

ESSENTIAL COMMANDS

Covers  
Fedora Linux

# LINUX

POCKET GUIDE



O'REILLY®

DANIEL J. BARRETT

# Programming with Shell Scripts

Earlier when we covered the shell (bash), we said it had a programming language built in. In fact, you can write programs, or *shell scripts*, to accomplish tasks that a single command cannot. Like any good programming language, the shell has variables, conditionals (if-then-else), loops, input and output, and more. Entire books have been written on shell scripting, so we'll be covering the bare minimum to get you started. For full documentation, run `info bash`.

## Whitespace and Linebreaks

bash shell scripts are very sensitive to whitespace and linebreaks. Because the “keywords” of this programming language are actually commands evaluated by the shell, you need to separate arguments with whitespace. Likewise, a linebreak in the middle of a command will mislead the shell into thinking the command is incomplete. Follow the conventions we present here and you should be fine.

## Variables

We described variables earlier:

```
$ MYVAR=6
$ echo $MYVAR
6
```

All values held in variables are strings, but if they are numeric the shell will treat them as numbers when appropriate.

```
$ NUMBER="10"
$ expr $NUMBER + 5
15
```

When you refer to a variable's value in a shell script, it's a good idea to surround it with double quotes to prevent certain runtime errors. An undefined variable, or a variable with spaces in its value, will evaluate to something unexpected if not surrounded by quotes, causing your script to malfunction.

```
$ FILENAME="My Document"           Space in the name
$ ls $FILENAME                       Try to list it
```

```
ls: My: No such file or directory  Oops! ls saw 2 arguments
ls: Document: No such file or directory
$ ls -l "$FILENAME"                List it properly
My Document                        ls saw only 1 argument
```

If a variable name is evaluated adjacent to another string, surround it with curly braces to prevent unexpected behavior:

```
$ HAT="fedora"
$ echo "The plural of $HAT is $HATs"
The plural of fedora is                Oops! No variable "HATs"
$ echo "The plural of $HAT is ${HAT}s"
The plural of fedora is fedoras        What we wanted
```

## Input and Output

Script output is provided by the `echo` and `printf` commands, which we described in “Screen Output” on page 144:

```
$ echo "Hello world"
Hello world
$ printf "I am %d years old\n" `expr 20 + 20`
I am 40 years old
```

Input is provided by the `read` command, which reads one line from standard input and stores it in a variable:

```
$ read name
Sandy Smith <ENTER>
$ echo "I read the name $name"
I read the name Sandy Smith
```

## Booleans and Return Codes

Before we can describe conditionals and loops, we need the concept of a Boolean (true/false) test. To the shell, the value 0 means true or success, and anything else means false or failure.

Additionally, every Linux command returns an integer value, called a *return code* or *exit status*, to the shell when the command exits. You can see this value in the special variable `?`:

```
$ cat myfile
My name is Sandy Smith and
```

```

I really like Fedora Linux
$ grep Smith myfile
My name is Sandy Smith and
$ echo $?
0
$ grep aardvark myfile
$ echo $?
1

```

*A match was found...*

*...so return code is "success"*

*No match was found...*

*...so return code is "failure"*

The return codes of a command are usually documented on its manpage.

## test and “[”

The test command (built into the shell) will evaluate simple Boolean expressions involving numbers and strings, setting its exit status to 0 (true) or 1 (false):

```

$ test 10 -lt 5
$ echo $?
1
$ test -n "hello"
$ echo $?
0

```

*Is 10 less than 5?*

*No, it isn't*

*Does the string "hello" have nonzero length?*

*Yes, it does*

A list of common test arguments are found in Table 12, for checking properties of integers, strings, and files.

test has an unusual alias, “[” (left square bracket), as a shorthand for use with conditionals and loops. If you use this shorthand, you must supply a final argument of “]” (right square bracket) to signify the end of the test. The following tests are identical to those before:

```

$ [ 10 -lt 5 ]
$ echo $?
1
$ [ -n "hello" ]
$ echo $?
0

```

Remember that “[” is a command like any other, so it is followed by *individual arguments separated by whitespace*. So if you mistakenly forget some whitespace:

```

$ [ 5 -lt 4 ]
bash: [: missing ']'

```

*No space between 4 and ]*

then test thinks the final argument is the string “4]” and complains that the final bracket is missing.

Table 12. Some common arguments for the test command

---

<b>File tests</b>	
-d <i>name</i>	File <i>name</i> is a directory
-f <i>name</i>	File <i>name</i> is a regular file
-L <i>name</i>	File <i>name</i> is a symbolic link
-r <i>name</i>	File <i>name</i> exists and is readable
-w <i>name</i>	File <i>name</i> exists and is writable
-x <i>name</i>	File <i>name</i> exists and is executable
-s <i>name</i>	File <i>name</i> exists and its size is nonzero
<i>f1</i> -nt <i>f2</i>	File <i>f1</i> is newer than file <i>f2</i>
<i>f1</i> -ot <i>f2</i>	File <i>f1</i> is older than file <i>f2</i>

---

<b>String tests</b>	
<i>s1</i> = <i>s2</i>	String <i>s1</i> equals string <i>s2</i>
<i>s1</i> != <i>s2</i>	String <i>s1</i> does not equal string <i>s2</i>
-z <i>s1</i>	String <i>s1</i> has zero length
-n <i>s1</i>	String <i>s1</i> has nonzero length

---

<b>Numeric tests</b>	
<i>a</i> -eq <i>b</i>	Integers <i>a</i> and <i>b</i> are equal
<i>a</i> -ne <i>b</i>	Integers <i>a</i> and <i>b</i> are not equal
<i>a</i> -gt <i>b</i>	Integer <i>a</i> is greater than integer <i>b</i>
<i>a</i> -ge <i>b</i>	Integer <i>a</i> is greater than or equal to integer <i>b</i>
<i>a</i> -lt <i>b</i>	Integer <i>a</i> is less than integer <i>b</i>
<i>a</i> -le <i>b</i>	Integer <i>a</i> is less than or equal to integer <i>b</i>

---

<b>Combining and negating tests</b>	
t1 -a t2	And: Both tests t1 and t2 are true
t1 -o t2	Or: Either test t1 or t2 is true
! <i>your_test</i>	Negate the test, i.e., <i>your_test</i> is false
\( <i>your_test</i> \)	Parentheses are used for grouping, as in algebra

---

## true and false

bash has built-in commands `true` and `false`, which simply set their exit status to 0 and 1, respectively.

```
$ true
$ echo $?
0
$ false
$ echo $?
1
```

These will be useful when we discuss conditionals and loops.

## Conditionals

The `if` statement chooses between alternatives, each of which may have a complex test. The simplest form is the `if-then` statement:

```
if command                If exit status of command is 0
then
    body
fi
```

For example:

```
if [ `whoami` = "root" ]
then
    echo "You are the superuser"
fi
```

Next is the `if-then-else` statement:

```
if command
then
    body1
else
    body2
fi
```

For example:

```
if [ `whoami` = "root" ]
then
    echo "You are the superuser"
else
    echo "You are an ordinary dude"
fi
```

Finally, we have the form `if-then-elif-else`, which may have as many tests as you like:

```
if command1
then
    body1
elif command2
then
    body2
elif ...
...
else
    bodyN
fi
```

For example:

```
if [ `whoami` = "root" ]
then
    echo "You are the superuser"
elif [ "$USER" = "root" ]
then
    echo "You might be the superuser"
elif [ "$bribe" -gt 10000 ]
then
    echo "You can pay to be the superuser"
else
    echo "You are still an ordinary dude"
fi
```

The case statement evaluates a single value and branches to an appropriate piece of code:

```
echo 'What would you like to do?'
read answer
case "$answer" in
    eat)
        echo "OK, have a hamburger"
        ;;
    sleep)
        echo "Good night then"
        ;;
    *)
        echo "I'm not sure what you want to do"
        echo "I guess I'll see you tomorrow"
        ;;
esac
```

The general form is:

```
case string in
  expr1)
    body1
    ;;
  expr2)
    body2
    ;;
  ...
  exprN)
    bodyN
    ;;
*)
  bodyelse
  ;;
esac
```

where *string* is any value, usually a variable value like `$myvar`, and *expr1* through *exprN* are patterns (run the command `info bash reserved case` for details), with the final `*` like a final “else.” Each set of commands must be terminated by `;;` (as shown):

```
case $letter in
  X)
    echo "$letter is an X"
    ;;
  [aeiou])
    echo "$letter is a vowel"
    ;;
  [0-9])
    echo "$letter is a digit, silly"
    ;;
  *)
    echo "I cannot handle that"
    ;;
esac
```

## Loops

The while loop repeats a set of commands as long as a condition is true.

```
while command           While the exit status of command is 0
do
  body
```

```
done
```

For example, if this is the script `myscript`:

```
i=0
while [ $i -lt 3 ]
do
    echo "again"
    i=`expr $i + 1`
done
```

```
$ ./myscript
0
1
2
```

The `until` loop repeats until a condition becomes true:

```
until command           While the exit status of command is nonzero
do
    body
done
```

For example:

```
i=0
until [ $i -gt 3 ]
do
    echo "again"
    i=`expr $i + 1`
done
```

```
$ ./myscript
0
1
2
```

The `for` loop iterates over values from a list:

```
for variable in list
do
    body
done
```

For example:

```
for name in Tom Jack Harry
do
    echo "$name is my friend"
```

```
done

$ ./myscript
Tom is my friend
Jack is my friend
Harry is my friend
```

The for loop is particularly handy for processing lists of files, for example, all files of a certain type in the current directory:

```
for file in *.doc
do
    echo "$file is a stinky Microsoft Word file"
done
```

For an infinite loop, use `while` with the condition `true`, or `until` with the condition `false`:

```
while true
do
    echo "forever"
done

until false
do
    echo "forever again"
done
```

Presumably you would use `break` or `exit` to terminate these loops based on some condition.

## Break and Continue

The `break` command jumps out of the nearest enclosing loop. Consider this simple script called `myscript`:

```
for name in Tom Jack Harry
do
    echo $name
    echo "again"
done
echo "all done"

$ ./myscript
Tom
again
```

```
Jack
again
Harry
again
all done
```

Now with a break:

```
for name in Tom Jack Harry
do
    echo $name
    if [ "$name" = "Jack" ]
    then
        break
    fi
    echo "again"
done
echo "all done"
```

```
$ ./myscript
Tom
again
Jack           The break occurs
all done
```

The continue command forces a loop to jump to its next iteration.

```
for name in Tom Jack Harry
do
    echo $name
    if [ "$name" = "Jack" ]
    then
        continue
    fi
    echo "again"
done
echo "all done"
```

```
$ ./myscript
Tom
again
Jack           The continue occurs
Harry
again
all done
```

`break` and `continue` also accept a numeric argument (`break N`, `continue N`) to control multiple layers of loops (e.g., jump out of  $N$  layers of loops), but this kind of scripting leads to spaghetti code and we don't recommend it.

## Creating and Running Shell Scripts

To create a shell script, simply put `bash` commands into a file as you would type them. To run the script, you have three choices:

### *Prepend `#!/bin/bash` and make the file executable*

This is the most common way to run scripts. Add the line:

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

to the very top of the script file. It must be the first line of the file, left-justified. Then make the file executable:

```
$ chmod +x myscript
```

Optionally, move it into a directory in your search path. Then run it like any other command:

```
$ myscript
```

If the script is in your current directory, but the current directory “.” is not in your search path, you'll need to prepend “./” so the shell finds the script:

```
$ ./myscript
```

The current directory is generally not in your search path for security reasons.

### *Pass to `bash`*

`bash` will interpret its argument as the name of a script and run it.

```
$ bash myscript
```

### *Run in current shell with “.”*

The preceding methods run your script as an independent entity that has no effect on your current shell.\* If you want your script to make changes to your current

shell (setting variables, changing directory, and so on), it can be run in the current shell with the “.” command:

```
$ . myscript
```

## Command-Line Arguments

Shell scripts can accept command-line arguments and options just like other Linux commands. (In fact, some common Linux commands *are* scripts.) Within your shell script, you can refer to these arguments as \$1, \$2, \$3, and so on.

```
$ cat myscript
#!/bin/bash
echo "My name is $1 and I come from $2"

$ ./myscript Johnson Wisconsin
My name is Johnson and I come from Wisconsin
$ ./myscript Bob
My name is Bob and I come from
```

Your script can test the number of arguments it received with \$#:

```
if [ $# -lt 2 ]
then
    echo "$0 error: you must supply two arguments"
else
    echo "My name is $1 and I come from $2"
fi
```

The special value \$0 contains the name of the script, and is handy for usage and error messages:

```
$ ./myscript Bob
./myscript error: you must supply two arguments
```

To iterate over all command-line arguments, use a for loop with the special variable \$@, which holds all arguments:

```
for arg in $@
do
    echo "I found the argument $arg"
```

\* Technically, it runs in a separate shell (a *subshell* or *child shell*) that inherits the attributes of the original shell, but cannot alter them in the original shell.

```
done
```

## Exiting with a Return Code

The `exit` command terminates your script and passes a given return code to the shell. By tradition, scripts should return 0 for success and 1 (or other nonzero value) on failure. If your script doesn't call `exit`, the return code is automatically 0.

```
if [ $# -lt 2 ]
then
    echo "Error: you must supply two arguments"
    exit 1
else
    echo "My name is $1 and I come from $2"
fi
exit 0

$ ./myscript Bob
./myscript error: you must supply two arguments
$ echo $?
1
```

## Beyond Shell Scripting

Shell scripts are fine for many purposes, but Linux comes with much more powerful scripting languages, as well as compiled programming languages. Here are a few.

Language	Program	To get started...
Perl	<code>perl</code>	<code>man perl</code> <a href="http://www.perl.com/">http://www.perl.com/</a>
Python	<code>python</code>	<code>man python</code> <a href="http://www.python.org/">http://www.python.org/</a>
C, C++	<code>gcc</code>	<code>man gcc</code> <a href="http://www.gnu.org/software/gcc/">http://www.gnu.org/software/gcc/</a>
Java	<code>javac<sup>a</sup></code>	<a href="http://java.sun.com/">http://java.sun.com/</a>
FORTRAN	<code>g77</code>	<code>man g77</code> <a href="http://www.gnu.org/software/fortran/fortran.html">http://www.gnu.org/software/fortran/fortran.html</a>

Language	Program	To get started...
Ada	gnat	info gnat <a href="http://www.gnu.org/software/gnat/gnat.html">http://www.gnu.org/software/gnat/gnat.html</a>

---

<sup>a</sup> Not included in Fedora, nor many other Linux distros.