Introductory Guide to Bash

For: COMP 206: Introduction to Software Systems

Author: Mairead Shaw

Version: 1.0

June 14, 2017

1 About this Guide

Hello, and welcome to the wonderful world of Bash! The aim of this guide is to explain what Bash is and how to use it. It is divided into eight sections:

1. Introduction
2. Variables
3. Passing Arguments to Scripts
4. Basic Operators
5. Basic String Operators
6. Decision-Making
7. Arrays
8. Loops

The sections follow a general structure. First, there is an explanation of the aspect of Bash being discussed. Then, there is a demonstration of the syntax required to use that aspect. Syntax is followed by a worked example using that aspect and exercises for you to work on independently. Finally, there are links for further readings that go into more detail, and that I referenced while writing the section.

Note: this guide assumes you have basic familiarity with using the command line, i.e. you know how to type commands into the command line.

Happy Bashing!

2 Introduction

2.1 What is Bash?

"Bash" is an acronym for "Bourne-again shell", so called because it replaced another shell, the Bourne shell. A shell program is one that allows users to interact with other programs.

Essentially, Bash is a program on your computer. It is designed to take commands from you, and you will use the Bash shell language to give those commands.

2.2 Modes of Use

There are two ways you can use Bash: interactively, and non-interactively.

In the interactive mode, Bash executes commands as you type them in. Between commands, it just waits. When you type commands like "ls", "cd", "mkdir", etc. into the command-line, you are using the interactive mode.

In the non-interactive mode, Bash runs a script from top to bottom. A script is a set of pre-written commands, and is generally saved in a file and used to automate commands.

2.3 Hello, world! Worked Example

Now, let's run through printing "Hello, world!" to the screen interactively and non-interactively. Don't worry about the use of the quotations around 'Hello, world!' just yet. We'll get to quotes in another section; this example is just to show you the different modes of interacting with Bash.

Interactively: type "echo 'Hello, world!'" (without the external "") into the command-line.

Non-interactively: follow these steps:

- At the command line, type "vi hw.sh" (without "")
  - This will open the text editor Vi
  - I'm calling it 'hw.sh' for 'helloworld.sh'
- To enter text, press "i" (for "insert")
- Type "#!/bin/bash" into the first line
  - This says that your file is being written for Bash
- Type "echo 'Hello, world!'" into the second line
- Press the escape key to exit text-entering mode

- Type ":", then "wq" to save the file and exit Vi
  - "wq" stands for "write" and "quit"
- Type "chmod +x hw.sh"
  - This makes the script executable, so that you can run it
- Type "./hw.sh" to execute the script

You may be thinking that the non-interactive method has too many steps to be useful. And if you were only to execute the one "Hello, world!" command once, you'd be right! But imagine if you had to execute it thousands of times, or if there were hundreds of commands within the script, or if you wanted it to automatically execute when you start up your command line. **Then** it becomes useful.

2.4 Exercise

Print "Hi, my name is *your name here*" to the screen using both the interactive and non-interactive modes of operation.

2.5 Further Reading

http://guide.bash.academy/inception/

https://www.learnshell.org/en/Hello%2C_World%21

3 Variables

Shell variables are used to store information, usually to make it easy to reference that information later. For example, let's say I want to reference the Welsh town "Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch" in my script. Instead of typing out that entire name every time, I could store it in a variable called "town", both saving my time and making my code more readable.

3.1 Variable Syntax

Syntax: VAR-NAME=value

3.2 Declaring Variables

Shell variables are created once they are assigned a value. Variables can hold numbers, characters, or strings.

Variable names are case-sensitive, and may contain letters, numbers, and underscores ("_").

Variable assignment is done using the assignment operator ("="). **Note that no spaces are allowed on either side of the "=" when assigning a value to a variable.**

```
For the above Welsh town example:
town='Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoc'
```

3.3 Referencing Variables

The "$" operator is used to reference a variable.

```
town='Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoc'
echo "There is a Welsh town called $town"
```

"There is a Welsh town called Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoc" will be printed. You can also use curly braces around the variable name for clarity. The following will print the same thing as the previous:

```
town='Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoc'
echo "There is a Welsh town called ${town}"
```

3.3.1 Escape Characters

Some characters require that you "escape" them, meaning you add a backslash ("\") before them, so that they don't perform their special function.

For example, the $ symbol is used to reference variables, but what if you wanted to print the price of an object? Let's look at an example:

```
PRICE_PER_APPLE=5
echo "A single apple costs \$ $PRICE_PER_APPLE"
echo "A single apple costs $ $PRICE_PER_APPLE"
```

By backslashing the first $, the statement will print as it should: "A single apple costs $ 5". The second statement, without the backslash, prints "A single apple costs 4167PRICE_PER_APPLE", for me, and will print something similar for you. This is because the computer looks for the variable "$$", which is a special variable yielding the shell's process ID.

3.4 Quotations

There are two kinds of quotations in Bash: single quotes (' '), and double quotes (" "). Single quotes interpret things literally, whereas double quotes allow for $ notation. Double quotes also preserve white space.

Note: there are also backticks, which look deceptively similar to quotes, but differ greatly in function. Information input in backticks is read as a Bash command. The backtick key is found with the tilde symbol, next to the number 1 on your keyboard.

For example:

```
myname="Bob"
whitespace="Hello    world!"
echo "$whitespace" #This will print: Hello    world!
echo '$whitespace' #This will print: Hello world!
echo "Hello, $myname!" #This will print: Hello, Bob!
echo 'Hello, $myname!' #This will print: Hello, $myname!
echo "The date is `date`" #This will print the date
```

3.5 Exercises

1. Declare a variable called "roommates", and fill it with the number of roommates that you have. Then print to the screen "I have ___ roommate(s)."
2. Print to the screen: "The contents of my current directory are: ____" (using the ls command and backticks).

3.6 Further Reading

https://www.learnshell.org/en/Variables

http://guide.bash.academy/commands/

http://guide.bash.academy/expansions/

4 Passing Arguments

You may want to pass arguments to your script to be used inside, and in Bash, you can! This is analogous to passing arguments to methods in Java or functions in C.

4.1 Syntax

Arguments can be passed to the script by writing them as a space-delimited list following the name of the script being executed.

For example, let's say we have a script called "healthcare.sh" that takes "dentist", "therapist", and "doctor" as arguments. To execute:

```
./healthcare.sh dentist therapist doctor
```

4.2 Inside the Script

Inside the script, you reference the passed arguments using the syntax $number-of-argument. At the command line, variables are labelled with a number starting from zero. The script itself is the zeroth variable, so the first argument is $1, the second $2, the third $3, and so on.

```
./healthcare.sh dentist therapist doctor
Inside healthcare.sh:
echo "My $1 monitors my dental hygiene."
echo "My $2 monitors my mental health."
echo "My $3 monitors my physical health."
Prints to screen:
My dentist monitors my dental hygiene.
My therapist monitors my mental health.
My doctor monitors my physical health.
```

4.3 Special Variables

Bash also has certain special variables. They are as follows:

- $0 Name of current shell or script
- $n nth argument passed to script
- $# Number of arguments passed to script
- $@ Returns all arguments passed to script as separate strings
- $* Returns all arguments passed to script as single string

- $? Exit status of last command executed
- $$ Process ID of current shell or script
- $! Process number of last command in background

./healthcare.sh dentist therapist doctor

Special variables that print when I run it:

$0 = healthcare.sh

$# = 3

$@ = "dentist" "therapist" "doctor" (if entered in double quotes, like "$@")

$* = "dentist therapist doctor" (if entered in double quotes, like "$*")

$? = 0

$$ = 5582 (you won't get the same number)

$! = nothing printed, because nothing executed in the background

4.4 Exercises

1. Create a script called "sports.sh" that takes as arguments basketball, cricket, and rugby, and prints "I am most interested in ___, second-most interested in ___, and least interested in ___."

2. Create a script called "family.sh" that takes as arguments the first names of your family members (immediate, extended, kith, whatever you want!). Print the number of family members, and all of the names using only one command.

4.5 Further Reading

https://www.learnshell.org/en/Passing_Arguments_to_the_Script

https://unix.stackexchange.com/questions/218270/which-are-bash-shell-special-parameters

https://www.learnshell.org/en/Special_Variables

5 Basic Operators

Bash can handle the simple arithmetic operations of addition, subtraction, multiplication, integer division, exponentiation, and modulo.

5.1 Syntax

Syntax: $((expression))

5.2 Examples

Addition: $((10+24))

Subtraction: $((33-16))

Multiplication: $((15*15))

Integer Division: $((28/7))

Exponentiation: $((10**3))

Modulo: $((10%3))

Note: you can have more than two operands within the brackets. For example: $((5+(4-3)*2)) is acceptable. Pay attention to the order of operations!

5.3 Exercises

1. You have a shopping cart containing three bananas, a dozen eggs, and a rotisserie chicken. Each banana costs $1, a dozen eggs is $3, and the rotisserie chicken is $7. Calculate how much your groceries will cost.

2. If you get 150 calories from a quarter of the chicken, 89 from one banana, and 155 per egg, how many calories are in your grocery cart? How many do you eat for breakfast the next morning, when you have two bananas and three eggs?

5.4 Further Reading

https://www.learnshell.org/en/Basic_Operators

6 Basic String Operators

You can perform some basic operations on strings in Bash.

6.1 String Length

${#*STRING*}

Note: Spaces are included in the calculation of string length.

Example:

names="Sally and John"

echo ${#names} #prints "14"

6.2 Substring Extraction

${*STRING*:*$POS*:*$LEN*}

This extracts a substring from STRING of length $LEN, starting at position $POS. If $LEN is removed, a substring is extracted from $POS to the end of the string.

Note: the $LEN variable is not the index to which you extract, but the length of the extraction. So ${STRING:3:5} does not extract from 3 to 5, but from 3 to 8.

Example:

string="This is a string."

echo ${string:1} #prints the entire string without the first character

echo ${string:5:2} #prints "is"

6.3 Substring Replacement

To replace the first instance of a substring:

${*STRING*[@]/*substring*/*replacement*}

To replace all instances, use two slashes before the substring:

${*STRING*[@]//*substring*/*replacement*}

Example:

nursery_rhyme="And on that farm there was a cow."

echo $nursery_rhyme

echo ${nursery_rhyme[@]/cow/horse}

This will print:

And on that farm there was a cow.

And on that farm there was a horse.

There are other substring replacements that can be done beyond these basic ones. You can read about them at the link in Further Reading.

6.4 Exercises

1. Print the number of characters in your full name.

2. Replace all instances of "woodchuck" in "How much wood would a woodchuck chuck if a woodchuck could chuck wood?" with a less tongue-tying word.

6.5 Further Reading

https://www.learnshell.org/en/Basic_String_Operations

7 Decision-Making

Bash supports logical decision making using the standard range of if statements, using numeric and string comparisons as conditions.

7.1 Comparisons

Bash allows for two types of comparisons: numeric (ex. 5 is less than 7) and string (ex. "Dave" is the same string as "Dave"), but uses different syntax for the two.

7.1.1 Numeric Comparisons

Comparison	Evaluates to true when
$a -lt $b	$a less than $b
$a -gt $b	$a greater than $b
$a -le $b	$a lesser than or equal to $b
$a -ge $b	$a greater than or equal to $b
$a -eq $b	$a equal to $b
$a -ne $b	$a not equal to $b

7.1.2 String Comparisons

Comparison	Evaluates to true when
"$a" = "$b"	$a is the same as $b
"$a" == "$b"	$a is the same as $b
"$a" != "$b"	$a is not the same as $b
-z "$a"	$a is an empty string

Note: when comparing strings, you must place the variables in quotes, and spaces are required around the operators (=, ==, !=).

7.1.3 Operations within Conditions

You can perform operations within your conditions, like simple arithmetic (covered in section 5) or logical comparisons (not covered in this document).

Example 1: Arithmetic

```
if [ $((10-3)) -eq 7 ]; then
    echo "Yay math!"
fi
```

Example 2: Logical Operators

```
name="Alex"
```

```
if [ "$name" == "Alex" || "$name" == "Alexandra" ]; then
    echo "Hi, Alex!"
fi
```

7.2 if Statements

7.2.1 Syntax

if [***expression*** **]; then**
    #statements to execute if expression true
fi

Some things to note about the above syntax:

- There must be one space between each of the square brackets containing the condition expression and the expression itself
- The spacing between the closing square bracket and the semi-colon does not matter, nor does the spacing between the semi-colon and "then"
- As in other languages, the text contained in the if block should be indented
- You signal the end of the if statement using "fi" (which is "if" backwards)

7.2.2 Examples

1. A numeric comparison with an if statement.

Inside "numeric-bedtime.sh":

```
time=11
if [ $time -ge 11 ]; then
    echo "It is my bedtime!"
fi
```

This will print "It is my bedtime!"

2. A string comparison with an if statement.

Inside "string-bedtime.sh":

```
time="11 PM"
if [ $time == "11 PM" ]; then
    echo "It is my bedtime!"
fi
```

This will print "It is my bedtime!"

7.3 if/else Statements

7.3.1 Syntax

```
if [ expression ]; then
    #statements to execute if expression true
else
    #statements to execute if condition false
 fi
```

Note: the "else" statement does not have the same "; then" aspect as the "if" statement.

7.3.2 Example

Inside "wakeup.sh"

```
time="to get up"
if [ "$time" == "to get up" ]; then
    echo "Get up!"
else
    echo "You can sleep for a little longer."
fi
```

7.4 if/else if/else Statements

7.4.1 Syntax

```
if [ expression1 ]; then
    #statements to execute if expression1 true
elif [ expression2 ]; then
    #statements to execute if expression2 true
else
    #statements to execute if both conditions false
fi
```

Note: Bash uses "elif" instead of "else if", and you must provide a condition using the same syntax as for the "if" part of the statement.

7.4.2 Example

Inside "get-to-class.sh"

```
minutes-until-class=0
if [ $minutes-until-class -ge 60 ]; then
    echo "You have plenty of time."
elif [ $minutes-until-class -ge 15 ]; then
    echo "Time to go."
else
    echo "Hurry!"        #this prints
fi
```

7.5 Switch Statement

You can also set up a switch statement for decision-making, which involves setting up different cases that a variable may fulfill, one of which will execute.

7.5.1 Syntax

```
case "$variable" in
    "$condition1") commands... ;;
    "$condition2") commands... ;;
esac
```

Notes:

- Conditions are separated from commands by a single closing parenthesis – ")"
- Commands are followed by *two* semi-colons.
- If no condition is met, nothing will happen

An example will help clarify the way that these statements work.

7.5.2 Example

Inside "letter-grade.sh":

```
grade=85          #declare variable called "grade"
case $grade in
     85) echo "An 85% is an A";; #this will print because the value stored in the $grade
                                 variable matches this case
     80) echo "An 80% is an A-";;
     75) echo "A 75% is a B+";;
     70) echo "A 70% is a B";;
    ...etc.
esac
```

7.6 Exercise

Change the values of the variables so that all statements evaluate to "true":

apples=10

```
pears=12
name="Dane"
if [ $apples -le 9 ]; then
    echo "You have fewer than 10 apples."
fi
if [ "$name" == "Daniel" ]; then
    echo "Hello, Daniel!"
else
    echo "You didn't change the variable correctly.
fi
if [ $(($apples + $pears)) -ge 25 ]; then
    echo "You have 22 units of fruit."
fi
```

7.7 Further Reading

https://www.learnshell.org/en/Decision_Making

8 Arrays

Bash has arrays; hooray!

8.1 Initializing Arrays

Array naming follows the same conventions as variable naming.

An array is initialized by assigned space-delimited list within parentheses to the array variable.

$$\boxed{\textit{array\_name}=(\textit{val1 val2 val3 val4})}$$

Notes:

- No spaces are allowed on either side of the assignment operator

- You may leave some values uninitialized by putting a space where you would otherwise put the value
- Array size is not fixed
- The first index is 0, the second index is 1, etc.
- If a value contains spaces, it must be enclosed in single or double quotes (otherwise it will be interpreted as separate values).

8.2 Assigning Array Values

After an array has been initialized, you may replace values or add new values using the following syntax:

***array_name*[*index*]="*new_or_replacing_value*"**

Note: to assign values containing spaces, you must enclose the value in quotations (ex. "introduction to software systems"). You don't need quotations for single words, but you can include them.

8.3 Accessing Array Values

You can access an array value using the following syntax:

${*array_name*[*index*]}

Note: you must enclose the "array_name[index]" in curly braces to access the value at that index.

You can print all values of the array using the following syntax:

${*array_name*[@]}

And print the number of values in an array using this syntax:

${#*array_name*[@]}

Note: uninitialized blank spaces do not count toward the total number of values in an array.

8.4 Example

Inside "camping.sh":

```
supplies=(tent "sleeping bag" food toiletries stove)
echo "I'm going to sleep in a ${supplies[0]} and a ${supplies[1]}."
echo "We are bringing ${#supplies[@]} items with us on this trip."
echo "We are bringing ${supplies[@]}."
echo "Forgot something!"
supplies[5]="matches"
echo "Now we have ${supplies[@]} and are ready to camp!"
```

This will print:

```
I'm going to sleep in a tent and a sleeping bag.
We are bringing 5 items with us on this trip.
We are bringing tent sleeping bag food toiletries stove.
Forgot something!
Now we have tent sleeping bag food toiletries stove matches and are ready to camp!
```

8.5 Exercise

1. Make an array with everything you would bring to a picnic. Then print all values contained in the array and the number of things you're bringing. Add "windbreaker" (you never know when you'll need a windbreaker!) and print "I added a windbreaker" by accessing the index at which windbreaker is stored.

8.6 Further Reading

https://www.learnshell.org/en/Arrays

9 Loops

Bash allows for the use of for, while, and until loops. You can also affect loop flow using break and continue statements.

9.1 For Loops

9.1.1 Syntax

for ***arg*** **in** ***[list]*** **; do**
    commands...
done

9.1.2 Example

Inside "names.sh"

```
NAMES=(Anastasia Kevin Abdullah)
for N in $NAMES[@] ; do
    echo "Hello! My name is $N."
done
```

This will print out:

```
Hello! My name is Anastasia.
Hello! My name is Kevin.
Hello! My name is Abdullah.
```

9.2 While Loops

9.2.1 Syntax

while [***condition***]**; do**
    commands...
done

Note: there are spaces between the square brackets and the condition contained therein.

9.2.2 Example

Inside "fruits.sh":

```
fruits=(apple orange banana pineapple)
COUNT=0
current_fruit=${fruits[COUNT]}
while [ "$current_fruit" != "pineapple" ]; do
    echo "$current_fruit"
    COUNT=$((COUNT+1))
    current_fruit=${fruits[COUNT]}
done
```

This will print:

```
apple
orange
banana
```

9.3 Until Loops

9.3.1 Syntax

```
until [ condition ]; do
    commands...
done
```

Note: as with while loop syntax, there must be spaces between the condition and the square brackets containing it.

9.3.2 Example

Inside "count-to-five.sh"

```
COUNT=0
```

```
until [ $COUNT -gt 10 ]; do
    echo "Value of count: $COUNT"
    COUNT=$((COUNT+1))
done
```

This will print:

```
Value of count: 0
Value of count: 1
Value of count: 2
Value of count: 3
Value of count: 4
Value of count: 5
Value of count: 6
Value of count: 7
Value of count: 8
Value of count: 9
Value of count: 10
```

9.4 Break and Continue Statements

9.4.1 Break

A break statement is used to skip all remaining iterations, and move on to the code immediately following the loop. To use a break statement, insert the word "break" into your loop.

9.4.2 Break Example

Inside "counting.sh":

```
COUNT=0
while [ $COUNT -ge 0 ]; do
    echo "Value of count is: $COUNT"
    COUNT=$((COUNT+1))
```

```
    if [ $COUNT -ge 5 ]; then
        break
    fi
done
echo "Loop has been broken."
```

This will print:

```
Value of count is: 0
Value of count is: 1
Value of count is: 2
Value of count is: 3
Value of count is: 4
Loop has been broken.
```

9.4.3 Continue

A continue statement is used to skip the current iteration. The loop then executes its next iteration. To use a continue statement, insert the word "continue" into your loop.

9.4.4 Continue Example

Inside "counting.sh":

```
COUNT=0
while [ $COUNT -lt 10 ]; do
    if [ $COUNT -eq 5 ]; then
        echo "Skipping echo of 5."
        COUNT=$((COUNT+1))      #Make sure you don't create an infinite loop.
        continue
    fi
    echo "Value of count is: $COUNT"
    COUNT=$((COUNT+1))
done
```

This will print:

Value of count is: 0

Value of count is: 1

Value of count is: 2

Value of count is: 3

Value of count is: 4

Skipping echo of 5

Value of count is: 6

Value of count is: 7

Value of count is: 8

Value of count is: 9

9.5 Exercises

1. Print all even values from 1 to 20 using:
   a) a for loop.
   b) a while loop.
   c) an until loop.
2. Print from 1 to 10, excluding 6.
3. Create an array containing your first, middle, and last names, and print the values using for, while, and until loops.

9.6 Further Reading

https://www.learnshell.org/en/Loops