Tip 32: Get to know your shell

Ben Klemens

4 December 2011

level: basic command-line habitant
purpose: use the conveniences available to you

My favorite lines from this video about UNIX[1]:
"We are trying to make computing as simple as possible."
"...we wanted...not just a good programming environment...but a system around which a community could form—a fellowship."
"The UNIX operating system is basically made up of three parts: the kernel ...the shell...the various utility programs, which perform specific tasks like editing a file or sorting a bunch of numbers or making a plot."

The first two are just something to make you go *hmm*, but the third is a real statement of policy: the shell that you just use to `cd` and `ls` around before pulling up your editor was intended to really be a core part of how you do work on a UNIX box. As such, the shell does a lot more than just walk around the filesystem and start programs.

A POSIX-standard shell will have

1. an interactive front-end—the command prompt—which may include lots of user-friendly tricks,
2. a system for recording and re-using everything you typed—history,
3. abundant macro options, in which your text is replaced with new text—i.e., an *expansion* syntax, and
4. a Turing-complete programming language.

There is *a lot* of shell scripting syntax, but the next four tips will cover a few pieces of low-hanging syntactic fruit for the above four categories. There are many shells to be had (and my last tip will be that you try a different one from the one you're using now), but most of these tips are are POSIX-standard, and so work in any shell.

bang star I'll start with item #2 from the list: history. If you don't want to reach all the way over to the up arrow, `!!` will repeat the prior command. [This `!` stuff isn't POSIX-standard, but seems pretty standard across shells.] I find this useful when I'm editing the source code for `goprogram.c` in one window and running it in another. Here's what

[1] `http://www.youtube.com/watch?v=JoVQTPbD6UY`

I usually wind up typing in that run window's command prompt to compile and run the program over and over:

```
make; ./goprogram
!!
!!
!!
!!
```

Now divide the command line into the first item (the command), and everything else (the command arguments). You can paste the command arguments into the current line with `!*`. To make a directory and then step into it without retyping:

```
mkdir /home/b/tech/code_snippets/try_this
cd !*
```

If you don't have an edit window/run window setup, then you can alternate between editing a Python script and running it with:

```
vi a_script_that_I_am_writing.py
python !*
vi !*
python !*
```

where you can replace `vi` with the editor of your choice. If you are using `vi` itself, then you can of course run an executable script without leaving the editor via the `:!. %` command. `vi` is wonderful like that.

[In many shells, `!p` re-runs the last command that started with a *p* (and `!pyth` pulls up the last command that started with *pyth*, but why do extra typing). In which case you could turn the above sequence into `vi script`, then `python !*`, `!v`, `!p`, `!v`, `!p`. But use this form sparingly and don't just fish through your history, because `!r` might pull up an `rm *` you forgot about.]

fc This is a command for turning your noodling on the shell into a repeatable script. Try

```
fc -l   #the l is for 'list' and is important
```

You now have on the screen a numbered list of your last few commands. Your shell may let you type `history` to get the same effect.

You can write history items 100 through 200 to a file via `fc -l 100 200 > a_script`. Cut out the line numbers [or use `fc -n -l` to not print them to begin with], remove all your experiments that didn't work, and you've converted your futzing on the command line into a clean shell script.

In most shells [not POSIX-standard, but if it works, use it], you can run the shell script via `source a_script`, or the convenient shorthand `. a_script`, which trades comprehensibility for brevity.

If you omit the `-l` flag, then `fc` becomes a much more immediate and volatile tool. It pulls up an editor immediately (which means if you redirect with > you're basically hung), doesn't display line numbers, and when you quit your editor, whatever is in that file gets executed immediately. This is great for a quick repetition of the last few lines, but can be disastrous if you're not careful. If you realize that you forgot the `-l` or are otherwise surprised to see yourself in the editor, delete everything on the screen to prevent unintended lines from getting executed.

But to end on a positive note, `fc` stands for *fix command*, and that is its simplest usage. With no options it edits the prior line only, so it's nice for when you need to make more elaborate corrections to a command than just a typo.