ use output redirection to store output from a command to a file
  - ✓ and piping to redirect the output of a command to another command.
- **Command substitution** allows you to assign the output of a command to a user variable in the shell script.
- After the output is stored in a variable, you can use standard Linux string manipulation commands (such as sort or grep) to manipulate the data before displaying it.
- To redirect the output of a command to a variable, you need to use one of two command substitution formats:
  - ✓ Placing backticks (`) around the command
  - ✓ Using the command within the $() function

Demonstrating command substitution

```
$ var1=`date`
$ echo $var1
Fri Feb 21 18:05:38 EST 2019

$ var2=$(whoami)
$ echo $var2
anisa
```

Performing Math

- The world revolves around numbers, and at some point you'll probably need to do some mathematical operations with your data.
- Unfortunately, this is one place where the Bash shell shows its age.
- However, there are a couple of ways to use simple mathematical functions in Bash shell scripts.
- To include mathematical expressions in your shell scripts, you use a special format.
- This format places the equation within the brackets:

```
result=$[ 25 * 5 ]
```

  - ✓ The `$[]` format allows you to use integers only;
    - it doesn't support floating-point values.

Performing Math

- If you need to do floating-point calculations, one solution is to use the `bc` command-line calculator program.
- The `bc` calculator is a tool in Linux that can perform floating-point arithmetic:

```
$ bc
```

- Unfortunately, the bc calculator has some limitations of its own.
  - ✓ The floating-point arithmetic is controlled by a built-in variable called `scale`.

```
$ bc -q
3.44 / 5
0
scale=4
3.44 / 5
.6880
```

Performing Math

- To embed a **bc** calculation into your script, things get a bit complicated.
- You must use command substitution to capture the output of the calculation into a variable, but there's a twist.
- The basic format you need to use is

```
variable=$(echo "options; expression" | bc)
```

  - ✓ The first parameter, options, allows us to set the **bc** variables, such as the scale variable.
  - ✓ The expression parameter defines the mathematical expression to evaluate using **bc**.

- Though this looks pretty odd, it works:

```
$ var1=$(echo "scale=4; 3.44 / 5" | bc)
$ echo $var1
.6880
```

Logic Statements

- So far all of the shell scripts presented process commands in a linear fashion, one command after another.
  - ✓ However, not all programming is linear.
- There are times when you'd like your program to test for certain conditions—such as whether a file exists or if a mathematical expression is 0—and perform different commands based on the results of the test.
  - ✓ For that, the Bash shell provides **logic statements**.
- Logic statements allow us to test for a specific condition and then branch to different sections of code based on whether the condition evaluates to a True or False logical value.

The if Statement

- The most basic logic statement is the `if` condition statement.

```
if [ condition ]
then
   commands
fi
```

- The square brackets used in the `if` statement are a shorthand way of using the test command.
  - ✓ The test command evaluates a condition and returns a **True** logical value if the test passes or a **False** logical value if the test fails.
  - ✓ If the condition passes, the shell **runs the commands** in the `then` section of code, or if the condition evaluates to a False logical value, the shell script **skips the commands** in the then section of code.
- The condition test expression has quite a few different formats in the Bash shell programming.
- There are built-in tests for numerical values, string values, and even files and directories.
- You can combine tests by using the Boolean AND **(&&)** and OR **(||)** symbols.

Using Conditional Expressions

```
if [ condition ]
then
        commands
fi
```

```
if [ -s /tmp/tempstuff ]
then
     echo “/tmp/tempstuff found; aborting!”
     exit
fi
```

```
if test -s /tmp/tempstuff
```

Using Conditional Expressions

```
if [ command ]
     then
          additional-commands
fi
```

```
if [ conditional-expression ]
     then
          commands
     else
          other-commands
fi
```

Condition tests

Test	Type	Description
n1 -eq n2	Numeric	Checks if n1 is equal to n2
n1 -ge n2	Numeric	Checks if n1 is greater than or equal to n2
n1 -gt n2	Numeric	Checks if n1 is greater than n2
n1 -le n2	Numeric	Checks if n1 is less than or equal to n2
n1 -lt n2	Numeric	Checks if n1 is less than n2
n1 -lt n2	Numeric	Checks if n1 is not equal to n2
str1 = str2	String	Checks if str1 is the same as str2
str1 != str2	String	Checks if str1 is not the same as str2
str1 < str2	String	Checks if str1 is less than str2
-n str1	String	Checks if str1 has a length greater than zero
-z str1	String	Checks if str1 has a length of zero

Test	Type	Description
-d file	File	Check if file exists and is a directory
-e file	File	Checks if file exists
-f file	File	Checks if file exists and is a file
-r file	File	Checks if file exists and is readable
-s file	File	Checks if file exists and is not empty
-w file	File	Checks if file exists and is writable
-x file	File	Checks if file exists and is executable
-O file	File	Checks if file exists and is owned by the current user
-G file	File	Checks if file exists and the default group is the same as the current user
file1 -nt file2	File	Checks if file1 is newer than file2
file1 -ot file2	File	Checks if file1 is older than file2

Using if condition statements

```
#!/bin/bash
# testing the if condition
if [ $1 -eq $2 ]
then
      echo "Both values are equal!"
      exit
fi
if [ $1 -gt $2 ]
then
      echo "The first value is greater than the second"
      exit
fi
if [ $1 -lt $2 ]
then
      echo "The first value is less than the second"
      exit
fi
```

```
$ chmod u+x test12.sh
$ ./test12.sh 10 5
The first value is greater than the second
```

The case Statement

- Often you'll find yourself trying to evaluate the value of a variable, looking for a specific value within a set of possible values.
- Instead of having to write multiple if statements testing for all of the possible conditions, you can use a case statement.
- The case statement allows you to check multiple values of a single variable in a list-oriented format:

```
case variable in
pattern1) commands1;;
pattern2 | pattern3) commands2;;
*) default commands;;
esac
```

- The case statement compares the variable specified against the different patterns.
  - ✓ If the variable matches the pattern, the shell executes the commands specified for the pattern.
  - ✓ The asterisk symbol is the catchall for values that don't match any of the listed patterns.

Using the case statement

```
$ cat test13.sh
#!/bin/bash
# using the case statement
case $USER in
anisa | root)
      echo "Welcome, $USER"
      echo "Please enjoy your visit";;
testing)
      echo "Special testing account";;
mohsen)
      echo "Don't forget to log off when you're done";;
*)
      echo "Sorry, you're not allowed here";;
esac
$ chmod u+x test6.sh
$ ./test13.sh
Welcome, anisa
Please enjoy your visit
```

Loops

- When you're writing scripts, you'll often find yourself in a situation where it would come in handy to repeat the same commands multiple times, such as applying a command against all of the files in a directory.
- The Bash shell provides some basic looping commands to accommodate that.

The for Loop

- The `for` statement iterates through every element in a series, such as files in a directory or lines in a text document.

```
for variable in series ; do
    commands
done
```

- The variable becomes a placeholder, taking on the value of each element in the series in each iteration.
- The commands can use the variable just like any other variable that you define in the script.
- A common use of the for statement is to iterate through a series of numbers.
- Instead of having to list all of the numbers individually, you can use the `seq` command.
  - ✓ The `seq` command outputs a series of numbers.
  - ✓ Just specify the start, end, and interval values needed for the series.

Using the for loop

```
$ cat test14.sh
#!/bin/bash
# iterate through the files in the Home folder
for file in $(ls | sort) ; do
        if [ -d $file ]
        then
                echo "$file is a directory"
        fi
        if [ -f $file ]
        then
                echo "$file is a file"
        fi
done
$ ./test14.sh
Desktop is a directory
Documents is a directory
...
test1.sh is a file
test2.sh is a file
...
```

```
for x in `seq 1 10` `; do
        commands
done
```

The while Loop

- Another useful loop statement is the `while` command.

```
while [ condition ] ; do
    commands
done
```

- The `while` loop keeps looping as long as the condition specified evaluates to a True logical value.
  - ✓ When the condition evaluates to a False logical value, the looping stops.
- The condition used in the `while` loop is the same as that for the if-then statement, so you can test numbers, strings, and files.
- The opposite of the `while` command is the `until` command.
  - ✓ It iterates through a block of commands `until` the test condition evaluates to a True logical value.

Calculating the factorial of a number

```
$ cat test15.sh
#!/bin/bash
number=$1
factorial=1
while [ $number -gt 0 ] ; do
      factorial=$[ $factorial * $number ]
      number=$[ $number - 1 ]
done
echo The factorial of $1 is $factorial
$ ./test15.sh 5
The factorial of 5 is 120
$ ./test15.sh 6
The factorial of 6 is 720
```

Functions

- **As you start writing more complex shell scripts, you'll find yourself reusing parts of code that perform specific tasks.**
  - ✓ Sometimes it's something simple, such as displaying a text message and retrieving an answer from the script users.
  - ✓ Other times it's a complicated calculation that's used multiple times in your script as part of a larger process.
- **Functions are blocks of script code that you assign a name to and then reuse anywhere in your code.**
  - ✓ Any time you need to use that block of code in your script, all you need to do is use the function name you assigned.
- **There are two formats you can use to create functions in Bash shell scripts.**

```
function name {
    commands
}
```

  - ✓ Each function you define in your script must be assigned a unique name.
- **When you call the function, the Bash shell executes each of the commands in the order they appear in the function, just as in a normal script.**

```
name() {
    commands
}
```

  - ✓ The empty parentheses after the function name indicate that you're defining a function.

Functions

```
$ cat test16.sh
#!/bin/bash
# using a function in a script
function func1 {
        echo "This is an example of a function"
}
count=1
while [ $count -le 5 ]
do
        func1
        count=$[ $count + 1 ]
done
echo "This is the end of the loop"
func1
echo "Now this is the end of the script"
$ ./test16.sh
This is an example of a function
This is an example of a function
...
Now this is the end of the script
```

Function Return Value

- The Bash shell uses the return command to exit a function with a specific exit status.
- The return command allows you to specify a single integer value to define the function exit status, providing an easy way for you to programmatically set the exit status of your function.

```
$ cat test17.sh
#!/bin/bash
# using the return command in a function
function dbl {
   read -p "Enter a value: " value
   echo "doubling the value"
   return $[ $value * 2 ]
}
dbl
echo "The new value is $?"
```

- Just as you can capture the output of a command to a shell variable, you can also capture the output of a function to a shell variable.

```
result=$(dbl)
```

RUNNING SCRIPTS IN BACKGROUND MODE

There are times when running a shell script directly from the command-line interface is inconvenient. Some scripts can take a long time to process, and you may not want to tie up the command-line interface waiting. While the script is running, you can't do anything else in your terminal session. Fortunately, there's a simple solution to that problem. The following sections describe how to run your scripts in background mode on your Linux system.

Running in the Background

- To run a shell script in background mode from the command-line interface, just place an ampersand symbol after the command:

```
$ ./test18.sh &
[1] 19555
$ This is test program
Loop #1
Loop #2
$ ls -l
total 8
-rwxr--r-- 1 rich rich 219 Feb 26 19:27 test18.sh
* $ Loop #3
```

- The number in the square brackets is the job number the shell assigns to the background process.
- When the background process finishes, it displays a message on the terminal:

```
[1]+ Done ./test18.sh
```

Running Multiple Background Jobs

- You can start any number of background jobs at the same time from the command-line prompt:

```
$ ./test18.sh &
[1] 19582
$ This is test program
Loop #1
$ ./test18.sh &
[2] 19597
$ This is test program
Loop #1
```

```
$ ./test18.sh &
[3] 19612
$ This is test program
Loop #1
Loop #2
Loop #2
Loop #2
```

- You can see that all of the scripts are running by using the ps command:

```
$ ps au
```

RUNNING SCRIPTS WITHOUT A CONSOLE

There will be times when you want to start a shell script from a terminal session and then let the script run in background mode until it finishes, even if you exit the terminal session.

You can do this by using the nohup command.

Running Scripts Without a Console

- The nohup command runs another command blocking any SIGHUP signals that are sent to the process.
  - ✓ This prevents the process from exiting when you exit your terminal session.
- You can combine the nohup command with the ampersand to run a script in background and not allow it to be interrupted:

```
$ nohup ./test18.sh &
[1] 19831
$ nohup: appending output to 'nohup.out'
```

- when you use the nohup command, the script ignores any SIGHUP signals sent by the terminal session if you close the session.
- To accommodate any output generated by the command, the nohup command automatically redirects output messages to a file, called nohup.out.

```
$ cat nohup.out
This is a test program
[...]
```

SENDING SIGNALS

The Bash shell can send control signals to processes running on the system. This allows you to stop or interrupt a runaway application process if necessary. There are two basic Linux signals you can generate using key combinations on the keyboard to interrupt or stop a foreground process.

Interrupting a Process

- The Ctrl+C key combination generates a SIGINT signal and sends it to any processes currently running in the shell.
- The SIGINT signal interrupts the running process, which for most processes causes them to stop.
- You can test this by running a command that normally takes a long time to finish and pressing the Ctrl+C key combination:

```
$ sleep 100
$
```

Pausing a Process

- Instead of terminating a process, you can pause it in the middle of whatever it's doing.
- The Ctrl+Z key combination generates a SIGTSTP signal, stopping any processes running in the shell.

```
$ sleep 100
[1]+ Stopped sleep 100
```

  - ✓ The number in the square brackets indicates the job number for the process in the shell.
- You can view the stopped job by using the ps command:

```
$ ps au
```

  - ✓ The ps command shows the status of the stopped job as T, which indicates the command is either being traced or is stopped.

JOB CONTROL

In the previous section, you saw how to use the Ctrl+Z key combination to stop a job running in the shell. After you stop a job, the Linux system lets you either kill or restart it. Restarting a stopped process requires sending it a SIGCONT signal.

The function of starting, stopping, killing, and resuming jobs is called job control. With job control, you have full control over how processes run in your shell environment.

This section describes the commands used to view and control jobs running in your shell.

Viewing Jobs

- The key command for job control is the jobs command.
- The jobs command allows you to view the current jobs being handled by the shell.
- The jobs command parameters:
  - ✓ `-l`
    - List the PID of the process along with the job number.
  - ✓ `-n`
    - List only jobs that have changed their status since the last notification from the shell.
  - ✓ `-p`
    - List only the PIDs of the jobs.
  - ✓ `-r`
    - List only the running jobs.
  - ✓ `-s`
    - List only stopped jobs.

Viewing Jobs

```
$ cat test19.sh
#!/bin/bash
# testing job control
echo "This is a test program $$"
count=1
while [ $count -le 10 ] ; do
    echo "Loop #$count"
    sleep 10
    count=$[ $count + 1 ]
done
echo "This is the end of the test program"
$ ./test19.sh
This is a test program 29011
Loop #1
[1]+ Stopped ./test19.sh
$ ./test19.sh > test19.sh.out &
[2] 28861
$ jobs
[1]+ Stopped ./test19.sh
[2]- Running ./test19.sh >test19.shout &
```

```
$ ./test19.sh
This is a test program 29075
Loop #1
[1]+ Stopped ./test19.sh
$ ./test19.sh
This is a test program 29090
Loop #1
[2]+ Stopped ./test19.sh
$ ./test19.sh
This is a test program 29105
Loop #1
[3]+ Stopped ./test19.sh
$ jobs -l
[1] 29075 Stopped ./test19.sh
[2]- 29090 Stopped ./test19.sh
[3]+ 29105 Stopped ./test19.sh
$ kill -9 29105
$ jobs -l
[1]- 29075 Stopped ./test19.sh
[2]+ 29090 Stopped ./test19.sh
```

Restarting Stopped Jobs

- To restart a job in background mode, use the `bg` command, along with the job number:

```
$ bg 2
[2]+ ./test20.sh &
Loop #2
$ Loop #3
Loop #4
$ jobs
[1]+ Stopped ./test20.sh
[2]- Running ./test20.sh &
Loop #10
This is the end of the test program
[2]- Done ./test20.sh
```

- To restart a job in foreground mode, use the `fg` command, along with the job number:

```
$ jobs
[1]+ Stopped ./test20.sh
$ fg 1
./test20
Loop #2
Loop #3
```

RUNNING LIKE CLOCKWORK

I'm sure that, as you start working with scripts, there'll be a situation in which you'll want to run a script at a preset time, usually at a time when you're not there. There are two common ways of running a script at a preselected time:

- ✓ The at command
- ✓ The cron table

Each method uses a different technique for scheduling when and how often to run scripts. The following sections describe each of these methods.

Scheduling a Job Using the at Command

- **The `at` command allows you to specify a time when the Linux system will run a script.**
  - ✓ It submits a job to a queue with directions on when the shell should run the job.
- **Another command, `atd`, runs in the background and checks the job queue for jobs to run.**
  - ✓ Most Linux distributions start this automatically at boot time.
- **The `atd` command checks a special directory on the system (usually `/var/spool/at`) for jobs submitted using the at command.**
  - ✓ By default, the atd command checks this directory every 60 seconds.
  - ✓ When a job is present, the atd command checks the time the job is set to be run.
  - ✓ If the time matches the current time, the atd command runs the job.

The at Command Format

- **at** - queue jobs for later execution

```
at [-f filename] time
```

- ✓ By default, the at command submits input from STDIN to the queue. You can specify a filename used to read commands (your script file) using the -f parameter.
- ✓ The time parameter specifies when you want the Linux system to run the job:
  - A standard hour and minute, such as 10:15
  - An AM/PM indicator, such as 10:15PM
  - A specific named time, such as now, noon, midnight, or teatime (4PM)
- ✓ Besides specifying the time to run the job, you can also include a specific date, using a few different date formats:
  - A standard date format, such as `MMDDYY`, `MM/DD/YY`, `or DD.MM.YY`
  - A text date, such as Jul 4 or Dec 25, with or without the year
  - You can also specify a time increment: Now + 25 minutes, 10:15PM tomorrow, 10:15 + 7 days
- ✓ When you use the at command, the job is submitted into a job queue.
  - Job queues are referenced using lowercase letters, a through z.
- ✓ If you want to run a job at a lower priority, you can specify the letter using the –q parameter.

Retrieving Job Output

- When the job runs on the Linux system, there's no monitor associated with the job.
- Instead, the Linux system uses the email address of the user who submitted the job.
  - ✓ Any output destined to STDOUT or STDERR is mailed to the user via the mail system.

```
$ date
Thu Feb 28 18:48:20 EST 2019
$ at -f test3.sh 18:49
job 2 at Thu Feb 28 18:49:00 2019
$ mail
Heirloom Mail version 12.5 7/5/10. Type ? for help.
"/var/spool/mail/rich": 1 message 1 new
[...]
```

- When the job completes, nothing appears on the monitor, but the system generates an email message.
  - ✓ The email message shows the output generated by the script.
  - ✓ If the script doesn't produce any output, it won't generate an email message, by default.
- You can change that by using the -m option in the at command.
  - ✓ This generates an email message, indicating the job completed, even if the script doesn't generate any output.

Listing Pending Jobs

- The atq command allows you to view what jobs are pending on the system:

```
$ at -f test21.sh 19:15
warning: commands will be executed using /bin/sh
job 7 at 2007-11-04 10:15
$ at -f test21.sh 4PM
warning: commands will be executed using /bin/sh
job 8 at 2007-11-03 16:00
$ at -f test21.sh 1PM tomorrow
warning: commands will be executed using /bin/sh
job 9 at 2007-11-04 13:00
$ atq
7 2007-11-04 10:15 a
8 2007-11-03 16:00 a
9 2007-11-04 13:00 a
```

- The job listing shows the job number, the date and time the system will run the job, and the job queue the job is stored in.

Removing Jobs

- When you know the information about what jobs are pending in the job queues, you can use the `atrm` command to remove a pending job.

```
$ atrm 8
$ atq
7 2007-11-04 10:15 a
9 2007-11-04 13:00 a
```

- Just specify the job number you want to remove.
- You can only remove jobs that you submit for execution.
  - ✓ You can't remove jobs submitted by others.

Building the cron Table

- All system users can have their own cron table (including the root user) for running scheduled jobs.
- Linux provides the crontab command for handling the cron table.
  - ✓ To list an existing cron table, use the -l parameter:

```
$ crontab -l
no crontab for anisa
```

- By default, each user's cron table file doesn't exist.
- To add entries to your cron table, use the -e parameter.
  - ✓ When you do that, the crontab command automatically starts the vi editor with the existing cron table, or an empty file if it doesn't yet exist.

```
$ crontab -e
```

Scheduling Regular Scripts

The cron Table

- The cron table uses a special format for allowing you to specify when a job should be run.
- The format for the cron table is
  - `min hour dayofmonth month dayofweek command`
- The cron table allows you to specify entries as specific values, ranges of values (such as 1–5), or as a wildcard character (the asterisk).
- To run a command at 10:15 a.m. every day: `15 10 * * * command`
- To specify a command to run at 4:15 p.m. every Monday: `15 16 * * 1 command`
  - You can specify the dayofweek entry either as a three-character text value (mon, tue, wed, thu, fri, sat, sun)
  - or as a numeric value, with 0 being Sunday and 6 being Saturday.
- to execute a command at 12 noon on the first day of every month: 00 12 1 * * command
- You can add any command-line parameters or redirection symbols you like, as a regular command line:

```
15 10 * * * /home/rich/test21.sh > test21out
```

- The cron program runs the script using the user account that submitted the job.