

Shell Scripting

Kyle Wheeler

Contents

1	Intro	1
1.1	What is a Shell	1
1.2	What are Shell Scripts?	1
1.3	Why Use Shell Scripts?	1
2	One Liners	2
2.1	Redirection	2
2.1.1	File Descriptors	2
2.1.2	Simple Redirection (<code>sort</code>)	2
2.1.3	Lots of Files!	2
2.1.4	Connecting Outputs	2
2.1.5	Appending	3
2.2	Conditionals	3
2.3	Environment Variables	3
2.4	Quotes, Strings, and Expansion	4
2.4.1	Wildcards	4
2.4.2	The Tilde	4
2.4.3	Lists	4
2.4.4	Variables	4
2.4.5	Quotes	5
3	Story Problems	5
3.1	Loops	5
3.1.1	For-Loops	5
3.1.2	While-Loops and Until-Loops	6
3.2	More Variables	6
3.2.1	Numbers	6
3.2.2	Better String Manipulation	6
3.3	Conditionals	7
4	Advanced Scripts	7
4.1	Pre-Defined Variables	7
4.1.1	The Numbers	7
4.1.2	The Rest	7
4.1.3	Arrays	8
4.2	Pre-Defined Functions	8
4.3	Functions	8
5	Beyond Scripts	8
5.1	Readline's Beauty	9
5.2	Unix Tools	9
5.3	Config Files	10
6	Examples	11
6.1	Change Background	11
6.2	Check <code>/var/log/messages</code>	11

1 Intro

This document is an introduction to shell scripting. It's not meant to be a complete or authoritative source, but by the same token, I'm pretty sure I know what I'm talking about. All examples in this document (unless otherwise noted) use `bash` syntax¹. If I mention a script in here, I try and include it at the end of this document in the **Examples** section.

1.1 What is a Shell

Many computers have some form of a command-line interface. That is, on many computers you can bring up a window with what is called a command-prompt where you can type in commands to get the computer to do something that you want it to do. In each of these interfaces, there is a program running that reads what characters you have typed in, and translates them into actual commands that the computer hardware can understand. This program is called the command-interpreter, or “the shell.”²

In the strictest of senses, DOS, as a command-line environment, has a shell, called `COMMAND.COM`. On Unix systems, there are many available shells.

1.2 What are Shell Scripts?

A shell script is a text file containing a set of commands to run within the shell. A simplistic example of a shell script would be a Windows `.BAT` file. Advanced shells support more complete or convenient command sets within such a file (script). In most cases, the contents of that file could be typed into the command-prompt and the effect would be the same as running the script.

In Unix, since there are many, many shells available, scripts can be written for any of the shells. And since each shell may use different syntax and may have different capabilities, each shell script must identify which shell it should be interpreted³ with to get the correct behavior. In Unix, the way that shell scripts identify the correct shell is the first line of the script. The first line of the file is a shebang⁴ followed by the absolute path to the shell (generally, shells are kept in `/bin`). Here is a standard example:

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

By convention, the hash symbol (`#`) is almost always a comment delimiter in shell scripts (and behaves similarly to the C++ comment delimiter, `//`).

1.3 Why Use Shell Scripts?

Some people look at shell scripts and think “Hey, that looks like another programming language! I'll just do it in the programming languages that I'm used to.” And for most things, you could. However, shell scripts make doing simple tasks easy. For example, if you wanted a program to look at a directory full of JPEG image files, select one of them at random, and set that JPEG file as the desktop picture, you *could* write a C program to do that. But then you'd have to use the various system calls, and then learn the Xwindows API commands for setting the desktop picture, trap for errors, compile, and then hunt down the bugs. Or, you could write a five-line shell script that uses the pre-made tools that come with X (twelve lines, if you want to be fancy). If you want to monitor `/var/log/messages`, and you want to have the computer email you a sorted copy of it when there are five lines that match a specific pattern, you *could* write a C program that opens `/var/log/messages`, then uses your hand-made pattern matching algorithm to look for this one specific pattern, and counts how many there are, saves them in memory, uses your hand-made sorting algorithm, opens a TCP port, connects to your mail server, talks SMTP (use

¹Because, as you will realize by the end of this document, `bash` is the “one true shell,” and all others are pretenders to the throne.

²It is called a “shell” because all (well, almost all) commands that you can execute will run as a child process of the shell program – and therefore will inherit (or be “in”) the shell's “environment.” (see the **EnVariables** section) Thus, because the shell provides the environment, it can be thought of as a shell around the commands that are given to it.

³“Interpreting” a shell script is almost always the same as running or executing the contents of the shell script, and most people use the terms interchangeably.

⁴A shebang is a hash symbol (`#`) followed by a bang, also known as an exclamation mark (!).

RFC 821) to send out a message, and closes the port. OR, you could write a seven line shell script that uses a for loop, `grep`, `wc`, `sort`, and pipes the results to `sendmail`.

In essence, the point behind writing a shell script is not only to be able to run the same commands over and over again without retyping them and without worrying about typos the hundredth time you run it, but also to use the many useful tools that Unix comes with, instead of re-inventing the wheel.

2 One Liners

Shell scripts can do an amazing number of things in one line. You may argue whether or not they are real shell scripts (although many shell scripts on a Unix machine are only two lines—the shell identifier, and the complex command), but they're good to know anyway.

2.1 Redirection

2.1.1 File Descriptors

Unix programs, when launched, have three file-descriptors⁵ open. These file descriptors are called Standard-Input (`stdin`), Standard-Output (`stdout`), and Standard-Error (`stderr`). Most command-line utilities will read in input from `stdin`, write errors to `stderr`, and write everything else to `stdout`. Each of these file descriptors has a number—`stdin` is number 0, `stdout` is number 1, and `stderr` is number 2 (this will be important later).

2.1.2 Simple Redirection (`sort`)

One very useful Unix tool is `sort`. `sort` takes input from `stdin`, sorts that input by line, and writes it in sorted form to `stdout`. Using a shell, you can connect a file to `sort`'s `stdin`. If, say, you have a file that is a list of words to sort,

```
sort < file
```

will print out the contents of that file, sorted. You can also direct the output of a command to another file, like this:

```
sort < file > file.sorted
```

(the order of redirection does not (usually) matter—you can use `sort > file.sorted < file` as well). Note that you have only directed `stdout` to the `file.sorted`, so errors will still be printed to the terminal. If you wish to redirect `stderr` to a file, you must remember that `stderr` is file descriptor number two (I told you it would be important), so to redirect it you do this:

```
sort < file > file.sorted 2> file.error
```

2.1.3 Lots of Files!

You can also use the shell to open new files, either for reading or writing, as long as you specify the number that the descriptor will have (although you are not guaranteed that the program will use it)—but that's for advanced users. If you're curious, here's an example:

```
sort < file > file.sorted 2>file.error 3>file.3 4< file2
```

2.1.4 Connecting Outputs

You can also connect the output together. For example, it is common that you want both `stdout` and `stderr` to be saved in the same file. Here's how it works:

```
sort <file 1>&2> file.out
```

⁵Everything in Unix is treated as a file, so while these may not be files per-se, treat them as such (for the most part).

2.1.5 Appending

Once you start playing around with redirection like this, you'll notice that every time you do it, your `file.sorted` is replaced with a new copy. If you do not want that behavior, and would rather that your command was *appending* to the end of the file, simply use `>>` instead of `>`.

2.2 Conditionals

When programs (commands) finish (or exit, quit, stop, or whatever you want to call it) they pass back to the shell something called a “return value.” This value indicates whether the command was successful or not. A common thing to want to do is to have one command contingent upon the successful completion of a previous command. For example, you may have a very important command to run, and you want it to email you if that command fails. Or you may have a series of commands, but you don't want it to execute the rest of the commands if one of them fails.

The way you do this is with conditionals (which are based upon the return values of the commands that you execute). Conditionals are traditional boolean operators—you use some reasonably standard symbols: `||` (or), `&&` (and), `!` (not), and `;`⁶. Commands are interpreted in a short-circuit fashion. So, in an `a && b` command, if `a` returns false, `b` won't be executed, but in an `a || b` command, if `a` returns true, `b` won't be executed.

Here's a command that will compile `foo.c`, and if there are no errors, will execute the output:

```
gcc foo.c && ./a.out
```

Here's another example that will compile `foo.c`, and if there are errors, it will edit the file:

```
gcc foo.c || vim foo.c
```

You can also stack them together. Here's a command that will compile `foo.c`, and if there are no errors execute it, but if there are errors will edit the file:

```
gcc foo.c && ./a.out || vim foo.c
```

2.3 Environment Variables

One of the concepts in Operating Systems that was invented a long time ago is “environment variables.” Basically, there is certain information, identified by a string of characters (a name) that is part of the “environment” in which a program runs. A copy of these environment variables is given to commands you run, and they're used, generally, to store some preferences and configuration settings. You set Environment Variables like this:

```
export HISTCONTROL="ignoredups"
```

Some of the important ones you will probably care about are:

HISTCONTROL

Set this to `ignoredups`, and thank me later. (It ignored duplicates when building your command-history. If you don't know what I'm talking about, you'll thank me even more when you read the **Readline** section.)

PS1 & PS2

These set the prompts. `PS1` is the standard prompt, `PS2` is the prompt you get if you've left a container open (containers are loops, quotes, parenthesis, and so on). There's a `PS3` too, but it's rarely used.

PAGER

This is the default program to be used for viewing text files. One program that uses this is `man`—you can use this to switch between using `more` and `less` to view man pages, for example. (Set it to `less`.)

⁶The `;` operator is a conditional that suffices as a command separator—subsequent commands are *always* executed with this separator.

EDITOR

This is the default program to be used for creating or editing text files. Most mail readers/composers use this.

PATH⁷

This is a colon-separated list of directories to look in when you type in a command (if the command is in one of those directories, it will be executed). The list is searched in order, from the beginning to the end, and the first one found will be used.

2.4 Quotes, Strings, and Expansion

2.4.1 Wildcards

When you express a string in a command, it is generally interpreted in some way before it is actually evaluated. For example, if you execute the following command

```
ls *.txt
```

a list of all the files in the current directory that end in `.txt` will be printed to the terminal. The `*` is used as a wildcard in the shell's primitive pattern matching, and is always used to select from file names. The shell expands the `*.txt` pattern, so if your current directory holds two text files named `foo.txt` and `bar.txt`, then `ls` thinks the command looked like this: `ls bar.txt foo.txt`

2.4.2 The Tilde

You've probably also seen references to files written like this:

```
cd ~/work
```

The `~` in that command is "expanded" to mean the current user's home directory (which may not always be in `/home`, depending on the system—for example, on MacOS X, it's `/Users`, and on Oak it's much more complicated). The `~` can also be used with other usernames to get the paths to the named user's home directory—like this:

```
ls ~ostermann/classbin
```

2.4.3 Lists

You can enumerate a number of possible strings using lists, and it will act like a limited wildcard. For example

```
rm foo.{aux,dvi,log,pdf,ps}
```

will delete `foo.aux`, `foo.dvi`, `foo.log`, `foo.pdf`, and `foo.ps`. These lists can be nested.

2.4.4 Variables

Variables are also interpreted in strings. The contents of the current environment are all variables, so to find out, for example, what is in your `PATH` variable, you can:

```
echo $PATH
echo ${PATH}
```

The two are equivalent, but the second one, in complicated scripts, is generally safer and harder to confuse. You can also define your own variables, and use them, like this:

```
F00=bar
echo $F00
```

⁷Do NOT put "." ANYWHERE in here.

The second line there is interpreted by the shell as if you had really typed in: `echo bar`. That means you could also do this:

```
F00='ls -l'
$F00 /usr/bin
```

And it would behave as if you had typed in `ls -l /usr/bin`.

2.4.5 Quotes

If you want to alter the standard method of interpretation, you can group things into strings using quotes. There are three kinds of quotes: double-quotes (`"`), single-quotes (`'`), and back-quotes (```).

Double-quotes group things into strings, so you can have an argument that contains spaces. Variables, tildes, wildcards, and other kinds of quotes are still interpreted or expanded within double-quotes.

Single-quotes group things into strings similarly to double-quotes, but nothing inside of single-quotes is interpreted further.

Back-quotes evaluate the contents as if it was another command.

Confused? Consider the following table of commands and results:

Double Quotes	Single Quotes	Back Quotes
<code>echo foo > file.txt</code>	<code>echo foo > file.txt</code>	<code>echo foo > file.txt</code>
<code>echo bar >> file.txt</code>	<code>echo bar >> file.txt</code>	<code>echo bar >> file.txt</code>
<code>THEFILE=file.txt</code>	<code>THEFILE=file.txt</code>	<code>THEFILE=file.txt</code>
<code>echo "wc -l \$THEFILE"</code>	<code>echo 'wc -l \$THEFILE'</code>	<code>echo `wc -l \$THEFILE`</code>
The Output:	The Output:	The Output:
<code>wc -l file.txt</code>	<code>wc -l \$THEFILE</code>	<code>2</code>

3 Story Problems

So, you've mastered the one-liner. You understand quotes, variables, wildcards, redirection, and you're on top of the world. Then you want to add a `.html` extension to every file in a directory.

3.1 Loops

Loops to the rescue! The shell supports a few different kinds of looping structures. There's a `for` loop, a `while` loop, and an `until` loop, among others.

3.1.1 For-Loops

There are two kinds of for-loops: C-style and list-style. C-style for-loops are easy to explain by example.

```
for (( I=1; $I < 4; I=$I+1 )) ; do
echo $I
done
```

This script will print out 1, 2, and 3 on separate lines. Just like a C for-loop, the first part initializes a variable, the second part is a test, and the third part is the increment.

The second kind of for-loops are based in the shell's ability to parse strings into lists. Here's an example:

```
for F in `ls ~/*`; do
echo $F
done
```

The shell parses the string between the "in" and the ";" and separates it into a list of things. For each item in the list, the `F` variable will be assigned the value of the item, and the contents of the loop will be executed. The shell separates things into lists by spaces and by newlines—so in this example, where you

would expect it to simply print out the name of every file in your home directory, if one of those files has a space in its name, the name of that file will be split and put on different lines.

So in my example, you can add the `.html` extension like this:

```
for F in *; do mv $F $F.html; done
```

Oh, didn't I tell you? Loops can be expressed as a single line. But you see how variable names can get confusing? It would be safer to write that script like this:

```
for F in *; do mv ${F} ${F}.html; done
```

3.1.2 While-Loops and Until-Loops

These loops perform exactly as you would expect them to. While-loops work like this:

```
F=1
while [ $F -lt 5 ]; do
F=$((F+1))
echo "Not done yet"
done
```

And until-loops are identical, except the test (in the example, the test was `[$F -lt 5]`) is negated.

3.2 More Variables

3.2.1 Numbers

You've probably been able to tell already that there is more to the shell's variable handling than meets the eye. Most of the time, variables are treated as strings, however, they sometimes can act as numbers. To treat a variable as a number, you can put an expression involving it inside double-parenthesis (like above: `F=$((F+1))`). Variables may also behave like numbers when the Unix tool `test` is evaluating them. Note: `test` is symlinked to `[`, and the `]` is thrown away! So if you want to look up the right syntax, use `man test`.

3.2.2 Better String Manipulation

Unfortunately, the shell's string manipulation abilities are limited at best. Fortunately, you can use standard Unix tools to achieve the same effect. What say you need to take a bunch of files with spaces in their names, and replace the spaces with underscores. That sounds like a job for `sed` (`sed` interprets standard regular expressions for you)! You could do something like this (note the different kinds of quotes):

```
for F in *; do
mv $F `echo $F | sed 's/ /_/g'`
done
```

Similarly, if you want to find out your IP address in a shell script, you can use the other beautiful tool, `awk`. `awk` is very powerful, but one common use is for identifying text strings separated by white space (spaces and tabs). Combine that with `cut` (which separates strings by specific characters), and you can do something like this:

```
MYIP=`/sbin/ifconfig eth0 | grep "inet addr:" | awk '{ print $2 }' | cut -d: -f2`
```

See how cool pipes can be?

3.3 Conditionals

There are two main conditionals (that I haven't mentioned already). The first is `if`. Its use is pretty easy to understand, and fairly predictable. However, its syntax is a little unusual. Here's how it works:

```
if [ $F -eq 1 ]; then
echo "Equals One"
elif [ $F -eq 2 ]; then
echo "Equals Two"
else
echo "Equals Neither"
fi
```

Notice that the close of the `if`-statement is `if` backwards? Case is the same way. Here's an example case-statement that does the same thing as that `if` statement.

```
case "$F" in
"1")
echo "Equals One"
;;
"2")
echo "Equals Two"
;;
*)
echo "Equals Neither"
;;
esac
```

4 Advanced Scripts

So what say you haven't learned how to use `make` yet, and you're compiling `.tex` files into `.pdf` files, and you're getting tired of manually compiling your \LaTeX into `.dvi`, `.ps`, and finally `.pdf`s. You have two choices: learn `make` (highly recommended), or write a shell script!

4.1 Pre-Defined Variables

4.1.1 The Numbers

Shell script text files, when you set the executable bit on them, can be executed just like regular programs. As regular programs, they may be given arguments, and you can find out what the value of those arguments are, using some of the pre-defined variables. The "numbers," as I call them, are the variables `$0`, `$1`, `$2`, and so on. In a shell script, `$0` will correspond to how the shell script was called, `$1` will correspond to the first argument given to it, and so on. If there was no argument, the variable contains nothing. So, to solve my `make-less` problem, I could write this shell script:

```
#!/bin/bash
latex $1.tex && dvips -o $1.ps $1.dvi && ps2pdf $1.ps
```

If, for example, I called that shell script `latexit`, and put it in my path, I could compile `foo.tex` file into `foo.pdf` with this command: `latexit foo`. The "numbers" are redefined for functions (which I'll talk about in a moment). In functions, they correspond to the arguments passed to the function instead of the ones passed to the shell script itself.

4.1.2 The Rest

There are more pre-defined variables in the shell. Here they are:

`$*`

This expands to the argument list passed to the script.

`$@`

This expands to the argument list passed to the script separated by spaces.

`$#`

This expands to the number of arguments passed to the script.

`$?`

This expands to the status of the most recently executed command (did it succeed or did it fail?)

`$$`

This expands to the PID of the shell.

`$!`

This expands to the PID of the last command (spawned process).

`$_`

This expands to the absolute file name of the shell script, or the last argument to the previous command (after expansion).

4.1.3 Arrays

I suppose it is worth mentioning that variables can be arrays, like this:

```
F00[1]=bar
echo ${F00[1]}
```

4.2 Pre-Defined Functions

There are literally hundreds of pre-defined functions built into the one true shell. If you want to memorize them all, read the man page. The two that I use the most in shell scripts are `read`, `unset`, and `eval`. They do predictable things, similar to how they work in other interpreted languages. `read` will read into a variable from a prompt or a file, `unset` will un-declare and destroy a variable, and `eval` will evaluate a string (or a variable) as a command.

4.3 Functions

This is the fun stuff—defining your own functions. This is another idea that is easiest to explain by example. Here's a function (that's in my `~/ .bashrc`, actually) for adding something to the `PATH`.

```
function add_to_path {
  INPATH=0
  for F in `echo $PATH | sed 's/:/ /g'`; do
    if [ $1 == $F ] ; then
      INPATH=1
      break
    fi
  done
  [ $INPATH -eq 0 ] && PATH="$PATH:$1"
  unset INPATH
}
```

5 Beyond Scripts

This section just collects little bits of information related to shells and scripting that some people might not know (and would make their lives considerably easier).

5.1 Readline's Beauty

Once upon a time, a Unix hacker was typing a multi-line script into his shell and made a mistake (a rare thing for this hacker). He reached for the backspace key, and there appeared in his terminal: `^H`. This greatly displeased the hacker extraordinaire, and he tried to use the arrow keys to go back and edit his command. Instead, he got `^[D`. As he was retyping his command, from scratch, he resolved that such a thing should never happen again. After he was done with his script, he wrote a library such that this thing should never happen in his shell again. He created `libreadline`, and it was good.

`Libreadline` is a library for reading from the terminal that any shell worth anything at all uses. It has command history (press the up-arrow to scroll through your previous commands), it allows you to edit your command using the arrow keys, it allows you to tab-complete the names of files and commands (to prevent typos, when you type the beginning of a long filename hit tab and the rest of it will automatically be typed in for you), it even allows you to use standard emacs or vi commands (emacs is default) to edit your command. It is, truly, a thing of beauty.

5.2 Unix Tools

This list, borrowed in part from Matt Hyclak, is a list of commands that you *should* know and are very useful if you do much shell-scripting.

`awk`

Scans its input for lines that match any of the patterns specified.

`cat`

Takes stdin and any files that you pass it and dumps them to stdout, in order.

`cut`

Prints selected parts of lines from its input to stdout.

`date`

Prints the current time and date (optionally some other time and date) in a specified interesting or beautiful format.

`diff`

Outputs the difference between two text files; returns 1 if two binary files differ.

`echo`

Prints its command line arguments to standard output.

`find`

Helps to find a file in your filesystem based on one of many characteristics including name, size, modification time, filetype... etc.

`grep`

`Grep` is used to search for a *regular expression* (complex pattern) in a file.

`less`

A GNU replacement for the basic `more` pager; superior in almost all respects.

`locate`

A better `find` than `find` if all you're concerned about is filename. `locate` periodically (daily, on most systems) creates a database of all your files and searches that instead of the life filesystem to improve search time tremendously.

`lynx` or `links`

A simple console mode web browser; `lynx` can really save you if you need to reference the web and X is misbehaving. `links` is a better `lynx`, supports more accurate html rendering, supports tables, and all of `lynx`'s syntax.

man
This is the Unix manual page browser. Not useful in scripts, but if you don't learn any other commands, learn this one!

more
This is the standard Unix pager; `more` displays its input to the screen, breaking at every screenful to wait for you to hit enter.

printf
A better echo, `printf` translates `\n` into newlines, among other things.

ps
Lists the currently running processes on the system in any number of wild and wonderful ways.

sed
Performs potentially complex operations on its input similar to those you might do in an editor.

sort
Takes its input, sorts it based on specified criteria, and outputs it.

test
Evaluates an expression and returns zero or 1 depending on its truth value.

wc
Gives you statistics about a file's content: number of characters, lines, words, etc.

which
Returns the complete (canonical) pathname of a given command.

who/w
Lists all the current users of the system (with some caveats).

xargs
Breaks its input into tokens and provides those tokens as arguments to a specified command.

5.3 Config Files

Guess what—config files are just little shell scripts that don't have a limited scope (they are executed directly by the shell, and not by a child of the shell)! Don't believe me? Check out a standard `.bashrc`:

```
# .bashrc

if [ -f ~/.bash_profile ]; then
  . ~/.bash_profile
fi

# shell info
PS1="/dev/null << "
PS2="<< "

# User specific aliases and functions
alias ls='ls --color'
alias lsl='ls --color -lh'
alias quota='quota -v kwheeler'

export PATH=$PATH:/usr/ccs/bin
```

Pretty slick, eh? `bash` loads `/etc/bashrc` and then `~/.bashrc` when it runs, and if it's a login shell (either you're logging in, or you passed it the `-l` option), it loads `~/.login` too.

6 Examples

6.1 Change Background

You can have cron run this.. I created it because I was bored. It switches the background (in X) to a random picture in a folder of pictures. It requires Esetroot, which is part of most Eterm packages.

```
#!/bin/bash
PICTURES=/home/kyle/bgs
COUNT='ls $PICTURES | wc -l'
WHICH=$((RANDOM%COUNT))
WHICH=$((WHICH+1))
COUNT=0
for F in $PICTURES/*; do
COUNT=$((COUNT+1))
if [ $COUNT -eq $WHICH ]; then
Esetroot "$F"
break
fi
done
```

6.2 Check /var/log/messages

I have no idea why you would want to do this, but I mentioned it in section 1.3, so here it is.

```
#!/bin/bash
PATTERN="FAILED"
while true; do
if [ 'grep "$PATTERN" /var/log/messages | wc -l' -eq 5 ]; then
printf 'To: me@foo.gov\nStuff Went Bad!' | /usr/sbin/sendmail -t
fi
done
```