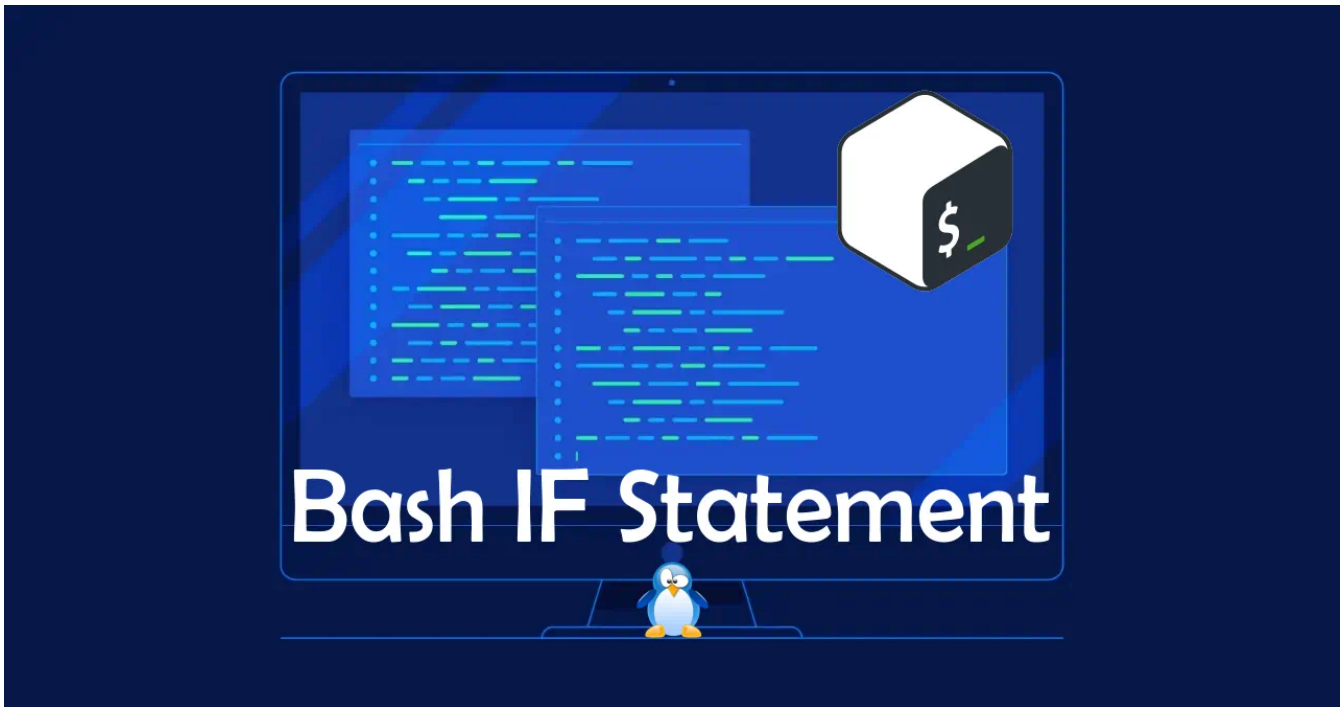
MYBLUELINUX.COM

WEB SITE FOR LINUX FANS AND PROGRAMMERS

Bash: if statement

2021-01-11 Bash, Programming



A **Shell script** usually needs to test *if* a command succeeds or a condition is met. In **Bash**, this test can be done with a Bash *if statement*. As with any other programming language, Bash comes with conditional expressions that allow you to test for conditions and alter the control flow if the condition is satisfied or not.

This post covers the bash *if* statement and the related clauses *then*, *else if* (*elif*), and *else*.

Introduction to the Bash If Statement

In Bash, the *if statement* is part of the conditional constructs of the programming language. The *if* in a Bash script is a shell keyword that is used to test conditions based on the exit status of a test command.

> An **exit status of zero, and only zero, is a success**, i.e. a condition that is **true**. **Any other exit status is a failure**, i.e. a condition that is **false**.

The syntax of the *if* statement in Bash is:

```
if first-test-commands; then
    consequent-commands;
[elif more-test-commands; then
    more-consequents;]
[else alternate-consequents;]
fi
```

What are the double Parentheses ((...)), single [...], and double [[..]] Square Brackets?

The `((...))`, `[...]`, and `[[...]]` constructs are often used to evaluate complex conditional expressions with comparison operators, and to return an exit status of `0` or `1` that can be used in a bash **if statement**.

The double parentheses `((...))` is used to test an arithmetic expression. You can read more about this construct in our post on bash arithmeticicon. It does support the `&&` and `||` binary operators.

The single square brackets `[...]` is the command `[` which is a shell builtin and an alias to the test command. The test command and the alias `[` are used to evaluate conditional expressions. This is part of the POSIX standard.

There are some notable differences between the double brackets and single bracket notation:

- double brackets perform pattern matching where the right-hand side can be a glob pattern. Note that globbing will not work if the right-hand string is quoted

- double brackets notation prevent word splitting, hence you can omit quotes around string variables.
- double brackets notation does not expand filenames.
- double brackets notation support the `&&` and `||` binary operators.
- double brackets notation support regex pattern matching when using the `=~` notation.

The double square brackets `[[...]]` is a shell keyword. It is similar in behavior to the single square bracket and is used to evaluate conditional expressions and is a Bash, Zsh, and Korn shell specific. This construct can handle more complex conditions and is less error-prone, see the FAQ on some examples of incorrect use of the single bracket command.

Note that the `((...))` and `[[...]]` constructs are Bash compound commands.

What are the Bash Conditional Expressions?

Conditional expressions are used by the `[[` compound command and the `test` and `[` builtin commands. Conditional Expressions can be unary (one operand) or binary (two operands). Unary operators are often used to test the status of a file, a variable, a shell option (optname), or a string.

Unary and Binary expressions are formed with the following *primaries*.

Conditional Expression	Meaning
-a `file`	True if `file` exists.
-b `file`	True if `file` exists and is a block special `file` .
-c `file`	True if `file` exists and is a character special `file` .
-d `file`	True if `file` exists and is a directory.
-e `file`	True if `file` exists.
-f `file`	True if `file` exists and is gular `file` .

Conditional Expression	Meaning
-g `file`	True if `file` exists and its set-group-id bit is set.
-h `file`	True if `file` exists and is a symbolic link.
-k `file`	True if `file` exists and its “sticky” bit is set.
-p `file`	True if `file` exists and is a named pipe (FIFO).
-r `file`	True if `file` exists and is readable.
-s `file`	True if `file` exists and has a size greater than zero.
-t `fd`	True if `file` descriptor `fd` is open and refers to a terminal.
-u `file`	True if `file` exists and its set-user-id bit is set.
-w `file`	True if `file` exists and is writable.
-x `file`	True if `file` exists and is executable.
-G `file`	True if `file` exists and is owned by the effective group id.
-L `file`	True if `file` exists and is a symbolic link.
-N `file`	True if `file` exists and has been modified since it was last read.
-O `file`	True if `file` exists and is owned by the effective user id.
-S `file`	True if `file` exists and is a socket.
`file1` -ef `file2`	True if `file1` and `file2` refer to the same device and inode numbers.
`file1` -nt `file2`	True if `file1` is newer (according to modification date) than `file2`, or if `file1` exists and `file2` does not.
`file1` -ot `file2`	True if `file1` is older than `file2`, or if `file2` exists and `file1` does not.

Conditional Expression	Meaning
-o `optname`	True if the shell option `optname` is enabled (see `set -o` for a list of options).
-v `varname`	True if the shell variable `varname` is set (has been assigned a value, even an empty value).
-R `varname`	True if the shell variable `varname` is set and is a name reference.
-z `string`	True if the length of `string` is zero.
-n `string`	True if the length of `string` is non-zero.
`string1` == `string2`	True if the strings are equal. It will perform pattern matching when used with the `[[` command. The `=` notation should be used with the test command for POSIX conformance.
`string1 = string2`	(same as `string1 == string2`).
`string1` != `string2`	True if the strings are not equal.
`string1` =~ regex	True if the strings match the Bash regular expression regex. Captured groups are stored in the BASH_REMATCH array variable.
`string1 < string2`	True if `string1` sorts before `string2` lexicographically.
`string1 > string2`	True if `string1` sorts after `string2` lexicographically.

The Conditional Expressions also support arithmetic binary operators as follows and where `arg1` and `arg2` are either positive or negative integers. When used with the `[[` command, `arg1` and `arg2` are evaluated as arithmetic expressions, hence the `((` compound command should be preferred.

Conditional Expression	Meaning
`arg1` -eq `arg2`	True if `arg1` equal to `arg2`
`arg1` -ne `arg2`	True if `arg1` not equal to `arg2`
`arg1` -lt `arg2`	True if `arg1` less than `arg2`
`arg1` -le `arg2`	True if `arg1` less than or equal to `arg2`
`arg1` -gt `arg2`	True if `arg1` greater than `arg2`
`arg1` -ge `arg2`	True if `arg1` greater than or equal to `arg2`

How to use an If Statement with Then, Else, Else If (elif) clauses

As we mentioned earlier, a **If Statement** must have a **then** clause and optionally can have an **else if** clause with the keyword **elif** followed by **then**, and/or an **else** clause. The **If Statement** always ends with the **fi** keyword.

The **if**, **then**, **else**, **elif** and **fi** keywords must be the last keyword of a line or they need to be terminated with a semi-colon **;** before any other keyword is being used.

```
if false; then
  echo 'This command will never run since condition is always false'
elif ((RANDOM%2)); then
  echo 'This command will execute only when $RANDOM % 2 equal to 0'
else
  echo 'This command will execute if no other condition is met.';
fi
```

Using a Bash If Statement with Conditional Expressions

Note that a condition doesn't need any special enclosing characters like parentheses, though they may be used to override the precedence of other operators. Depending on the test to be performed, a command can be used directly, or the use of the `[[` compound command, or the `test` and `[` builtin commands. The `((` compound

command is reserved for Arithmetic Expansion. Blank spaces between keywords and commands matters.

For example, if we want to test whether a file exists and is a regular file (not a symlink), we could use the `-f` primary with any of the following notation.

```
#!/bin/bash

touch myfile

if [[ -f myfile ]]; then
    echo "myfile exists. If Statement Condition equal $?.";
fi
# output:
echo "myfile exists. If Statement Condition equal $?.";

if [ -f myfile ]; then
    echo "myfile exists. If Statement Condition equal $?.";
fi
# output:
echo "myfile exists. If Statement Condition equal $?.";

if test -f myfile; then
    echo "myfile exists. If Statement Condition equal $?.";
fi
# output
echo "myfile exists. If Statement Condition equal $?.";
```

You can negate a condition using the `!` keyword.

```
#!/bin/bash

touch myfile
rm myfile

if [[ ! -f myfile ]]; then
    echo "myfile does not exist. If Statement Condition equal $?."
fi

# output:
myfile does not exist. If Statement Condition equal 0.
```

Using a Bash If Statement with multiple conditions

As we mentioned above, you can use the binary operators `&&` (**and**) and `||` (**or**) in the double brackets `[[` compound notation. This is similar to using the `-a` (**and**) and `-o` (**or**) in a single bracket `[` .

```
#!/bin/bash

touch myfile

if [[ -f myfile &&  -r myfile ]]; then
echo "File exists and is Readable.";
fi
# output:
File exists and is Readable.

if [ -f myfile -a  -r myfile ]; then
     echo "File exists and is Readable.";
fi
# output:
File exists and is Readable.
```

Note that if you use the binary operators in a single bracket notation you will end up with an error `bash: [: missing ``]'` . This is because `[` is a command and expect `]` as the last argument. Similarly, when using `test` , the command would fail with `bash: -r: command not found` as `&&` terminate the previous command and expect a new command right after. When using `&&` or `||` with single brackets, you will need to use them outside of the brackets or test command.

```
#!/bin/bash

touch myfile

if [ -f myfile &&  -r myfile ]; then
    echo "File exists and is Readable.";
fi

if [ -f myfile ] && [ -r myfile ]; then
    echo "File exists and is Readable.";
fi
```

```
if test -f myfile &&  -r myfile; then
    echo "File exists and is Readable.";
fi

if test -f myfile && test -r myfile; then
    echo "File exists and is Readable.";
fi

# output:
./if.sh: line 5: [: missing `]'
File exists and is Readable.
./if.sh: line 13: -r: command not found
File exists and is Readable.
```

Using Nested If Statements

A `nested if statement` is an `if statement` inside a clause of another `if statement`. Nothing prevents multiple levels of `if statement` in a shell script and in Bash.

```
if first-test-commands; then
  if second-level-test-commands; then
    consequent-commands;
  else second-level-alternate-consequents;
[elif more-test-commands; then
  more-consequents;]
[else alternate-consequents;]
fi
```


BASH

« PREVIOUS
Bash: Guide to Bash IFS variable

NEXT »
GNU Regex POSIX character classes