DEVELOPMENT ENVIRONMENT DEVELOPMENT

An infinite field of recursive rabbit holes

ZOLTÁN NAGY

Development Environment Development

An infinite field of recursive rabbit-holes

Zoltán Nagy

This book is for sale at http://leanpub.com/devenv

This version was published on 2020-06-13

Contents

Preface

The Pitch

Maybe you've worked with them. Maybe you saw them in a hacker (or Hackers) movie. You know the type. Fingers flying on the keyboard, weird UI on the screen that doesn't look like yours at all, windows showing up for seconds, always filling the whole screen, always showing the right information. Always the right tool at hand, every action a shortcut, every command an alias, always doing just the right thing. Idea to shell to editor to shell to passing tests in seconds, faster than the untrained eye can follow.

From the outside, it just looks like that person is typing really fast – and they probably are. It may feel like they're just more experienced – and they might well be. But that's not the real source of magic. The real power comes from a toolbox that has the right tool for whatever you want to accomplish. Where each tool is sharpened to an edge that can split a TCP stream in two. There's no need to think about where each tool is, or how to use it, because of plain old practice. Just like the guitarist of your favorite band doesn't consciously think about where to put their finger for that next chord, you shouldn't need to think about whether to use `sed` or `ngrep`. After a point, it becomes muscle memory.

Not to mention: if you know what you want to do *exactly*, you have an easier time finding the right tool. If you have a variety of tools, you have an easier time pattern-matching and realizing what exactly it is you want to do. It's a virtuous cycle.

That's what a customized development environment feels and looks like, at its best. At its worst, it's fighting obscure bugs and weird systems. It's finding a bug tracker showing your exact problem as `WONTFIX` from six years ago. Somewhere in the middle is research, hard work, mindfulness, a special mindset, hard work, obsession with details, lots of learning, and some more work – though the work feels more like play.

This book aims to show you a path that leads to the good parts, without getting bogged down in the bad parts, no matter how much time you have to invest. At its core, putting in the effort to improve your development environment is spending time you can spare, to be more efficient when you don't have any time to spare (say, when fixing an outage). Even when you're not in a crisis, effortlessly and efficiently using your tools cuts down on context switches, helping you stay in the flow.

What do you mean, “development environment”?

There’s the obvious – your IDE or editor. But let’s widen our horizons a bit. Is your terminal part of your development environment? How about your browser? Would you say the computer hardware running all that is part of the environment you develop on? These, and more, are all fair game when looking for ways to make your work more enjoyable and efficient.

It’s a frustrating, joyous, fumbling, exciting journey through blogs, bug trackers, manuals and, dotfiles. I’m glad you joined the trip.

Who is This Book For?

You'll get the most out of this book if you're motivated to work in that magical way, and are ready to improve the way you use your development environment to that end, but didn't yet put in a lot of time towards getting there. You're ready to jump in with both feet – this book is your diving board. You're in the right place.

Maybe you are motivated, but don't have the time to pore through all the `man` pages and years of forum posts to figure it all out. You can expect some quick wins and techniques you can implement right away, as well as warning signs on time-sinks to avoid. This will be your cookbook.

But maybe you're on the fence about whether spending time on improving your devenv is worth it. Again, the quick wins and time-sink warning signs should be useful. But more importantly, you'll also be able to get a feel for the journey – if you get a taste for it, you can always dive deeper. If not, take the quick wins. Either way, I'm happy for you.

Or maybe you already went down the rabbit hole, and write Elisp code to make Emacs send birthday reminders to your family using the MTA you run on your VPS. For you, this book may help you organize your knowledge. It may point out blind spots, open new avenues of research, show new perspectives and approaches.

What Technologies are Covered?

I must also mention what technologies you can expect to see here. The biggest divide is across the axis of operating systems. The primary focus of the content is Linux, and macOS (you may know it as OSX). With minor exceptions, the same techniques and technologies apply to BSD as well – but if you're using BSD, I don't need to tell you where that breaks down.

If you develop on Windows, and for Windows, and you'll only ever touch Windows, then I'll be straight with you: there will be biggish chunks of content not directly applicable to your environment. You'll still learn some ways of thinking that you can apply, and get ideas on what to start looking for, but much of the hands-on content will be irrelevant. In case you're wondering, yes, I'm open and totally willing to add that Windows-specific content you were about to send to abesto0@gmail.com.

How to Read This Book

While you can certainly read most of this book on an e-book reader, there are a lot of code snippets and screenshots that are going to be way less colorful. You'll definitely want to check at least those parts out on your computer. Ideally on the one where you're tweaking your development environment, for easy copy-pasting of snippets. Going further, I strongly encourage you to download the PDF version for reading on a computer. While easy to quickly check something out, reading online (with the LeanPub app)

will not serve you well.

I encourage you to read the next chapter, "Appetizer", straight through. It introduces some basic concepts, and provides a taste for the kind of wins you can get by tweaking your environment.

For the "Main course", reading back to front is perfectly fine as well. Some sections build on each other – for instance, we evaluate terminal emulators first, then talk about choosing a shell to run in that terminal emulator. If any of the sections are not relevant for you, or if one of them seem especially exciting, feel free to skip around. You won't miss anything. If you do jump around, make sure to check out the two Interludes – they provide ways of working that are useful across any part of your development environment.

"Dessert" contains deep dives, tips, and extra-geeky ideas you can try out. Who knows, maybe some of them will click for you. There's little to no connection between the topics covered here, feel free to choose and pick the ones that look exciting. You may still want to skim through them all, just to get a picture of what's out there.

About the Author

As cheesy as it sounds, a development environment is a highly personal thing. Especially so if you invest time and effort to customize it, to make it really yours – which is exactly what this book is for. That means it makes sense for us to get to know each other a bit before diving in. I hope knowing where I'm coming from will also help put some of my opinions and conclusions in context. They may not always make sense for you – and that's alright!

One of my axioms (values?) is never to waste anyone's time. I'm not very altruistic about it, that also includes my own. I get annoyed when I – or anyone – needs to spend more time doing something than they'd strictly have to, in an ideal world. So far, that's not unique.

Where it gets weird is what I think about how much time things should take in an ideal world. It usually depends on how often you do something. For something that's as common as opening a terminal, `cd`-ing or whatever to your project, and starting its unit tests? I want that whole thing to take somewhere around 2 seconds tops. Switching to documentation / editor / e-mail? O(1), a single shortcut. It shouldn't matter how many windows you have open.

I'm most at home with tooling, infrastructure, Linux, and backend code, but I tinker with a lot of software things. I learned monads. Three times. Know them? Hell no. But there's just something inherently fun in taking a piece of technology, learning it, then bending it as far as it will go. Wonder if I can make React talk to that Bacon.js stream? Speaking of JS, how does Emacs handle JS embedded in HTML? React DOM elements in JS embedded in HTML? How about a script tag in a React DOM element in JS embedded in HTML? Will that break syntax highlighting? How about in Vim?

A big chunk of that tinkering necessarily went into the development environment itself. Let's see if I can give you the result of that tinkering, without you needing to spend the same hours – after all, we all hate wasted time.

What to Expect from this Book

I'll do my best to make all these weird tweaks accessible, quick to implement, and quick to learn. This does *not* mean that this book will let you sit down, spend thirty minutes reading the chapter on Vim, and be a Vim expert. We'll cover many diverse fields that are deep in themselves. This book scratches the surface of these areas, explores why and how you may want to incorporate them into your workflow, and then provides pointers on diving deep, if you want to dive deep.

Don't expect yourself to dive deep into *all* those topics. Any one of them can take months, if not years, of your free time, if you choose to spend your time there. Staying with the easy-to-use example of editors and IDEs, there's no point to mastering five different flavors of Vim and Emacs and three IDEs. You *can* do it, if that's your idea of fun (it is mine). You may *have* to do it in some very specific lines of work, but otherwise just invest enough time so you can make an educated decision, and go with it. Each chapter ends with an "In a Hurry?" section with my personal recommendation, so you don't even need to do that if you don't want to. Even so, I recommend skimming all the sections, just to get a taster of what's out there.

Once again: you don't have to learn *all* this stuff. Take what's useful, leave the rest, maybe look it up when it becomes useful because your work changes. My background is in infrastructure, the cloud, and distributed systems, and my perspective on what's useful is heavily influenced by this. Don't take my word for it. Evaluate it for yourself.

All this takes time. Why would you invest that time? There's a simple way to make that decision: do you expect the time you invest to make a positive return on investment, during your developer lifetime? If yes, it's probably a good investment to make. To make that call, you'll need understand where you could be investing time. A good rule of thumb is that you should first sharpen the tool you use the most.

Another reason to invest time may be that it's *fun*. I get a feeling of satisfaction when the whole system works exactly like it should, after tweaking it for hours. "Like it should" is absolutely subjective – I can give you my list of shell aliases, and they'll annoy you. Because they're not solving a problem you have, in a way that makes sense to you. So instead, I'll show you why and how you may want to create *your own* aliases. Realize that the development environment you're using is made of the same stuff that you work on day to day. It’s software, it’s code. You can understand it, you can tweak it, you can change it – especially if it’s well designed.

Finally, let’s tackle a topic that comes up a lot when discussing this kind of development environment tweaking. It seems there is a correlation between how customized ones environment is, and how efficient an engineer they are in general. I think this works in a number of ways. Obviously, if your environment helps you more than my environment helps me, then you can work faster than me. This book contains many specific snippets, and quick techniques to build your own customizations, so you can reap this benefit without spending years to dive deep into everything. But that seems to not explain all the effects we’re seeing.

During the course of working on your environment, you learn a lot – you pick up lexical knowledge of how the components you’re touching work. Over time this can galvanize into an intuitive understanding of these *kinds* of systems, leading to sentences like “I expect this to work like X, and I can test this by performing experiment Y”. The Main Course of this book provides some of that lexical knowledge on just-in-time basis. The Dessert chapters at the discuss some of the topics touched upon in more depth than is strictly needed for tweaking your development environment. This is most useful if your development environment share components with your actual work items, like if you both work on a Linux desktop and manage Linux servers. If that’s not the case, some of that knowledge is useful just to understand your own desktop. I also expect that at least *some* of this knowledge and intuition transfers into other areas, but we’re getting to rather shaky ground here.

Even more importantly, your development environment is a great training course for software development in general. You get to explore your own requirements in depth. You get to understand how the existing tools solve those requirements, and then you get to make changes to said tools, so that they better solve the requirements. Most importantly, you hit bugs and problems, and you fix them. This loop of identifying a requirement or problem, debugging it, learning as you go, and finally fixing the problem, builds a habit of fixing stuff. It teaches you that you can fix everything; there’s always a bigger hammer. Your own development environment provides visceral feedback, and iterations tend to be short – much shorter than any kind of project work. You make a change, maybe restart an application, and *boom*, it’s in production. Make sure to use virtual or test environments for experiments though!

The conviction that the complex and weird behavior you’re seeing is something you can fix translates into any kind of software engineering. Unfortunately this book can’t do this part for you – you need to make your own mistakes, debug them, and fix them. The point here is not to have the problem fixed – it’s to have fixed the problem. The best we can do here is give you the lexical knowledge so you know where to start debugging. Or at least, know what to break in the first place.

However! Messing around with your development environment is *not* the only way to gain that confidence for fixing problems. It is not *the way* to becoming a great engineer. Spending hours and months and years tweaking your development environment, making new mistakes and learning new things, seems to “cause” good software engineering skills. But there are *other ways* to get there. I’m not saying this is even the *right* way, I’m saying it’s *one* way. And it’s the one journey this book takes you on.

Conventions Used

Information

Blocks like this contain information tangentially related to the topic being discussed. They don't provide all the details, but are instead starting points for more research, should you be interested.

Exercises

You're encouraged to follow along with all examples along the book. Exercise boxes like this call out specific things to try that are no obvious.

Warning

This is information that highlights pitfalls, especially ones that can have consequences outside of "just" messing up the part of your devenv you're working on.

Tips

Ideas, tips you can try out when the situation arises. Not something you can *do* right now, more of description of a method for approaching things.

Capitalization of names of software follows the most standard way I can find - this is usually the way the name appears on the website and documentation of the software in question: tmux, GNU Screen, Bash.

When mentioned in the context of a command to run from the command-line, software names are in `monospace`, and capitalization follows the name of the executable: `tmux`, `screen`, `bash`.

The string "development environment" tends to come up a lot when talking about, you know, how to build a great *development environment*. To make sentences shorter and more comprehensible, the string "devenv" will usually be used instead of the long form "development environment".

```python
class CodeExamples:
   "Look like this, obviously"
   def run(self):
       print('With syntax highlighting, when appropriate')
```

```
$ echo 'Interactive shell sessions always include the prompt'
Interactive shell sessions always include the prompt
$ echo -e 'To separate the command\nfrom the output\nLike this\n\n'
To separate the command
from the output
Like this


$
```

```bash
# Some longer sections designed for copy-pasting into a terminal
echo 'are formatted like this, with no prompts'
echo 'so that you can really just take the whole thing and paste it into your shell'
```

Appetizer

In this chapter we'll first look at the philosophy behind all the customizations and optimizations we'll do later on. We'll also take a detour into philosophy, and look at a few specific techniques you can start applying today without up-front investment. Finally, we'll touch on how to make sure you don't need to find the same tutorial more than once.

Identifying your Motivation

Different people will have different reasons for optimizing a development environment. One thing's for sure: whatever the first links in the causal chain, the last item is always "therefore, I want to reduce the friction of my development environment". Put it differently: I want to do my work more quickly, fight less with the environment. I want to make the environment support my workflow, as opposed to changing my workflow to suit the workflow of the tools at hand.

Unfortunately, as they say, there are no free lunches. Just as energy doesn't come from nothing, that speed improvement, that time saving, also needs to come from somewhere. You can think of it as the law of conservation of development time. Or maybe it's really just conservation of energy – put in energy to improve your environment, and you'll need to spend less energy later to do your work.

Either way, there's a critical realization hiding here. In terms of invested energy, your development environment is a bit like a savings account – you can put in energy now, and realize the saved time days, or weeks, or months later. Better yet, any improvement you make *keeps on giving*. It doesn't run out.

What also helps is that all time is not made equal. You have your work-time and free-time, obviously, but there's also crunch-time, off-time, tea-time, outage-fixing-time, hacking time, time for side projects, time for cooking, and time for sleeping. The art of improving your development environment is founded on finding time you can spare (be that off-time, hacking time, or sleep time) and investing that so that you can be more efficient when time is at a premium (like recovering from an outage).

Find the Time

Think about your weekly schedule. When can you carve out an hour or two to implement improvements on your devenv?

Remember to have fun as well. Since you're doing this in time you can spare, it's OK to go down more rabbit holes than usual. No need to justify it, just let go and geek out!

So fine, good, we say we want to do work more efficiently when we’re low on time. What does that look like? For one, it means minimizing mental context switches. Every time you need to think about how to accomplish what you want, you experience a micro-context switch. You go from “I want to know which server is generating error logs” to “is the server name field 3 or field 4 in the access logs?”. You go from “I want to rename this function” to “does this IDE properly refactor this language?”

A related, but not equivalent, idea is staying in the flow. Context switches obviously kill flow, but so does the wrong kind of frustration. A problem needs to be challenging enough for you to stay in the flow, but the half-second input lag of your terminal should not be that challenge. Conversely, there’s a satisfaction to the well-choreographed dance of meet problem – deploy right tool – meet next problem – deploy the next right tool – meet next problem.

All that is a long way of saying: we want to minimize the time and number of actions needed to get from “I know exactly what I want to do” to “what I wanted to do has happened”. Note that this does not solve the problem of “I don’t exactly know what I want to do”. Only analyzing the problem and examining the available tools can do that. Having a wide array of tools ready to use definitely helps, though.

When you know *what* you want, but not *how* to get there, having more efficient tools can also help. Imagine you can think of five ways to refactor a piece of code. The refactorings are not conceptually hard, but they need a lot of code to be moved around. If your IDE can do most of that work in a few clicks, you’re quite likely to try all five ways, and choose the one that’s actually the cleanest one. Similarly, you might try all of them if you’re extremely efficient in using the text editing functions of your IDE or editor. But you’re quite likely to call “good enough” after the second one if each refactoring takes an hour, because your tools don’t support you.

Mindfulness Applies Everywhere

Let’s talk a bit about a soft, squishy topic before jumping into our first hard tech discussion.

According to Wikipedia, mindfulness “is the psychological process of bringing one’s attention to the internal and external experiences occurring in the present moment”. Mindfulness meditation is traditionally practiced sitting down, eyes closed, focusing on the breathing. For me personally, it didn’t change my life, but if nothing else, it’s a fascinating experience.

In the context of software development, and improving your development environment, this is the very first step to take: to notice what can be improved. There are broad strokes that are true for any developer, but to make yourself *really* efficient, you need to become conscious of your own habits and practices, and make *those* practices flow smoothly in your environment.

Of course the traditional practice of sitting eyes closed, hands in your laps doesn’t mesh well with paying attention to your development process. Not all meditation needs to be done like that, though. There’s walking meditation, running meditation, even meditation while doing repetitive power exercises. But coding meditation is not something I’ve read or heard about – maybe flow is a bit like that.

Either way, we don’t need to get stuck on the practice of meditation – the concept of mindfulness, of paying attention to internal and external processes, is useful in itself, and we can build our own practice around it.

Profile Your Workflow

Choose a workday when you have some slack-time. Have a big release tomorrow? Try this afterwards. Make sure you have a notebook on you, or have a text file you like. Every hour during the day, or at the end of each Pomodoro if you’re into that, stop and ask yourself two questions:

1. What action in my devenv did I repeat more than two times?
2. What intentions took long to implement? What tools did I struggle with?

Note your answers. Later, when you have the time, review the notes. Make a note (mental or otherwise) to look especially closely at the sections dealing with the tools you had problems with.

That's a structured, somewhat forced way of experiencing what it feels like to pay attention to your workflow, the interaction of yourself and your development environment. Strive for a habit of continually noting opportunities for improvement. Usually you shouldn't break your flow to fix them, but do make sure you know what they were once you have the time. Personally, I collect my issues on GitHub, at https://github.com/abesto/ansible-devenv/issues.

After noting down the issues, the next step is of course to go ahead and fix them. Generally, this should be an iterative improvement – tweaking a bit here, adding an argument there, installing a package, changing a configuration option. There will be times for big changes, designing, researching, and planning though – especially when you first set out to make your development environment truly your own. The next section, "Main course", will take you through some of those big changes. Before that though, let's take a look at a specific change you can start implementing today, iteratively.

On Premature Customization

It may be tempting to copy-paste all the cool-looking examples you find on public GitHub repositories, or in the environments of coworkers, or even in this book, and see what happens. While that's not a bad approach, make sure you have a recovery method ready, in case things go wrong. For instance, if you're messing with your `.bashrc`, what will you do if it turns out there's an `exit 1` in the snippet you pasted? (You'd run `bash --norc`). Better yet, make sure you understand the context of the tweak you're about to adopt (is it for Bash or Zsh?), and what it does, before pasting it into your configuration. This book aims to give you enough knowledge and tools to do both.

Supercharging your Shell

Shell aliases are one of the easiest, quickest, most ubiquitous ways to sharpen your devenv. Let’s first unpack what a “shell alias” is.

What’s a Shell?

Strictly speaking, a shell is any piece of software built with the intention of allowing access to functions of an operating system. The command prompt of MS-DOS is a shell, Windows 3.1 is another shell, and Bash is also a shell. However, in daily use, we don’t usually refer to graphical interfaces as shells – that word is reserved for text-based “shells” using a CLI (command-line interface). That’s also what “shell” will mean in the rest of this book.

Shells have, without exception, two modes of operation: an *interactive*, and a batch mode. If you know of an exception, let me know – I love esoteric software. Interactive mode is what everyone who’s ever seen a terminal knows: you have a *prompt* at the left-hand side with some information about the state of the system and shell. It might look something like this:

```
[username@hostname]$
```

This is followed by the area where commands are entered (usually one line at a time). Once a command is finished, the output is printed, then the prompt again, and on and on it goes. In code listings of showing shell sessions, the prompt will be just the character `$`. Unless otherwise indicated, the shell used is Bash version 4.3.

On the other hand, *batch* mode is about capturing a series of commands to execute in a script, and running them at once – in a batch. These scripts are called shell scripts (except for Windows, where they’re called batch scripts). At their simplest, they’re just a series of commands to run, one after the other. For example, this is how you’d create a directory, create a file inside it, then delete the directory, in an interactive shell:

```
$ mkdir -v awesome-dir
mkdir: created directory 'awesome-dir'
$ touch awesome-dir/awesome-file
$ rm -rv awesome-dir
removed 'awesome-dir/awesome-file'
removed directory 'awesome-dir'
```

Now we can create a file called `awesome-script.sh` with the following contents:

```
mkdir -v awesome-dir
touch awesome-dir/awesome-file
rm -rv awesome-dir
```

These are exactly the commands issued above in the interactive shell session. When we run the script, we get:

```
$ bash awesome-script.sh
mkdir: created directory 'awesome-dir'
removed 'awesome-dir/awesome-file'
removed directory 'awesome-dir'
```

That's exactly the output we got above, without the interleaved prompts. Creating shell scripts is a way of automating repetitive work, especially useful when that work is error-prone. We'll look at constructs and practices making that as painless as possible in the chapter dedicated to command-line productivity. For now, let's focus on ways of making the interactive mode of operation easier, without diving deep into hard-core shell scripting.

What's a Shell Alias?

I imagine you use a shell at least a couple of times a day. I further imagine some of the commands you type are the same, over and over again. Someone working on Node.js projects might type `npm test` over and over (then again, someone working on Node.js projects is probably using something to automatically re-run tests as the code changes). Someone working on Python projects might find themselves typing `nosetests` over and over. Pretty much everyone these days is using some version control system. If they use the CLI, they'll be typing things like `git checkout master`, `git commit -a`, and `git pull` all day.

A *shell alias* is a way to say "when I say *this*, I mean *that*". Oh and: you usually want to make *this* shorter and easier to type than *that*. For example, you can try this in your shell right now:

```
$ alias la='ls -lhat'
$ la
total 8.0K
drwxrwxr-x  2 abesto abesto 4.0K Apr  5 21:55 .
-rw-rw-r--  1 abesto abesto    0 Apr  5 21:55 awesome-file-3
-rw-rw-r--  1 abesto abesto    0 Apr  5 21:52 awesome-file-2
-rw-rw-r--  1 abesto abesto    0 Apr  5 21:52 awesome-file
drwxr-xr-x 63 abesto abesto 4.0K Apr  5 21:52 ..
```

All examples in this chapter use Bash as their shell. They should work just fine if you're using any other POSIX-compatible shell as well. Then again, if you're using something funky like `tcsh` or `fish`, you don't need me to tell you that.

Let's break down what happened there. First, what are all those arguments to `ls`?

- `-l` turns on the long listing format, providing a big bunch of metadata about each file
- `-h` prints file sizes in a human-readable format like `12M` for 12 megabytes, instead of printing the number of bytes
- `-a` prints all files, including hidden ones
- `-t` sorts the listing by modification time, newest first.

So that would be `ls -l -h -a -t`. We can contract short command-line flags into just `ls -lhat`, or `ls -hatl`, or any other ordering – the order of the flags here doesn't matter.

Next, we have quotes around the command on the right-hand side. Why is that? Notice also that there's no space around the `=`, unlike what you would see in any cultured programming language. The construct `lefthand=righthand` is how variable assignment looks like in Bash; `lefthand` must be a valid variable name, then comes `=` without any spaces on either side, then `righthand`, which must be a single string. In Bash, the unquoted character sequence `foo bar` represents two strings - `foo` and `bar`. To create a single string containing both words, we can escape the space like this: `foo\ bar`, but that's not very easy to read. Better, we can quote the string: `'foo bar'`. Note the use of single quotes - variable and subshell interpolation happens inside double quotes, so if you just want to represent a string verbatim, it's safest to go with the single quotes. Putting all that together, here's what variables look like in use:

```
$ apology='sorry if this is too trivial'
$ echo "$apology"
sorry if this is too trivial
```

Fine, so what does all this have to do with aliases? Only this: creating an alias uses exactly the same syntax and rules as defining a variable, preceded with the special keyword `alias`. That is, the command `alias lefthand=righthand` tells Bash to, whenever it sees `lefthand`, pretend it saw `righthand` instead, taking all the above rules into account. Putting all that together, it's now quite clear that in `alias la='ls -lhat'` defines an alias called `la` that expands to `ls -lhat`, as well as how it does that.

Note that any arguments you pass to the invocation of the alias will be passed to the expanded command. You might type `ls -lhat awesome-dir` to list the contents of that awesome directory. Having defined `alias la='ls -lhat'`, you can equivalently type `la awesome-dir`.

Why would you want to do this though? Couple of reasons:

1. It's **shorter**. Fewer key-presses means not only shorter time to type, but also fewer chances to mistype a character.
2. It **encapsulates knowledge**. You may need to search the internet and read `man ls` to get all the options you want – and chances are, if you wanted them once, you'll need them again. So create an alias, and remember that `la` is that thing with the verbose output, instead of "`l` for long, `a` for hidden, and what was it for the human-readable file-sizes again?"
3. It opens the way to **iterative improvement**. For instance, you might learn that adding `--color=auto` to the arguments of `ls` causes it to use color in its output to highlight different kinds of files – blue for directories, light-blue for symlinks, green for executables. You can now extend the alias you defined previously. This increases the other two gains – compared to the full command, the alias is even shorter. It encapsulates more knowledge. The best part? You don't need to commit anything to working memory. Just keep using the alias you've been using all this time, except now it's better.

Persisting Aliases

Unfortunately any aliases you define in an interactive session are lost the moment that interactive session ends – when you close the terminal, in plain English. Not to worry though - `~/.bashrc` to the rescue!

When an interactive Bash session starts, all the lines in `~/.bashrc` are evaluated as if you had entered them directly on the command line. This is different from just running it as a script in a number of significant ways - again, we'll cover those in the chapter dedicated to each shell. For now, it's enough to know that to persist an alias, as well as any other customization of your shell, just open up `~/.bashrc` in your favorite text editor and add the commands, same way as you would type them in an interactive shell.

> On startup, Bash loads (ie. executes the commands in) a bunch of files. Depending on whether we're in an interactive shell, a login shell, and the phase of Mercury, this includes some combination of `~/.bashrc`, `~/.bash_profile`, `~/.profile`, and `/etc/profile`. For a proper description, refer to https://www.gnu.org/software/bash/manual/html_node/Bash-Startup-Files.html.

When Aliases are Not Enough

One of my favorite aliases goes like this (line-breaks inserted to fit on a page):

```
alias glg="git log --graph \
--pretty=format:'%Cred%H%Creset -%C(yellow)%d%Creset \
%s %Cgreen(%cr) %C(bold blue)<%an>%Creset' \
--abbrev-commit"
```

There's no way in hell I'd want to type this more than once. I don't even know what half of it does. I probably didn't even write it myself, just copy-pasted it from somewhere on the internet because I like the output. It prints an extremely fancy and nice-to-read version of the history of the git repository I'm in.

And this is about the limit of what aliases can do. If you want something more complex – for example, something that needs flow control, you should absolutely not use an alias. It's possible to do. Technically, anything in Bash can be a one-liner, it's just the line will be extremely long. But there's a better way.

Instead of diving into deep and complex examples though, consider this: what if the logical argument of an "alias" is not the last string on the command line? For instance, in the configuration management system Chef, whose main command-line interface is the command `knife`, sometimes it's necessary to “force” uploading a “cookbook”. (If you’ve never touched Chef, don’t fret; the functionality of the command is irrelevant for the example. Feel free to imagine this is the CLI of a developer-friendly collaborative cookbook-collecting site). Here’s what it looks like:

```
knife cookbook upload <name-of-cookbook> --force
```

That’s quite a mouthful, so of course defining an alias for it makes sense. But oh no! If we define the alias as `alias kcuf='knife cookbook upload --force'`, then the name of the cookbook to upload can only be passed as an argument to `kcuf`, which means the expanded command will be `knife cookbook upload --force <name-of-cookbook>`.

The solution is to use a function instead of an alias:

```
kcuf() {
    knife cookbook upload "$1" --force
}
```

The syntax `identifier() {` starts a function in Bash. It’s designed to look like the definition of a function in any C-like language, but it doesn’t have any of the features you’d expect: you can’t define the argument list inside the parentheses. You can’t even move the `{` to the next line, even if you prefer one of the *inferior* coding standards out there. `}` of course ends the function definition. Finally, `$1` is a built-in special variable in Bash – its value is the first argument passed to the function it’s used in. When used outside a function, it’s the first argument passed to the process it’s used in. In case you’re wondering, it has a few friends like `$2` and `$3`. There’s also `$0`, then name of the function (or the “name” of the process, when used outside a function).

All in all, this is a tiny bit more typing than defining an alias, but it elegantly resolves the limitation of aliases around argument order. You can also use any and all Bash features inside the body of a function, which can lead to powerful abstractions, as shown in the story below.

Aliases: A Case-Study

I used to work quite a bit on projects based on the Django web framework. For reasons that are not worth detailing, I didn’t have a way to run the unit test suite from the IDE – I had to switch to a terminal, optionally switch to the right project, activate the virtualenv of the project, and then invoke `./manage.py test`.

> Virtualenv provides isolated Python environments – it's a bit like `nvm` plus `npm` in Node-land, or `rvm` for Ruby. Each virtualenv has its own Python interpreter (versions of which may differ between environments), as well as its own set of Python packages, instead of using globally installed packages. This allows development and packaging of applications without having to synchronize any versions between different projects.

At a guess, I would do this around ten to thirty times a day. It quickly became annoying. The part I'd do most often was running the tests. That's pretty easy to fix in bash. It's as simple as adding this line to `~/.bashrc`:

```
alias mt='./manage.py test'
```

This saves some keystrokes, but more importantly it erases the mental burden of "how do I run the tests again?" - `mt` is simple enough that it can go straight to muscle memory. Alt-Tab mt return. Tests are running, elapsed time: ~0.3 seconds. Nice.

Sometimes I'd want to run the unit tests with the env var `TEST_WITH_REMOTE` set to `1`. That's quite a lot of typing, and easy to get wrong, and the only indication I'll get of getting it wrong is that tests fail in the wrong way. No surprise, I guess, here's our alias to fix that:

```
alias rmt='TEST_WITH_REMOTE=1 ./manage.py test'
```

> *Environment variables* (colloquially "*env vars*", or sometimes even ENV vars) are a way to pass string values from a parent process to a child process. In this example, the parent process is your shell, while the child process is the Python test runner. Test code can then check for the value of this *environment variable*, and change its behavior as needed. Like the name suggests, environment variables encode information specific to the *environment* a process is running in. For example, you might use env vars to tell a web application whether it's running on a development machine (and so should expose debugging facilities) or in production (and so should optimize whatever it can, and should *not* expose any debugging features).

Working outwards, next up is activating the virtualenv. In all our projects, the virtualenv is always stored in the root of the project, a directory called `virtualenv`, and you have to activate it using the magic incantation `. virtualenv/bin/activate`, where the dot (`.`) tells bash to "source" that file - pretend that you typed the contents of the file directly, instead of running it as a command. Typing that becomes second nature over time, and is easy to always get right using auto-completion, but it's still time that could be saved. Not to mention, if you're not in the root of the project, you need to put the right number of `../`s in front of `virtualenv/bin/activate`. That's mental effort that does nothing, at all, to get you closer to your goal, whatever that may be. It's a function of the environment – and you can change the environment. Here's one way of doing that – again, in `~/.bashrc`:

```
v() {
 for candidate in virtualenv ../virtualenv ../../virtualenv; do
  if [ -f $candidate/bin/activate ]; then
   . $candidate/bin/activate
   return
  fi
 done
}
```

This looks a bit different – the logic is complex enough that it doesn't fit in a simple alias. We have to define a function. The idea is to check all the usual locations where the virtualenv can be, and activate it once we find it. Note that it could be generalized in a lot of ways, it could traverse the directory structure more generally to generate more possible candidates. But it hits the golden standard of It Works.

Now for how impactful such a small tweak can be: an engineer I worked together with at the time asked what `v` does when he saw me use it in my shell. I explained it to him; he got excited, we copy-pasted it into his `.bashrc`, he was happy, then I forgot about it. A year or so later a new hire asked me for help to solve a tricky problem he hit while setting up the virtualenv for a project. As he was explaining the steps to reproduce the problem, he said "then I use `v` to activate the virtualenv". I got suspicious; I asked him what that is and where it comes from. He showed me the exact same code, which he got from his tech-lead - *not* the engineer I showed it to. Turns out, this function was so useful that it got passed around across a chain of at least three engineers. Not only that, it was so useful that it was included in the welcome package of a newly hired developer!

For completeness' sake, `v` is only complete with its counterpart `d` that deactivates the virtualenv. Virtualenvs define a function `deactivate` to deactivate themselves, so this is trivial:

```
alias d=deactivate
```

Navigating the File System in Constant Time

… in a shell, that is. There’s a tool called `autojump`. It latches on to your shell, and goes with you where-ever you `cd`, and it keeps track of the directories. Then, you can invoke it with a partial name of a directory you’ve visited previously. It will do its level best to figure out which directory you mean, and take you straight there.

Here’s an idea: set up https://github.com/wting/autojump, do a few days of work as usual so that it can learn your patterns. Then focus on using it for another day, see what it feels like.

If you want more magic, https://github.com/clvv/fasd has more magic.

Tying it back to the original story of running unit tests: given all this preparation, I now had a sequence of keystrokes, in muscle memory, that I could use to run tests. Starting from the IDE being open, it goes like this, and takes around 1.2 seconds from “I want to run tests” to “tests are running”:

- Alt-Tab (switches from IDE to terminal)
- `j auth` (uses `autojump` to go to the project, called `authservice` in this case)
- `v` (activates the virtualenv)
- `mt` (runs tests)

If you’re wondering about the weirdly accurate timings: I once gave a little internal talk on optimizing the development workflow on Django services – I recorded my screen, and measured how long each step takes.

So what should you turn into aliases / functions? How much is too much? Should you even use aliases? Won’t that make things confusing when you need to work at another computer? Generally speaking, the right balance is what feels right for you, personally. If you’re just starting out on the path of customizing your shell environment, I’d suggest starting slow and adding just a few aliases, seeing how they work, tweaking them over time. Slowly expanding your arsenal allows you time to recognize the situations when you can use that alias. Initially you can expect to have a tiny shock when you use an alias. Your brain gets confused. It usually takes another 3-5 seconds to implement that intention, and your brain tends to turn off for that time. Now that the intention takes a fraction of a second to complete, you must unlearn that pause.

Adoption

Be aware that actually benefiting from your new shell improvements requires that you actively use them, and that takes time. You need to remember to use them and then use them enough times, until they become muscle memory. This is one more reason to go bit by bit: you can't instantly remember 6 new magic spells and also recognize the situations in which they should be useful. I have this friend who started to use autojump three times, but then always forgot to use `j` instead of `cd`, and then she just forgot about it. You need to actively practice the use of your new tool.

DRY your shell usage!

DRY, short for Don't Repeat Yourself, is a principle we apply liberally to code – repeated code is prime target for refactoring. Apply the same principle to your use of the shell. What operations do you repeat? Can you automate them? You have the full power of shell scripting at your hands.

Once you start looking for ways to customize your environment (shell or otherwise), you'll inevitably come across situations where you just want to have the same features that other developer seems to have. Copy-pasting their configuration might do just that, or it might mess up your environment or workflow. Keep in mind that you can only customize what you understand. In one extreme case, I've helped someone debug a messed-up shell where they copy-pasted configuration into their `.bashrc` that was written *for another shell*.

Some of the systems you'll need to understand will be ones specific to your company or project. I can't *directly* help you with those; but hopefully by the end of this book you'll understand enough of the underlying open-source, standard technologies they're built on to be confident in exploring and customizing them.

Also Try

- `huffshell` suggests new aliases based on your shell history
- `commandlinefu` is a crowdsourced repository of shell commands that do useful, complex stuff

End of the Sample

Here's a teaser of the rest of what's available in the full book currently:

- Version Control for your dotfiles
- Automating Environment Setup
- Editors and IDEs: Choosing and Customizing
- Picking an Operating System
  - macOS, the Safe Option
  - Ubuntu: Beginner-Friendly and Stable
  - Arch Linux: Minimal, Always Up to Date
- Terminal Emulators
- Command-line Productivity