8 examples of Bash if statements to get you started

This article shows examples of how to use BASH if statements in scripts and command line. This is a good starter tutorial for those who haven't quite mastered BASH yet.

Written by Benjamin Cane (https://twitter.com/madflojo) on 2014-01-27 08:00:00 | *5 min read*

Shell scripting is a fundamental skill that every systems administrator should know. The ability to script mundane & repeatable tasks allows a sysadmin to perform these tasks quickly. These scripts can be used for anything from installing software, configuring software or quickly resolving a known issue.

A fundamental core of any programming language is the `if` statement. In this article I am going to show several examples of using `if` statements and explain how they work.

If value equals 1

The first example is one of the most basic examples, if true.

```
if [ $value -eq 1 ]
then
  echo "has value"
fi
```

Now this alone doesn't seem all that amazing but when you combine it with other commands, like for example checking to see if a username exists in the passwd file.

```
value=$( grep -ic "benjamin" /etc/passwd )
if [ $value -eq 1 ]
then
  echo "I found Benjamin"
fi
```

The `grep -ic` command tells grep to look for the string and be case `i` nsensitive, and to `c` ount the results. This is a simple and fast way of checking whether a string exists within a file and if it does perform some action.

Adding the else

The above `if` statement works great for checking if the user exists but what happens when the user doesn't exist? Right now, nothing but we can fix that.

```
value=$( grep -ic "benjamin" /etc/passwd )
if [ $value -eq 1 ]
then
  echo "I found Benjamin"
else
  echo "I didn't find Benjamin"
fi
```

The `else` statement is part of an `if` statement, its actions are only performed when the `if` statements comparison operators are not true. This is great for performing a check and executing a specific command if the test is true, and a different command if the test is false.

Checking if value is greater or less than

In the previous two examples you can see the use of the `-eq` equals operator, in this example I am going to show the `-gt` greater than and `-lt` less than operators.

First let us start with the greater than operator.

```
value=$( grep -ic "benjamin" /etc/passwd )
if [ $value -gt 5 ]
then
  echo "I found a lot of Benjamins..."
fi
```

Second we will use the less than operator.

```
value=$( grep -ic "benjamin" /etc/passwd )
if [ $value -lt 5 ]
then
  echo "I found only a few Benjamins..."
fi
```

Using else if

While it would be easy enough to simply add an else to either the less than or greater than examples to handle conditions where I found more or less "Benjamins" than the if statement is looking for. I can also use the `elif` statement to perform an additional `if` statement if the first one wasn't found to be true.

```
value=$( grep -ic "benjamin" /etc/passwd )
if [ $value -eq 1 ]
then
  echo "I found one Benjamin"
elif [ $value -gt 1 ]
then
  echo "I found multiple Benjamins"
else
  echo "I didn't find any Benjamins"
fi
```

The order of this if statement is extremely important, you will notice that I first check if the value is specifically 1. If the value is not specifically 1 I then check if the value is greater than 1, if it isn't 1 or greater then 1 I simply tell you that I didn't find any Benjamins. This is using the `elif` or else if statement.

Nested if statements

A nested `if` statement is where you have an `if` statement inside of an existing `if` statement.

```
value=$( grep -ic "benjamin" /etc/passwd )
if [ $value -ge 1 ]
then

  if [ $value -eq 1 ]
  then
    echo "I found Benjamin"
  elif [ $value -eq 2 ]
  then
    echo "I found two Benjamins"
  else
    echo "There are too many Benjamins"
  fi

fi
```

Let us break down how the above statements work together. First we execute the `grep` and send its value to the `value` variable. Our `if` statement will check if the value of the `value` variable is `-ge` greater than or equal to 1. If it is than we will execute the second `if` statement and see if it is either equal to 1, and if not equal to exactly 1 if it is equal to exactly 2. If it is not 1 or 2 it must be much larger so at this point it will give up checking and say "There are too many Benjamins".

Checking if a string value is set

The above examples show some good examples of using integer based operators. If you are wondering what the heck an operator is, than let me explain. The `-eq` in the statement is an operator or in simpler terms a comparison, it is used to tell bash an operation to perform to find true or false. An if statement is always going to be true or false, either the operator is correct and it is true, or the operator is incorrect and it is false.

There are a ton of operators in bash that will take more than 8 examples to get you through but for now, the examples in today's article should get you started.

In this example I am going to show how you can check if a variable has a string value.

```
if [ -n $value ]
then
  echo "variable value has a value or $value"
fi
```

The `-n` operator is for checking if a variable has a string value or not. It is true if the variable has a string set. This is a great way to test if a bash script was given arguments or not, as a bash scripts arguments are placed into variables `$1`, `$2`, `$3` and so on automatically.

Usually though in a bash script you want to check if the argument is empty rather than if it is not empty, to do this you can use the `-z` operator.

```
if [ -z $1 ]
then
  echo "sorry you didn't give me a value"
  exit 2
fi
```

If value is not true

The `-z` operator is the opposite of `-n`, you could get the same results by performing this `if` statement with the `!` NOT operator. When a `!` operator is added to an `if` statement it takes the existing operator and inverts the statement. So a `! -lt` turns into "not less than".

```
if [ ! -n $1 ]
then
  echo "sorry you didn't give me a value"
elif [ ! -z $1 ]
then
  echo "hey thanks for giving me a value"
fi
```

The "not" operator can be extremely useful, to be honest I didn't even know the `-n` operator existed until writing this article. Usually whenever I wanted to see if a string was not empty I would simply use `! -z` to test it. Considering the `!` operator can be used in any `if` statements test, this can be a huge time saver sometimes.

Using AND & OR in if statements

The final `if` statement example that I am going to show is using `&&` AND & `||` OR in an `if` statement. Let's say you have a situation where you are accepting an argument from a bash script and you need to not only check if the value of this argument is set but also if the value is valid.

That's where AND comes in.

```
if [[ -n $1 ]] && [[ -r $1 ]]
then
  echo "File exists and is readable"
fi
```

The above `if` statement will check if the `$1` variable is not empty, and if it is not empty than it will also check if the file provided in `$1` is readable or not. If it is, then it will echo "File exists and is readable", if either of these two tests are false. Than the `if` statement will not be executed.

Using our bash script example, many times you are going to want to check if either `$1` has no value or the file is unreadable in order to alert the user and exit the script. For this we will use the OR statement.

```
if [[ -z $1 ]] || [[ ! -r $1 ]]
then
  echo "Either you didn't give me a value or file is unreadable"
  exit 2
fi
```

In the above example we want to exit the script if either of these conditions are true, that is exactly what OR provides for us. If either one of these conditions is true the `if` statement will cause the script to exit.

You may notice in the above `if` statements I am using a double brackets rather than single brackets. When using `&&` or `||` it is generally a good idea to use the double brackets as this opens up some additional functionality in bash. You will also find in older implementations of bash a single bracket being used with `&&` can cause some syntax issues. Though this seems to have been remediated in newer implementations, it is always a good idea to assume the worst case and write your scripts to handle older bash implementations. You never know where you might find yourself running a script on a system that hasn't been updated in a while.

The above should get you started on writing `if` statements in bash, I am have barely touched on many of the cool ways you can perform tests with bash. I am sure some of you readers may have some that you want to share, if you do drop by the comments and share away.