

I Don't Do Windows



Bash Shortcuts

By David S. Jackson

During your daily life at the command prompt, you sometimes wonder when it will get easier—when the command prompt will be as natural as, say, a file manager. Well, in time you can be so comfortable at the command line that a file manager or graphical tool of any kind will seem primitive by comparison. But it takes time—and some mentoring along the way.

Here are some tips that have made my life at the prompt comfortable and slick, as well as infinitely preferable to graphical tools. My remarks pertain primarily to Bash and to Bourne-compatible shells.

Regular Expressions

These are symbols that allow you to express volumes with just a few keystrokes. At the command prompt, you most often are selecting files on which to perform some operation. You can waste a lot of keystrokes and mouse clicks on selecting just the right files, or you can use regular expressions that help you select only the needles in just the haystack you desire.

Some of the most common special characters used in regular expressions are the asterisk (*), the period (.), the question mark (?), the backslash (\), the curly braces ({}), the square brackets ([]), and parentheses ().

For example, instead of checking the size of all graphic files in a directory by listing them individually, a single command can display all the results you seek, and save you what could be a very tedious task:

```
du -sh *.{jpg,gif,png}
```

This command runs the `du` command (disk space usage) on all graphic files in the current directory. The curly braces let you isolate all files with any suffix in the braces separated by a comma and no space. The shell looks for all files with either a `.jpg` suffix, a `.gif` suffix, or a `.png` suffix. I've combined three different searches into a single command.

Some other handy structures are specifying a range of characters using the square braces:

```
rm $HOME\[a-zA-Z0-9]+\sw[a-z]{1,3}
```

This command removes all `vi` swap files in the current

directory. Ever open a `vi` file and get its complaint about a swap file existing, “Do you want to restore the current file from the swap file?” It can be annoying. And when you know you're fastidious about saving your files anyway, sometimes those swap files just get in the way, yet you don't want to turn off the swap file feature, just in case the power does go off while you're editing a file.

Anyway, this little command uses ranges of characters within the square braces to specify permissible characters and their quantities you want the

shell to return to you. The plus sign and the numbers in braces specify the number of characters that are permissible in the sequence you're examining.

The first period is “escaped” (meaning it has a backslash in front of it), because the shell can interpret the character ambiguously. The range of characters includes all lowercase characters (`a-z`) and all uppercase characters (`A-Z`), and all numerals (`0-9`). Now, this range of characters applies to a single character space. In this case, we want to specify one or more characters, so we use the plus sign. If we wanted to specify zero or many, we could use the asterisk. For zero or one, we use the question mark. If we wanted to specify exactly four characters and only four characters, we could say:

```
[a-zA-Z0-9]{4}\sw[a-z]
```

If we wanted to specify at least two, but not more than four characters, we could have said:

```
[a-zA-Z0-9]{2,4}
```

But for our purpose, we want to specify all swap files in the current directory, and they can be named anything, with letters and numbers preceding a period and a suffix starting with the letters `sw`.

What if there are other dot files in your directory that start with `sw` that you don't want to delete? You can use a pipe and the `xargs` command. Try this:

```
ls -a $HOME\[a-zA-Z0-9]+\sw[a-z]{1,3} | xargs -J % rm -i %
```

Take a look and “man `xargs`” or “info `xargs`” for more details on this handy tool. `Xargs` syntax might vary slightly between implementations. If you wanted to be asked on a file-by-file basis, you could try:

```
for file in `ls -a $HOME\[a-zA-Z0-9]+\sw[a-z]{1,3}`; do  
> rm -i ${file}  
> done
```

We do occasionally find files with oddball characters in them. Sometimes underscores, hyphens, dollar signs, carets, and other characters make their way into filenames. It's unfortunate that they do, because these characters are special to the shell. They're called metacharacters.

Sometimes it's worthwhile to specify a range of characters that do not appear in a filename:

```
rm $HOME\.[^0-9]+\sw[a-z]
```

This command removes only swap files whose prefixes are not composed of digits. This type of pattern or expression normally comes in handy when you're dealing with specific strings of characters, such as with date or time stamps on files:

```
rm `ls -l|egrep "Jul [2][2-8]"`
```

Variables and Pattern-Matching Operators

These are big words for moving mountains with steam shovels instead of teaspoons. The true power of regular expressions kicks in when you use several of the shell's features together in concert. The UNIX philosophy calls for many small tools doing specific tasks very well and working well together.

A common task is renaming groups of files in a directory. Many times you have irregularly named files that you want to name according to a desired pattern. For example, let's say you want to take all the "file name.MP3" files in a directory and remove excess spaces, as well as remove any existing suffix to replace with a lowercase "mp3" suffix.

The first part of the problem we'll deal with is replacing the existing suffix, if any, with a lowercase "mp3" suffix. First, we'll determine if a file is actually an MP3 file. If so, we'll capture the basename of the file and append the desired suffix. If a file is not an MP3 file, we'll perform no action on it. The command might look like this:

```
for file in `ls -1`; do
file $file | egrep -iq mp3 && mv ${file} ${file%.*}.mp3
done
```

This command would work on 10 files or 100 files. The nice thing about the shell is that you can write an entire script as a single command. The shell won't execute it until you enter the final closing character and press <Return>.

Above, the `${file}` and `${file%.*}` constructions are a way to expand variables and substitute filename parts before or after the dot separating the basename from the suffix. The `${file%.*}` says remove any part of `${file}`, the variable containing the name of the file, after the dot, if

there is a dot. Notice, we replace the dot and anything after it with a dot and the lowercase "mp3" suffix. But this works only on MP3 files as identified by the file command.

The command above doesn't take blank spaces into consideration. How do we remove them?

We have to prevent the shell from letting the "for" built-in command interpret the spaces as field separators (which is what "for" normally does). So, instead of saying:

```
for filename in `command`; do
```

we'll say,

```
for filename in *; do
```

It basically means the same, but we avoid some of the ambiguities in how ls treats blank spaces.

Next, we'll use variable substitution to upgrade the pattern matching shown previously:

```
for file in *; do
if file $file|grep -iq mp3
then
mv $file $(echo $file|sed 's//g')
fi
```

This will work on all MP3 files, but the renaming will be meaningful only for those files with spaces in the filenames. The spaces will be removed. Remember, if you want to test a regular expression, simply place an echo command ahead of the actual mv command. It will echo the command and how the shell interprets the regular expression rather than trying to execute the command.

You'll notice I use pipes, conditional processing, command exit status checking, command substitution, variable expansion, positional parameters, and multiple lines of commands all on a single command prompt. The shell lets you do all this in a single command line, and it rewards you with enormous power in a few keystrokes for your effort.

If you were to try this in a file manager, you would probably have to repeat yourself many, many times with redundant mouse clicks. The shell lets you accomplish it all with enormous brevity. □