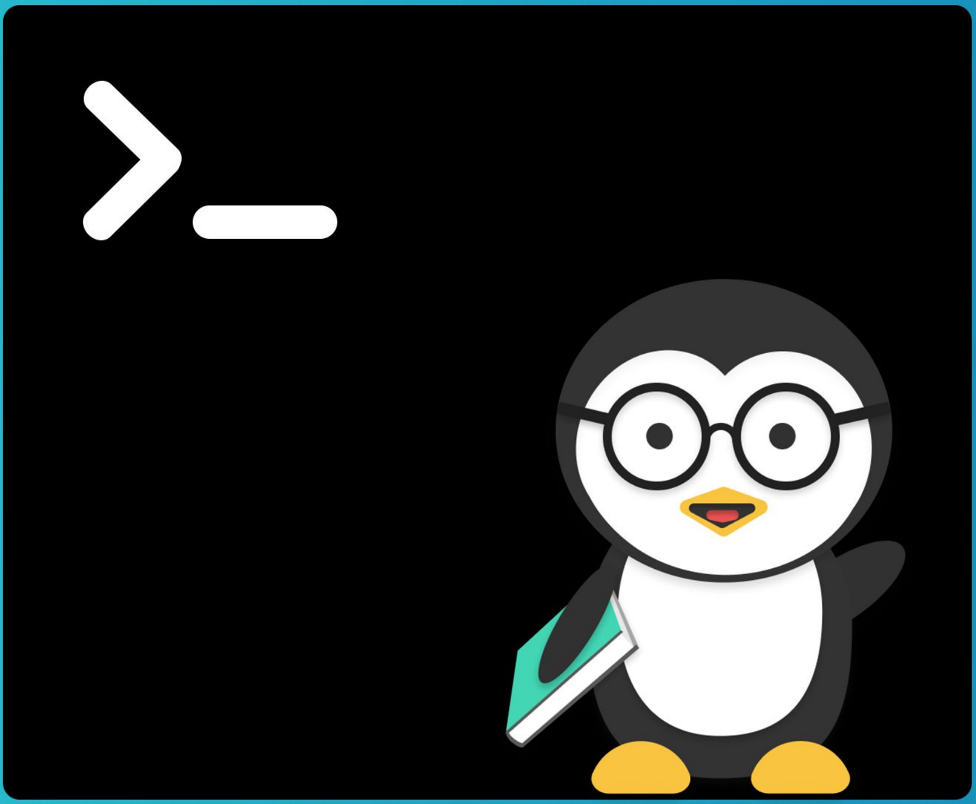
LEARN BASH QUICKLY

A Friendly Guide with Exercises to Easily Get Started with Bash Scripting



AHMED ALKABARY

A LINUX HANDBOOK PUBLICATION

Learn Bash Quickly

I dedicate this book to my friend and coach, Hatem Mersal. You inspired me to become the best version of myself.

BRIEF CONTENTS

Acknowledgments vii

Introduction viii

Chapter 1: Hello World 1

Chapter 2: Bash Variables 6

Chapter 3: Bash Script Arguments 11

Chapter 4: Bash Arrays 16

Chapter 5: Basic Arithmetic Operations 24

Chapter 6: Bash Strings 30

Chapter 7: Decision Making in Bash 37

Chapter 8: Bash Loops 45

Chapter 9: Bash Functions 53

Chapter 10: Automation with Bash 60

Chapter 11: echo 'See you later!' 66

Chapter 12: Solutions to Bash Exercises 67

CONTENTS IN DETAIL

Acknowledgments **vii**

Introduction **viii**
About the Author . . . viii
Why this Book? . . . viii
How to use this Book . . . viii
Contacting Me . . . viii
Don't Forget . . . ix

Chapter 1: Hello World **1**
Create and run your first shell script . . . 1
Convert your shell script into bash script . . . 2
The shebang line . . . 3
Editing your PATH variable . . . 3
Knowledge Check . . . 5

Chapter 2: Bash Variables **6**
Using variables in bash shell scripts . . . 6
Variables and Data Types . . . 8
Constant variables . . . 8
Command substitutions . . . 9
A smarter Hello World script . . . 10
Knowledge Check . . . 10

Chapter 3: Bash Script Arguments **11**
Passing argument to a bash script . . . 11
Passing multiple arguments to a bash script . . . 12
Getting creative with arguments . . . 13
Special bash variables . . . 14
Knowledge Check . . . 14

Chapter 4: Bash Arrays **16**
Your first array . . . 16
Accessing array elements . . . 17
Adding array elements . . . 18
Deleting array elements . . . 19
Creating hybrid arrays . . . 21
Knowledge Check . . . 22

Chapter 5: Basic Arithmetic Operations **24**
Addition and Subtraction . . . 24
Multiplication and Division . . . 25
Powers and Remainders . . . 27
Celsius to Fahrenheit Calculator . . . 28
Knowledge Check . . . 29

Chapter 6: Bash Strings **30**
Get string length 30
Concatenating two strings 30
Finding substrings 31
Extracting substrings 32
Replacing substrings 33
Deleting substrings 33
Converting upper and lowercase letters 34
Knowledge Check 35

Chapter 7: Decision Making in Bash **37**
Using if statement 37
Using if-else statement 38
Using elif statement 38
Using nested if statements 39
Using case statement 40
Test conditions in bash 41
Bonus: if else statement in one line 43
Knowledge Check 44

Chapter 8: Bash Loops **45**
For loops in bash 45
C-style for loops 45
List/Range for loops 46
While loops in bash 47
Until loops in bash 48
Traversing array elements 49
Using break and continue statements 49
Infinite loops in bash 50
Knowledge Check 51

Chapter 9: Bash Functions **53**
Creating functions in bash 53
Returning function values 54
Passing arguments to functions 55
Local and global variables 56
Recursive functions 57
Knowledge Check 59

Chapter 10: Automation with Bash **60**
Automating user management 60
Automating backups 62
Monitoring disk space 63
Knowledge Check 65

Chapter 11: echo 'See you later!' **66**
Where to go next? 66
Keep In Touch 66

Chapter 12: Solutions to Bash Exercises **67**
Exercise 1: Print number of CPU cores 67
Exercise 2: Print calendar of a given year 68
Exercise 3: Convert case 69
Exercise 4: Sort an array 70
Exercise 5: Calculate net salary 71
Exercise 6: Remove asterisks from string 72
Exercise 7: Check if year is a leap year 73
Exercise 8: Ping a bunch of servers 74
Exercise 9: Calculate the greatest common divisor 75
Exercise 10: Automatic downtime monitor 76

Acknowledgments

I am very thankful to everyone who's supported me in the process of creating this book.

First and foremost, I would like to thank Abhishek Prakash who served as the main editor of the book. Without Abhishek, this book would be full of random errors!

I am also very grateful for the support I have from the Linux Foundation; I was the recipient of the 2016 & 2020 LiFT scholarship award, and I was provided with free training courses from the Linux Foundation that have benefited me tremendously in my career.

A big shoutout also goes to all readers on **Linux Handbook** and my 160,000+ students on **Udemy**. You guys are all awesome. You gave me the motivation to write this book and bring it to life.

Last but not least, thanks to Linus Torvalds, the creator of Linux. You have changed the lives of billions on this earth for the better. God bless you!

Introduction

Learn Bash Quickly is a fully practical hands-on guide for learning bash scripting. It will get you up and running with bash scripting in no time.

About the Author

Ahmed Alkabary is a professional Linux/UNIX system administrator working at IBM Canada. He has over seven years of experience working with various flavors of Linux systems. He also works as an online technical trainer/instructor at Robertson College.

Ahmed holds two BSc degrees in Computer Science and Mathematics from the University of Regina. He also holds the following certifications:

- Red Hat Certified System Administrator (RHCSA).
- Linux Foundation Certified System Administrator (LFCS).
- AWS Certified DevOps Engineer - Professional.
- AWS Certified Solutions Architect - Associate.
- Azure DevOps Engineer Expert.
- Azure Solutions Architect Expert.
- Cisco Certified Network Associate Routing & Switching (CCNA).

Why this Book?

If you are tired of spending countless hours doing the same tedious task on Linux over and over again then this book is for you! *Learn Bash Quickly* will teach you all the skills you need to automate borings tasks in Linux. You will be much more efficient working on Linux after reading this book, more importantly, you will get more sleep, I promise you! *Learn Bash Quickly* does assume prior Linux knowledge and that you have experience working on the Linux command line.

How to use this Book

To get the most out of this book, you should read it in a chronological order as every chapter prepares you for the next, and so I recommend that you start with the first chapter and then the second chapter and so on till you reach the finish line. Also, you need to write and run every single bash script in the book by yourself, do not just read a bash script!

Contacting Me

There is nothing I like more in this life than talking about Linux; you can reach me at my personal email *ahmed.alkabary@gmail.com.* I get a lot of emails every day, so please include *Learn Bash Quickly* or *#LBQ* in the subject of your email.

Don't Forget

I would appreciate any reviews or feedback, so please don’t forget to leave a review after reading the book. Cheers!

Chapter 1: Hello World

If you have to do it more than once, automate it!

You will often find yourself repeating a single task on Linux over and over again. It may be a simple backup of a directory or it could be cleaning up temporary files or it can even be cloning of a database.

Automating a task is one of the many useful scenarios where you can leverage the power of bash scripting.

Let me show you how to create a simple bash shell script, how to run a bash script and what are the things you must know about shell scripting.

Create and run your first shell script

Let's first create a new directory named scripts that will host all our bash scripts:

```
kabary@handbook:~$ mkdir scripts
kabary@handbook:~$ cd scripts/
kabary@handbook:~/scripts$
```

Now inside this 'scripts directory', create a new file named **hello.sh** using the cat command:

```
kabary@handbook:~/scripts$ cat > hello.sh
```

Insert the following line in it by typing it in the terminal: Insert the following line in it by typing it in the terminal:

```
kabary@handbook:~/scripts$ cat > hello
echo 'Hello, World!'
```

Press **Ctrl+D** to save the text to the file and come out of the cat command.

You can also use a terminal-based text editor like Vim, Emacs or Nano. If you are using a desktop Linux, you may also use a graphical text editor like Gedit to add the text to this file.

So, basically you are using the echo command to print "Hello World". You can use this command in the terminal directly but in this test, you'll run this command through a shell script.

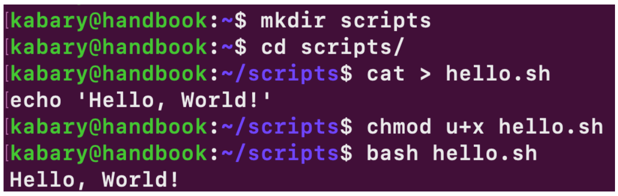
Now make the file **hello.sh** executable by using the chmod command as follows:

```
kabary@handbook:~/scripts$ chmod u+x hello.sh
```

And finally, run your first shell script by preceding the **hello.sh** file with your desired shell "bash":

```
kabary@handbook:~/scripts$ bash hello.sh
Hello, World!
```

You'll see Hello, World! printed on the screen. That was probably the easiest Hello World program you have ever written, right?
Here's a screenshot of all the steps you saw above:



```
kabary@handbook:~$ mkdir scripts
kabary@handbook:~$ cd scripts/
kabary@handbook:~/scripts$ cat > hello.sh
echo 'Hello, World!'
kabary@handbook:~/scripts$ chmod u+x hello.sh
kabary@handbook:~/scripts$ bash hello.sh
Hello, World!
```

Figure 1: First Bash Script

Convert your shell script into bash script

Confused? Don't be confused just yet. I'll explain things to you.

Bash which is short for "Bourne-Again shell" is just one type of many available shells in Linux.

A shell is a command line interpreter that accepts and runs commands. If you have ever run any Linux command before, then you have used the shell. When you open a terminal in Linux, you are already running the default shell of your system.

Bash is often the default shell in most Linux distributions. This is why bash is synonymous to shell.

The shell scripts often have almost the same syntaxes, but they also differ sometimes. For example, array index starts at 1 in Zsh instead of 0 in bash. A script written for Zsh shell won't work the same in bash if it has arrays.

To avoid unpleasant surprises, you should tell the interpreter that your shell script is written for bash shell. How do you do that? You use shebang!

The shebang line

The line “#!/bin/bash” is referred to as the shebang line and in some literature, it’s referred to as the hashbang line and that’s because it starts with the two characters hash ‘#’ and bang ‘!’.

```
#!/bin/bash

echo 'Hello, World!'
```

When you include the line “#!/bin/bash” at the very top of your script, the system knows that you want to use bash as an interpreter for your script. Thus, you can run the **hello.sh** script directly now without preceding it with bash:

```
kabary@handbook:~/scripts$ cat hello.sh
#!/bin/bash

echo 'Hello, World!'
kabary@handbook:~/scripts$ ./hello.sh
Hello, World!
```

Editing your PATH variable

You may have noticed that I used **./hello.sh** to run the script; you will get an error if you omit the leading ./

```
kabary@handbook:~/scripts$ hello.sh
hello.sh: command not found
```

Bash thought that you were trying to run a command named **hello.sh**. When you run any command on your terminal; they shell looks for that command in a set of directories that are stored in the **PATH** variable.

You can use echo to view the contents of that **PATH** variable:

```
kabary@handbook:~/scripts$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin
:/bin:/usr/games:/usr/local/games:/snap/bin
```

The colon character (:) separates the path of each of the directories that your shell scans whenever you run a command.

Linux commands like echo, cat, etc can be run from anywhere because their executable files are stored in the bin directories. The bin directories are included in the **PATH** variable. When you run a command, your system checks the **PATH** for all the possible places it should look for to find the executable for that command.

If you want to run your bash script from anywhere, as if it were a regular Linux command, add the location of your shell script to the PATH variable.

First, get the location of your script's directory (assuming you are in the same directory), use the pwd command:

```
kabary@handbook:~/scripts$ pwd
/home/kabary/scripts
```

Use the export command to add your scripts directory to the **PATH** variable:

```
kabary@handbook:~/scripts$ export PATH=$PATH:/home/kabary/scripts
```

Notice that I have appended the 'scripts directory' to the very end of our PATH variable. So that the custom path is searched after the standard directories.

Let's make sure the **scripts** directory is appended to the **PATH** variable:

```
kabary@handbook:~/scripts$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin
:/bin:/usr/games:/usr/local/games:/snap/bin:/home/kabary/scripts
```

The moment of truth is here; run **hello.sh**:

```
kabary@handbook:~/scripts$ hello.sh
Hello, World!
```

It works! This takes us to end of this chapter. I hope you now have some basic idea about shell scripting.

Knowledge Check

Bash Exercise: Print number of CPU cores

Create a bash script named **cores.sh** that prints out the number of CPU cores you have in your system.

Hint: Use the nproc command.

Solution to the exercise is mentioned at the end of the book.

Chapter 2: Bash Variables

Time changes, and so do variables!

You must have played with variables quite a bit if you did any sort of programming.

If you never worked with variables before, you can think of them as a container that stores a piece of information that can vary over time.

Variables always come in handy while writing a bash script and in this tutorial, you will learn how to use variables in your bash scripts.

Using variables in bash shell scripts

In the last chapter, you learned to write a hello world program in bash.

```
#!/bin/bash

echo 'Hello, World!'
```

That was a simple Hello World script. Let's make it a better Hello World.

Let's improve this script by using shell variables so that it greets users with their names. Edit your **hello.sh** script and use read command to get input from the user:

```
kabary@handbook:~/scripts$ cat hello.sh
#!/bin/bash

echo "What's your name, stranger?"

read name

echo "Hello, $name"
```

Now if you run your **hello.sh** script; it will prompt you for your name and then it will greet you with whatever name you provide to it:

```
kabary@handbook:~/scripts$ ./hello.sh
What's your name, stranger?
Elliot
Hello, Elliot
kabary@handbook:~/scripts$
```

In the above example, I entered **Elliot** as my name and then the script greeted me with "Hello, Elliot". That's definitely much better than a generic "Hello, World" program. Don't you agree?

Now let's go over the script line by line here to make sure that you understand everything.

I first included the shebang line to explicitly state that we are going to use bash shell to run this script.

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

Next, I ask the user to enter his/her name:

```
echo "What's your name, stranger?"
```

That's just a simple echo command to print a line to the terminal; pretty self-explanatory.

Now here's the line where all the magic happens:

```
read name
```

Here, I used the read command to transfer the control from running script to the user, so that the user can enter a name and then store whatever user entered, in the 'name' variable.

Finally, the script greets the user with their name:

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

Notice here, you have to precede the variable name with a dollar sign to get the value stored in the variable name. If you omit the dollar sign, "Hello, name" would be displayed instead.

This dollar sign is known as the dereference operator in bash scripting.

Variables and Data Types

Let's mess around a little bit more with the variables.

You can use the equal sign to create and set the value of a variable. For example, the following line will create a variable named **age** and will set its value to 27:

```
kabary@handbook:~/scripts$ age=27
```

After you have created the **age** variable, you can change its value as much as you want:

```
kabary@handbook:~/scripts$ age=3
```

The above command changes the value of the variable **age** from 27 to 3. If only times can go back, I can hear you saying!

Variables can hold different types of data; variables can store integers, strings, and characters:

```
letter='c'
color='blue'
year=2020
```

Constant variables

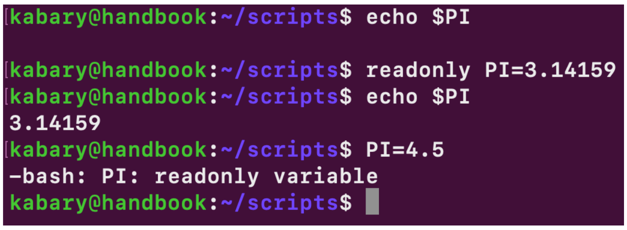
You can also create a constant variable, that is to say, a variable whose value will never change! This can be done by preceding your variable name with the readonly command:

```
kabary@handbook:~/scripts$ readonly PI=3.14159
```

The above command will create a constant variable **PI** and set its value of 3.14159. Now, you can't' change the value of constant variable, if you try, you will get an error:

```
kabary@handbook:~/scripts$ PI=123
bash: PI: readonly variable
```

As you can see, you can only read the value of a constant variable, but you can never change its value after it is created.



```
kabary@handbook:~/scripts$ echo $PI

kabary@handbook:~/scripts$ readonly PI=3.14159
kabary@handbook:~/scripts$ echo $PI
3.14159
kabary@handbook:~/scripts$ PI=4.5
-bash: PI: readonly variable
kabary@handbook:~/scripts$
```

Figure 2: Constant Variables

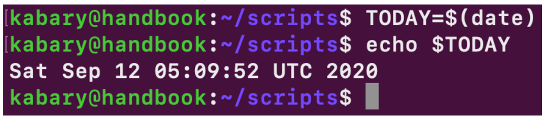
Command substitutions

The ability to store the output of a command into a variable is called command substitution and it's by far one of the most amazing features of bash.

The date command is a classic example to demonstrate command substitution:

```
kabary@handbook:~/scripts$ TODAY=$(date)
```

The above command will store the output of the command date into the variable **TODAY**. Notice, how you need to enclose the date command within a pair of parentheses and a dollar sign (on the left).



```
kabary@handbook:~/scripts$ TODAY=$(date)
kabary@handbook:~/scripts$ echo $TODAY
Sat Sep 12 05:09:52 UTC 2020
kabary@handbook:~/scripts$
```

Figure 3: Command substitution

Alternatively, you can also enclose the command within a pair of back quotes:

```
kabary@handbook:~/scripts$ TODAY=`date`
```

The back quote method is the old way of doing command substitution, and so I highly recommend that you avoid it and stick with the modern approach:

```
variable=$(command)
```

A smarter Hello World script

Now since you just learned how to do command substitution, it would make sense to visit the Hello World script one last time to perfect it!

Last time, you asked the user to enter his/her name so the script greets them; this time, you are not going to ask, your script already knows it!

Use the whoami command along with command substitution to greet whoever run the script:

```
kabary@handbook:~/scripts$ cat hello.sh
#!/bin/bash

echo "Hello, $(whoami)"
```

As you can see, you only needed just two lines! Now run the script:

```
kabary@handbook:~/scripts$ ./hello.sh
Hello, kabary
```

Alright, this brings us to the end of this chapter. I hope you have enjoyed working with shell variables as much as me. In the next chapter, you'll learn about passing arguments to your shell scripts.

Knowledge Check

Bash Exercise: Print calendar of a given year

Create a bash script named **cal.sh** that would display the calendar of given year.

The script would prompt the user to enter a year; then, it would display the corresponding year's calendar.

Hint: Use the cal command.

Solution is at the end of the book.

Chapter 3: Bash Script Arguments

Arguments can be useful, especially with Bash!

So far, you have learned how to use variables to make your bash scripts dynamic and generic, so it is responsive to various data and different user input.

In this chapter, you will see how you can pass variables to bash scripts from the command line.

Passing argument to a bash script

The following **count_lines.sh** script will output the total number of lines that exist in whatever file the user enters:

```
kabary@handbook:~/scripts$ cat count_lines.sh
#!/bin/bash

echo -n "Please enter a filename: "
read filename
nlines=$(wc -l < $filename)

echo "There are $nlines lines in $filename"
```

For example, the user can enter the **/etc/passwd** file and the script will spit out the number of lines as a result:

```
kabary@handbook:~/scripts$ ./count_lines.sh
Please enter a filename: /etc/passwd
There are 34 lines in /etc/passwd
```

This script works fine; however, there is a much better alternative!

Instead of prompting the user for the filename, we can make the user simply pass the filename as a command line argument while running the script as follows:

```
./count_lines.sh /etc/passwd
```

The first bash argument (also known as a positional parameter) can be accessed within your bash script using the **$1** variable.

So in the **count_lines.sh** script, you can replace the **filename** variable with **$1** as follows:

```
kabary@handbook:~/scripts$ cat count_lines.sh
#!/bin/bash

nlines=$(wc -l < $1)
echo "There are $nlines lines in $1"
```

Notice that I also got rid of the read and first echo commands as they are no longer needed!

Finally, you can run the script and pass any file as an argument:

```
kabary@handbook:~/scripts$ ./count_lines.sh /etc/group
There are 59 lines in /etc/group
```

Passing multiple arguments to a bash script

You can pass more than one argument to your bash script. In general, here is the syntax of passing multiple arguments to any bash script:

```
script.sh arg1 arg2 arg3  …
```

The second argument will be referenced by the **$2** variable, the third argument is referenced by **$3**, etc.

The **$0** variable contains the name of your bash script in case you were wondering!

Now we can edit our **count_lines.sh** bash script so that it can count the lines of more than one file:

```
kabary@handbook:~/scripts$ cat count_lines.sh
#!/bin/bash

n1=$(wc -l < $1)
n2=$(wc -l < $2)
n3=$(wc -l < $3)

echo "There are $n1 lines in $1"
echo "There are $n2 lines in $2"
echo "There are $n3 lines in $3"
```

You can now run the script and pass three files as arguments to the bash script:

```
kabary@handbook:~/scripts$ ./count_lines.sh /etc/passwd /etc/group /etc/hosts
There are 34 lines in /etc/passwd
There are 59 lines in /etc/group
There are 9 lines in /etc/hosts
```

As you can see, the script outputs the number of lines of each of the three files; and needless to say that the ordering of the arguments matters, of course.

Getting creative with arguments

There are a whole lot of Linux commands out there.

Some of them are a bit complicated as they may have long syntax or a long array of options that you can use.

Fortunately, you can use bash arguments to turn a hard command into a pretty easy task!

To demonstrate, take a look at the following **find.sh** bash script:

```
kabary@handbook:~/scripts$ cat find.sh
#!/bin/bash

find / -iname $1 2> /dev/null
```

It's a very simple script that yet can prove very useful! You can supply any filename as an argument to the script and it will display the location of your file:

```
kabary@handbook:~/scripts$ ./find.sh boot.log
/var/log/boot.log
```

You see how this is now much easier than typing the whole find command! This is a proof that you can use arguments to turn any long complicated command in Linux to a simple bash script.

If you are wondering about the **2> /dev/null**, it means that any error message (like file cannot be accessed) won't be displayed on the screen. This is explained in detail in my other book *Learn Linux Quickly*.

Special bash variables

Bash has a lot of built-in special variables that are quite handy and are available at your disposal.

The table below highlights the most common special built-in bash variables:

Special variable	Description
$0	The name of the bash script.
$1, $2...$n	The bash script arguments.
$$	The process id of the current shell.
$#	The total number of arguments passed to the script.
$@	The value of all the arguments passed to the script.
$?	The exit status of the last executed command.
$!	The process id of the last executed command.

Table 1: Special Bash Variables

To see these special variables in action; take a look at the following **variables.sh** bash script:

```
kabary@handbook:~/scripts$ cat variables.sh
#!/bin/bash

echo "Name of the script: $0"
echo "Total number of arguments: $#"
echo "Values of all the arguments: $@"
```

You can now pass any arguments you want and run the script:

```
kabary@handbook:~/scripts$ ./variables.sh Apple Banana Orange
Name of the script: ./variables.sh
Total number of arguments: 3
Values of all the arguments: Apple Banana Orange
```

Alright, this brings us to the end of this chapter. I hope you now realize how powerful and useful bash arguments can be. In the next chapter, you'll learn about arrays.

Knowledge Check

Bash Exercise: Convert case

Create a bash script named **upper.sh** that would display a file content in upper case letters. The file path must be passed to the script as an argument.

Hint: Use the tr command.

Solution is provided at the end of the book.

Chapter 4: Bash Arrays

Arrays to the rescue!

So far, you have used a limited number of variables in your bash script, you have created few variables to hold one or two filenames and usernames.

But what if you need more than few variables in your bash scripts; let's say you want to create a bash script that reads a hundred different input from a user, are you going to create 100 variables?

Luckily, you don't need to because arrays offer a much better solution.

Your first array

Let's say you want to create a **timestamp.sh** bash script that updates the timestamp of five different files.

First, use the naïve approach of using five different variables:

```
kabary@handbook:~/scripts$ cat timestamp.sh
#!/bin/bash

file1="f1.txt"
file2="f2.txt"
file3="f3.txt"
file4="f4.txt"
file5="f5.txt"

touch $file1
touch $file2
touch $file3
touch $file4
touch $file5
```

Now, instead of using five variables to store the value of the five filenames, you create an array that holds all the filenames, here is the general syntax of an array in bash:

```
array_name=(value1 value2 value3 … )
```

So now you can create an array named **files** that stores all the five filenames you have used in the **timestamp.sh** script as follows:

```
files=("f1.txt" "f2.txt" "f3.txt" "f4.txt" "f5.txt")
```

As you can see, this is much cleaner and more efficient as you have replaced five variables with just one array!

Accessing array elements

The first element of an array starts at index 0 and so to access the n^{th} element of an array; you use the $n - 1$ index.

For example, to print the value of the 2^{nd} element of your **files** array, you can use the following echo statement:

```
echo ${files[1]}
```

and to print the value of the 3^{rd} element of your **files** array, you can use:

```
echo ${files[2]}
```

and so on.

The following **reverse.sh** bash script would print out all the five values in your **files** array in reversed order, starting with the last array element:

```
kabary@handbook:~/scripts$ cat reverse.sh
#!/bin/bash

files=("f1.txt" "f2.txt" "f3.txt" "f4.txt" "f5.txt")

echo ${files[4]}
echo ${files[3]}
echo ${files[2]}
echo ${files[1]}
echo ${files[0]}
```

Now if you run this script, you'll see the arrays elements displayed in a reversed order:

```
kabary@handbook:~/scripts$ ./reverse.sh
f5.txt
f4.txt
f3.txt
f2.txt
f1.txt
```

I know you might be wondering why so many echo statements and why don't I use a loop here. This is because I am dedicating an entire chapter to discuss looping in bash.

You can also print out all the array elements at once:

```
echo ${files[*]}

f1.txt f2.txt f3.txt f4.txt f5.txt
```

You can print the total number of the **files** array elements, i.e. the size of the array:

```
echo ${#files[@]}

5
```

You can also update the value of any element of an array; for example, you can change the value of the first element of the **files** array to "a.txt" using the following assignment:

```
files[0]="a.txt"
```

Adding array elements

Let's create an array that contains name of the popular Linux distributions:

```
distros=("Ubuntu" "Red Hat" "Fedora")
```

The **distros** array current contains three elements. You can use the += operator to add (append) an element to the end of the array.

For example, you can append Kali to the **distros** array as follows:

```
distros+=("Kali")
```

Now the **distros** array contains exactly four array elements with Kali being the last element of the array.

Here is the complete **distros.sh** script:

```
kabary@handbook:~/scripts$ cat distros.sh
#!/bin/bash

distros=("Ubuntu" "Red Hat" ""Fedora)

echo ${distros[*]}

distros+=("kali")

echo ${distros[*]}
```

If you run this script, you'll get the following output:

```
kabary@handbook:~/scripts$ ./distros.sh
Ubuntu Red Hat Fedora
Ubuntu Red Hat Fedora kali
```

Deleting array elements

Let's first create a **num** array that will stores the numbers from 1 to 5:

```
num=(1 2 3 4 5)
```

You can print all the values in the **num** array:

```
echo ${num[*]}

1 2 3 4 5
```

You can delete the 3rd element of the num array by using the unset shell built-in command:

```
unset num[2]
```

Now if you print all the values of the **num** array:

```
echo ${num[*]}
1 2 4 5
```

As you can see, the third element of the **num** array has been deleted.

You can also delete the whole **num** array in the same way:

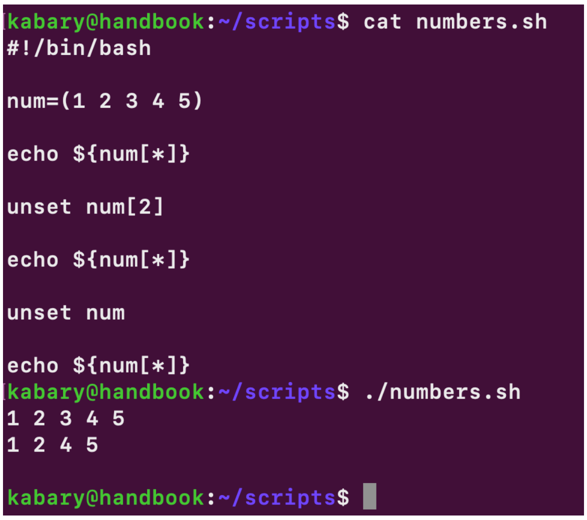
```
unset num
```

Here is the complete **numbers.sh** script:

```
kabary@handbook:~/scripts$ cat numbers.sh
#!/bin/bash

num=(1 2 3 4 5)

echo ${num[*]}

unset num[2]

echo ${num[*]}

unset num

echo ${num[*]}
```

And here's the output if your run the **numbers.sh** script:

```
kabary@handbook:~/scripts$ ./numbers.sh
1 2 3 4 5
1 2 4 5
```



```
kabary@handbook:~/scripts$ cat numbers.sh
#!/bin/bash

num=(1 2 3 4 5)

echo ${num[*]}

unset num[2]

echo ${num[*]}

unset num

echo ${num[*]}
kabary@handbook:~/scripts$ ./numbers.sh
1 2 3 4 5
1 2 4 5

kabary@handbook:~/scripts$
```

Figure 4: Running **number.sh**

Creating hybrid arrays

In bash, unlike many other programming languages, you can create an array that contains different data types. Take a look at the following **user.sh** bash script:

```
kabary@handbook:~/scripts$ cat user.sh
#!/bin/bash

user=("john" 122 "sudo,developers" "bash")

echo "User Name: ${user[0]}"
echo "User ID: ${user[1]}"
echo "User Groups: ${user[2]}"
echo "User Shell: ${user[3]}"
```

Notice the **user** array contains four elements:

1. "John" $\longrightarrow$ String Data Type
2. 122 $\longrightarrow$ Integer Data Type
3. "sudo,developers" $\longrightarrow$ String Data Type
4. "bash" $\longrightarrow$ String Data Type

So, it's totally ok to store different data types into the same array. Isn't that awesome?

```
kabary@handbook:~/scripts$ ./user.sh
User Name: john
User ID: 122
User Groups: sudo,developers
User Shell: bash
```

This brings us to the end of this chapter. In the next chapter, you will learn how to do some math in bash!

Knowledge Check

Bash Exercise: Sort an array

Consider the following **sorted.sh** bash script:

```
#!/bin/bash

num=(1 2 3 5 4)

echo "Before sorting array num: "

echo ${num[@]}

You_code_goes_here

echo "After sorting array num: "

echo ${num[@]}
```

You need to edit the **sorted.sh** script so that the **num** array becomes sorted as you can see in the following output:

```
kabary@handbook:~/scripts$ ./sorted.sh
Before sorting array num:
1 2 3 5 4
After sorting array num:
1 2 3 4 5
```

Restrictions: You are only allowed to add and delete elements from the array.

Hint: Swap the last two elements of the array; use unset to delete the fourth element of array and then add it back!

Solution is provided at the end of the book.

Chapter 5: Basic Arithmetic Operations

Let's do some bash math!

While writing your bash scripts, you will often find yourself wanting to figure out the result of an arithmetic calculation to determine a remaining disk space, file sizes, password expiry dates, number of hosts, network bandwidth, etc.

In this chapter of the bash beginner series, you will learn to use bash operators to carry out various arithmetic calculations.

To refresh your memory, here are the arithmetic operators in bash:

Operator	Description
+	Addition
-	Substraction
*	Multiplication
/	Integer division (without decimal numbers)
%	Modulus division (gives only remainder)
**	Exponentiation (x to the power y)

Table 2: Bash Arithmetic Operators

Addition and Subtraction

Let's create a bash script named **addition.sh** that will simply add two file sizes (in bytes) and display the output.

You must be familiar with arguments in bash scripts by now. I do hope you are also familiar with the cut and du commands.

The du command gives you the size of the file along with the file name. This is where the cut command is used to extract the first column (i.e. the file size) from the output. Output of the du command is passed to the cut command using a pipe redirection.

Here's the **addition.sh** script:

```
kabary@handbook:~/scripts$ cat addition.sh
#!/bin/bash

fs1=$(du -b $1 | cut -f1)
fs2=$(du -b $2 | cut -f1)

echo "File size of $1 is: $fs1"
echo "File size of $2 is: $fs2"
```

```
total=$(($fs1 + $fs2))

echo "Total size is: $total"
```

Notice that you will pass the two filenames as arguments to the script. For example, here I run the script and pass the two files **/etc/passwd** and **/etc/group** as arguments:

```
kabary@handbook:~/scripts$ ./addition.sh /etc/passwd /etc/group
File size of /etc/passwd is: 1723
File size of /etc/group is: 827
Total size is: 2550
```

The most important line in the **addition.sh** script is:

```
total=$(($fs1 + $fs2))
```

Where you used the + operator to add the two numbers **$fs1** and **$fs2**. Notice also that to evaluate any arithmetic expression you have to enclose between double parenthesis as follows:

```
$((arithmetic-expression))
```

You can also use the minus operator (-) to for subtraction. For example, the value of the **sub** variable in the following statement will result to seven:

```
sub=$((10-3))
```

Multiplication and Division

Let's create a bash script named **giga2mega.sh** that will convert Gigabytes (GB) to Megabytes (MB):

```
kabary@handbook:~/scripts$ cat giga2mega.sh
#!/bin/bash

GIGA=$1
MEGA=$(($GIGA * 1024))

echo "$GIGA GB is equal to $MEGA MB"
```

Now let's run the script to find out how many Megabytes are there in four Gigabytes:

```
kabary@handbook:~/scripts$ ./giga2mega.sh 4
4 GB is equal to 4096 MB
```

Here I used the multiplication (*) operator to multiply the number of Gigabytes by 1024 to get the Megabytes equivalent:

```
MEGA=$(($GIGA * 1024))
```

It's easy to add more functionality to this script to convert Gigabytes (GB) to Kilobytes (KB):

```
KILO=$(($GIGA * 1024 * 1024))
```

I will let you convert Gigabytes to bytes as a practice exercise!

You can also use the division operator (/) to divide two numbers. For example, the value of the **div** variable in the following statement will evaluate to five:

```
div=$((20 / 4))
```

Notice that this is integer division and so all fractions are lost. For instance, if you divide 5 by 2, you will get 2 which is incorrect, of course:

```
kabary@handbook:~/scripts$ div=$((5 / 2))
kabary@handbook:~/scripts$ echo $div
2
```

To get a decimal output; you can make use of the bc command. For example, to divide 5 by 2 with the bc command, you can use the following statement:

```
echo "5/2" | bc -l
2.50000000000000000000
```

Notice that you can use other operators as well with the bc command whenever you are dealing with decimal numbers:

```
kabary@handbook:~/scripts$ echo "5/2" | bc -l
2.50000000000000000000
kabary@handbook:~/scripts$ echo "2.5*3" | bc -l
7.5
kabary@handbook:~/scripts$ echo "1.3+2" | bc -l
3.3
kabary@handbook:~/scripts$ echo "4.1-0.5" | bc -l
3.6
```

Powers and Remainders

Let's create a power calculator! I am going to create a script named **power.sh** that will accept two numbers a and b (as arguments) and it will display the result of a raised to the power of b:

```
kabary@handbook:~/scripts$ cat power.sh
#!/bin/bash
a=$1
b=$2
result=$((a**b))
echo "$1^$2=$result"
```

Notice that I use the exponentiation operator (**) to calculate the result of a raised to the power of b.

Let's do a few runs of the script to make sure that it yields the correct answers:

```
kabary@handbook:~/scripts$ ./power.sh 2 3
2^3=8
kabary@handbook:~/scripts$ ./power.sh 3 2
```

```
3^2=9
kabary@handbook:~/scripts$ ./power.sh 5 2
5^2=25
kabary@handbook:~/scripts$ ./power.sh 4 2
4^2=16
```

You can also use the modulo operator (%) to calculate integer remainders. For instance, the value of the **rem** variable in the following statement will evaluate to 2:

```
rem=$((17%5))
```

The remainder here is 2 because 5 goes into 17 three times, and two remains!

Celsius to Fahrenheit Calculator

Let's wrap up this chapter by creating a script named **c2f.sh** that will convert Celsius degrees to Fahrenheit degrees using the equation below:

```
F = C x (9/5) + 32
```

This will be a good exercise for you to try the new things you just learned in this bash tutorial.

Here's a solution (there could be several ways to achieve the same result):

```
kabary@handbook:~/scripts$ cat c2f.sh
#!/bin/bash

C=$1
F=$(echo "scale=2; $C * (9/5) + 32" | bc -l)

echo "$C degrees Celsius is equal to $F degrees Fahrenheit."
```

I used the bc command because we are dealing with decimals and I also used "scale=2" to display the output in two decimal points.

Let's do a few runs of the script to make sure it is outputting the correct results:

```
kabary@handbook:~/scripts$ ./c2f.sh 2
2 degrees Celsius is equal to 35.60 degrees Fahrenheit.
kabary@handbook:~/scripts$ ./c2f.sh -3
-3 degrees Celsius is equal to 26.60 degrees Fahrenheit.
kabary@handbook:~/scripts$ ./c2f.sh 27
27 degrees Celsius is equal to 80.60 degrees Fahrenheit.
```

Perfect! This brings us to the end of this chapter. I hope you have enjoyed doing some math with bash and in the next chapter you will learn how to manipulate strings!

Knowledge Check

Bash Exercise: Calculate net salary

Create a bash script named **salary.sh** that would calculate the total net salary of an employee. The script would prompt the user to enter a monthly gross salary (before) and a tax rate (in percentage). Finally, the script would calculate and output the total net annual salary (after tax).

Hint: Use the bc command to handle decimals.

Your output should be similar to this:

```
kabary@handbook:~/scripts$ ./salary.sh
Please enter your monthly gross salary: 5000
Please enter your tax rate (in percentage): 10
Your total net annual salary is: 54000.00
```

Solution to the exercise can be found at the end of the book.

Chapter 6: Bash Strings

Let's manipulate some strings!

If you are familiar with variables in bash, you already know that there are no separate data types for string, int etc. Everything is a variable.

But this doesn't mean that you don't have string manipulation functions.

In this chapter, you will learn how to manipulate strings using a variety of string operations. You will learn how to get the length of a string, concatenate strings, extract substrings, replace substrings, and much more!

Get string length

Let's start with getting the length of a string in bash.

A string is nothing but a sequence (array) of characters. Let's create a string named **distro** and initialize its value to "Ubuntu":

```
distro="Ubuntu"
```

Now to get the length of the **distro** string, you just have to add # before the variable name. You can use the following echo statement:

```
kabary@handbook:~/scripts$ echo ${#distro}
6
```

Do note that the echo command is for printing the value. **{#string}** is what gives the length of string.

Concatenating two strings

You can append a string to the end of another string; this process is called string concatenation.

To demonstrate, let's first create two strings **str1** and **str2** as follows:

```
str1="hand"
str2="book"
```

Now you can join both strings and assign the result to a new string named **str3** as follows:

```
str3=$str1$str2
```

It cannot be simpler than this, can it? Here's the script:

```
kabary@handbook:~/scripts$ cat con.sh
#!/bin/bash

str1="hand"
str2="book"

str3=$str1$str2

echo $str3
```

And here's its output:

```
kabary@handbook:~/scripts$ ./con.sh
handbook
```

Finding substrings

You can find the position (index) of a specific letter or word in a string. To demonstrate, let's first create a string named **str** as follows:

```
kabary@handbook:~/scripts$ str="Bash is Cool"
```

Now you can get the specific position (index) of the substring cool. To accomplish that, use the expr command:

```
kabary@handbook:~/scripts$ word="Cool"
kabary@handbook:~/scripts$ expr index "$str" "$word"
9
```

The result 9 is the index where the word “Cool” starts in the **str** string.

> I am deliberately avoiding using conditional statements such as if, else because conditional statements will be covered later in the book.

Extracting substrings

You can also extract substrings from a string; that is to say, you can extract a letter, a word, or a few words from a string.

To demonstrate, let’s first create a string named **foss** as follows:

```
kabary@handbook:~/scripts$ foss="Fedora is a free operating system"
```

Now let’s say you want to extract the first word “Fedora” in the **foss** string. You need to specify the starting position (index) of the desired substring and the number of characters you need to extract.

Therefore, to extract the substring “Fedora”, you will use 0 as the starting position and you will extract 6 characters from the starting position:

```
kabary@handbook:~/scripts$ echo ${foss:0:6}
Fedora
```

Notice that the first position in a string is zero just like the case with arrays in bash. You may also only specify the starting position of a substring and omit the number of characters. In this case, everything from the starting position to the end of the string will be extracted.

For example, to extract the substring “free operating system” from the **foss** string; we only need to specify the starting position 12:

```
kabary@handbook:~/scripts$ echo ${foss:12}
free operating system
```

Replacing substrings

You can also replace a substring with another substring; for example, you can replace "Fedora" with "Ubuntu" in the **foss** string as follows:

```
kabary@handbook:~/scripts$ echo ${foss/Fedora/Ubuntu}
Ubuntu is a free operating system
```

Let's do another example, let's replace the substring "free" with "popular":

```
kabary@handbook:~/scripts$ echo ${foss/free/popular}
Fedora is a popular operating system
```

Since you are just printing the value with the echo command, the original string is not altered and remains intact.

Deleting substrings

You can also remove substrings. To demonstrate, let's first create a string named **fact** as follows:

```
kabary@handbook:~/scripts$ fact="Sun is a big star"
```

You can now remove the substring "big" from the **fact** string:

```
kabary@handbook:~/scripts$ echo ${fact/big}
Sun is a star
```

Let's create another string named **cell**:

```
kabary@handbook:~/scripts$ cell="112-358-1321"
```

Now let's say you want to remove all the dashes from the **cell** string; the following statement will only remove the first dash occurrence in the **cell** string:

```
kabary@handbook:~/scripts$ echo ${cell/-}
112358-1321
```

To remove all dash occurrences from the **cell** string, you have to use double forward slashes as follows:

```
kabary@handbook:~/scripts$ echo ${cell//-}
1123581321
```

Notice that you are using echo statements and so the **cell** string is intact and not modified; you are just displaying the desired result!

To modify the string, you need to assign the result back to the string as follows:

```
kabary@handbook:~/scripts$ echo $cell
112-358-1321
kabary@handbook:~/scripts$ cell=${cell//-}
kabary@handbook:~/scripts$ echo $cell
1123581321
```

Converting upper and lowercase letters

You can also convert a string to lowercase or uppercase letters. Let's first create two strings named **legend** and **actor**:

```
kabary@handbook:~/scripts$ legend="john nash"
kabary@handbook:~/scripts$ actor="JULIA ROBERTS"
```

You can convert all the letters in the **legend** string to uppercase:

```
kabary@handbook:~/scripts$ echo ${legend^^}
JOHN NASH
```

You can also convert all the letters in the **actor** string to lowercase:

```
kabary@handbook:~/scripts$ echo ${actor,,}
julia roberts
```

You can also convert only the first character of the **legend** string to uppercase as follows:

```
kabary@handbook:~/scripts$ echo ${legend^}
John nash
```

Likewise, you can convert only the first character of the **actor** string to lowercase as follows:

```
kabary@handbook:~/scripts$ echo ${actor,}
jULIA ROBERTS
```

You can also change certain characters in a string to uppercase or lowercase; for example, you can change the letters **j** and **n** to uppercase in the **legend** string as follows:

```
kabary@handbook:~/scripts$ echo ${legend^^[jn]}
JohN Nash
```

Awesome! This brings us to the end of this chapter. I hope you have enjoyed doing string manipulation in bash. Next, you'll learn to add decision-making skills to your bash scripts!

Knowledge Check

Bash Exercise: Remove asterisks from string

Create a bash script named **trim.sh** that would do the following:

- Ask the user to enter a string with asterisks.
- Remove all asterisks (*) in the string.
- Change all the letters in the string to uppercase.
- Output the updated string to the terminal.

Restriction: You are only allowed to use the echo command!

Hint: Use the escape character with the asterisk!

Your output should be similar to this:

```
kabary@handbook:~/scripts$ ./trim.sh
Please enter a string: h***an***db***oo********k
Updated string: HANDBOOK
```

Solution to the exercise can be found at the end of the book.

Chapter 7: Decision Making in Bash

Let's make our bash scripts intelligent!

In this chapter, you will learn how to use conditional statements in your bash scripts to make it behave differently in different scenarios and cases.

This way you can build much more efficient bash scripts and you can also implement error checking in your scripts.

Using if statement

The most fundamental construct in any decision-making structure is an if condition. The general syntax of a basic if statement is as follows:

```
if [ condition ]; then
  your code
fi
```

The **if** statement is closed with a **fi** (reverse of if).

Pay attention to white space!

- There must be a space between the opening and closing brackets and the condition you write. Otherwise, the shell will thrown out an error.
- There must be space before and after the conditional operator (=, ==, <=, etc). Otherwise, you'll see an error like "unary operator expected".

Now, let's create an example **root.sh** script. This script will echo the statement "you are root" only if you run the script as the root user:

```
kabary@handbook:~/scripts$ cat root.sh
#!/bin/bash

if [ $(whoami) = 'root' ]; then
        echo "You are root"
fi
```

The whoami command outputs the username. From the bash variables chapter, you know that **$(command)** syntax is used for command substitution and it gives you the output of the command.

The condition **$(whoami) = 'root'** will be true only if you are logged in as the root user.

Don't believe me? You don't have to. Run it and see it for yourself:

```
kabary@handbook:~/scripts$ ./root.sh
kabary@handbook:~/scripts$ sudo su
root@handbook:/home/kabary/scripts# ./root.sh
You are root
```

Using if-else statement

You may have noticed that you don't get any output when you run the **root.sh** script as a regular user. Any code you want to run when an if condition is evaluated to false can be included in an **else** statement as follows:

```
kabary@handbook:~/scripts$ cat root.sh
#!/bin/bash

if [ $(whoami) = 'root' ]; then
        echo "You are root"
else
        echo "You are not root"
fi
```

Now when you run the script as a regular user, you will be reminded that you are not the almighty root user:

```
kabary@handbook:~/scripts$ ./root.sh
You are not root
```

Using elif statement

You can use an **elif** (else-if) statement whenever you want to test more than one expression (condition) at the same time.

For example, the following **age.sh** bash script takes your age as an argument and will output a meaningful message that corresponds to your age:

```
kabary@handbook:~/scripts$ cat age.sh
#!/bin/bash

AGE=$1

if [ $AGE -lt 13 ]; then
        echo "You are a kid."
elif [ $AGE -lt 20 ]; then
        echo "You are a teenager."
elif [ $AGE -lt 65 ]; then
        echo "You are an adult."
else
        echo "You are an elder."
fi
```

Now do a few runs of the **age.sh** script to test out with different ages:

```
kabary@handbook:~/scripts$ ./age.sh 11
You are a kid.
kabary@handbook:~/scripts$ ./age.sh 18
You are a teenager.
kabary@handbook:~/scripts$ ./age.sh 44
You are an adult.
kabary@handbook:~/scripts$ ./age.sh 70
You are an elder.
```

Notice that I have used the **-lt** (less than) test condition with the **AGE** variable.

Also be aware that you can have multiple **elif** statements but only one **else** statement in an **if** construct and it must be closed with an **fi**.

Using nested if statements

You can also use an if statement within another if statement. For example, take a look at the following **weather.sh** bash script:

```
kabary@handbook:~/scripts$ cat weather.sh
#!/bin/bash

TEMP=$1

if [ $TEMP -gt 5 ]; then
        if [ $TEMP -lt 15 ]; then
                echo "The weather is cold."
        elif [ $TEMP -lt 25 ]; then
                echo "The weather is nice."
        else
                echo "The weather is hot."
        fi
else
        echo "It's freezing outside ..."
fi
```

The script takes any temperature as an argument and then displays a message that reflects what would the weather be like. If the temperature is greater than five, then the nested (inner) **if-elif** statement is evaluated. Let's do a few runs of the script to see how it works:

```
kabary@handbook:~/scripts$ ./weather.sh 0
It's freezing outside ...
kabary@handbook:~/scripts$ ./weather.sh 8
The weather is cold.
kabary@handbook:~/scripts$ ./weather.sh 16
The weather is nice.
kabary@handbook:~/scripts$ ./weather.sh 30
The weather is hot.
kabary@handbook:~/scripts$ ./weather.sh -20
It's freezing outside ...
```

Using case statement

You can also use **case** statements in bash to replace multiple if statements as they are sometimes confusing and hard to read. The general syntax of a **case** construct is as follows:

```
case "variable" in
        "pattern1" )
                Command … ;;
        "pattern2" )
                Command … ;;
        "pattern2" )
                Command … ;;
esac
```

Pay attention!

- The patterns are always followed by a blank space and)
- Commands are always followed by double semicolon ;; White space is not mandatory before it.
- Case statements end with **esac** (reverse of **case**).

Case statements are particularly useful when dealing with pattern matching or regular expressions. To demonstrate, take a look at the following **char.sh** bash script:

```
kabary@handbook:~/scripts$ cat char.sh
#!/bin/bash

CHAR=$1

case $CHAR in
        [a-z])
                echo "Small Alphabet." ;;
        [A-Z])
                echo "Big Alphabet." ;;
        [0-9])
                echo "Number." ;;
        *)
                echo "Special Character."
esac
```

The script takes one character as an argument and displays whether the character is small/big alphabet, number, or a special character:

```
kabary@handbook:~/scripts$ ./char.sh a
Small Alphabet.
kabary@handbook:~/scripts$ ./char.sh Z
Big Alphabet.
kabary@handbook:~/scripts$ ./char.sh 4
Number.
kabary@handbook:~/scripts$ ./char.sh $
Special Character.
```

Notice that I have used the wildcard asterisk symbol (*) to define the default case which is the equivalent of an else statement in an if condition.

Test conditions in bash

There are numerous test conditions that you can use with if statements. Test conditions varies if you are working with numbers, strings, or files. Think of them as logical operators in bash.

I have included some of the most popular test conditions in the table below:

Condition	**Meaning**
$a -lt $b	$a < $b
$a -gt $b	$a > $b
$a -le $b	$a <= $b
$a -ge $b	$a >= $b
$a -eq $b	$a is equal to $b
$a -ne $b	$a is not equal to $b
-e $FILE	$FILE exists
-d $FILE	$FILE exists and is a directory
-f $FILE	$FILE exists and is a regular file
-L $FILE	$FILE exists and is a soft link
$STRING1 = $STRING2	$STRING1 is equal to $STRING2
$STRING1 != $STRING2	$STRING1 is not equal to $STRING2
-z $STRING1	$STRING1 is empty

Table 3: Bash Test Conditions

Luckily, you don't need to memorize any of the test conditions because you can look them up in the **test** man page:

```
kabary@handbook:~/scripts$ man test
```

Let's create one final script named **filetype.sh** that detects whether a file is a regular file, directory or a soft link:

```
kabary@handbook:~/scripts$ cat filetype.sh
#!/bin/bash

if [ $# -ne 1 ]; then
        echo "Error: Invalid number of arguments"
         exit 1
fi

file=$1

if [ -f $file ]; then
        echo "$file is a regular file."
```

```
elif [ -L $file ]; then
        echo "$file is a soft link."
elif [ -d $file ]; then
        echo "$file is a directory."
else
        echo "$file does not exist"
fi
```

I improved the script a little by adding a check on number of arguments. If there are no arguments or more than one argument, the script will output a message and exit without running rest of the statements in the script.

Let's do a few runs of the script to test it with various types of files:

```
kabary@handbook:~/scripts$ ./filetype.sh weather.sh
weather.sh is a regular file.
kabary@handbook:~/scripts$ ./filetype.sh /bin
/bin is a soft link.
kabary@handbook:~/scripts$ ./filetype.sh /var
/var is a directory.
kabary@handbook:~/scripts$ ./filetype.sh
Error: Invalid number of arguments
```

Bonus: if else statement in one line

So far all the if else statement you saw were used in a proper bash script. That's the decent way of doing it but you are not obliged to it.

When you just want to see the result in the shell itself, you may use the if else statements in a single line in bash.

Consider the **root.sh** bash script:

```
kabary@handbook:~/scripts$ cat root.sh
#!/bin/bash

if [ $(whoami) = 'root' ]; then
        echo "You are root"
else
        echo "You are not root"
fi
```

You can use all the if else statements in a single line like this:

```
if [ $(whoami) = 'root' ]; then echo "root"; else echo "You are not root"; fi
```

You can copy and paste the above in terminal and see the result for yourself.

Basically, you just add semicolons after the commands and then add the next if-else statement.

Awesome! This should give you a good understanding of conditional statements in Bash. I hope you have enjoyed making your bash scripts smarter!

Knowledge Check

Bash Exercise: Check if year is a leap year

Create a bash script named **isleap.sh** that would determine whether or not a given year is a leap year. A year is a leap year if one of the following conditions is satisfied:

- Year is multiple of 400.
- Year is multiple of 4 and not multiple of 100.

You can pass the year as an argument to your script.

Hint: Use the remainder operator!

Your output should be similar to this:

```
kabary@handbook:~/scripts$ ./isleap.sh 2014
2014 is not a leap year.
kabary@handbook:~/scripts$ ./isleap.sh 2016
2016 is a leap year.
kabary@handbook:~/scripts$ ./isleap.sh 2020
2020 is a leap year.
kabary@handbook:~/scripts$ ./isleap.sh 2100
2100 is not a leap year.
kabary@handbook:~/scripts$ ./isleap.sh 2200
2200 is not a leap year.
kabary@handbook:~/scripts$ ./isleap.sh 2204
2204 is a leap year.
```

Solution to the exercise can be found at the end of the book.

Chapter 8: Bash Loops

Beware of infinite loops!

The ability to loop is a very powerful feature of bash scripting. Loops have a variety of use cases.

In this chapter, you will explore the three different bash loop structures. You will also learn how to use loops to traverse array elements.

Furthermore, you will learn how to use break and continue statements to control loops, and finally, you will learn how to create infinite loops.

For loops in bash

For loops are one of three different types of loop structures that you can use in bash. There are two different styles for writing a for loop:

- C-styled for loops
- Using for loop on a list/range of items

C-style for loops

If you are familiar with a C or C++ like programming language, then you will recognize the following for loop syntax:

```
for ((initialize ; condition ; increment)); do
    [COMMANDS]
done
```

Using the aforementioned C-style syntax, the following **for.sh** bash script will print out "Hello Friend" ten times:

```
kabary@handbook:~/scripts$ cat for.sh
#!/bin/bash

for ((i = 0 ; i < 10 ; i++)); do
        echo "Hello Friend"
done
```

The for loop first initialized the integer variable i to zero then it tests the condition ($i < 10$); if true, then the loop executes the line echo "Hello Friend" and increments the variable i by 1, and then the loop runs again and again until i is no longer less than 10:

```
kabary@handbook:~/scripts$ ./for.sh
Hello Friend
Hello Friend
Hello Friend
Hello Friend
Hello Friend
Hello Friend
Hello Friend
Hello Friend
Hello Friend
Hello Friend
```

List/Range for loops

Another syntax variation of for loop also exists that is particularly useful if you are working with a list of files (or strings), range of numbers, arrays, output of a command, etc. The list/range syntax for loop takes the following form:

```
for item in [LIST]; do
  [COMMANDS]
done
```

For example, the following for loop does exactly the same thing as the C-style for loop you had created in the previous section:

```
for i in {1..10}; do
        echo "Hello Friend"
done
```

The **var.sh** script below will print out the names for all the files and directories that exists under the **/var** directory:

```
kabary@handbook:~/scripts$ cat var.sh
#!/bin/bash

for i in /var/*; do
        echo $i
done
```

Below is sample output when you run the **var.sh** script:

```
kabary@handbook:~/scripts$ ./var.sh
/var/backups
/var/cache
/var/crash
/var/lib
/var/local
/var/lock
/var/log
/var/mail
/var/opt
/var/run
/var/snap
/var/spool
/var/tmp
```

While loops in bash

The while loop is another popular and intuitive loop you can use in bash scripts. The general syntax for a while loop is as follows:

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

For example, the following **3x10.sh** script uses a while loop to print the first ten multiples of the number three:

```
kabary@handbook:~/scripts$ cat 3x10.sh
#!/bin/bash

num=1
while [ $num -le 10 ]; do
        echo $(($num * 3))
        num=$(($num+1))
done
```

Here's the output of the **3x10** script:

```
kabary@handbook:~/scripts$ ./3x10.sh
3
6
9
```

```
12
15
18
21
24
27
30
```

The script first initialized the **num** variable to 1; then, the while loop will run as long as **num** is less than or equal to 10. Inside the body of the while loop, the echo command print out the value of **num** multiplied by three and then it increments **num** by 1.

Until loops in bash

If you are coming from a C/C++ background, you might be looking for a do-while loop but that one doesn't exist in bash.

There is another kind of loop that exists in bash. The until loop follows the same syntax as the while loop:

```
until [ condition ]; do
        [COMMANDS]
done
```

The key difference between until loop and while loop is in the test condition. A while loop will keep running as long as the test condition is true; on the flip side, an until loop will keep running as long as test condition is false!

For example, you can easily recreate the **3x10.sh** script with an until loop instead of a while loop; the trick here is to negate the test condition:

```
#!/bin/bash

num=1
until [ $num -gt 10 ]; do
        echo $(($num * 3))
        num=$(($num+1))
done
```

Notice that the negation of the test condition [**$num -le** 10]; is [**$num -gt** 10];

Traversing array elements

For loops are often the most popular choice when it comes to iterating over array elements.

For example, the following **prime.sh** script iterates over and prints out each element in the **prime** array:

```
kabary@handbook:~/scripts$ cat prime.sh
#!/bin/bash

prime=(2 3 5 7 11 13 17 19 23 29)

for i in "${prime[@]}"; do
        echo $i
done
```

This is the output of the **prime.sh** script:

```
kabary@handbook:~/scripts$ ./prime.sh
2
3
5
7
11
13
17
19
23
29
```

Using break and continue statements

Sometimes you may want to exit a loop prematurely or skip a loop iteration. To do this, you can use the break and continue statements.

The **break** statement terminates the execution of a loop and turn the program control to the next command or instruction following the loop.

For example, the following loop would only print the numbers from one to three:

```
for ((i=1;i<=10;i++)); do
        echo $i
        if [ $i -eq 3 ]; then
                break
        fi
done
```

You can also use a **continue** statement to skip a loop iteration. The loop continues and moves to the next iteration but the commands after the continue statements are skipped in that partcular iteration.

For example, the following **odd.sh** script would only print the odd numbers from one to ten as it skips over all even numbers:

```
kabary@handbook:~/scripts$ cat odd.sh
#!/bin/bash

for ((i=0;i<=10;i++)); do
        if [ $(($i % 2)) -ne 1 ]; then
                continue
        fi
        echo $i
done
```

Here's the output of the **odd.sh** script:

```
kabary@handbook:~/scripts$ ./odd.sh
1
3
5
7
9
```

Infinite loops in bash

An infinite loop is a loop that keeps running forever; this happens when the loop test condition is always true.

In most cases, infinite loops are a product of a human logical error.

For example, someone who may want to create a loop that prints the numbers 1 to 10 in descending order may end up creating the following infinite loop by mistake:

```
for ((i=10;i>0;i++)); do
        echo $i
done
```

The problem is that the loop keeps incrementing the variable *i* by 1. To fix it, you need to change i++ with i-- as follows:

```
for ((i=10;i>0;i--)); do
        echo $i
done
```

In some cases, you may want to intentionally create infinite loops to wait for an external condition to be met on the system. You can easily create an infinite for loop as follows:

```
for ((;;)); do
        [COMMANDS]
done
```

If you want to create an infinite while loop instead, then you can create it as follows:

```
while [ true ]; do
        [COMMANDS]
done
```

Awesome! This brings us to the end of this chapter. I hope you have enjoyed looping around in bash! In the next chapter, you'll learn to reuse code by creating functions.

Knowledge Check

Bash Exercise: Ping a bunch of servers

Create a bash script named **subnet.sh** that would ping every single server in the 23.227.36.x subnet where x is a number between 0 and 255. If a ping succeeded, display the statement "Server 23.227.36.x is up and running". Otherwise, if a ping failed, display the statement "Server 23.227.36.x is unreachable."

Hint: Use a for loop and the ping command!

Your output should be similar to this:

```
kabary@handbook:~/scripts$ ./subnet.sh
Server 23.227.36.0 is unreachable.
Server 23.227.36.1 is up and running.
Server 23.227.36.2 is up and running.
Server 23.227.36.3 is up and running.
Server 23.227.36.4 is up and running.
Server 23.227.36.5 is up and running.
Server 23.227.36.6 is up and running.
Server 23.227.36.7 is up and running.
Server 23.227.36.8 is up and running.
Server 23.227.36.9 is up and running.
Server 23.227.36.10 is unreachable
Server 23.227.36.11 is unreachable.
.
.
.
```

Solution to the exercise can be found at the end of the book.

Chapter 9: Bash Functions

Never rewrite code, use functions instead!

When your bash scripts get bigger and bigger, things can get very messy!

You may find yourself rewriting the same pieces of code again and again in different parts of your bash scripts.

Luckily, you can avoid rewriting code by using functions in bash which will make your scripts more organized and readable.

In this chapter, you will learn to create functions, return function values, and pass function arguments in bash shell scripts.

Furthermore, you will learn how variables scope work and how to define recursive functions.

Creating functions in bash

There are two different syntaxes for declaring bash functions. The following syntax is the most common used way of creating bash functions:

```
function_name () {
    commands
}
```

The second less commonly used of creating bash functions starts with the reserved work function followed by the function name as follows:

```
function function_name {
    commands
}
```

Now there are a couple of things you should be well aware of when working with functions:

- A function will never run/execute unless you invoke/call the function.
- The function definition must precede any calls to the function.

Anytime you want a function to run, you just need to call it! A function call is done by simply referencing the function name.

Take a look at the following **fun.sh** bash script:

```
kabary@handbook:~/scripts$ cat fun.sh
#!/bin/bash

hello () {
        echo "Hello World"
}

hello
hello
hello
```

I defined a function named **hello** that simply echo's the line "Hello World" to the terminal. Notice that I did three **hello** function calls and so if you run the script, you will see the "Hello World" line printed three times on the screen:

```
kabary@handbook:~/scripts$ ./fun.sh
Hello World
Hello World
Hello World
```

Returning function values

In many programming languages, functions do return a value when called; however, this is not the case with bash as bash functions do not return values.

When a bash function finishes executing, it returns the exit status of the last executed command captured in the **$?** variable. Zero indicates a successful execution and a non-zero positive integer $(1-255)$ indicates failure.

You can use a **return** statement to alter the function's exit status. For example, take a look at the following **error.sh** script:

```
kabary@handbook:~/scripts$ cat error.sh
#!/bin/bash

error () {
        blabla
        return 0
}

error
echo "The return status of the error function is: $?"
```

If you run the **error.sh** bash script, you might be surprised by the output:

```
kabary@handbook:~/scripts$ ./error.sh
./error.sh: line 4: blabla: command not found
The return status of the error function is: 0
```

Without the **return 0** statement, the **error** function would have never returned a non-zero exit status as **blabla** results in a command not found error.

So as you can see, even though bash functions do not return values, I made a workaround by altering function exit statuses.

You should also be aware that a return statement immediately terminates a function.

Passing arguments to functions

You can pass arguments to a function just like you can pass arguments to a bash script. You just include the arguments when you do the function call.

To demonstrate, let's take a look at the following **iseven.sh** bash script:

```
kabary@handbook:~/scripts$ cat iseven.sh
#!/bin/bash

iseven () {
        if [ $(($1 % 2)) -eq 0 ]; then
                echo "$1 is even."
        else
                echo "$1 is odd."
        fi
}

iseven 3
iseven 4
iseven 20
iseven 111
```

The **iseven** function tests whether a number is even or odd. I did four function calls to **iseven**. For each function call, I supplied one number which is the first argument to the **iseven** function and is referenced by the **$1** variable in the function definition.

Let's run the **iseven.sh** bash script to make sure it works:

```
kabary@handbook:~/scripts$ ./iseven.sh
3 is odd.
4 is even.
20 is even.
111 is odd.
```

You should also be well aware that bash function arguments and bash script arguments are two different things. To contrast the difference, take a look at the following **funarg.sh** bash script:

```
kabary@handbook:~/scripts$ cat funarg.sh
#!/bin/bash

fun () {
        echo "$1 is the first argument to fun()"
        echo "$2 is the second argument to fun()"
}

echo "$1 is the first argument to the script."
echo "$2 is the second argument to the script."

fun Yes 7
```

Run the script with a couple of arguments and observe the result:

```
kabary@handbook:~/scripts$ ./funarg.sh Cool Stuff
Cool is the first argument to the script.
Stuff is the second argument to the script.
Yes is the first argument to fun()
7 is the second argument to fun()
```

As you can see, even though you used the same variables **$1** and **$2** to refer to both the script arguments and the function arguments, they produce different results when called from within a function.

Local and global variables

Bash variables can either have a global or local scope. You can access a global variable anywhere in a bash script regardless of the scope. On the contrary, a local variables can only be accessed from within their function definition.

To demonstrate, take a look at the following **scope.sh** bash script:

```
kabary@handbook:~/scripts$ cat scope.sh
#!/bin/bash

v1='A'
v2='B'

myfun() {
        local v1='C'
        v2='D'
        echo "Inside myfun(): v1: $v1, v2: $v2"
}

echo "Before calling myfun(): v1: $v1, v2: $v2"
myfun
echo "After calling myfun(): v1: $v1, v2: $v2"
```

I first defined two global variables **v1** and **v2**. Then inside **myfun** function definition, I used the **local** keyword to define a local variable **v1** and modified the global variable **v2**. Note that you can use the same variable name for local variables in different functions.

Now let's run the script:

```
kabary@handbook:~/scripts$ ./scope.sh
Before calling myfun(): v1: A, v2: B
Inside myfun(): v1: C, v2: D
After calling myfun(): v1: A, v2: D
```

From the script output, you can conclude the following:

- A Local variable that have the same name as a global variable will take precedence over global variables inside a function body.
- You can change a global variable from within a function.

Recursive functions

A recursive function is a function that calls itself! Recursive functions come in handy when you attempt to solve a programming problem that can be broken down to smaller subproblems.

The factorial function is a classic example of a recursive function. Take a look at the following **factorial.sh** bash script:

```
kabary@handbook:~/scripts$ cat factorial.sh
#!/bin/bash

factorial () {

        if [ $1 -le 1 ]; then
                echo 1
        else
                last=$(factorial $(( $1 -1)))
                echo $(( $1 * last ))
        fi
}

echo -n "4! is: "
factorial 4
echo -n "5! is: "
factorial 5
echo -n "6! is: "
factorial 6
```

Any recursive function must begin with a base case which is necessarily to end the chain of recursive function calls. In the **factorial** function, the base case is defined as follows:

```
if [ $1 -le 1 ]; then
    echo 1
```

Now derive the recursive case for the **factorial** function. To calculate the factorial of a number n where n is a positive number greater than one, you can multiply n by the factorial of $n - 1$:

$$\textbf{factorial}(n) = n * \textbf{factorial}(n - 1)$$

Let's use the above equation to write this recursive case:

```
last=$(factorial $(( $1 -1)))
echo $(( $1 * last ))
```

Now run the script to make sure you get the correct results:

```
kabary@handbook:~/scripts$ ./factorial.sh
4! is: 24
5! is: 120
6! is: 720
```

As an additional exercise, try to write a recursive function to calculate the n^{th} Fibonacci number. First, try to come up with the base case and then the recursive case; you got this!

Knowledge Check

Bash Exercise: Calculate the greatest common divisor

Create a script named **gcd.sh** that will calculate the greatest common divisor of two numbers: **num1** and **num2** (passed as script arguments).

In mathematics, the greatest common divisor of two or more integers, which are not all zero, is the largest positive integer that divides each of the integers.

In your script, create a function named **common_divisor** that tests whether a number evenly divides **num1** and **num2**.

Hint: Remainders and loops are your best friends!

Your output should be similar to this:

```
kabary@handbook:~/scripts$ ./gcd.sh 25 5
The gcd of 25 and 5 is: 5
kabary@handbook:~/scripts$ ./gcd.sh 100 80
The gcd of 100 and 80 is: 20
kabary@handbook:~/scripts$ ./gcd.sh 1000 600
The gcd of 1000 and 600 is: 200
kabary@handbook:~/scripts$ ./gcd.sh -100 -88
The gcd of -100 and -88 is: 1
kabary@handbook:~/scripts$ ./gcd.sh 999 777
The gcd of 999 and 777 is: 111
```

Solution to the exercise can be found at the end of the book.

Chapter 10: Automation with Bash

You're either the one that creates the automation, or you're getting automated.

You can now use all the bash skills you have learned in this book to create very useful bash scripts that would help you in automating boring repetitive administrative tasks.

Automation should really be your ultimate goal whenever you write a bash script.

In this chapter, I will show you some automation scripts that you can later extend on to automate any task you want. These scripts will utilize bash arrays, if-else, loops and other concepts you have learned in this series.

Automating user management

Creating a user on multiple servers maybe something that you would do on a daily basis as a sysadmin. It is a tedious a task, and so let's create a bash script that automates it.

First, create a text file that includes all the server hostnames or IP addresses where you wish to add the new user on.

For example, here I created the file **servers.txt** that includes five different servers:

```
kabary@handbook:~/scripts$ cat servers.txt
server1
server2
server3
server4
server5
```

Keep in mind, that I have used server hostnames as I have already included the IP addresses of the servers in my **/etc/hosts** file.

Now take a look at the following **adduser.sh** bash script:

```
kabary@handbook:~/scripts$ cat adduser.sh
#!/bin/bash

servers=$(cat servers.txt)

echo -n "Enter the username: "
read name
echo -n "Enter the user id: "
```

```
read uid

for i in $servers; do
        echo $i
        ssh $i "sudo useradd -m -u $uid ansible"
        if [ $? -eq 0 ]; then
                echo "User $name added on $i"
        else
                echo "Error on $i"
        fi
done
```

The **adduser.sh** script would first ask you to enter the username and the user id of the user who you want to add; then, it will loop over and connect to all the servers in the **servers.txt** file via SSH and add the requested user.

Let's run the script and see how it works:

```
kabary@handbook:~/scripts$ ./adduser.sh
Enter the username: ansible
Enter the user id: 777
server1
User ansible added on server1
server2
User ansible added on server2
server3
User ansible added on server3
server4
User ansible added on server4
server5
User ansible added on server5
```

The script ran successfully and user ansible was added on all five servers. There are a couple of important points here you need to understand:

- You can either use empty **ssh** pass phrases or run **ssh-agent** to avoid getting prompted to enter a key (or password) while the script is running.
- You must have a valid account that has super user access (without a password requirement) on all the servers.

Imagine that you need to add a user on 100+ different Linux servers! The **adduser.sh** script can definitely save you countless hours of work.

Automating backups

Taking backups is something that we all do on a regular basis so why not automate it? Take a look at the following **backup.sh** script:

```
kabary@handbook:~/scripts$ cat backup.sh
#!/bin/bash

backup_dirs=("/etc" "/home" "/boot")
dest_dir="/backup"
dest_server="server1"
backup_date=$(date +%b-%d-%y)

echo "Starting backup of: ${backup_dirs[@]}"

for i in "${backup_dirs[@]}"; do
        sudo tar -Pczf /tmp/$i-$backup_date.tar.gz $i
        if [ $? -eq 0 ]; then
                echo "$i backup succeeded."
        else
                echo "$i backup failed."
        fi
        scp /tmp/$i-$backup_date.tar.gz $dest_server:$dest_dir
        if [ $? -eq 0 ]; then
                echo "$i transfer succeeded."
        else
                echo "$i transfer failed."
        fi
done

sudo rm /tmp/*.gz
echo "Backup is done."
```

So, the script first creates an array named **backup_dirs** that stores all directory names that we want to backup. Then, the script creates three other variables:

- **dest_dir** $\longrightarrow$ To specify the backup destination directory.
- **dest_server** $\longrightarrow$ To specify the backup destination server.
- **backup_time** $\longrightarrow$ To specify the date of the backup.

Next, for all the directories in the **backup_dirs** array, the script creates a gzip compressed tar archive in **/tmp**, then uses the scp command to send/copy the backup to the destination server. Finally, the script removes all the gzip archives from **/tmp**.

Here is a sample run of the **backup.sh** script:

```
kabary@handbook:~/scripts$ ./backup.sh
Starting backup of: /etc /home /boot
/etc backup succeeded.
etc-Aug-30-20.tar.gz 100% 1288KB 460.1KB/s   00:02
/etc transfer succeeded.
/home backup succeeded.
home-Aug-30-20.tar.gz 100% 2543KB 547.0KB/s   00:04
/home transfer succeeded.
/boot backup succeeded.
boot-Aug-30-20.tar.gz 100%  105MB 520.2KB/s   03:26
/boot transfer succeeded.
Backup is done.
```

You may want to run take the backups every day at midnight. In this case, you can schedule the script to run as a cron job:

```
kabary@handbook:~/scripts$ crontab -e
0 0 * * * /home/kabary/scripts/backup.sh
```

Monitoring disk space

Filesystems are destined to run out of space, the only thing you can do is to act fast before your system crashes! You can use the df command to see the remaining space on any filesystem:

```
kabary@handbook:~$ df -h / /apps /database
Filesystem Size  Used Avail Use% Mounted on
/dev/sda5 20G  7.9G   11G  44% /
/dev/mapper/vg1-applv 4.9G  2.4G  2.3G  52% /apps
/dev/mapper/vg1-dblv 4.9G  4.5G  180M  97% /database
```

My **/database** filesystem is almost out of space as it's currently sitting at 97% usage. I can display only the usage if I use the awk command to only show the fifth field.

Now take a look at the following **disk_space.sh** bash script:

```
kabary@handbook:~/scripts$ cat disk_space.sh
#!/bin/bash
```

```
filesystems=("/" "/apps" "/database")

for i in ${filesystems[@]}; do
        usage=$(df -h $i | tail -n 1 | awk '{print $5}' | cut -d % -f1)
        if [ $usage -ge 90 ]; then
                alert="Running out of space on $i, Usage is: $usage%"
                echo "Sending out a disk space alert email."
                echo $alert | mail -s "$i is $usage% full" your_email
        fi
done
```

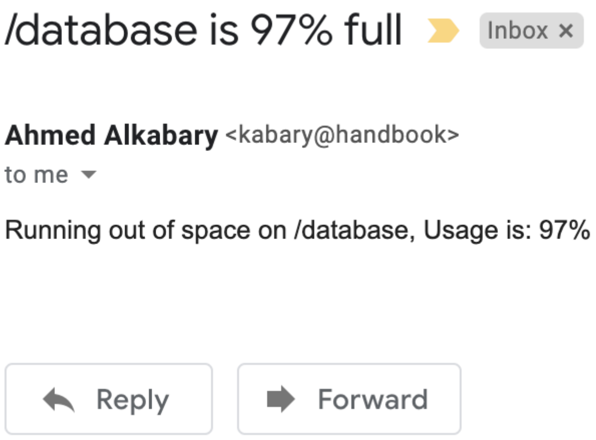
First, you created a **filesystems** array that holds all the filesystems that you want to monitor. Then, for each filesystem, you grab the usage percentage and check to see if it's bigger than or equal to 90. If usage is above 90%, the script sends an alert email indicating that the filesystem is running out of space.

Notice that you need to replace **your_email** in the script with your actual email.

I ran the script:

```
kabary@handbook:~/scripts$ ./disk_space.sh
Sending out a disk space alert email.
```

And shortly after, I got the following email in my inbox:


/database is 97% full

Ahmed Alkabary <kabary@handbook>

to me

Running out of space on /database, Usage is: 97%

Figure 5: Disk Space Alert Email

You may want to run the **disk_space.sh** script six hours or so. In this case, you can also schedule the script to run as a cron job:

```
kabary@handbook:~/scripts$ crontab -e
0 */6 * * * /home/kabary/scripts/disk_space.sh
```

Awesome! This brings us to the end of the book. I hope you have enjoyed learning bash scripting. With bash scripting in your skills arsenal, you can automate any boring tedious task on Linux!

Knowledge Check

Bash Exercise: Automatic downtime monitor

Create a bash script named **outage.sh** that would automatically detect whether a server in your environment is down! Store all your server hostnames that you want to monitor in a file named **servers.txt** as follows:

```
kabary@handbook:~/scripts$ cat servers.txt
server1
server2
server3
server4
server5
```

If there is server outage, send an email to yourself specifying which server is unreachable. Your script should run every 2 hours.

Hint: Use mail and cron.

Solution to the exercise can be found at the end of the book.

Chapter 11: echo 'See you later!'

I want to congratulate you on finishing reading the book and learning how to automate borings tasks with bash scripting. I hope you have enjoyed reading the book as much as I have enjoyed writing it!

Where to go next?

You now may be wondering, ”where do I go next from here?”; most people ask the same question after learning a new skill, and here’s my two cents on what to do after learning Bash Scripting:

- Put your new skill to work! If you don’t keep practicing what you have learned, you will eventually lose it. Always try and write a bash script for anything that you do repetitively on Linux.
- Make money! **Linux** is in huge demand; Start applying for Linux jobs.
- Learn a newer enterprise automation tool like **Ansible** or **Puppet**.

Keep In Touch

You can connect with me on LinkedIn. Also, do not hesitate to send me an email if you ever want to enroll in any of my Udemy courses; I will be more than happy to send you a free coupon! Also, do not forget to subscribe to ***Linux Handbook*** so you don’t miss out on the amazing Linux content.

Chapter 12: Solutions to Bash Exercises

Exercise 1: Print number of CPU cores

```
kabary@handbook:~/scripts$ cat cores.sh
#!/bin/bash

echo -n "Number of CPU(s): "; nproc
kabary@handbook:~/scripts$ ./cores.sh
Number of CPU(s): 2
```

Exercise 2: Print calendar of a given year

```
kabary@handbook:~/scripts$ cat cal.sh
#!/bin/bash

echo -n "Please enter a year: "
read year
echo "Calendar of $year"
cal $year
```

Exercise 3: Convert case

```
kabary@handbook:~/scripts$ cat upper.sh
#!/bin/bash

echo "Displaying Content of $1 in upper case."

cat $1 | tr [:lower:] [:upper:]
```

Exercise 4: Sort an array

```
#!/bin/bash

num=(1 2 3 5 4)

echo "Before sorting array num: "

echo ${num[@]}

You_code_goes_here

echo "After sorting array num: "

echo ${num[@]}
```

Exercise 5: Calculate net salary

```
kabary@handbook:~/scripts$ cat salary.sh
#!/bin/bash

echo -n "Please enter your monthly gross salary: "
read mgross
echo -n "Please enter your tax rate (in percentage): "
read trate

mdeduct=$(echo "scale=2; ($trate/100)*$mgross" | bc -l)
mnet=$(echo "$mgross-$mdeduct" | bc -l)
tnet=$(echo "$mnet*12" | bc -l)

echo "Your total net annual salary is: $tnet"
```

Exercise 6: Remove asterisks from string

```
#!/bin/bash

echo -n "Please enter a string: "
read str1

str1=`echo ${str1//\*}`
str1=`echo ${str1^^}`

echo "Updated string: $str1"
```

Exercise 7: Check if year is a leap year

```
kabary@handbook:~/scripts$ cat isleap.sh
#!/bin/bash

year=$1

if [ $(($year % 400)) -eq 0 ]; then
    echo "$year is a leap year."
elif [ $(($year % 4)) -eq 0 ] && [ $(($year % 100)) -ne 0 ]; then
    echo "$year is a leap year."
else
    echo "$year is not a leap year."
fi
```

Exercise 8: Ping a bunch of servers

```
kabary@handbook:~$ cat subnet.sh
#!/bin/bash

for((i=0;i<256;i++)); do
        ping -c 1 23.227.36.$i >> /dev/null 2>&1
        if [ $? -eq 0 ]; then
                echo "Server 23.227.36.$i is up and running."
        else
                echo "Server 23.227.36.$i is unreachable."
        fi
done
```

Exercise 9: Calculate the greatest common divisor

```
kabary@handbook:~/scripts$ cat gcd.sh
#!/bin/bash

num1=$1
num2=$2
gcd=1

common_divisor() {
x=$1

if [ $(($num1 % $x)) -eq 0 ] && [ $(($num2 % $x)) -eq 0 ]; then
        return 1
else
        return 0
fi
}

for ((i=1;i<=$num1 && i<=$num2;i++)); do

        common_divisor $i
        if [ $? -eq 1 ]; then
                gcd=$i
        fi
done

echo "The gcd of $num1 and $num2 is: $gcd"
```

Exercise 10: Automatic downtime monitor

```
kabary@handbook:~/scripts$ cat outage.sh
#!/bin/bash

servers=$(cat servers.txt)

for i in $servers; do
        ping -c 1 $i >> /dev/null 2>&1
        if [ $? -ne 0 ]; then
                echo "Sending out an outage email."
                echo "$i is unreachable." | mail -s "$i Outage" your_email
        fi
done
```