Globbing and Quoting

This lesson will introduce you to the joy of quoting in bash. You will learn what globbing is and what its special characters are. It'll also explain the difference between single and double quotes, and you'll learn the difference between globs and regular expressions.

WE'LL COVER THE FOLLOWING

If you've come across it before, you may have wondered what the `*` in bash commands really means, and how it is different from *regular expressions*.

> Note: Do not panic if you don't know what regular expressions are. Regular expressions are patterns used to search for matching strings. Globs look similar and perform a similar function, but are not the same. That's the key point to understand about globs vs regular expressions.

How Important is this Lesson?

Globbing and quoting are essential topics when using bash. It's rare to come across a set of commands or a script that doesn't depend on knowledge of them.

Globbing

Type these commands into the terminal:

```
touch file1 file2 file3
ls *
echo *
```

Type the above code into the terminal in this lesson.

- **Line 1** creates three files (`file1`, `file2`, `file3`)
- **Line 2** runs the `ls` command, asking to list the files matching `*`
- **Line 3** runs the `echo` command using `*` as the argument to `echo`

Type all the commands in this lesson out in the terminal below.

> Note: continue to use this terminal during this lesson. Throughout this course it is assumed that you complete each lesson by typing in the commands in the provided terminal for that lesson in the order the commands are given.

You should be able to see the three files listed when both, **line 4** and **line 5**, are executed.

The shell took the `*` character and converted it to match all the files in the current working directory. In other words, it has converted the `*` character into the string `file1 file2 file3` and then processed the resulting command.

Quoting

What do you think will be output if we run these commands?

Think about it first, make a prediction, and then type it out!

```
ls '*'
ls "*"
echo '*'
echo "*"
```

Type the above code into the terminal in this lesson.

- **Line 1** lists files matching the `*` character in single quotes
- **Line 2** lists files matching the `*` character in double quotes
- **Line 3** echoes the `*` character in single quotes
- **Line 4** echoes the `*` character in double quotes

You will see that the first two commands result in an error, and the second two just show the raw `*` on the terminal. In other words, both types of quote do nothing with the `*` character.

This is difficult to predict even if you are very experienced in bash!

Was the output what you expected? Can you explain it? Ironically it may be harder to explain if you have experience of quoting variables in bash!

Quoting in bash is a very tricky topic. The point to be noted is that quoting globs removes their effect. But in other contexts single and double quotes have different meanings. You will see example of this in the next lesson on variables.

Quoting changes the way bash can read the line, making it decide whether to take these characters and transform them into something else, or just leave them be. We will see more examples of this throughout the course, and specifically in the next lesson

What you should take from this is that “quoting in bash is tricky” and be prepared for some head-scratching later!

Other glob characters

`*` is not the only globbing primitive. Other globbing primitives are:

- `?` - matches any single character
- `[abd]` - matches any character from a, b or d
- `[a-d]` - matches any character from a, b, c or d

Try running these commands and see if the output is what you expect:

```
ls *1
ls file[a-z]
ls file[0-9]
```

Type the above code into the terminal in this lesson.

- **Line 1** lists all the files that end in '1'
- **Line 2** lists all files that start with 'file' and end with a character from a to z
- **Line 3** lists all files that start with 'file' and end with a character from 0 to 9

Dotfiles

Dotfiles are like normal files, except their name begins with a dot. Create some with `touch` and `mkdir`:

```
touch .adotfile
mkdir .adotfolder
touch .adotfolder/file1 .adotfolder/.adotfile
```

Type the above code into the terminal in this lesson.

You've now created some dotfiles and folders. If you run `ls`:

```
ls
```

Type the above code into the terminal in this lesson.

those files don't show up. So these files are hidden from us in normal view. What if we try to use a * as a glob?

```
ls *
```

Type the above code into the terminal in this lesson.

Same result. Those files are hidden. While this may seem (and sometimes is) annoying, having files that don't match even a `*` glob is very useful.

Frequently you want to have a file that sits alongside other files but that is generally ignored.

For example, you might write some code that reformats a set of text files in a folder, but you don't want to reformat a dotfile that contains information about what's in those text files.

Unfortunately, it can be annoying when you really do want to see *all* the files in a folder. To achieve this, type:

```
echo .*
```

Type the above code into the terminal in this lesson.

The leading dot tells bash that you really do want all the files to be matched.

If you do the same with `ls`:

```
ls .*
```

Type the above code into the terminal in this lesson.

you get a slightly more complicated output, as `ls` returns richer output depending on whether the item is a file or a folder. If it's a folder, it shows every file within that folder under a separate heading.

You'll also get a couple of extra 'special' folders returned that you may not have been aware of before.

The single dot folder (`.`) is a special folder that represents the folder that you are in. For example, if you type:

```
cd .
```

Type the above code into the terminal in this lesson.

You will go nowhere! You've changed directory to the same folder that you are in.

The double dot folder (`..`) is a special folder that represents the folder that represents the parent folder of the one you are in.

What do you think happens at the root folder (`/`)? Have a look and find out.

Differences with Regular Expressions

While globs look similar to regular expressions (regexes), they are used in different contexts and are separate things.

The * character in this command has a different significance depending on whether it is being treated as a glob or a regular expression.

```
rename -n 's/(.*)/new$1/' *
```

Type the above code into the terminal in this lesson.

You should see output like this:

```
rename(file1, newfile1)
rename(file2, newfile2)
rename(file3, newfile3)
```

- **Line 1** contains the command that renames all filenames to prepend ‘new’ in front. The `-n` flag tells rename to just print out the files that would be changed, and not actually carry out the renaming

The first `*` character is treated as regular expressions, because it is not interpreted by the shell, but rather by the `rename` command. The reason it is not interpreted by the shell is because it is enclosed in single quotes. The last `*` is treated as a glob by the shell, and expands to all the files in the local directory.

Again, the key takeaway here is that context is key.

Note that `.` has no meaning as a glob, and that some shells offer more powerful extended globbing capabilities. Bash is one of the shells that offers extended globbing, which we do not cover here, as it would potentially confuse the reader further. Just be aware that more sophisticated globbing is possible.

Globbing and Quoting Quiz

1

What will this command output in an empty folder?

```
ls *
```

COMPLETED 0%

1 of 2

What You Learned

- What a *glob* is
- What a *dotfile* is
- *Globs* and *regexes* are different
- Single and double quotes around *globs* can be significant!

What Next?

Next up is another fundamental topic: **variables**.

Exercises

1) Create a folder containing files with very similar names and use globs to list one and not the other.

2) Research regular expressions online.

3) Research the program `grep`. If you already know it, read the `grep` `man`

3) Research the program `grep`. If you already know it, read the `grep` `man` page. (Type `man grep`).