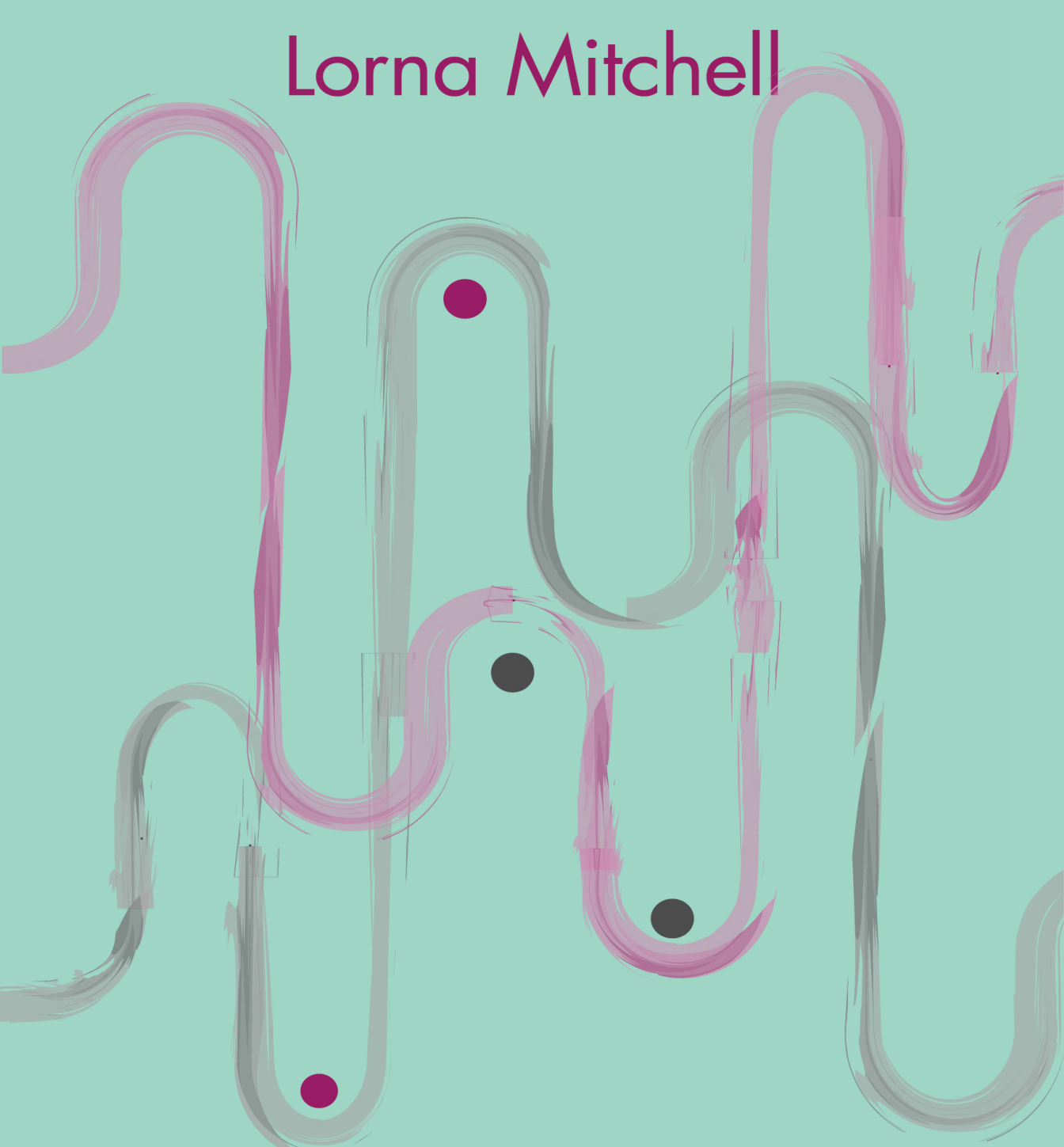
Lorna Mitchell



Git Workbook

Self-Study Guide to Git

Git Workbook

Self-Study Guide to Git

Lorna Mitchell

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

This version was published on 2018-01-15

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Lorna Mitchell by spreading the word about this book on Twitter!

The suggested hashtag for this book is #gitworkbook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#gitworkbook

Also By Lorna Mitchell

N Ways To Be A Better Developer

Zend Certification Preparation Pack

Contents

About This Workbook

This workbook is designed to equip you with the skills you need to use the Git source control tool. It assumes absolutely no pre-requisite knowledge at all. To get the best out of this book you will need:

- an attitude for participation
- a GitHub.com account (sign up if you don't have one, it's free)
- some time. Little stolen moments of time is fine, this is all about the small pieces

There is no easy way to pick up new skills, and git is no exception (and also there's a lot of git skills!). In recognition of that, this workbook encapsulates everything you will need for all the various areas, by explaining it, showing you, and then asking you to try it yourself. There are some puzzles and quizzes along with the exercises - these are there to hack your brain into remembering this stuff! It's tempting to skip through or imagine what you would write, but please embrace the experience and work through all the exercises and puzzles, I promise it will help to make everything end up in your brain.

There are some supporting materials in git repositories on GitHub: you can find all of them under the Git Workbook organisation here: https://github.com/gitworkbook[1].

If you have an comments or questions, then I would love to hear them: lorna@lornajane.net. Have fun!

Each section has a box for you to tick ☑ so you can track your progress.

☐ I have read the introduction and am ready to begin with git

[1] https://github.com/gitworkbook

Make A Repository

Git is a distributed version control system, which means that you do your local work on an actual repository, and there are other remotes which are also repositories in exactly the same way. We'll start by making a standalone repository and then we'll discuss how to link it to other repositories or "repos" as they are known.

The command to create a repo is `git init [dir]`. If you run `git init` on its own, your new repo will be in the current directory. If you run `git init myproject` then you'll get a new directory called `myproject` with a git repo inside it. You can check if a directory is a git repo by running `git status` (if it says "not a git repository", check you changed into the directory you created the repository in) or by checking that there is a hidden `.git/` directory inside there.

Assignment

1) Create a new git repository in the directory `project1`.

2) Check that this directory exists and is a git repository

 I now have a repository to use for the next examples

Handle Interruptions with Git Stash

Do you ever get interrupted at work? Me too!

When something comes up and you are right in the middle of a task, you have two options, neither of them are good. You can either commit half a thought, creating a strange, unfinished story, or you can try to fix another thing with a half-chewed changeset in your working area. Actually the third option is to throw away all your changes so you have a clean repo, but let's stay away from that one.

Enter the `stash` command, which lets you safely put your work aside for a moment while you deal with something else. In particular it's useful when you can't switch branches because you have something in your working area that would be overwritten by a change in the other branch.

You can stash multiple stashes, they are stored in a stack so by default you'll always get the most recent thing you stashed - but you can also manipulate that list. Run `git help stash` for more instructions.

Final note: it's possible to lose work from your stash, so unless you really are putting it aside for just a few minutes, I'd recommend creating a branch and committing instead - you can make your history logical again later (see the section on interactive rebasing).

Assignment

This is a pretty simple feature so the easiest way to show you is to let you have a go.

1) Make a change to whatever branch you are on, but don't commit it. Instead type `git stash`

2) Run `git status` and marvel at the absence of the work you just did

3) Make more changes and type `git stash` again

4) Look at the stashes by running `git stash list`

5) Apply the newest one by running `git stash apply`, then run `git status` and then `git stash list` again. What happened?

 I can safely put my work aside for a moment, and still get it back

Rewrite History With Rebase Interactive

Another use of the rebase command is to quite literally rewrite history - the history of your branch.

Before we go any further, there are some caveats. It is not recommended to rewrite shared history; this means that you don't rebase commits that you have already pushed[2], or that instead you should create a new branch name as we did when we transplated a branch with the rebase command in the previous section.

Now I've got the warnings out of the way, let's get on. This feature is called "interactive rebasing" and it allows you to take a range of recent commits, and change them in any way you like. Your choices are:

Action	Outcome
`pick`	keep the commit (this is the default)
`reword`	get the chance to change the commit message
`edit`	edit the commit
`fixup`	combine these changes into the previous commit. Useful for the "fixed the bug", followed by "no really fixed it this time" situations
`squash`	still combines commits but prompts you with an editor with all the commit messages present in order for you to make the correct commit
remove	to pretend a commit never happened: simply delete the line for it

Pretty powerful stuff. One thing that always confuses me is that the order of the commits is in the order that they happened, with the oldest first. Which is exactly the opposite way round from the way I usually see my history, via the `git log` command!

[2] Where "you have already pushed" really means "someone has already pulled". If you've pushed a branch to your own repo and you weren't expecting anyone to pull or collaborate on it, then you can do as you please!

First of all: `git log`

```
$ git log
commit e11e694f62832dc84473350d838040aa94977280
Author: Lorna Mitchell <lorna@lornajane.net>
Date:   Mon Oct 20 10:49:47 2014 +0100

    Bug definitely fixed by this commit

commit c8c923986d5df6e9281a77884af4f031e5355c15
Author: Lorna Mitchell <lorna@lornajane.net>
Date:   Mon Oct 20 10:49:29 2014 +0100

    Fixed bug 123 that's been causing all the issues
```

Oops! I think you can see the problem here. The interactive rebase is exactly the right thing to iron out this history. Since I know I only want to operate on those last 2 commits, I can use this command rather than looking up which revision I want to go back to: `git rebase -i HEAD~2`.

When I do so, my editor opens a file that looks like this (the gotcha is that these commits show in time order, oldest first, unlike git log. This confuses me every time):

```
pick c8c9239 Fixed bug 123 that's been causing all the issues
pick e11e694 Bug definitely fixed by this commit
```

There are also some comments in the file, which is basically the documentation for the various options we showed above. Each commit is listed with its revision number and commit message, and starting with the word `pick`. Next, I will edit the file to describe what should happen, and then git works through line by line to actually do it. At each point where my input is needed, it will pause and let me either edit the message or actually use git. `git status` will warn me that I am still mid-rebase while that's the case, but I can type `git rebase --abort` to undo it all at any time during the rebase.

If I don't change anything and just save and close the file that git prompted me with, then git will simply take and reapply all the changes, one at a time. This is the default

behaviour and the edits that I make to the file tell git what to do differently this time around.

For the situation above, I'm simply going to squash that commit into the first one, so it looks like I can actually fix a bug properly:

```
pick c8c9239 Fixed bug 123 that's been causing all the issues
fixup e11e694 Bug definitely fixed by this commit
```

When I save and close the file, git will make a new commit (with a new identifier) containing all the changes needed to fix this bug. Since I chose the fixup option, I don't need to supply any more information or write a new commit message.

Now my git log looks much tidier:

```
$ git log
commit 83347acaf9e9efeaf948fea8718cfa40af0fb069
Author: Lorna Mitchell <lorna@lornajane.net>
Date:   Mon Oct 20 10:49:29 2014 +0100

    Fixed bug 123 that's been causing all the issues
```

Exactly like when we merge or rebase a branch, conflicts can occur here too. Git will stop, and give you the space you need to resolve the conflict exactly like you do for a merge conflict: identify the conflict, edit the files, then add and commit. There's no difference in what you need to do and when you are finished, git will continue with its rebase.

Quick Quiz: Rebasics

For each of these situations, would you use rebase, or rebase interactive?

- To fix a branch that branched from the wrong place?
- To make a branch be based off the current tip of master and therefore include all the changes in master?
- To edit the history on a single branch?

Quiz answers can be found at the end of the workbook

Assignment

1) Get onto the `master` branch and create three commits here.

2) Look at the history with `git log`, then use `git rev-parse HEAD~3` to check which revision we'll be rebasing back to.

3) Rebase all three: delete the first commit you made and combine the other two using `squash`

4) Look at the history again to check that you now have one commit as you expected

 I can rewrite (unpushed) history in git