Shell-Tricks

Intro

Here are some ways to speed up your shell use and make using a terminal more enjoyable. Depending on what you're working on, your mileage may vary, so change and adapt this stuff until you're comfortable.

Usually when you open up a terminal, you're using Bash, but you may also be using Zsh. You can pick one or the other (see chsh for more info). To figure out which one you're using, open up a terminal and run:

```
echo $SHELL
```

Interactive stuff

Vi mode or Readline?

Interactive shells give you some tools to edit the current command you're entering, to search your command history, and do other useful stuff. By default if you're running Bash, you'll be using GNU Readline (which defaults to Emacs-like hotkeys).

If you're using Zsh, you'll be using something like readline, which either defaults to emacs-like keyboard shortcuts, or vi-like shortcuts. This depends on whether you've specifically asked for one or the other (see bindkey), what `$EDITOR` is set to, and possibly other things.

Why is this important?

It's more important to know that there are at least two different sets of keybindings, than to choose one. If you're used to Vim, realize that vi-mode is missing some stuff you might have gotten used to (try ciw with your cursor in the middle of a word, or ca" when inside double quotes).

Readline

The bash manual has a good reference for the emacs-like keybindings. The ones I use most are:

- `C-r` to search backwards through command history, keep pressing `C-r` to find the next item. (where `C` is the Control/Ctrl key)
- `C-s` to search forwards through command history
- `M-f` move forwards one word (where `M` is the Meta/Option/Alt key)
- `M-b` move back one word
- `C-k` delete text from the cursor to the end of the line
- `C-/` undo the last action (just for editing text, not undoing commands; how cool would that be though?)

Vi mode

I had trouble finding an official list of keybindings for vi-mode, but this list looks right, and I found a blog post that talks about configuring vi mode in Zsh.

When in vi mode you're either running commands (normal/command mode), or inserting text (insert mode). To show which mode you're in, check out this stackexchange answer for Zsh. It doesn't appear that you can tell which mode you're in when running bash, so try to remember which mode you're in, or just press escape a lot.

Job Control

Running commands in your shell are sometimes referred to as "jobs", sometimes they take over the shell (like when you're running vim or emacs). Here are some useful things you can do to running commands:

- Stop a running command (including editors) with `C-z`
- See which commands are running with the `jobs` command
- Bring a job to the foreground with the `fg` command
- Bring a specific command (as seen in the output of `jobs`) to the foreground with `fg %1` where 1 is the id of the command

Suspending programs and running others can be useful when some of your workflow requires an interactive program (like an editor), but you need a quick shell here and there (like when you're running recently-edited test code). To me, this feels faster than switching terminals or using something like tmux. For more info about job control, see the bash manual.

Aliases for quick job control

```
alias f=fg # foreground the command that was last suspended
alias j=jobs # show the list of background jobs
alias f1='fg %1' # bring the first suspended job to the foreground
alias f2='fg %2' # note: this isn't a function key,
alias f3='fg %3' # but the letter 'f' followed by the number '1'
alias f4='fg %4'
alias f5='fg %5'
```

Scripts aren't run "interactively"

When you run a script that you saved to a file, usually all the fancy aliases and other things that you use when typing in your shell aren't available, because they're not needed. The bash manual has a good explanation of interactive shells.

Whether your shell is interactive affects:

- which files are loaded when the shell starts (see the bash manual or the zsh manual)
  - this means your aliases usually won't be loaded when scripting, which is a good things (your scripts will be more portable), but can be confusing when you're running a shell from another program like `:!` command in vi.
- other things, like job control

Switch between directories

You might find that you spend a lot of time switching directories. Here are a few things you can use to speed this up:

```
cd - # the last directory you were in
cd # the home directory, same as `cd ~` or `cd $HOME`
```

Useful aliases for moving up directories

```
alias ..='cd ..' # move up one directory
alias ..2='cd ../..' # move up two directories
alias ..3='cd ../../..'
alias ..4='cd ../../../..'
```

Switching between projects

It's also good to keep things organized. Personally I keep my code/work in `~/projects`, which (with a simple shell function and Zsh's `compdef` for completion magic) makes it really easy to switch to different projects, or get back to a project's root directory. `p <project>` will bring you to a specific project's folder. This is a bit nicer than `cd` because it doesn't matter what directory you're in, `p` will always switch you back to `~/projects` (when given no arguments), or `~/projects/something` if you type `p something`.

Run previously ran commands

```
!! # the last command

sudo !! # the last command, but with an added prefix

!-2 # the second to last command, for instance:
touch something/file
#=> touch: cannot touch ‘something/file’: No such file or directory
mkdir something
!-2 # runs ‘touch something/file‘

!1234 # the '1234'th history item, use ‘history‘ to see them all
```

I just learned these are called "event designators", for more info, see the bash manual.

One-character aliases

You might notice that you run certain commands very often. Why not make these as short as possible by aliasing them to a single letter? Here are mine so far:

```
alias e='TERM=xterm-16color emacs -nw' # edit text/code with emacs
alias v=vim # edit text/code with vim
alias m=mutt # read mail

alias g=git # manage changes to files (http://git-scm.com/)

alias f=fg # bring a background job to the foreground
alias j=jobs # list background jobs

alias l=ls # list files

# start or join a tmux session:
alias t='tmux attach || tmux new -s "$(basename $PWD | sed "s/\W/-/g")"'

# update an Arch Linux system:
alias u="su -c 'pacman --sync --refresh --sysupgrade'"
# update an OS X system:
alias u='sudo softwareupdate --download --install --all && brew update && brew upgrade
```

Adding, editing, and pruning aliases

Even though I pair program at work, I still make plenty of aliases. Some people avoid aliases all together because they pair. It's frustrating to type in a command you're used to working and get a "command not found" error, or worse: get some behavior you didn't expect (I've heard of people aliasing `git co` to `git commit`, where some of us would expect it to run `git checkout`). Here are some things to keep in mind when making aliases:

Am I overriding something the system provides?

See if the command already exists before aliasing it. You can do this in a number of ways:

- Running the command (possibly dangerous). If you get an "`ab`: command not found" or similar error, you're probably clear to alias it away.
- Search your distribution's package archives. You can usually do this with your OS's package manager right from the terminal

- You may be able to search for existing binaries/commands using something like pkgfile

Is what I'm about to alias destructive in any way? Can it's effects be reversed?

Be aware that making destructive commands easier to type means you'll be typing them faster, maybe even without thinking. If you were to accidentally run whatever it is you're about to alias (for instance you might alias `rm -f` to `r` to avoid typing the flag all the time), can you reverse whatever it did? How hard would that be? Sometimes it might be worth leaving certain commands more tedious so you don't lose work. Also some commands can be made safer with options. You could alias `rm` to `rm -i` or `rm -I` to make it ask for confirmation when deleting files.

What would other people expect this to do?

Depending on how much you pair with other people, it may be important to think about what kinds of commands they'll be running on your machine (or what kinds of commands you'll expect to be available on their machine). Personally I don't put too much weight into this, because if an alias or some configuration is messing one of you up, you can just remove it (or add it if you're missing an alias you use a lot).

You should change your aliases

Be aware of aliases that you often need to pass options to. For instance, if you've aliased `gc` to `git commit`, and you usually end up typing `gc --patch` or `gc -p`, why not just add that option to your alias?

Know how to discover what's available on whatever machine you're working on

Knowing how to change your setup to match your preferences is almost as important as knowing how to figure out what's been set up. Commands like `which` and `command -v` can help you figure out whether a command is an alias, binary, or function. These are so useful I've aliased `which` to `?` on my machines:

```
alias '?'=which
```

```
? la
#=> la: aliased to ls --almost-all

? sh
#=> /usr/bin/sh

? cd
#=> cd: shell built-in command
```

You can do similar things with git:

```
alias g=git
g config --global alias.h help

g h c
#=> ‘git c’ is aliased to ‘commit --verbose’

g h commit
# shows the manual for ‘git commit‘
# you can’t shadow (alias over) built-in git subcommands
```

Conclusion

I hope you've learned something that will make you more productive. If you have any comments on the post, or shell tips for me, I'm on twitter (link at the bottom), and my setup is open source, so fork away!