Introduction to Advanced Bash Scripting

Estimated time needed: **5** minutes

In the hands-on lab portion of the final project, you will be using more advanced scripting commands and concepts that the course has not covered yet. This reading will familiarize you with these more advanced concepts, so you can complete the lab with confidence.

Objectives

After completing this reading, you will be able to create Bash scripts that:

- use conditional statements to run a set of commands only if a specified condition is `true`
- apply logical operators to create `true`/`false` comparisons
- perform basic arithmetic calculations
- create list-like arrays and access their elements
- implement `for` loops to execute operations repeatedly, based on a looping index

Conditionals

Conditionals, or `if` statements, are a way of telling a script to do something only under a specific condition.

Bash script conditionals use the following `if-then-else` syntax:

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
```

```
1. if [ condition ]
2. then
3.     statement_block_1
4. else
5.     statement_block_2
6. fi
```

Copied!

If the `condition` is `true`, then Bash executes the statements in `statement_block_1` before exiting the conditional block of code. After exiting, it will continue to run any commands after the closing `fi`.

Alternatively, if the `condition` is `false`, Bash instead runs the statements in `statement_block_2` under the `else` line, then exits the conditional block and continues to run commands after the closing `fi`.

> **Tips:**
>
> - You must always put spaces around your condition within the square brackets `[ ]`.
> - Every `if` condition block must be paired with a `fi` to tell Bash where the condition block ends.

- The `else` block is optional but recommended. If the condition evaluates to `false` without an `else` block, then nothing happens within the `if` condition block. Consider options such as echoing a comment in `statement_block_2` to indicate that the condition was evaluated as `false`.

In the following example, the condition is checking whether the number of command-line arguments read by some Bash script, `$#`, is equal to 2.

```
1
2
3
4
5
6
```

```
if [[ $# == 2 ]]
then
  echo "number of arguments is equal to 2"
else
  echo "number of arguments is not equal to 2"
fi
```

Copied!

Notice the use of the double square brackets, which is the syntax required for making integer comparisons in the condition `[[ $# == 2 ]]`.

You can also make string comparisons. For example, assume you have a variable called `string_var` that has the value `"Yes"` assigned to it. Then the following statement evaluates to `true`:

```
1
```

```
`[ $string_var == "Yes" ]`
```

Copied!

Notice you only need single square brackets when making string comparisons.

You can also include multiple conditions to be satified by using the "and" operator `&&` or the "or" operator `||`. For example:

```
1
2
3
4
5
6
```

```
if [ condition1 ] && [ condition2 ]
then
    echo "conditions 1 and 2 are both true"
else
    echo "one or both conditions are false"
fi
```

Copied!

```
1
2
3
4
5
6
```

```
if [ condition1 ] || [ condition2 ]
then
```

```
3.     echo "conditions 1 or 2 are true"
4. else
5.     echo "both conditions are false"
6. fi
```

Copied!

Logical operators

The following logical operators can be used to compare integers within a condition in an `if` condition block.

`==`: is equal to

If a variable `a` has a value of 2, the following condition evaluates to `true`; otherwise it evalutes to `false`.

```
1. 1
```

```
1. $a == 2
```

Copied!

`!=`: is not equal to

If a variable `a` has a value different from 2, the following statement evaluates to `true`. If its value is 2, then it evalutes to `false`.

```
1. 1
```

```
1. a != 2
```

Copied!

> **Tip**: The `!` logical negation operator changes `true` to `false` and `false` to `true`.

`<=`: is less than or equal to

If a variable `a` has a value of 2, then the following statement evaluates to `true`:

```
1. 1
```

```
1. a <= 3
```

Copied!

and the following statement evalutates to `false`:

```
1. 1
```

```
1. a <= 1
```

Copied!

Alternatively, you can use the equivalent notation `-le` in place of `<=`:

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
```

```
1. a=1
```

```
2. b=2
3. if [ $a -le $b ]
4. then
5.    echo "a is less than or equal to b"
6. else
7.    echo "a is not less than or equal to b"
8. fi
```

Copied!

We've only provided a small sampling of logical operators here. You can explore resources such as the Advanced Bash-Scripting Guide to find out more.

Arithmetic calculations

You can perform integer addition, subtraction, multiplication, and division using the notation `$(())`. For example, the following two sets of commands both display the result of adding 3 and 2.

```
1. 1
```

```
1. echo $((3+2))
```

Copied!

or

```
1. 1
2. 2
3. 3
4. 4
```

```
1. a=3
2. b=2
3. c=$(($a+$b))
4. echo $c
```

Copied!

Bash natively handles integer arithmetic but does not handle floating-point arithmetic. As a result, it will always truncate the decimal portion of a calculation result.

For example:

```
1. 1
```

```
1. echo $((3/2))
```

Copied!

prints the truncated integer result, `1`, not the floating-point number, `1.5`.

The following table summarizes the basic arithmetic operators:

Symbol	Operation
`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division

Table: **Arithmetic operators**

Arrays

The **array** is a Bash built-in data structure. An array is a space-delimited list contained in parentheses.ʊTo create an array, declare its name and contents:

```
1
```

```
my_array=(1 2 "three" "four" 5)
```

Copied!

This statement creates and populates the array `my_array` with the items in the parentheses: `1`, `2`, `"three"`, `"four"`, and `5`.

You can also create an empty array by using:

```
1
```

```
declare -a empty_array
```

Copied!

If you want to add items to your array after creating it, you can add to your array by appending one element at a time:

```
1
2
```

```
my_array+=("six")
my_array+=(7)
```

Copied!

This adds elements `"six"` and `7` to the array `my_array`.

By using indexing, you can access individual or multiple elements of an array:

```
1
2
3
4
5
6
7
8
```

```
# print the first item of the array:
echo ${my_array[0]}

# print the third item of the array:
echo ${my_array[2]}

# print all array elements:
echo ${my_array[@]}
```

Copied!

> **Tip:** Note that array indexing starts from 0, not from 1.

`for` loops

You can use a construct called a `for` loop along with indexing to iterate over all elements of an array.

For example, the following `for` loops will continue to run over and over again until every element is printed:

```
1
2
3
```

```
for item in ${my_array[@]}; do
  echo $item
done
```

Copied!

or

```
1
2
3
```

```
for i in ${!my_array[@]}; do
  echo ${my_array[$i]}
done
```

Copied!

The `for` loop requires a `; do` component in order to cycle through the loop. Additionally, you need to terminate the `for` loop block with a `done` statement.

Another way to implement a `for` loop when you know how many iterations you want is as follows. For example, the following code prints the number 0 through 6.

```
1
2
3
4
```

```
N=6
for (( i=0; i<=$N; i++ )) ; do
  echo $i
done
```

Copied!

You can use `for` loops to accomplish all sorts of things. For example, you could count the number of items in an array or sum up its elements, as the following Bash script does:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

```
#!/usr/bin/env bash
# initialize array, count, and sum
my_array=(1 2 3)
count=0
sum=0
for i in ${!my_array[@]}; do
```

```
  # print the ith array element
  echo ${my_array[$i]}
  # increment the count by one
  count=$(($count+1))
  # add the current value of the array to the sum
  sum=$(($sum+${my_array[$i]}))
done
echo $count
echo $sum
```

Copied!

Go ahead and try running this script, so you get a sense of how this loop works.

Summary

In this lab, you learned that:

- Conditional statements can be used to run commands based on whether a specified condition is `true`
- Logical operators do `true`/`false` comparisons
- Arithmetic operators perform basic arithmetic calculations
- You can create list-like arrays and access their individual elements
- `for` loops execute operations repeatedly, based on a looping index

Congratulations! You are now ready to practice your newly acquired knowledge in the following hands-on lab.

Authors

Jeff Grossman
Sam Prokopchuk

Other Contributors

Rav Ahuja

Change Log

Date (YYYY-MM-DD)	Version	Changed By	Change Description
2023-12-11	1.9	K Sundararajan	Update in a code under `Conditionals` section for accuracy
2023-08-01	1.8	K Sundararajan	Updated the `shebang` line in the last code
2023-05-17	1.7	Jeff Grossman	Add `&&`, `||` conditional logic
2023-05-17	1.6	Jeff Grossman	Add for loop w/ known #iterations
2023-05-12	1.5	Nick Yi	Review and edits
2023-05-10	1.4	Jeff Grossman	Add table of arithmetic operators
2023-05-10	1.3	Jeff Grossman	Add string comparison
2023-03-23	1.2	Nick Yi	ID review
2023-03-17	1.1	Jeff Grossman	edit and expand content
2022-04-05	1.0	Sam Prokopchuk	create inital reading