**Globbing and Quoting**

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 [#](https://www.educative.io/courses/master-the-bash-shell/xo6MRvjyjzz#globbing)

Type these commands into the terminal:

1

2

3

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

### Quoting [#](https://www.educative.io/courses/master-the-bash-shell/xo6MRvjyjzz#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!

1

2

3

4

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!

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

### Other glob characters [#](https://www.educative.io/courses/master-the-bash-shell/xo6MRvjyjzz#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:

1

2

3

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 [#](https://www.educative.io/courses/master-the-bash-shell/xo6MRvjyjzz#dotfiles)

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

1

2

3

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:

1

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?

1

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:

1

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:

1

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:

1

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 [#](https://www.educative.io/courses/master-the-bash-shell/xo6MRvjyjzz#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.

1

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.

### Basic Variables [#](https://www.educative.io/courses/master-the-bash-shell/NE8LYJ3zkM2#basic-variables)

Start by creating a variable and echoing it.

1

2

MYSTRING=astring

echo $MYSTRING





Type the above code into the terminal in this lesson.

Simple enough: you create a variable by

* Stating its name
* Immediately adding an equals sign
* Immediately stating the value

Variables don’t need to be capitalized, but they generally are by convention.

To get the value out of the variable, you have to use the dollar sign to tell bash that you want the variable dereferenced.

### Variables and Quoting [#](https://www.educative.io/courses/master-the-bash-shell/NE8LYJ3zkM2#variables-and-quoting)

Things get more interesting when you start quoting.

Quoting can be used to group different words into a single variable value:

1

2

3

MYSENTENCE=A sentence     # This command will not work

MYSENTENCE="A sentence"   # The quotes group the words together

echo $MYSENTENCE





Type the above code into the terminal in this lesson.

Since (by default) the shell reads each word separated by a space, it thinks the word sentence is not related to the variable assigment, and treats it as a program. To store a sentence with space in it inside a variable, you can enclose it in the double quotes, as shown above.

Things get even more interesting when we embed other variables in the quoted string:

1

2

3

4

MYSENTENCE="A sentence with $MYSTRING in it"

echo $MYSENTENCE                               # Outputs the sentence with the variable dereferenced

MYSENTENCE='A sentence with $MYSTRING in it'

echo $MYSENTENCE                               # Outputs the sentence with the variable not dereferenced





Type the above code into the terminal in this lesson.

If you were expecting similar behaviour to the [previous lesson](https://www.educative.io/collection/page/5164406595911680/5419374779301888/6261306883571712) you may have gotten a surprise!

This illustrated an important point if you’re reading shell scripts: the bash shell translates the variable into its value if it’s in double quotes, but does not if it’s in single quotes.

Remember from the previous lesson that this is not true when globbing!

Try out the code below in the terminal and see the output. Like always, make sure you think about the output you expect before you see it:

1

2

3

4

5

6

7

8

MYGLOB=\*        # No quotes around the glob

echo $MYGLOB    # Glob is interpreted

MYGLOB="\*"      # Double quotes around the glob

echo "$MYGLOB"  # Glob is not interpreted

MYGLOB='\*'      # Single quotes around the glob

echo "$MYGLOB"  # Glob is not interpreted

echo '$MYGLOB'  # Variable is not interpreted

echo $MYGLOB    # Glob is interpreted





Type the above code into the terminal in this lesson.

Globs are not expanded when in either single or double quotes. Confusing isn’t it?

### Shell Variables [#](https://www.educative.io/courses/master-the-bash-shell/NE8LYJ3zkM2#shell-variables)

Some variables are special, and set up when bash starts:

1

2

3

echo $PPID

PPID=nonsense

echo $PPID





Type the above code into the terminal in this lesson.

* **Line 1** - PPID is a special variable set by the bash shell. It contains the bash’s parent process id
* **Line 2** - Try and set the PPID variable to something else
* **Line 3** - Output PPID again

Can you work out what happened there?

You couldn’t set the variable, because this is a readonly variable.

If you want to make a variable read-only, put readonly in front of it, like this:

1

2

readonly MYVAR=astring

MYVAR=anotherstring





Type the above code into the terminal in this lesson.

### The export Command [#](https://www.educative.io/courses/master-the-bash-shell/NE8LYJ3zkM2#the-export-command)

Type in these commands, and try to predict what will happen:

1

2

3

4

5

6

7

8

9

10

11

MYSTRING=astring               # Set the MYSTRING variable

bash                           # Enter a fresh bash shell

echo $MYSTRING                 # Has the string made it to the new bash shell?

exit                           # Quit the bash shell

echo $MYSTRING                 # Is the string still there?

unset MYSTRING                 # Unset the string

echo $MYSTRING                 # Check it's not there

export MYSTRING=anotherstring  # Do the same, but export it this time

bash                           # Enter a fresh bash shell

echo $MYSTRING                 # Has it made it this time?

exit





Type the above code into the terminal in this lesson.

Based on this, what do you think export does?

You’ve already seen that a variable set in a bash terminal can be referenced later by using the dollar sign.

But what happens when you set a variable, and then start up another process?

In this case, you set a variable (MYSTRING) to the value astring, and then start up a new bash shell process. Within that bash shell process, MYSTRING does not exist, so an error is thrown. In other words, the variable was not inherited by the bash process you just started.

After exiting that bash session, and unsetting the MYSTRING variable to ensure it’s gone, you set it again, but this time export the variable, so that any processes started by the running shell will have it in their environment. You show this by starting up another bash shell, and it echoes the new value anotherstring to the terminal.

It’s not just shells that have environment variables! All processes have environment variables.

### Outputting Exported and Shell Variables [#](https://www.educative.io/courses/master-the-bash-shell/NE8LYJ3zkM2#outputting-exported-and-shell-variables)

Wherever you are, you can see the exported variables that are set by running env:

1

env





Type the above code into the terminal in this lesson.

The output of env will likely be different wherever you run it.

That isn’t all the variables that are set in your shell, though. It’s just the environment variables that are exported to processes that you start in the shell.

If you want to see all the variables that are available to you in your shell, type:

1

compgen -v





Type the above code into the terminal in this lesson.

compgen is a command that generates list of possible ‘word completions’ in bash when you hit tab repeatedly. The -v flag shows all the variables that could be completed from where you type. Hence, it lists all variables, exported and local to the shell, in the environment where you are.

### Arrays [#](https://www.educative.io/courses/master-the-bash-shell/NE8LYJ3zkM2#arrays)

Also worth mentioning here are arrays. One such built-in, read only array is BASH\_VERSINFO. As in other languages, arrays in bash are zero-indexed.

Type out the following commands, which illustrate how to reference the version information’s major number:

1

2

3

4

echo $BASH\_VERSINFO

echo $BASH\_VERSINFO[0]   # Does not work

echo ${BASH\_VERSINFO[0]} # Will work

echo ${BASH\_VERSINFO}





Type the above code into the terminal in this lesson.

Arrays can be tricky to deal with, and bash doesn’t give you much help!

The first thing to notice is that the array will output the item at the index 0 if no index is given.

The second thing to notice is that simply adding [0] to a normal variable reference does not work. Bash treats the square bracket as a character not associated with the variable and appends it to the end of the array.

You have to tell bash to treat the whole string BASH\_VERSION[0] as the variable to be dereferenced. You do this by using the curly braces.

These curly braces can be used on simple variables too:

1

2

echo $BASH\_VERSION\_and\_some\_string     # Won't find the variable

echo ${BASH\_VERSION}\_and\_some\_string   # The curly quotes tell bash where the variable is





Type the above code into the terminal in this lesson.

In fact, ‘simple variables’ can be treated as arrays with one element!

1

echo ${BASH\_VERSION[0]}





Type the above code into the terminal in this lesson.

So all bash variables are ‘really’ arrays!

Bash has 6 items (0-5) in its BASH\_VERSINFO array:

1

2

3

4

5

6

  echo ${BASH\_VERSINFO[1]}

  echo ${BASH\_VERSINFO[2]}

  echo ${BASH\_VERSINFO[3]}

  echo ${BASH\_VERSINFO[4]}

  echo ${BASH\_VERSINFO[5]}

  echo ${BASH\_VERSINFO[6]}





Type the above code into the terminal in this lesson.

As ever with variables, if the item does not exist then the output will be an empty line.

### Associative Arrays [#](https://www.educative.io/courses/master-the-bash-shell/NE8LYJ3zkM2#associative-arrays)

Bash also supports associative arrays.

With associative arrays, you use a string instead of a number to reference the value:

1

2

3

4

5

6

7

declare -A MYAA=([one]=1 [two]=2 [three]=3)  # declare the associative array

MYAA[one]="1"                                # set the 'one' array element

MYAA[two]="2"

echo $MYAA                                   # outputs nothing

echo ${MYAA[one]}                            # outputs '1'

WANT=two

echo ${MYAA[$WANT]}                          # outputs '2'





Type the above code into the terminal in this lesson.