



Linux for Researchers

Chapter 5: Shell Scripts

When you find yourself typing the same set of commands over and over, it's time to write a script to automate the process. As we've seen, Unix has a modular design. Each command typically does one simple, well-defined task, and commands can be connected together to do complex jobs. The shell just provides the plumbing to connect these commands together.

Today, we'll start off by looking at how we can put those commands and the necessary plumbing together into a script file that we can run.

We'll look at a few more commands that are particularly useful in scripts.

Then we'll look at the variables and flow control mechanisms that make a script more useful than just a list of commands.

Part 1: Writing Shell Scripts



- What's a shell script?
 - A simple shell script is a file that contains some commands that you would normally type at the command line.
- Why is it useful to learn about shell scripts?
 - Shell scripts are widely used, so sooner or later you'll encounter a script you'll need to understand.
 - You'll inevitably want to easily repeat a series of commands in a consistent way (Avoiding questions like "What switches did I use the last time I compiled that program?" and "Do I really have to type almost the same command a thousand times to process all of these files?")

When is a Shell Script Appropriate?:

Generally, a shell script is appropriate when you need to do something repeatedly (either many times right now, or at various times in the future), when you don't need to work with complex data structures like arrays or do much math, and when speed isn't particularly important.

Command Line	Shell Script	Interpreted Language	Compiled Language
tcsh,bash	Bourne Shell	Perl, Python	C, Fortran
One-shots	Repeated	Complex Data Structures or Math	Speed

The Bourne shell is good at doing the flow-control and plumbing necessary to write a script that automates collections of shell commands. It's not very good at math, though, or at arrays (even one-dimensional ones). My rule of thumb is that, as soon as you start thinking about doing arrays in your shell script, it's time to start learning Perl.

Life Decisions....

HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
(ACROSS FIVE YEARS)

	HOW OFTEN YOU DO THE TASK					
	50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
1 SECOND	1 DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
5 SECONDS	5 DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
30 SECONDS	4 WEEKS	3 DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
1 MINUTE	8 WEEKS	6 DAYS	1 DAY	4 HOURS	1 HOUR	5 MINUTES
5 MINUTES	9 MONTHS	4 WEEKS	6 DAYS	21 HOURS	5 HOURS	25 MINUTES
30 MINUTES		6 MONTHS	5 WEEKS	5 DAYS	1 DAY	2 HOURS
1 HOUR		10 MONTHS	2 MONTHS	10 DAYS	2 DAYS	5 HOURS
6 HOURS				2 MONTHS	2 WEEKS	1 DAY
1 DAY					8 WEEKS	5 DAYS

<https://xkcd.com/1205/>

You be the judge: Is it worth your time?

Choice of Shell-Scripting Language:

I highly recommend you write your scripts in the **Bourne shell** (e.g. bash):

- It's **universally available** on Unix-like operating systems.
- It's the **default login shell** for user accounts under Linux.
- **Very few** shell scripts are written in any other language.
- You'll **need to understand it anyway**, because it's so ubiquitous.
- You'll be able to benefit from many **examples** and much **expertise** among your peers.

On the other hand, I recommend you **avoid non-standard features** of bash, that aren't shared by other Bourne shell implementations. Even if you expect your script to always run under Linux, it doesn't hurt to make it **portable from the outset**.

Because of the above, assume that everything I say today is specific to bash, unless I say otherwise.

Regarding non-standard features, there's a set of standards called "POSIX" that specifies how a Unix-like system should behave. Among other things, these standards list certain minimal features that the operating system's Bourne shell scripting language should support. For the examples I'll use today, I've used only features that should be available in any POSIX-compliant implementation of the Bourne shell.

An Example Script:

mkthumbs

```
#!/bin/sh
DIR=$1
# Confirm this is what you want to do:
echo "Do you really want to make thumbnails in $DIR\? (y/n):"
read ANS
if [ "$ANS" = "y" ]
then
    # Change into target directory:
    cd $DIR

    # Get list of images:
    FILES=`ls *.jpg | grep -v '^t_ '`

    # Make thumbnails:
    for F in $FILES
    do
        echo "Processing $F..."
        convert -define jpeg:size=200x200 $F \
            -thumbnail '100x100>' t_`F`
    done
fi
```

This script makes thumbnails for each jpeg image in a given directory. In the following slides we'll look at what each part of such a script does.

Creating and Running a Script:

You can create a shell script with any **text editor**. Just take the commands you'd normally type at the command line and put them into a file. Let's call the file "myprogram". There are then two ways to run your program:

1. Explicitly **tell the shell** to run the commands in the file:

```
sh myprogram
```

Or, 2. make the program **executable**, and then just type its name (assuming that the current directory is in your search path):

```
chmod +x myprogram  
myprogram
```

If you're using tcsh as your login shell, you might need to type "rehash" before you'll be able to run your script by just typing its name. That's because tcsh keeps a cache of the names of all of the executable files in your search path. If a new executable (like your script) is added while you're logged in, the rehash command will tell tcsh to refresh that cache, so that it finds your newly-executable script.

Using the “shebang”:

- You should always begin your shell scripts with the characters “#!/bin/sh” by themselves on the first line.
- When the shell runs an executable file, it looks for the two characters “#!” as the first two bytes of the file. This set of characters is collectively known as a “shebang”.
- When the shell finds them, it knows that the file is a shell script, not a binary executable. The characters following the “#!” tell the shell **which scripting language** should be used to interpret the script.
- Lets you write scripts in **any language you want**, regardless of what your login shell is.

So, when a file begins with “#!/bin/sh” and you invoke it by typing the file's name, the shell actually does something like “/bin/sh myprogram” invisibly, behind the scenes.

On most Linux systems, **/bin/sh** is just a symbolic link to /bin/bash. All Unix-like operating systems should have something called /bin/sh that understands Bourne shell syntax.

Debugging Scripts:

If you want to see what commands the script is using, as they're executed, you can always invoke it like this:

```
sh -x myprogram
```

This is often useful for debugging.

Comments, and Very Long Commands:

- Any line beginning with "#" is considered a comment, and ignored by the computer when running the script:

```
# Create the user account:  
useradd $userid  
  
# Set the user's password:  
passwd $userid
```

- Long commands can be continued on **multiple lines**. A line ending in a **backslash** is continued onto the next line. Note that there can be no characters (**not even a space**) after the backslash:

```
ls -l /tmp | \  
grep bkwl1a | \  
awk '{print $5/1000,$NF}' | \  
sed -e 's/\..*$//' | \  
/bin/mail -s Information bryan@virginia.edu
```

Part 2: Shell Script Plumbing

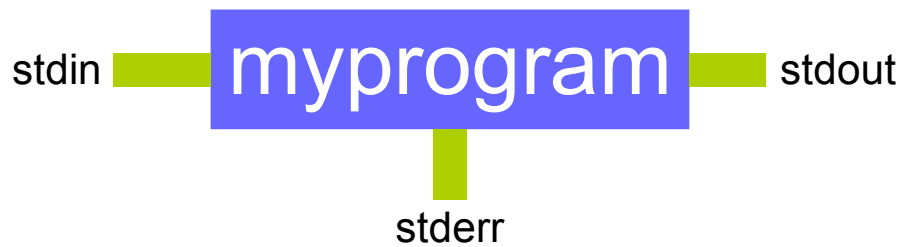


In addition to letting you run commands, the shell provides some plumbing that allows you to connect commands together. The output of one command can be sent to a file, or even to the input of another command. We talked about this earlier, but let's do a quick review.

Standard I/O Channels:

Each running program has three input/output "channels" associated with it:

- 0 "Standard Input" (stdin)
- 1 "Standard Output" (stdout)
- 2 "Standard Error" (stderr)



The program reads input from stdin, writes normal output to stdout, and writes any errors to stderr. When you run a command from the command line, all three of these channels just point to your screen and keyboard.

Redirecting stdout and stderr:

You can redirect stdout to a file (with stderr still going to the screen):

```
myprogram > myprog.out
```

If you already have a file, and you want a command to append onto the end of it, you can use ">>":

```
myprogram >> myprog.out
```

You can send stdout and stderr into separate files:

```
myprogram 1> myprog.out 2> myprog.err
```

Or you can send them both into the same file:

```
myprogram > myprog.out 2>&1
```

Redirecting stdin:

You can also redirect stdin. Normally, commands read input from your keyboard, but you can tell them to read it from someplace else. Look at the following example:

```
myprogram < myprog.in 1> myprog.out 2> myprog.err
```

In the command above, we tell the program to read input from inputfile.dat, then write normal output into outputfile.dat and errors into errorfile.dat. The input might be answers to any questions the program would ask us.

Connecting Commands Together:

What if we connect the output of one program to the input of another?

That's easy to do, and it's one of the most powerful features of shells in Unix-like operating systems.

Consider the following command:

```
myprogram | grep bryan
```

The "|" symbol is called a "pipe", and in the command above it connects the stdout of myprogram to the the stdin of grep.

The "grep" command looks for strings that match a given pattern. Run this way, it reads data from stdin, searches for the pattern, and prints matching lines out on stdout (where they would appear on your screen unless you redirected them into a file).

You'll use grep a lot in shell scripts. We'll describe how it works shortly.

Part 3: Using Variables



A variable is a “box” in which we can temporarily store some information. Just like most programming languages, the shell provides mechanisms for using variables.

Defining Variables:

Variables in shell scripts are used in ways that you're probably already familiar with:

```
DELAY=24
NAME=Bryan
FULLNAME="Bryan Wright"
ADDRESS="Right Here"
NAMEANDADDRESS="$NAME at $ADDRESS"
```

- Note that there can be **no spaces** on either side of the equals sign.
- Values containing spaces need to be **quoted**.
- The last example shows that you can get the value of a variable by putting a **dollar sign** in front of its name.

Double-Quotes and Single-Quotes:

A string enclosed in single-quotes is interpreted literally, but a string enclosed in double-quotes is subject to variable expansion and other processing:

```
~/demo> echo 'My home is $HOME'  
My home is $HOME  
  
~demo> echo "My home is $HOME"  
My home is /home/elvis
```

or...

```
~/demo> echo 'Here I am: `pwd`'  
Here I am: `pwd`  
  
~demo> echo "Here I am: `pwd`"  
Here I am: /home/elvis/demo
```

Special Characters:

The characters `*`, `?`, ```, `$`, `!` and `\` may cause grief if used between double-quotes. You can make these literal by prepending a `\`, as `"\"`:

```
echo "this is a dollar sign: \$"  
echo "this is a question mark: \?"
```

Also note that this won't work...

```
echo 'I'm here!' nor this: echo 'I\'m here!'
```

...but this will:

```
echo 'I'""''m here'
```

The “read” Command:

You can accept input from the user and store it in a variable using the “read” command:

```
#!/bin/sh
```

```
echo "Tell me your name:"
```

```
read ANSWER
```

```
echo "You said your name was $ANSWER."
```

Using Backticks:

Backticks are another way of putting values into variables.
Here's an example:

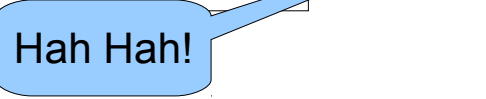
```
CFILES=`ls *.c`
```

In the command above, the shell executes what's between the backticks, then sticks the output of the command into the variable called CFILES. In this example, the variable would contain a list of files with names like *.c in the current directory.

Arithmetic Evaluation:

Doing simple arithmetic in bash isn't so simple. For example:

```
~/demo> VARIABLE=0  
~/demo> VARIABLE=$VARIABLE+1  
~/demo> echo $VARIABLE  
0+1
```

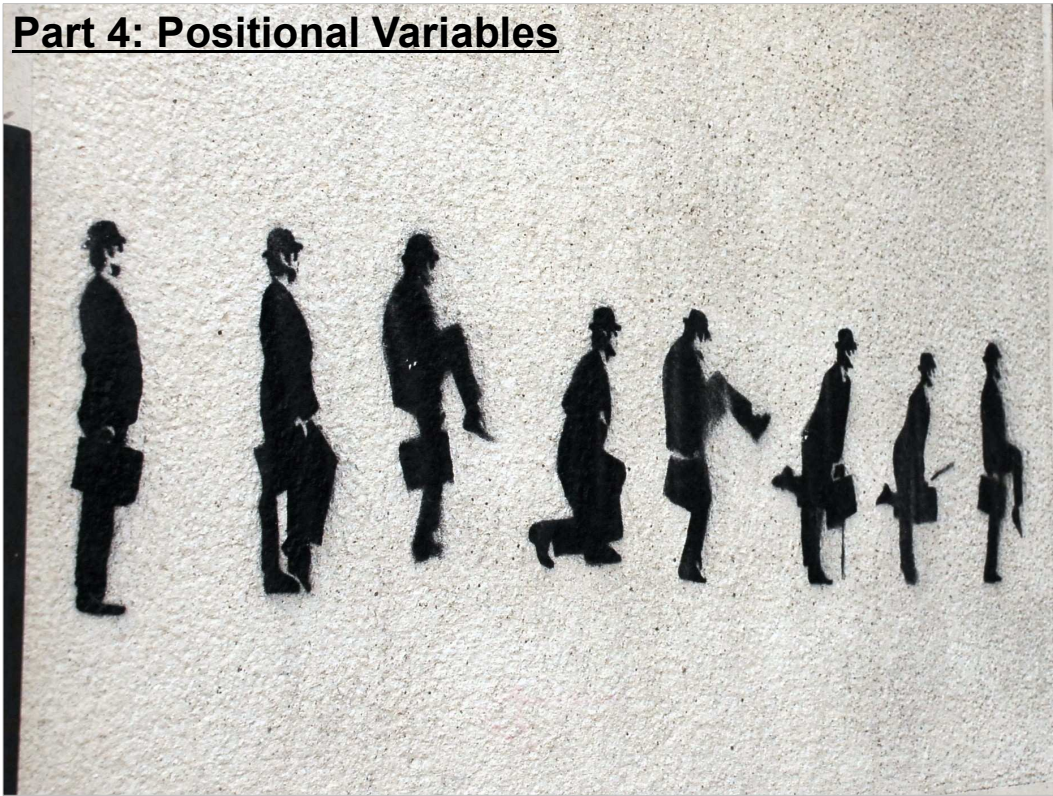


Hah Hah!

However, expressions enclosed between “((“ and “))” will be evaluated arithmetically:

```
~/demo> VARIABLE=0  
~/demo> ((VARIABLE=$VARIABLE+1))  
~/demo> echo $VARIABLE  
1
```

Part 4: Positional Variables



When we run a script there are some variables that are automatically defined for us. Among them are the "positional variables", which can help you make your script more flexible.

Positional Variables:

What if we want to give our program arguments on the command line?

```
~/demo> myprogram file1.txt file2.txt file3.txt
```

The command line arguments are available within the program through a set of pre-defined “positional variables”. These are special variables with the names \$0, \$1, \$2, \$3, etc.. In the example above, these variables would have the following values:

```
$0 = myprogram  
$1 = file1.txt  
$2 = file2.txt  
$3 = file3.txt
```

The variables \$* and @\$ each contain the whole list of arguments, starting with \$1, but they behave differently when enclosed in double-quotes:

Var	Value
\$*	file1.txt file2.txt file3.txt
“\$*”	“file1.txt file2.txt file3.txt”
\$@	file1.txt file2.txt file3.txt
“\$@”	“file1.txt” “file2.txt” “file3.txt”

See “man bash” for more special variables in bash.

Part 5: Useful Components for Building Scripts



In this section we'll look at a collection of commands that are particularly useful in shell scripts. As you'll see, these commands mostly follow the Unix philosophy that says each command should do one simple thing well, and that complicated things can be done by plugging multiple simple commands together.

The “grep” Command:

```
~/demo> ps aux | grep acroread
bkw1a 20422 2.7 1.9 111516 38696 pts/19 S 10:49 0:00 /usr/bin/acroread cluster.pdf
bkw1a 20512 0.0 0.0 5032 664 pts/19 S+ 10:50 0:00 grep acroread

~/demo> ps aux | grep acroread | grep -v grep
bkw1a 20422 2.7 1.9 111516 38696 pts/19 S 10:49 0:00 /usr/bin/acroread cluster.pdf

~/demo> ps aux | grep ^bkw1a
bkw1a 30922 0.0 0.1 7472 2720 pts/19 Ss Feb02 0:00 -tcsh
bkw1a 20422 2.7 1.9 111516 38696 pts/19 S 10:49 0:00 /usr/bin/acroread cluster.pdf
bkw1a 20748 0.0 0.0 5488 996 pts/19 R+ 10:54 0:00 ps auxf
bkw1a 20512 0.0 0.0 5500 664 pts/19 S+ 10:54 0:00 grep ^bkw1a

~/demo> ps aux | grep pdf$
bkw1a 20422 2.7 1.9 111516 38696 pts/19 S 10:49 0:00 /usr/bin/acroread cluster.pdf

~/demo> grep important readme.txt
This line is very important!
```

Pick lines with the string “acroread”

Ignore lines with “grep”
 (“v” = “veto”)

Pick lines ending in “pdf”

Pick lines beginning with “bkw1a”

The “grep” command can be used to select particular lines from an **input stream** or a **file**. In the first examples above, the output of “ps” is piped into grep. In the last example, grep searches within a file. With grep, search terms are specified as “**regular expressions**”. Regular expressions allow you to combine text and wild-card specifiers in complex ways to pick out the lines you want.

Regular expressions (sometimes referred to as “regex”) were invented by Ken Thompson, the same guy who invented Unix.

The name “grep” comes from the syntax of an ancient editor called “ed”. In ed, when you wanted to search for a string and print out lines that contained it, you'd use the command “g/re/p”, where “re” was some regular expression you wanted to search for.

One oddity you should be aware of: If you want to search for something beginning with a dash, you'll need to do something like “grep -- -string”. The “--” tells grep that we're done giving it switches, and everything else following is a literal search string.

Some Regular Expression Syntax:

Symbol	Meaning	-E?
.	Match any single character.	
*	Match zero or more of the preceding item.	
+	Match one or more of the preceding item.	y
?	Match zero or one of the preceding item.	y
{n,m}	Match at least n, but not more than m, of the preceding item.	y
^	Match the beginning of the string.	
\$	Match the end of the string.	
[abc123]	Match any of the enclosed list of characters .	
[^abc123]	Match any character not in this list.	
[a-zA-Z0-9]	Match any of the enclosed ranges of characters .	
this that	Match “this” or “that”.	y
\., *, etc.	Match a literal “.”, “*”, etc.	

Note that some of the things above won't work unless you give grep the “-E” switch, to enable “extended” regular expressions. These are noted in the last column, above. Also note that any expression that uses the characters [] { } ? * needs to be enclosed in single-quotes, to prevent the shell from interpreting these.

Note that regular expressions are different from the pattern matching that we've used before, with commands like “ls *.txt”. That's called “glob” pattern matching, or “globbing”, because it uses a short expression to refer to a whole glob of files.

Regular expression pattern matching is a different thing altogether. For example, you can see in the slide above that * has a completely different meaning in a regular expression.

Some grep Examples:

```
grep elvis
```

```
grep ^elvis$
```

```
grep '[eE]lvis'
```

```
grep 'B[oO][bB]'
```

Match Bob, BOB, BoB, or BOb

```
grep '^$'
```

Match blank lines

```
grep '[0-9][0-9]'
```

```
grep '[a-zA-Z]'
```

Match phone numbers

```
grep '[^a-zA-Z0-9]'
```

```
grep -E '[0-9]{3}-[0-9]{4}'
```

```
grep '^.$'
```

```
grep '^\\.'
```

Match lines with just one character

```
grep '^\\.[a-z][a-z]'
```

Exit Status, “grep -q”, “&&” and “||”:

Whenever a command finishes, it returns an “**exit status**” to the shell, telling the shell whether the command failed or completed successfully. You can use the “**&&**” or “**||**” operators to tell the shell to do something if the command succeeds or fails, respectively.

```
~/demo> w
11:03:09 up 45 days, 40 min, 3 users, load average: 0.05, 0.10, 0.03
USER      TTY      FROM    LOGIN@   IDLE   JCPU   PCPU WHAT
elvis     pts/12   :1000  29Jan09 11days 3.02s  0.10s -bin/tcsh
elvis     pts/17   :1001  09:46    1:16m  1.96s  0.09s -bin/tcsh

~/demo> w | grep -q elvis && echo Found Elvis
Found Elvis

~/demo> w | grep -q bryan || echo No bryans
No bryans

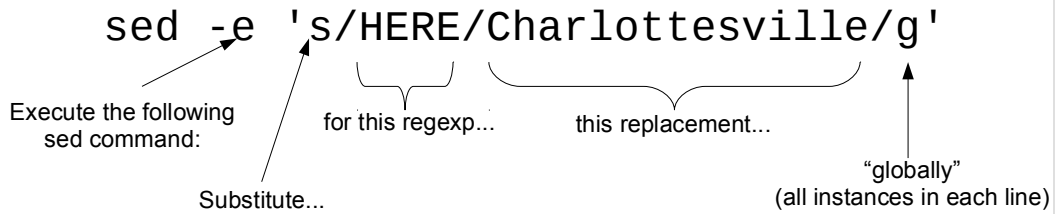
~/demo> w | grep -q elvis && echo Found Elvis || echo No Elvises
Found Elvis
```

The “**-q**” switch tells grep not to write any output, but only return an exit status to the shell. Grep tells the shell it was successful if it finds the regular expression it was looking for. Otherwise, grep tells the shell it failed.

The “sed” Command:

The “sed” command is a “stream editor”. It performs **editing operations** on its input stream, **one line at a time**, and sends the modified data to its output stream. The example below shows the most common way that sed is used. Sed works with the same set of basic regular expressions as grep. To use the “extended” set of regular expressions with sed, give it the “-r” switch.

```
~/demo> cat file.txt | sed -e 's/HERE/Charlottesville/g' > newfile.txt  
~/demo> echo 555-1212 | sed -e 's/-//'  
5551212
```



If you leave off the “g” at the end, sed would only replace the first occurrence of “HERE” in each line. For information about the other things sed can do, see “man sed”.

The “awk” Command:

Awk is actually a complete text processing language, but it's most often used for one simple task: to **extract columns** from an input stream. You might think of this function as orthogonal to grep:

```
~/demo> ls -l
```

```
total 104
-rw-r----- 1 bkw1a bkw1a 20601 Jan 18 10:51 cluster.pdf
-rw-r----- 1 bkw1a demo 983 Jan 18 10:53 cpuinfo.dat
drwxr-x--- 3 bkw1a bkw1a 4096 Jan 18 11:35 phase1
drwxr-x--- 2 bkw1a bkw1a 4096 Jan 19 11:03 phase2
-rw-r----- 1 bkw1a demo 72 Jan 18 10:52 readme.txt
-rw-r----- 1 bkw1a bkw1a 9552 Jan 18 10:52 ReadMe.txt
drwxrwsr-x 2 bkw1a demo 4096 Jan 27 13:18 shared
```

```
Column: 1 2 3 4 5 6 7 8 9
```

awk '{print \$5}'

```
~/demo> ls -l | awk '{print $5}'
20601
983
4096
4096
72
9552
4096
```

grep phase2

Awk was invented by Alfred Aho, Peter Weinberger and Brian Kernighan (hence “AWK”) at AT&T Bell Labs in the 70s.

For chopping data into columns, you can also use the “cut” command, but it's less versatile.

Awk Examples:

awk '{print \$1, \$5}' ← Print columns 1 and 5

awk '{print \$3/2, 5+\$6}' ← Do some arithmetic before printing

awk '{print \$NF}' ← Print the last column (NF = "Number of Fields")

awk '{print \$(NF-1)}' ← Print the next-to-last column

awk '{print "The file size is", \$5}'

awk -F: '{print \$1}' ← Add some text to each line of output

Use ":" as the field separator

Awk numbers the columns **beginning with 1** (not zero). By default, awk chops each line into columns by looking for white-space (spaces or tabs) between characters. You can tell awk to use a different "field separator" with the "-F" switch.

The “sort” Command:

By default, “sort” **sorts** the lines from its input stream in **dictionary order**, and writes out the sorted data. You can use the “-n” switch to sort the lines in **numerical order**, instead.

```
~/demo> cat /etc/passwd | awk -F: '{print $1}' | sort
adm
apache
bin
bkw1a
cyrus
daemon
dbus
distcache
elvis
...etc.
```

“sort” versus “sort -n”:

```
~/demo> ls -l|awk '{print $5,$NF}' | sort
11 cluster.pdf
20601 cluster.pdf
29 data-for-everybody.1.dat
4096 phase1
4096 phase2
4096 shared
41 ForYourEyesOnly.dat
72 readme.txt
9552 ReadMe.txt
983 cpuinfo.dat
```

```
~/demo> ls -l|awk '{print $5,$NF}' | sort -n
11 cluster.pdf
29 data-for-everybody.1.dat
41 ForYourEyesOnly.dat
72 readme.txt
983 cpuinfo.dat
4096 phase1
4096 phase2
4096 shared
9552 ReadMe.txt
20601 cluster.pdf
```

In the first example, we're using awk to pick out the username field from /etc/passwd. In the second and third examples, we're using awk to pick out the file size field from the output of ls.

The “uniq” Command:

```
~/demo> ls -l /tmp | awk '{print $3}' | sort
abk5s
abk5s
bj6h
bj6h
bj6h
cb8nw
cd6j
...etc.
```

```
~/demo> ls -l /tmp | awk '{print $3}' | sort | uniq
abk5s
bj6h
cb8nw
cd6j
c12jd
cw5xj
cx4d
...etc.
```

```
~/demo> ls -l /tmp | awk '{print $3}' | \
sort | uniq -c

      2 abk5s
      3 bj6h
      1 cb8nw
      5 cd6j
     18 c12jd
      3 cw5xj
      2 cx4d
...etc.
```

```
~/demo> ls -l /tmp | awk '{print $3}' | \
sort | uniq -c | sort -n

      1 zlh9w
      1 cb8nw
      1 jhh7t
      ...
      8 ema9u
      8 mab3ed
      9 rjh2j
     15 hg4c
     15 root
     18 c12jd
```

The “uniq” command eliminates **duplicate lines** from its input. It can also be used to **count** the number of duplicates, by using the “-c” switch.

Combined with “sort -n”, as below, uniq can help you produce a **ranked list** of how frequently particular terms appear in the input.

The “wc” command:

The “wc” command (“word count”) can count the number of characters, words or lines in its input. By default, wc shows all three counts.

```
~/demo> cat readme.txt | wc
      3      12      72
      Lines    Words  Characters
```

Use the “-l”, “-w” or “-c” switches to report only line count, word count or character count, respectively.

```
~/demo> cat /etc/passwd | wc -l
73
```

Number of accounts

The "tail" and "head" commands:

What if you just want to get the last line of a file? For that, there's the "tail" command:

```
tail -1 file.dat
```

This would print out the last line of the file. To get the last two lines, type "tail -2", and so forth. With no number, the last ten lines are printed.

There's a similar "head" command to get the first n lines of a file.

These can also be used in pipelines:

```
ls -l /tmp | awk '{print $3}' | sort -n | tail -1
```

The “find” Command:

Find is a tool for **finding files**. It can find files based on name, date, size, ownership, permissions or depth within the directory hierarchy.

```
find . -type f -print
```

Begin in this directory,
and descend down

Select files that match
some criteria...

...and do something.

Some Available Actions:

-print Print the file's name (default action).
-ls Show the file's properties.
-exec Execute a given command, inserting the file's name.

Examples:

```
find . -type f  
find . -type f -ls  
find . -mtime -1  
find . -type f -exec gzip {} \;
```

Some Available Criteria:

-name Match a file name.
-type Match regular files (f), directories (d), symbolic links (l), etc.
-mtime Time since file was last modified, in days. Use “+” for “more than” and “-” for “less than”.
-user Specify user who owns file.
-perm Specify permissions of file.
-size Specify file size. “+” and “-” function as for mtime.

The command “find .” will just descend from the current directory and print the names of all the files and directories it finds.

The “-name” criterion accepts glob-style wild-cards. For example, you could type:

```
find . -name '*.dat'
```

Note that you'll need to enclose the expression in single-quotes, to prevent the shell from trying to expand “*.dat” before calling “find”.

Finally, note that bash (in some implementations) tries to be “helpful”. When an unquoted wildcard pattern can't be matched, it passes the wildcard pattern, just as it's typed, to the program being run, just as though you'd enclosed it in single quotes. The problem with this approach is that the behavior will be different depending on whether or not a file in the current directory happens to match the pattern. I recommend you avoid relying on this feature, and instead always use quotes explicitly to clarify your intent.

The “test” or “[“ Command:

The “test” command is a general-purpose tool to check to see if a given condition is true, and return an appropriate exit status to the shell. This command is so commonly used that it has an alternate one-character name, “[”. The command ignores a trailing “]” on its command line, so the most common way to use test is something like this:

```
[ condition to test for ] && echo True
```

These spaces are mandatory

Because the “test” command is so frequently used in system shell scripts, you should never name an executable “test”. This is very likely to cause odd behavior.

Using "test":

Strings:

STRING1 = STRING2	the strings are equal
STRING1 != STRING2	the strings are not equal

Integers:

INTEGER1 -eq INTEGER2	INTEGER1 is equal to INTEGER2
INTEGER1 -ge INTEGER2	INTEGER1 is greater than or equal to INTEGER2
INTEGER1 -gt INTEGER2	INTEGER1 is greater than INTEGER2
INTEGER1 -le INTEGER2	INTEGER1 is less than or equal to INTEGER2
INTEGER1 -lt INTEGER2	INTEGER1 is less than INTEGER2
INTEGER1 -ne INTEGER2	INTEGER1 is not equal to INTEGER2

Files:

-e FILE	FILE exists
-f FILE	FILE exists and is a regular file
-d FILE	FILE exists and is a directory
-L FILE	FILE exists and is a symbolic link
-s FILE	FILE exists and has a size greater than zero
-x FILE	FILE exists and is executable

Booleans:

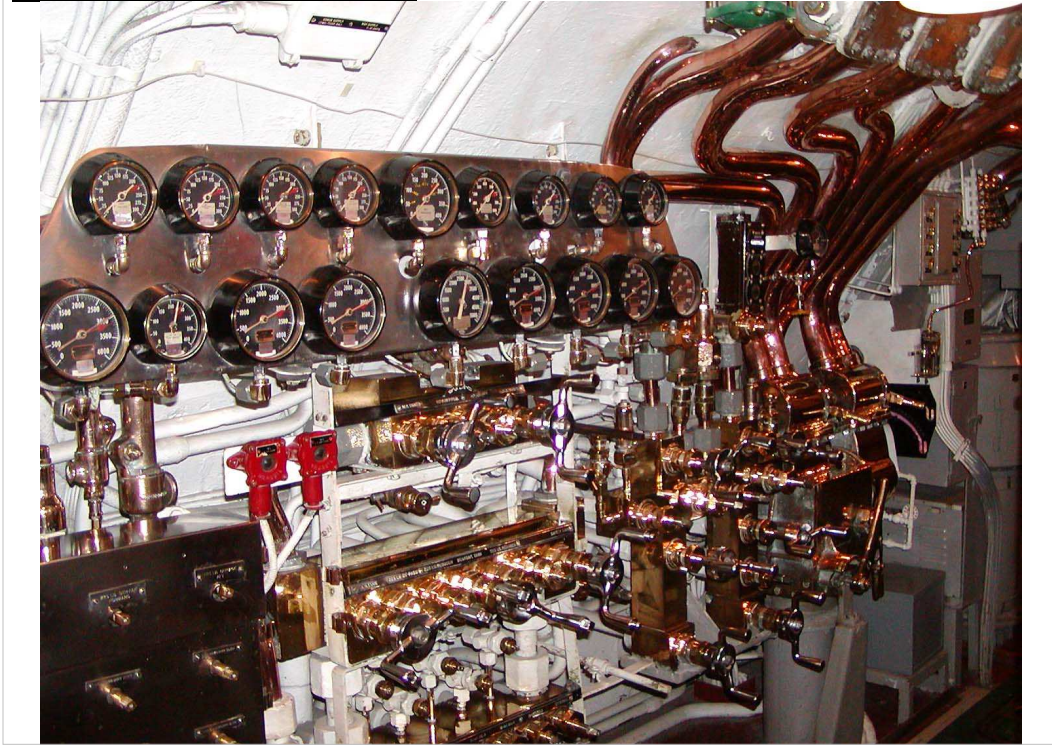
EXPRESSION1 -a EXPRESSION2	both EXPRESSION1 and EXPRESSION2 are true
EXPRESSION1 -o EXPRESSION2	either EXPRESSION1 or EXPRESSION2 is true
! EXPRESSION	EXPRESSION is false

Examples of the “test” Command:

```
~/demo> [ -f /etc/motd ] && cat /etc/motd
~/demo> [ -x /local/bin/update ] && /local/bin/update
~/demo> [ -d /tmp/mytemp ] || mkdir /tmp/mytemp
~/demo> [ "$ANSWER" = "yes" ] && echo Okay. || echo Nope.
~/demo> [ "$ANSWER" \!= "no" ] && echo Didn't SAY no...
~/demo> [ $SIZE -gt 10000 ] && echo Found a big file.
~/demo> [ $SIZE -le 1000 -o -f /etc/preserve ] && echo OK.
```

Note that some shells will require that you put a backslash in front of any “!” characters, to prevent the shell from interpreting them before they're passed to “test”.

Part 6: Flow Control

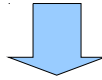


Finally, the shell provides us with tools for controlling the flow of our script. We can create loops to repeat a set of commands, with variations, and we can create conditional sections that will only be executed when certain conditions are met.

Making Loops with “for”:

Here's an example of a “for” loop:

```
#!/bin/sh  
  
for FILE in "$@"  
do  
    echo "Processing file $FILE..."  
done
```



```
~/demo> myprogram file1.txt file2.txt file3.txt  
Processing file file1.txt...  
Processing file file2.txt...  
Processing file file3.txt...
```

This loops through the list of values given after “in”, setting “FILE” successively to each value and executing all commands between “do” and “done” each time.

Examples of “for” Loops:

```
# Convert postscript files to PDF:
for FILE in *.ps
do
    echo "Processing $FILE..."
    ps2pdf $FILE
    echo "Done."
done

# Renice all of my processes:
for PID in `ps aux | grep bkwl1a | awk '{print $1}'`
do
    renice 15 $PID
done

# Find all text files and convert to Unix line ends:
LIST=`find . -name '*.txt'`
for F in $LIST
do
    dos2unix $F
done
```

There's also a numerical version of the “for” statement, similar to the “for” statement found in C. The syntax for this is:

```
for (( expr1 ; expr2 ; expr3 )) ; do list ; done
```

First, the arithmetic expression `expr1` is evaluated according to the rules described below under `ARITHMETIC EVALUATION`. The arithmetic expression `expr2` is then evaluated repeatedly until it evaluates to zero. Each time `expr2` evaluates to a non-zero value, `list` is executed and the arithmetic expression `expr3` is evaluated. If any expression is omitted, it behaves as if it evaluates to 1. The return value is the exit status of the last command in `list` that is executed, or false if any of the expressions is invalid.

You can also use the “seq” command to generate a list of numbers for use with “for” or anything else. See “man seq” for details.

The “if” Statement:

```
#!/bin/sh

USER=$1

if grep -q ^$USER: /etc/passwd
then
    echo "User already exists"
else
    echo "Creating user $USER..."
    useradd $USER
fi
```

The “if” statement looks at the exit status of the following command, and does different things depending on whether the command succeeds or fails.

Note that I've grabbed \$1 and put it into \$USER. It's good practice to put positional variables into new variables with more descriptive names. This should be done right at the top of your script.

Examples of “if” Statements:

```
# Checking the result of a “test” command ([):
if [ -f /tmp/junk.dat ]
then
    rm /tmp/junk.dat
fi

# Many options:
if [ “$ANSWER” = “yes” ]
then
    echo 'Yay!'
elif [ “$ANSWER” = “maybe” ]
then
    echo 'There's still hope!'
elif [ “$ANSWER” = 'Huh?' ]
then
    echo 'Try again.'
else
    echo 'sigh.'
fi
```

The “case” Statement:

If you're expecting to deal with a lot of options, its often convenient to use a “**case**” statement instead of a bunch of if/elif/elif/elif/elif..... statements.

```
case $ANSWER in
yes|YES|y|Y)
    echo 'Yay!'
    ;;
no|NO|n|N)
    echo 'Phooey!'
    ;;
maybe|MAYBE|Maybe)
    echo 'Hope remains... '
    ;;
*)
    echo "Please try again"
    ;;
esac
```

The “while” Statement:

Like “for”, you can also use “while” to do loops:

```
#!/bin/sh

echo 'How many should I do?'
read LIMIT

VALUE=1
while [ $VALUE -le $LIMIT ]
do
    echo "The value is now $VALUE."
    ((VALUE = $VALUE+1))
done
```

“while” looks at the exit value of the following command, and continues to loop until the command fails.



Thanks!