**Here Docs**

Note: In this lesson, leading spaces are tabs. We have covered this in a [previous lesson](https://www.educative.io/collection/page/5164406595911680/5419374779301888/4584985879117824), but as a reminder: to get a tab character in your shell, type \C-v, and then hit the TAB button.

### How Important is this Lesson? [#](https://www.educative.io/courses/master-the-bash-shell/m7EzWVEjoxE#how-important-is-this-lesson)

**Here documents**, once learned, are used frequently in bash scripts or on the command line. It’s a piece of knowledge that separates the experienced user from the junior.

### Basic Here Docs [#](https://www.educative.io/courses/master-the-bash-shell/m7EzWVEjoxE#basic-here-docs)

Type this in to see the basic form of the here doc:

The first line starts with cat followed by a redirection to a file.

Then you use two left chevrons and follow that with a string that represents a marker for the end of the file’s contents. In this case, the word is END.

1

cat > afile.txt << END





Type the above code into the terminal in this lesson

Then you type in whatever you want the file to contain.

1

2

    A file can contain

            whatever you like





Type the above code into the terminal in this lesson.

(The whitespace in the second line above are spaces.)

When you’re done, you can finish the file by typing the string you used as a marker on the first line alone on its own line:

1

END





Type the above code into the terminal in this lesson.

You can check the contents of the file by just cating it:

1

cat afile.txt





Type the above code into the terminal in this lesson.

The marker word does not need to be END! It could be anything you choose. END is generally used as a convention. Sometimes you see EOF, or STOP, or something similar. If you have a document with END in it, for example, you might want to avoid problems with the document ending early by choosing a different word.

### More Advanced Here Docs [#](https://www.educative.io/courses/master-the-bash-shell/m7EzWVEjoxE#more-advanced-here-docs)

Now you‘re going to put a here document in a function. The function takes one argument. This argument is used as a filename, and the function creates a simple script with that filename that echoes the first argument given to that script.

Will this work? Read it carefully, predict the outcome, and then run it:

1

2

3

4

5

6

7

8

9

$ function write\_echoer\_file {

    cat > $1 << END

#!/bin/bash

echo $1

END

    chmod +x $1

}

write\_echoer\_file echoer

./echoer 'Hello world'





Type the above code into the terminal in this lesson.

* **Lines 1-7** creates a function called write\_echoer
* **Lines 2-5** creates a file with the name of the first argument to the write\_echoer\_file function that echoes its own name
* **Line 6** makes the newly-created file executable
* **Line 8** uses write\_echoer\_file to create a program called echoer
* **Line 9** tries to call the newly-created echoer program to output Hello world

Hmmm. That didn’t work, because the $1 got interpreted in the write\_echoer\_file function as being the filename we passed in. In the here doc, we wanted the $1 characters to be put into the script without being interpreted.

Try this instead:

1

2

3

4

5

6

7

8

9

function write\_echoer\_file {

    cat > $1 << 'END'

#!/bin/bash

echo $1

END

    chmod +x $1

}

write\_echoer\_file echoer

./echoer 'Hello world'





Type the above code into the terminal in this lesson.

Do you see the difference? This time, the delimiter word END was wrapped in single quotes. This made sure that the echo $1 was not interpreted by the shell when being typed in.

Can you see why we needed to use single quotes here? What happens when you use double quotes?

This kind of confusion can happen all the time when writing bash scripts, so it’s really important to get these differences clear in your mind.

Our function is working now, but we could still make it better.

Try this (remember, the leading spaces are tabs - see the note above for how to input a tab):

1

2

3

4

5

6

7

8

9

function write\_echoer\_file {

    cat > $1 <<- 'END'

        #!/bin/bash

        echo $1

        END

    chmod +x $1

}

write\_echoer\_file echoer

./echoer





Type the above code into the terminal in this lesson.

The difference there was that we added a hyphen after the << in **line 2**. This tells the shell to ignore leading tabs in the following text. This makes it much neater and easier to read scripts that use here docs, as the indentation can be consistent.

What if END is part of the here doc within another line?

1

2

3

4

5

6

7

8

9

10

function write\_echoer\_file {

    cat > $1 <<- 'END'

        #!/bin/bash

        echo $1

        echo Is this the END?

        END

    chmod +x $1

}

write\_echoer\_file echoer

./echoer





Type the above code into the terminal in this lesson.

No problem if it is not the only thing on the line.

Experiment to try and see what happens if it is.

### Here Strings [#](https://www.educative.io/courses/master-the-bash-shell/m7EzWVEjoxE#here-strings)

Related to the here doc, a here string can be applied in the same way with the <<< operator:

1

2

3

4

function write\_here\_string\_to\_file {

    cat > $1 <<< $2

}

write\_here\_string\_to\_file afile.txt "Write this out"





Type the above code into the terminal in this lesson.

**Bash History**

### How Important is this Lesson? [#](https://www.educative.io/courses/master-the-bash-shell/qV2Evqz28V7#how-important-is-this-lesson)

Bash’s history features are used at the command line so often that it’s difficult to understate how important they are. It’s a rich subject, but here I keep to the features I use most of the time.

### Bash and History [#](https://www.educative.io/courses/master-the-bash-shell/qV2Evqz28V7#bash-and-history)

Bash keeps a history of commands you have run. It keeps this in memory.

1

history





Type the above code into the terminal in this lesson.

### Using Your History [#](https://www.educative.io/courses/master-the-bash-shell/qV2Evqz28V7#using-your-history)

It can be tedious to type out often-used commands and arguments again and again, so bash offers several ways to save your effort.

Type this out and try and figure out what is going on:

1

2

3

4

5

6

7

8

9

10

11

echo nowhere

cd !$                              # !$ is replaced by 'nowhere', the last argument to the previous command

echo 'About bash history' > file1

echo 'Another file' > file2

grep About file1

!!                                 # Repeat the last command

grep About file2

grep Another !$

rm file2

!e                                 # Repeat the last command beginning with 'e'

!gr                                # Repeat the last command beginning with 'gr'





Type the above code into the terminal in this lesson.

That introduced a few tricks you haven’t necessarily seen before.

All of them start with the ! (or so-called bang) sign, which is the sign used to indicate that the bash history is being referred to.

* The simplest, and most frequently seen is the double bang !!, which just means: re-run the previous command
* The one I use most often, though, is the second one you come across in the listing above: !$, or bang dollar. This one I must use dozens of times every day. It tells bash to re-use the last argument of the previous command.
* Finally, a bang followed by ‘normal’ characters re-runs the last command that matches those starting letters. The !e looks up the last command that ran starting with an e and runs that. Similarly, the !gr runs the last command that started with a gr, ie the grep.

Notice that the command that’s rerun is the evaluated command. For that grep, what is re-run is as though you typed: grep Another file2, and not: grep Another !$.

### How to Learn History Shortcuts [#](https://www.educative.io/courses/master-the-bash-shell/qV2Evqz28V7#how-to-learn-history-shortcuts)

The history items above are enough to be going on with if you’ve not seen them before. There‘s little point listing them all as you‘ll likely forget them before you finish this course.

So before you go on, a quick note about learning these things: it’s far more important to learn to use these tricks than understand them. To understand them is pretty easy - I’m sure you understood the passage above without much difficulty.

The way to learn these is to ‘get them under your fingers’ to the point where you don’t even think about it. The way I recommend to do that is to concentrate on one of them at a time, and as you’re working, remember to use that one where appropriate. Gradually you’ll add more and more to your repertoire, and you will soon look like a whizz at the terminal.

### More Advanced History Usage [#](https://www.educative.io/courses/master-the-bash-shell/qV2Evqz28V7#more-advanced-history-usage)

You might want to stop there, as trying to memorise/learn much more in one go can be overwhelming.

But there are many more tricks to learn like this in bash, so I’m going to lay them out now so you might return to them later when you’re ready.

Carrying on from where you left off above:

1

2

grep Abnother file1

^Ab^a^





Type the above code into the terminal in this lesson.

The carets (^) are used to replace a string from the previous command. In this case, Ab is replaced with: a. This is often handy if you made a spelling mistake.

Next up are the position command shortcuts, or ‘word designators’:

1

2

3

grep another file1 | wc -l

# Is that output correct? I want to check the file by eye:

cat !:2





Type the above code into the terminal in this lesson.

Starting with the bang sign to indicate we’re referring to the history, there follows a colon. Then, you specify the word with a number. The numbers are zero-based, so the arguments start with 1:

1

2

grep another file1

fgrep !:1-$          # A more sophisticated 'word designator'





Type the above code into the terminal in this lesson.

In the above example, you want to run the same command as before, but use the fgrep command instead of grep (fgrep is a ‘faster’ grep, which doesn’t really help us here, but is just an example). To achieve this you use the so-called ‘word designators’.

Here you add a dash indicating you want a ‘range’ of words, and the $ sign indicates we want all the arguments up to the end of the previous command. Recall that !$ means give me the last argument from the previous command, and so is itself a shortcut for !:$

Finally, another trick I use all the time:

1

2

3

4

LGTHWDIR=$(PWD)

cd /tmp

cat ${LGTHWDIR}/file1

cd !$:h





Type the above code into the terminal in this lesson.

The trick is the :h modifier to the !$ history shortcut on **line 4**. This is one of several modifiers available, but the only one I regularly use.

When using a history shortcut, you can place a modifier at the end that starts with a colon. Here, the

* !$ takes the last word from the previous command, (which you set to full directory path to the freshly-created file1 file)
* Then, the modifier :h strips off the file at the end, leaving just the directory name. I use this all the time to quickly hop into a folder of a file I just looked at

### History Env Vars [#](https://www.educative.io/courses/master-the-bash-shell/qV2Evqz28V7#history-env-vars)

A quick note on environment variables that affect the history kept.

Type this in and try and figure out what’s going on:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

bash

HISTFILE=~/.bash\_history

HISTTIMEFORMAT="%d/%m/%y %T "

history | tail

HISTTIMEFORMAT="%d/%m/%y "

history | tail

HISTSIZE=2

ls

pwd

history | tail

tail ~/.bash\_history

exit

history

tail ~/.bash\_history





Type the above code into the terminal in this lesson.

* The HISTFILE variable (**line 2**) sets the file that command history will be saved to when the shell exits
* The HISTSIZE variable (**line 7**) must be set to the number of commands you want stored in the HISTFILE file when bash exits
* There is also a HISTFILESIZE variable which determines the size of the history file itself. I did not get you to reduce the size of this to `1’, as it would have wiped your history file and that would be more confusing! But you can play with it if you want
* Finally, the HISTTIMEFORMAT (**lines 3 and 5**) determines what time format should be shown with the bash history item. By default it’s unset, so I usually set mine everywhere to be %d/%m/%y %T

You should have noticed that the ~/.bash\_history file did not get updated with the ls and pwd commands until bash exited. It’s a common source of confusion that the bash history is not written out until you exit. If your terminal connection freezes, your history from that session may never be written out. This frequently annoys me!

### History Control [#](https://www.educative.io/courses/master-the-bash-shell/qV2Evqz28V7#history-control)

There’s another history-controlling environment variable worth understanding:

1

2

3

4

5

6

7

HISTCONTROL=ignoredups:ignorespace

ls

ls

 pwd    # <- note the space before the 'pwd'

pwd

ls

history | tail





Type the above code into the terminal in this lesson.

Was the output of history what you expected? HISTCONTROL can determine what gets stored in your history. The directives are separated by colons. Here we use ignoredups to tell history to ignore commands that are repeats of the last-recorded command. In the above input, the two consecutive ls commands are combined into one in the history. If you want to be really severe about your history, you can also use erasedups, which adds your latest command to the history, but then wipes all previous examples of the same command out of the history. What would this have done to the history output above?

ignorespace tells bash to not record commands that begin with a space, like the pwd in the listing above.

### CTRL-r [#](https://www.educative.io/courses/master-the-bash-shell/qV2Evqz28V7#ctrl-r)

Bash offers you another means to use your history.

Hit CTRL and hold it down. Then hit the ‘r’ key. You should see this on your terminal:

(reverse-i-search)`':

Let go. Now type grep. You should see a previous grep command. If you keep hitting CTRL+r you will cycle through all commands that had grep in them, most recent first.

If you want to cycle forward (if you hit CTRL+r too many times and go past the one you want (I do this a lot)), hit CTRL+s.

**Putting It Together - Some Useful Examples**

### How Important is this Lesson? [#](https://www.educative.io/courses/master-the-bash-shell/mENXkMnNkp3#how-important-is-this-lesson)

You can easily skip this lesson if you want as nothing here is new, but following this can embed some concepts and keep your motivation going before the final part!

### Output With Time [#](https://www.educative.io/courses/master-the-bash-shell/mENXkMnNkp3#output-with-time)

Frequently, I want to get the output of a command along with the time. Let’s say I’m running vmstat while a server is having problems, and I want to ensure that I’ve logged the time that vmstat relates to. Type this in:

1

2

3

4

5

6

7

function dateit() {

    while read line

    do

        echo "$line $(date '+ %m-%d-%Y %H:%M:%S')"

    done

}

vmstat 1 | dateit





Type the above code into the terminal in this lesson.

Note: vmstat is a program available on most Linux flavours that gives you a view of what the system resource usage looks like. It is not available on Mac OSes. If vmstat does not exist for you, then replace with vm\_stat, which should be available.

You should be able to follow what’s happening there based on what you’ve learned so far in

* [Functions](https://www.educative.io/collection/page/5164406595911680/5419374779301888/6423084577849344)
* [Loops](https://www.educative.io/collection/page/5164406595911680/5419374779301888/6109281717846016)
* [Command Substitution](https://www.educative.io/collection/page/5164406595911680/5419374779301888/5633923864330240/draft)
* [Pipes and Redirects](https://www.educative.io/collection/page/5164406595911680/5419374779301888/4797268530036736/draft)

The one exception is the while read line line. read is a shell builtin that takes input from standard input until a newline is seen (or an ‘end of file’ byte on its own line).

1

2

read

some input





Type the above code into the terminal in this lesson.

It returns a true exit code if input was seen:

1

echo $?





Type the above code into the terminal in this lesson.

and false if an end of file character is seen (with \C-d):

1

2

3

read

^D

echo $?





Type the above code into the terminal in this lesson.

If an argument is given to read, then a variable is populated with the input:

1

2

3

read myinput

some input

echo $myinput





Type the above code into the terminal in this lesson.

So in the dateit function we input above, the while loop keeps taking input from standard input and acts on each line as it’s read in, echoing each line to standard output until is gets an end-of-file character, at which point the while loop terminates as read returned a false exit code.

You will see the date appended to each line in the output. Experiment with the function to place the date before the line, or even on a separate line. See also the exercises below.

### Where Am I? [#](https://www.educative.io/courses/master-the-bash-shell/mENXkMnNkp3#where-am-i)

You may be familiar with the pwd builtin, which gives you your current working directory (cwd). Similarly, there is an environment variable (PWD) that bash normally sets that stores the cwd.

1

2

pwd

echo $PWD





Type the above code into the terminal in this lesson.

Very often in scripts, you will want to know where the current working directory of the process is.

But also (very often) you will want to know where the script you are running is located from within the script.

For example, if you are running a script that is found in your PATH (ie not in your local folder), and you want to refer to another file relative to that script from within that script, then you will need to know where that script is located.

1

2

3

4

5

6

cat > /tmp/pwdscript.sh << 'EOF'

echo My pwd is: $PWD

echo I am running in: $(dirname ${BASH\_SOURCE[0]})

EOF

chmod +x /tmp/pwdscript.sh

/tmp/pwdscript.sh





Type the above code into the terminal in this lesson.

Have a play with this command and read the code over to see what it does. If you can’t work something out try the hint below.

Show Hint

Was the output of history what you expected? HISTCONTROL can determine what gets stored in your history. The directives are separated by colons. Here we use ignoredups to tell history to ignore commands that are repeats of the last-recorded command. In the above input, the two consecutive ls commands are combined into one in the history. If you want to be really severe about your history, you can also use erasedups, which adds your latest command to the history, but then wipes all previous examples of the same command out of the history. What would this have done to the history output above?

ignorespace tells bash to not record commands that begin with a space, like the pwd in the listing above.

### CTRL-r [#](https://www.educative.io/courses/master-the-bash-shell/mENXkMnNkp3#ctrl-r)

Bash offers you another means to use your history.

Hit CTRL and hold it down. Then hit the r key. You should see this on your terminal:

(reverse-i-search)`':

Let go. Now type grep. You should see a previous grep command. If you keep hitting CTRL+r you will cycle through all commands that had grep in them, most recent first.

If you want to cycle forward (if you hit CTRL+r too many times and go past the one you want (I do this a lot)), hit CTRL+s.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

function extract() {

    if [ -z "$1" ]

    then

        echo "Usage: extract <file\_name>.<zip|rar|bz2|gz|tar|tbz2|tgz|Z|7z|xz|ex|tar.bz2|tar.gz|tar.xz>"

    else

        if [ -f $1 ] ; then

           case $1 in

             \*.7z)        7z x $1        ;;

             \*.bz2)       bunzip2 $1     ;;

             \*.exe)       cabextract $1  ;;

             \*.gz)        gunzip $1      ;;

             \*.tar.bz2)   tar xvjf $1    ;;

             \*.tar.gz)    tar xvzf $1    ;;

             \*.tar.xz)    tar xvJf $1    ;;

             \*.tar)       tar xvf $1     ;;

             \*.tbz2)      tar xvjf $1    ;;

             \*.tgz)       tar xvzf $1    ;;

             \*.Z)         uncompress $1  ;;

             \*.xz)        unxz $1        ;;

             \*.lzma)      unlzma $1      ;;

             \*.rar)       unrar x -ad $1 ;;

             \*.zip)       unzip $1       ;;

             \*)           echo "extract: '$1' - unknown archive method" ;;

           esac

        else

            echo "$1 - file does not exist"

        fi

    fi

}

mkdir lbthw\_misc

cd lbthw\_misc





Type the above code into the terminal in this lesson.

Challenge: explain what each line does.

You should be able to do this based on the lessons on:

* [Tests](https://www.educative.io/collection/page/5164406595911680/5419374779301888/4990562191015936)
* [Functions](https://www.educative.io/collection/page/5164406595911680/5419374779301888/6423084577849344)
* [Loops](https://www.educative.io/collection/page/5164406595911680/5419374779301888/6109281717846016)

### Output Absolute File Path [#](https://www.educative.io/courses/master-the-bash-shell/mENXkMnNkp3#output-absolute-file-path)

Quite often I want to give co-workers an absolute reference on a server to a file that I am looking at. One way to do this is to cut and paste the output of pwd, add a / to it, and then add the filename I want to share.

This takes a few seconds to achieve, and since it happens regularly, it’s a great candidate for a time-saving function:

1

2

3

4

5

6

7

8

function sharefiles() {

    for file in $(ls "$@"); do

        echo -n $(pwd)

        [[ $(pwd) != "/" ]] && echo -n /

        echo $file

    done

}

sharefiles





Type the above code into the terminal in this lesson.

You should be able to work out what is going on in the above function based on the lessons:

* [Functions](https://www.educative.io/collection/page/5164406595911680/5419374779301888/6423084577849344)
* [Loops](https://www.educative.io/collection/page/5164406595911680/5419374779301888/6109281717846016)
* [Variables](https://www.educative.io/collection/page/5164406595911680/5419374779301888/5676808240365568)

The only novel element is the special parameter $@. This (very useful) variable gives you all the arguments passed to the function (or program, if called outside a function).

Saving time by writing a function is often a great idea, but deciding what is worth automating is a non-trivial task.

Here are some things you want to think about when deciding what to automate:

* How often do you perform the task?
* How much effort is it to automate (it’s easy to under-estimate this!)
* Will the automation require effort to maintain?
* Do you always perform the task on the same machine?
* Do you control that machine?

My experience is that the effects of automation can be very powerful, but the above factors can also limit the return on investment.

Think about what you could automate today!