Bash History

This lesson gives you a pragmatic overview of bash's history features, which can save you lots of time when at the terminal. You will learn about where your history is stored, how previous commands can be referenced, and various options and variables that can be set when working with your shell history.

WE'LL COVER THE FOLLOWING

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

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

```
history
```

Type the above code into the terminal in this lesson.

Terminal

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:

```
echo nowhere
cd !$                              # !$ is replaced by 'nowhere', the last argument to the pr
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

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

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:

```
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':

```
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`:

```
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:

```
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

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

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

```
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

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

```
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`

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` one you want (I do this a lot)), hit `CTRL+S`.

History Quiz

1 What does this shortcut do?

`!$`

COMPLETED 0%

1 of 2

What You Learned

- Where bash keeps a *history* of commands
- How to refer to previous commands
- How to re-run a previous command with simple adjustments
- How to pick out specific arguments from the previous command
- How to control the *history* output
- How to control the commands that are added to the *history*
- How to search through your *history* dynamically

What Next?

Next you will tie these things together in a series of useful scripts that finish off this part.

Exercises

1) Remember to use one of the above practical tips every day until you don't think about using it. Then learn another one.

2) Read up on all the history shortcuts. Pick ones you think will be useful.

3) Amend your bash startup files to control history the way you want it.

4) Think about where your time goes at the command line (eg typing out directories or filenames) and research whether there is a way to speed it up.