Learn Bash The Hard Way

An introduction to Bash using 'The Hard Way' method.

Chapter 1. Core Git

This bash course has been written to help bash users to get to a deeper understanding and proficiency in bash. It doesn't aim to make you an expert immediately, but you will be more confident about using bash for more tasks than just one-off commands.

1.1. Assumptions

It assumes some familiarity with very basic shell usage and commands. For those looking to get to that point, I recommend following this set of mini-tutorials:

https://learnpythonthehardway.org/book/appendixa.html

It also assumes you are equipped with a bash shell and a terminal. If you're unsure whether you're in bash, type:

```
echo $BASH_VERSION
```

into your terminal. If you get a bash version string output then you in bash.

In this book, listings have a '$' sign

1.2. How The Course Works

The course *demands* that you type out all the exercises to follow it.

Frequently, the output will not be shown in the text, or even described.

Explanatory text will assume you typed it out.

This is really important: you must get used to working in bash, and figuring out what's going on by scratching your head and trying to work it out before I explain it to you.

Each section is self-contained, and must be followed in full. To help show you where you are, the shell commands are numbered 1-n, eg:

```
1$ #first command
2$ #second command
```

At the end of each section is a set of 'cleanup' commands (where needed) if you want to use them.

1.3. What You Will Get

This course aims to give students:

- A hands-on, practical understanding of bash
- Enough information to understand what is going on as they go deeper into bash

- A familiarity with advanced bash usage

1.4. Structure TODO

1.5. Introduction

1.5.1. What is bash?

Bash is a shell program.

A shell program is typically an executable binary that takes commands that you type and (once you hit return), translates those commands into (ultimately) system calls to the Operating System API.

1.5.2. Other shells

Other shells include:

- sh
- ash
- dash
- ksh
- csh
- tclsh

These other shells have different rules, conventions, logic, and histories that means they can look similar.

Because other shells are also programs, they can be run from in each other!

Here you run csh from within your bash terminal. Note that you get a different prompt (by default):

```
1$ csh
2%
```

Typically, a csh will give you a prompt with a percent sign, while bash will give you a prompt with a dollar sign. This is configurable, though, so your setup may be different.

1.5.3. History of bash

This diagram helps give a picture of the history of bash:

[shell history] | *diagrams/shell_history.png*

Bash is called the 'Bourne Again SHell'. It is a descendant of the Thompson Shell and then the Bourne 'sh' shell. Bash has other 'siblings' (eg ksh), 'cousins' (eg tcsh), and 'children', eg 'zsh'.

The details aren't important, but it's important to know that different shells exist and they can be

related and somewhat compatible.

Bash is the most widely seen and used shell as of 2017. However, it is still not uncommon to end up on servers that do not have bash!

1.5.4. What you learned

- What a shell is
- How to start up a different shell
- The family tree of shells

1.5.5. Exercises

1) Run 'sh' from a bash command line. What happens? 1) What commands can you find that are work in 'bash', but do not work in 'sh'?

1.6. Unpicking the shell: Globbing and Quoting

You may have wondered what the '*' in bash commands really means, and how it is different from regexps. This section will explain all, and introduce you to the joy of quoting in bash.

1.6.1. Globbing

Type these commands into your terminal

```
1$ mkdir lbthw_tmp           # Line 1
2$ cd lbthw_tmp              # Line 2
3$ touch file1 file2 file3   # Line 3
4$ ls *                      # Line 4
5$ echo *                    # Line 5
```

- Line 1 above makes a new folder that should not exist already.
- Line 2 moves into that folder.
- Line 3 creates three files (file1,2,3).
- Line 4 runs the 'ls' command, which lists files, asking to list the files matching '*'
- Line 5 runs the echo command using '*' as the argument to echo

What you should have seen was the three files listed in both cases.

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

1.6.2. Quoting

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

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

```
6$ ls '*'         # Line 1
7$ ls "*"         # Line 2
8$ echo '*'       # Line 3
9$ echo "*"       # Line 4
```

- Line 1 lists files matching the '*' character in single quotes
- Line 2 lists files matching the '*' character in double quotes
- Line 3 'echo's the '*' character in single quotes
- Line 4 'echo's the '*' character in double quotes

This is difficult even if you are an expert 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. You may want to take from this that quoting globs removes their effect. But in other contexts single and double quotes have different meanings.

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.

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

1.6.3. 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:

```
10$ ls *1             # Line 1
11$ ls file[a-z]      # Line 2
12$ ls file[0-9]      # Line 3
```

- Line 1 list all the files that end in '1'
- Line 2 list all files that start with 'file' and end with a character from a to z

- Line 3 list all files that start with 'file' and end with a character from 0 to 9

Differences with regexes

While globs look similar regexes, they are used in different contexts and are separate things.

The '*' characters in this command:

```
13$ rename -n 's/(.*)(.*)/new$1$2/' *      # Line 1
'file1' would be renamed to 'newfile1'   # Line 2
'file2' would be renamed to 'newfile2'   # Line 3
'file3' would be renamed to 'newfile3'   # Line 4
```

- Line 1 prints the files that would be renamed by the rename command if the -n flag were removed
- Lines2-4 show the files that would be renamed

have a different significance

NOTE	this assumes you have the program 'rename' installed.

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

Note that '.' has no meaning as a glob, and that some shells offer extended globbing capabilities. Bash offers 'extended globbing', which we do not cover here.

Cleanup

Now clean up what you just did:

```
14$ cd -                      # Line 1
15$ rm -rf lbthw_tmp          # Line 2
```

1.6.4. What you learned

- What a glob is
- Globs and regexes are different
- Single and double quotes around globs can be important

1.6.5. Exercises

1) TODO

1.7. Variables in Bash

As in any programming environment, variables are critical to an understanding of bash. In this

section you'll learn about variables in bash and some of their subtleties.

1.7.1. Basic Variables

Start by creating a variable and echoing it.

```
1$ MYSTRING=astring
2$ echo $MYSTRING
```

Simple enough: you create a variable by stating its name, immediately adding an equals sign, and then immediately stating the value.

Variables don't need to be capitalised, 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.

1.7.2. Variables and Quoting

Things get more interesting when you start quoting.

Quoting used to group different 'words' into a variable value:

```
3$ MYSENTENCE=A sentence
4$ MYSENTENCE="A sentence"
5$ echo $MYSENTENCE
```

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

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

```
6$ MYSENTENCE="A sentence with $MYSTRING in it"
7$ MYSENTENCE='A sentence with $MYSTRING in it'
```

If you were expecting similar behaviour to the previous section you may have got 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 section that this is not true when globbing! Globs are not expanded when in either single or double quotes. Confusing isn't it?

1.7.3. Shell Variables

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

```
8$ echo $PPID          # Line 1
9$ PPID=nonsense       # Line 2
10$ echo $PPID          # Line 3
```

- 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.

What happened there?

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

```
11$ readonly MYVAR=astring
12$ MYVAR=anotherstring
```

1.7.4. env

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

```
13$ env
TERM_PROGRAM=Apple_Terminal
TERM=xterm-256color
SHELL=/bin/bash
HISTSIZE=1000000
TMPDIR=/var/folders/mt/mrfvc55j5mg73dxm9jd3n4680000gn/T/
PERL5LIB=/home/imiell/perl5/lib/perl5
GOBIN=/space/go/bin
Apple_PubSub_Socket_Render=/private/tmp/com.apple.launchd.2BE31oXVrF/Render
TERM_PROGRAM_VERSION=361.1
PERL_MB_OPT=--install_base "/home/imiell/perl5"
TERM_SESSION_ID=07101F8B-1F4C-42F4-8EFF-1E8003E8A024
HISTFILESIZE=1000000
USER=imiell
SSH_AUTH_SOCK=/private/tmp/com.apple.launchd.uNwbe2XukJ/Listeners
__CF_USER_TEXT_ENCODING=0x1F5:0x0:0x0
PATH=/home/imiell/perl5/bin:/opt/local/bin:/opt/local/sbin:/Users/imiell/google-cloud-
sdk/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/X11/bin:/space/git/shutit:/s
pace/git/work/bin:/space/git/home/bin:~/.dotfiles/bin:/space/go/bin
PWD=/space/git/work
LANG=en_GB.UTF-8
XPC_FLAGS=0x0
HISTCONTROL=ignoredups:ignorespace
XPC_SERVICE_NAME=0
HOME=/Users/imiell
SHLVL=2
PERL_LOCAL_LIB_ROOT=/home/imiell/perl5
LOGNAME=imiell
GOPATH=/space/go
DISPLAY=/private/tmp/com.apple.launchd.lwUJWwBy9y/org.macosforge.xquartz:0
SECURITYSESSIONID=186a7
PERL_MM_OPT=INSTALL_BASE=/home/imiell/perl5
HISTTIMEFORMAT=%d/%m/%y %T
HISTFILE=/home/imiell/.bash_history
_=/usr/bin/env
OLDPWD=/Users/imiell/Downloads
```

The output will be different wherever you run it.

1.7.5. export

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

```
14$ MYSTRING=astring
15$ bash
16$ echo $MYSTRING
17$ exit
18$ echo $MYSTRING
19$ unset MYSTRING
20$ echo $MYSTRING
21$ export MYSTRING=anotherstring
22$ bash
23$ echo $MYSTRING
24$ exit
```

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 'echo's the new value "anotherstring" to the terminal.

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

1.7.6. Arrays

Worth mentioning here also 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:

```
25$ bash --version
26$ echo $BASH_VERSINFO
27$ echo $BASH_VERSINFO[0]
28$ echo ${BASH_VERSINFO[0]}
29$ echo ${BASH_VERSINFO}
```

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

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

The second thing to notice is that simply adding [0] to a normal array 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:

```
30$ echo $BASH_VERSION_and_some_string
31$ echo ${BASH_VERSION}_and_some_string
```

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

```
32$ echo ${BASH_VERSION[0]}
```

So all bash variables are 'really' arrays!

Bash has 6 items (0-5) in its BASH_VERSINFO array:

```
33$ echo ${BASH_VERSINFO[1]}
34$ echo ${BASH_VERSINFO[2]}
35$ echo ${BASH_VERSINFO[3]}
36$ echo ${BASH_VERSINFO[4]}
37$ echo ${BASH_VERSINFO[5]}
38$ echo ${BASH_VERSINFO[6]}
```

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

1.7.7. What you learned

- TODO

1.7.8. Exercises

1) Take the output of 'env' in your shell and work out why each item is there and what it might be used by. You may want to use 'man bash', or use google to figure it out. Or you could try re-setting it and see what happens.

2) Find out what the items in BASH_VERSINFO mean.

1.8. Functions in Bash

From one angle, bash can be viewed as a programming language, albeit a quite slow and primitive one.

One of the language features it has are the capability to create and call functions.

This leads us onto the topic of what a 'command' can be in bash: function, alias, program or 'builtin'.

1.8.1. Basic Functions

Start by creating a simple function

```
1$ function myfunc {
    echo Hello World
}
2$ myfunc
```

By declaring a function, and placing the block of code that needs to run inside curly braces, you can then call that function on the command line as though it were a program.

1.8.2. Arguments

Unlike other languages there is no checking of functions' arguments.

Predict the output of this, and then run it:

```
3$ function myfunc {
    echo $1
    echo $2
}
4$ myfunc "Hello World"
5$ myfunc Hello World
```

Can you explain the output?

Arguments to functions are numbered, from 1 to n. It’s up to the function to manage these arguments.

1.8.3. Variable Scope

Variables can have scope in bash. This is particularly useful in functions, where you don’t want your variables to be accessible from outside the function.

These commands illustrate this:

```
6$ function myfunc {
    echo $myvar
}
7$ myfunc
8$ myvar="Hi from outside the function"
9$ myfunc
```

Bash functions have no special scope. variables outside are visible to it.

There is, however, the capability within bash to declare a variable as local:

```
10$ function myfunc {
    local myvar="Hi from inside the function"
    echo $myvar
}
11$ myfunc
12$ echo $myvar
13$ local myvar="Will this work?"
```

The variable declared with 'local' is only viewed and accessed within the function, and doesn't interfere with the outside.

'local' is an example of a bash 'builtin'. Now is a good time to talk about the different types of commands.

1.8.4. Functions, Builtins, Aliases and Programs

Now is a good time to point out another area of bash which can cause confusion.

There are at least 4 ways to call commands in bash:

- Builtins
- Functions
- Programs
- Aliases

Let's take each one in turn.

Builtins

Builtins are commands that come 'built in' to the bash shell program. Normally you can't easily tell the difference between a builtin, a program or a function, but after reading this you will be able to.

Two such builtins are the familiar 'cd' and one called 'builtin'!

```
$ builtin cd /tmp
$ cd -
$ builtin grep
$ builtin notaprogram
```

As you've probably guessed from typing the above in, the 'builtin' builtin calls the builtin program (this time 'cd'), and throws an error if no builtin exists

In case you didn't know 'cd -' returns you to the previous directory you were in.

Functions

Functions we have covered above, but what happens if we write a function that clashes with a builtin? What if you create a function called 'cd'?

```
$ function cd() {
    echo 'No!'
}
$ cd /tmp
$ builtin cd /tmp
$ cd -
$ unset -f cd
$ cd /tmp
$ cd -
```

At the end there you 'unset' the function 'cd'. 'unset' is also . You can also 'unset -v' a variable. Or just leave the -v out, as it will assume you mean a variable by default.

```
$ declare -f
$ declare -F
```

If you want to know what functions are set in your environment, run 'declare -f'. This will output the functions and their bodies, so if you just want the names, use the '-F' flag.

Programs

which TODO

Aliases

alias unalias

and yes, you can alias alias.

1.8.5. What you learned

- TODO

1.8.6. Exercises

1) TODO 2) Run typeset -f 3) alias alias, override cd. Try and break things. Have fun. If you get stuck, close down your terminal, or exit your bash shell (if you haven't overridden exit!).

1.9. Pipes and redirects

Pipes are redirects are used very frequently in bash. This can cause a problem in that they are used so often by all users of bash that many don't understand their subtleties or how their full power.

This section will lay a firm foundation for you to understand these concepts as we move onto deeper bash topics.

1.9.1. Basic Redirects

Start off by creating a file:

```
1$ mkdir lbthw_pipes_redirects
2$ cd lbthw_pipes_redirects
3$ echo "contents of file1" > file1
```

1.9.2. Basic pipes

Type this in:

```
4$ cat file1 | grep -c file
```

NOTE if you don't know what grep is, you will need to learn. This is a good place to start: https://en.wikipedia.org/wiki/Grep

Normally you'd run a grep with the filename as the last argument, but instead here we 'pipe' the contents of file into the grep command by using the 'pipe' operator: '|'.

A pipe takes the standard output of one command and passes it as the input to another. What, then is standard output, really? You will find out soon!

```
5$ cat file2
```

What was the output of that?

Now run this, and try and guess the result before you run it:

```
6$ cat file2 | grep -c file
```

Was that what you expected? If it was, you can explain to the rest of the class :)

If it wasn't, then the answer is related to standard output and other kinds of output.

1.9.3. Standard output vs standard error

In addition to 'standard output', there is also a 'standard error' channel. When you pass a non-existent file to the cat command, it throws an error message out to the terminal. Although the message looks the same as the contents of a file, it is in fact sent to a different output channel. In this case it's 'standard error' rather than 'standard output'.

As a result, it is NOT passed through the pipe to the grep command, and grep counts 0 matches in its output.

To the viewer of the terminal, there is no difference, but to bash there is all the difference in the world!

There is a simpler way to refer to these channels. A number is assigned to each of them by the operating system.

These are the numbered 'file descriptors', and the first three are assigned to the numbers 0,1 and 2.

- 0 is 'standard input'
- 1 is 'standard output'
- 2 is 'standard error'

When you redirect standard output to a file, you use the redirection operator '>'. Implicitly, you are using the '1' file descriptor.

Type this to see an example of redirecting '2', which is 'standard error'.

```
7$ command_does_not_exist
8$ command_does_not_exist 2> /dev/null
```

In the second line above 2 ('standard error') is directed to a file called /dev/null

/dev/null is a special file, created by UNIX kernels. It is effectively a black hole into which data can be pumped: anything written to it will be absorbed and ignored.

Another commonly seen redirection operator is '2>&1'.

```
9$ command_does_not_exist 2>&1
```

What this does is tell the shell to send the output on standard error (2) to whatever standard output is pointed at at that point in the command.

Since standard output is pointed at the terminal at that time, standard error is pointed at the terminal. From your point of view you see no difference, since by default . But when we try and redirect to files things get interesting.

Now type these in and try and figure out why they produce different output:

```
10$ command_does_not_exist 2>&1 > outfile
11$ command_does_not_exist > outfile 2>&1
```

This is where things get tricky and you need to think carefully!

Remember that the redirection operator 2>&1 points (standard error) at whatever 1 (standard

output) was pointed to at the time. If you read the first line carefully, at the point 2>&1 was used, standard output was pointed at the terminal. So standard error is pointed at the terminal. After that point, standard output is redirected (with the '>' operator) to the file 'outfile'.

So at the end of all this, the standard error of the output of the command 'command_does_not_exist' points at the terminal, and the standard output points at the file 'outfile'.

In the second line, what is different?

The order of redirections is changed. Now, the standard output of the command 'command_does_not_exist' is pointed at the file 'outfile', and the redirection operator 2>&1 points 2 (standard error) to whatever 1 (standard output) is pointed at. So in effect, both standard out and standard error are pointed at the file

This pattern of sending all the output to a single file is seen very often, and few understand why it has to be in that order. Once you understand, you will never pause to think about which way round the operators should go again!

1.9.4. Difference between pipes and redirects

To recap:

- A pipe passes 'standard output' as the 'standard input' to another command
- A redirect sends a channel of output to a file

A couple of other commonly used operators are worth mentioning here:

```
12$ grep -c file < file1
```

The '<' operator redirects standard input to the command from a file, in this case just as 'cat file1 | grep -c file' did.

```
13$ echo line1 > file3
14$ echo line2 > file3
15$ echo line3 >> file3
16$ cat file3
```

The first two lines above use the '>' operator, while the third one uses the '>>' operator. The '>' operator effectively creates the file anew whether it already exists or not. The '>>' operator, by contrast, appends to the end of the file. As a result, only line2 and line3 are added to file3.

Cleanup

Now clean up what you just did:

```
17$ cd -
18$ rm -rf lbthw_pipes_redirects
```

1.9.5. What you learned

- TODO

1.9.6. Exercises

1) TODO

1.10. Scripts and Startups

TODO

1.10.1. Shell scripts

Cleanup

Now clean up what you just did:

```
$ cd -
$ rm -rf lbthw_scripts_and_startups
```

1.10.2. What you learned

-

1.10.3. Exercises

1) TODO