

Using the shell

Introduction

The *BASH* shell provides your working environment

The most popular shell client under *GNU/Linux* remains the *Bourne Again Shell* (or *BASH*). It provides you with the command prompt, and helps with relaying things to your terminal. But, as a mediator between you and the operating system, it provides much else of use too, like allowing you to juggle various jobs at once, put tasks in the background so you can get on with things in the foreground. Shells also provide ways of automating tasks by placing sequences of commands, one after another, in *shell scripts*. To hold all these commands together, most shells also provide a *scripting language*, which allows it to make decisions and move logically through a number of possible predefined scenarios.

One can use other shells, but *BASH* remains the default

Remember, the operating system runs your shell just like it does any other program. As such, you can find a number of shells made available to you, each with its own scripting language, keyboard shortcuts and configuration options. Nevertheless, a newly installed *GNU/Linux* system almost invariably uses *BASH* as its default shell program, so learn it first.

Use **Ctrl + Alt + F1** to switch to the text-mode virtual console

You have probably already logged in via *ssh*. If you launch a local terminal emulator/console program on your local system, it probably loads *BASH* as its shell immediately. The operating system also provides a simple, stark console in the full-screen text-mode you see while your system boots up. If you use *GNU/Linux* on your local machine, and the graphical interface begins to play up, you can jump into this text-mode console in the background by holding down **Ctrl + Alt** and pressing the **F1** key at the top of your keyboard, which brings up a login prompt on its own screen. The other function keys, up to **F5**, do the same, allowing you to have multiple text-mode *virtual consoles* open at once. Press **Ctrl + Alt + F7** to return to graphical mode, if it remains available.

Logout by typing **exit**, **logout** or **Ctrl + d**

To exit from a shell, you can type `exit` or `logout`. Some shells also let you press **Ctrl + d**, although others bar it. Sometimes, a shell has run *another* shell on top of it. As such, you could need to type this command more than once to get right out!

Finding your way round the shell

Verify what shell your account uses with `echo $SHELL`

Firstly, you need to make sure that your account uses *BASH*. To do so, type the following and press return:

```
echo $SHELL
```

The `echo` command, you've encountered before – it simply outputs what you ask it to. Here, `$SHELL` indicates that you wish to see the contents of the *environmental variable* called 'SHELL', which automatically became available in the background when you logged in. As such, on pressing return, you see the full path of your shell program, probably `/bin/bash`. If you find yourself on a friend's machine, and the shell acts oddly, you can find out what it runs with this command.

Move to the end of the line with `Ctrl + e`

On some keyboards, pressing the End key gets you to the end of the current line – useful if you have gone back to the beginning to make some edits, and now wish to complete it. However, if this doesn't work, the pressing `Ctrl + e` should.

Move to the beginning of the line with `Ctrl + a`

The Home key could bring you back to the beginning of the line. If not, use `Ctrl + a` – to remember it, remember that 'a' begins the alphabet.

Delete your whole current line with `Ctrl + u`

Sometimes, you want to delete what lies on your current command line, and start again. To do so, press `Ctrl + u` and it disappears. Remember this command for those situations when the system prompts you for your password, where you cannot see what you type, but you believe you've made a typo: instead of pressing return and waiting for it to complain, just press `Ctrl + u`, and the invisible bodged-up password invisibly disappears, allowing you to type it again.

Delete from the cursor to the end of the line with `Ctrl + k`

If you don't wish to delete the whole line, but only to kill from the cursor to its end, press `Ctrl + k`. The shell obliterates the text to the right of the cursor, but preserves that to its left.

Save typing with *BASH*'s completion and history

BASH saves you keystrokes with a few shortcuts

Some find the command-line interface frustrating because of the apparent amount of repetitive typing required to get things done. In fact, modern shells like *BASH* provide facilities to alleviate this, firstly by allowing you to set up short-cut commands, which we look at later; secondly, *BASH* lets you access commands you previously typed, saving having to type them again; thirdly, since many commands refer to files and directories, the shell can try to complete these without your having to type in the whole things. We look at the latter two facilities now, starting with re-using commands you have previously typed.

Reuse command previously typed with *BASH*'s history facility

BASH remembers all the commands you type, until a threshold set up by your system administrator, at which point it begins cycling through the history list again. Take a look at what *BASH* has remembered about you by typing `hi story` at the command line, and pressing return. It scrolls by quickly, so probably best pipe it through the less pager program:

```
hi story | less
```

If you remember, vaguely, a command previously typed and wish a reminder thereof, `grep` comes in useful (see the later section):

```
hi story | grep ls
```

Move through your history with your up and down arrows

Although the `hi story` command acts as a useful *aide memoir*, it doesn't save much typing. Fortunately, the shell provides useful ways of actually accessing that history. First of all, any time you find yourself at the shell prompt, you can bring the last command you typed back onto your command line simply by pressing your keyboard's "up arrow" key. If you press the up arrow again, the shell replaces that last command, in turn, with your second last command. Continue pressing your up arrow to go all the way back through your history. When you have the command you like, simply make any edits you wish and press return to run it, or press `Ctrl + c` to bring a new shell prompt without having to commit to anything. Try it: First, type `pwd` and press return. Then type `ls -la` and press return. Finally, type `echo $SHELL` and press return. Then press the up arrow once, and notice that the `echo` command reappears. Press the up arrow again, and the `ls` command reappears. Another press of the up arrow, and the `pwd` replaces it. Move your down arrow at this point, and the `ls` reappears, followed by the `echo`. Press return at any time, and the command currently on your the command line runs.

Rerun a command from the history with `!` and its history number

Moving up and down the history with your arrow keys can itself become time consuming if the command you wish to reattain lies far up the history list. You can run a specific number from your history list. Type `history` again, and note how it numbers each command you had typed. To run a command from the past again, you can simply type an exclamation mark, followed by this history number. Try the following (we discuss the `grep` text filter command in a later section at length):

```
history | grep echo
```

This filters the history list for only those lines in which you typed `echo`. To the command's left lies its history number. Assume it has the number 475. To run it, simply type the following, and press return:

```
!475
```

If you know you need to repeat that command later on, simply remember its history number, and it remains available to you for your whole shell session. You can also add additional commands to the line. For example:

```
!475 runs as my shell
```

Above, it expands whatever command 475 did (the `echo` command), and adds “runs as my shell” to the `echo` command's output.

Rerun the last command you typed with `!!`

A double exclamation mark acts as a synonym for “the last command I typed”. Usually, pressing your up arrow once, of course, achieves the same results.

Rerun a command with a similar name with `!keyword` and a keyword

Rather than type the history number after the exclamation mark, you can also type the beginning of your previous command. Let's say that the latest command beginning with “`ls`” existed in your history as:

```
ls -lap
```

Later on in the shell, `!ls` executes that command. Take care with this feature, though, as you could misremember the last command associated with the keyword, meaning you could end up running a command other than the one you expect! Always `grep` your `history` if in doubt.

Replace `^something^with something else^` in the last command

Imagine you have typed the following:

```
ls -l pictures/holiday2003/overseas/*.jpg
```

You then wished also to list your overseas holidays for “2004”. You could, of course, retype the whole line, except with “2004” instead of “2003”. Or you could take the shortcut of pressing the up arrow, and the moving the cursor to “2003” and changing it as appropriate, and then pressing return. As simple alternative presents itself. You can tell the shell to run a previous command, but with some specified replacements. At a shell, simply press the caret ^ character, then type what you wish to replace, then another caret, followed by that which you wish to replace, followed by a final caret. So, you to see 2004's pictures, you could just type:

```
^2003^2004^
```

After pressing return, the shell automatically turns the command into:

```
ls -l pictures/holiday2004/overseas/*.jpg
```

In fact, since the digit 3 appeared nowhere else in the original command except in the year, you could have got away with this:

```
^3^4^
```

Do a live reverse search in your command line with `Ctrl + r`

Even some aficionados don't know about this one. When you press `Ctrl + r`, the shell prompts you with the following:

```
(reverse-i-search)` `:
```

As you begin typing, begins a live search through you history. For example, if you type `l`, it shows you the last command beginning with that letter. Follow this with an `s`, and it shows the last `ls` command you typed. But let's say you want an earlier `ls` command – simply continue typing until you have a difference, and it switches to the older command. When ready, edit the command as desired (move forward with the right arrow) and press return to run it. To understand this best, you need to try it. Run the following commands and press return – the output doesn't matter:

```
ls -lap --reverse
ls -la
```

Now press `Ctrl + r` and press the letter `l`. Notice that your last, simpler `ls` command appears – it tries, by default, to show you the latest possible match. But now, continue typing until the letter “`p`”. Press that, and, bam, it changes to the more complex but older command. It does this, because at the point of your typing “`p`”, the commands diverge, and it realises you require the only one that matches.

Save typing by pressing `tab` to complete your file names

Most people find having to type long file names over and over again one of the chief deficiencies of a command line. After all, the graphical interface, where you merely click on the icon to do something with the file, rather than having to type in the file's name, seems a clear winner. You have seen some ways in which the command line regains some efficiency points, for example with wildcard pattern matching, where you can refer to every file in a directory beginning with `pic` by specifying "`pic*`".

Nevertheless, *BASH* provides you an additional time saver: tab completion. The "*tab*" here refers to the tabulator key on your keyboard (probably above Caps Lock); and "completion" refers to the fact that, on pressing this key, the shell tries to complete a file name without your having to type the whole thing out.

Again, you need to use this feature to understand it. First, enter `posi ski l l s/` (create it if you have previously removed it):

```
cd ~/posi ski l l s
```

Make a directory called `tabtest/` in which we can try things out cleanly:

```
mkdi r tabtest/
```

Enter the `tabtest/` directory with `cd tabtest/`

Then, let's create some directories within the `tabtest/` directory:

```
mkdi r -p save/typi ng/today/
```

Then, run:

```
mkdi r -p save/typi ng/tomorrow/
```

Finally, run this:

```
mkdi r -p save/the/whal e/
```

So, you have created a directory called `save/` which, in turn, holds two branches of child directories.

Let's see *tab* completion in action. Imagine we wish to enter the `save/` directory. Type `cd`, and press the *spacebar*, but do not press return. Instead, press the *tab* key. Immediately, `save/` pops up after the `cd` command.

How did it know to do this? Well, it looks in your *cwd* and finds no other file or directory except `save/`. With no ambiguity, it knows that you must have wanted to complete that directory name.

Now press the `tab` key again. The shell continues to try to complete the path, but this time, it implicitly looks in the `save/` directory. Unlike before, it has two options. It can't just show the complete directory immediately, and so merely shows a “t”, and beeps. This happens because the `save/` directory contains two objects, both beginning with “t”: “`typing/`” and “`the/`”. As such, the `tab` can only complete up until the first divergence, because after the first `t`, an ambiguity arises, which only you can resolve.

Do so by pressing the `y` key, followed by the `tab`. No other object begins “`ty`”. No further ambiguity remains. The shell can complete your request without further decision points and their keystrokes.

So far, we have:

```
cd save/typing/
```

We achieved this with four keystrokes: `tab`, `tab`, `y` and `tab`.

Now press `tab` again. This time, it displays “`to`” but gets no further. Two objects begin with “`to`” in the `typing/` directory and, once again, the shell cannot read your mind as to which you want. Let's imagine, however, that you cannot remember. Press the `tab` again, and it shows you the possible contenders, including `tomorrow/`. Pressing `tab` again after such an ambiguity always reveals your options. Press the `m` key, and then the `tab`. No further ambiguities remain possible, so it completes the full path. Press `return`, and your command runs.

The more you practice `tab` completion, the more indispensable it becomes. You can type long paths and file names with the minimal of keystrokes.