[Set-Up 10](#_Toc31972549)

[Features of GitHub 11](#_Toc31972550)

[set-up 11](#_Toc31972551)

[Username advice 12](#_Toc31972552)

[4.2 Free private repos 13](#_Toc31972553)

[Install or upgrade R and RStudio 14](#_Toc31972554)

[5.1 How to think about upgrading R and RStudio 14](#_Toc31972555)

[Chapter 6 Install Git 15](#_Toc31972556)

[6.1 Git already installed? 15](#_Toc31972557)

[6.2 Windows 16](#_Toc31972558)

[6.4 Linux 17](#_Toc31972559)

[Chapter 7 Introduce yourself to Git 17](#_Toc31972560)

[7.1 More about git config 18](#_Toc31972561)

[7.1.1 Configure the Git editor 18](#_Toc31972562)

[**Chapter 8 Install a Git client** 19](#_Toc31972563)

[8.1 What is a Git client? Why would you want one? 19](#_Toc31972564)

[8.2 A picture is worth a thousand words 20](#_Toc31972565)

[8.3 No one is giving out Git Nerd merit badges 21](#_Toc31972566)

[8.4 Recommended Git clients 22](#_Toc31972567)

[Can you hear me now? 23](#_Toc31972568)

[Chapter 9 Connect to GitHub 23](#_Toc31972569)

[9.1 Make a repo on GitHub 24](#_Toc31972570)

[9.2 Clone the repo to your local computer 24](#_Toc31972571)

[9.3 Make a local change, commit, and push 26](#_Toc31972572)

[9.3.1 Windows and line endings 27](#_Toc31972573)

[9.4 Confirm the local change propagated to the GitHub remote 28](#_Toc31972574)

[9.5 Am I really going to type GitHub username and password on each push? 28](#_Toc31972575)

[9.6 Clean up 29](#_Toc31972576)

[Chapter 10 Cache credentials for HTTPS 29](#_Toc31972577)

[10.1 You might not need to do anything! 29](#_Toc31972578)

[10.1.1 Windows 30](#_Toc31972579)

[10.1.2 macOS 30](#_Toc31972580)

[10.2 Special consideration re: two-factor authentication 30](#_Toc31972581)

[10.3 Get a test repository 30](#_Toc31972582)

[10.4 Verify that your Git is new enough to have a credential helper 31](#_Toc31972583)

[**10.5 Turn on the credential helper** 32](#_Toc31972584)

[10.5.0.4 Linux 32](#_Toc31972585)

[10.5.1 Trigger a username / password challenge 32](#_Toc31972586)

[Chapter 11 Set up keys for SSH 33](#_Toc31972587)

[11.1 SSH keys 33](#_Toc31972588)

[11.2 SSH outline and advice 33](#_Toc31972589)

[11.3 Do you already have keys? 34](#_Toc31972590)

[11.3.1 From RStudio 34](#_Toc31972591)

[11.3.2 From the shell 34](#_Toc31972592)

[11.4 Create an SSH key pair 35](#_Toc31972593)

[11.4.1 Option 1: Set up from RStudio 35](#_Toc31972594)

[11.4.2 Option 2: Set up from the shell 35](#_Toc31972595)

[11.4.3 Add key to ssh-agent 36](#_Toc31972596)

[11.4.3.2 Windows 36](#_Toc31972597)

[11.4.3.3 Linux 37](#_Toc31972598)

[11.5 Provide public key to GitHub 37](#_Toc31972599)

[11.5.1 RStudio to clipboard 37](#_Toc31972600)

[11.5.2 Shell to clipboard 38](#_Toc31972601)

[11.5.3 On GitHub 38](#_Toc31972602)

[11.6 Troubleshooting 38](#_Toc31972603)

[11.6.1 HTTPS vs SSH 38](#_Toc31972604)

[11.6.2 git2r – or some other tool – can’t find SSH keys on Windows 39](#_Toc31972605)

[11.6.3 Other 40](#_Toc31972606)

[Chapter 12 Connect RStudio to Git and GitHub 41](#_Toc31972607)

[12.1 Prerequisites 41](#_Toc31972608)

[12.2 Make a repo on GitHub 41](#_Toc31972609)

[12.3 Clone the new GitHub repository to your computer via RStudio 42](#_Toc31972610)

[12.4 Make local changes, save, commit 42](#_Toc31972611)

[12.5 Push your local changes online to GitHub 43](#_Toc31972612)

[12.6 Confirm the local change propagated to the GitHub remote 43](#_Toc31972613)

[12.7 Were you challenged for GitHub username and password? 43](#_Toc31972614)

[12.8 Clean up 44](#_Toc31972615)

[Chapter 13 Detect Git from RStudio 44](#_Toc31972616)

[13.1 Do you have a problem? 44](#_Toc31972617)

[13.2 Find Git yourself 45](#_Toc31972618)

[13.3 Tell RStudio where to find Git 45](#_Toc31972619)

[Chapter 14 RStudio, Git, GitHub Hell 47](#_Toc31972620)

[14.1 I think I have installed Git but damn if I can find it 47](#_Toc31972621)

[14.2 RStudio Git pane disappears on Mac OS 48](#_Toc31972622)

[14.3 Dysfunctional PATH 49](#_Toc31972623)

[14.4 Push/Pull buttons greyed out in RStudio 49](#_Toc31972624)

[14.5 I have no idea if my local repo and my remote repo are connected. 51](#_Toc31972625)

[14.6 Push fail at the RStudio level 51](#_Toc31972626)

[14.7 Push rejected, i.e. fail at the Git/GitHub level 51](#_Toc31972627)

[14.8 RStudio is not making certain files available for staging/committing 51](#_Toc31972628)

[14.9 I hear you have some Git repo inside your Git repo 52](#_Toc31972629)

[Get started with GitHub 52](#_Toc31972630)

[Chapter 15 New project, GitHub first 52](#_Toc31972631)

[15.1 Make a repo on GitHub 53](#_Toc31972632)

[15.2 New RStudio Project via git clone 53](#_Toc31972633)

[15.2.1 Optional: peek under the hood 54](#_Toc31972634)

[15.3 Make local changes, save, commit 54](#_Toc31972635)

[15.4 Push your local changes to GitHub 55](#_Toc31972636)

[15.5 Confirm the local change propagated to the GitHub remote 55](#_Toc31972637)

[15.6 Make a change on GitHub 56](#_Toc31972638)

[15.7 Pull from GitHub 56](#_Toc31972639)

[15.8 The end 56](#_Toc31972640)

[Chapter 16 Existing project, GitHub first 57](#_Toc31972641)

[16.1 Make a repo on GitHub 57](#_Toc31972642)

[16.2 New RStudio Project via git clone 57](#_Toc31972643)

[16.3 Bring your existing project over 58](#_Toc31972644)

[16.4 Stage and commit 58](#_Toc31972645)

[16.5 Push your local changes to GitHub 58](#_Toc31972646)

[16.6 Confirm the local change propagated to the GitHub remote 59](#_Toc31972647)

[16.7 The end 59](#_Toc31972648)

[Chapter 17 Existing project, GitHub last 59](#_Toc31972649)

[17.1 Make or verify an RStudio Project 60](#_Toc31972650)

[17.2 Make or verify a Git repo 60](#_Toc31972651)

[17.3 Stage and commit 60](#_Toc31972652)

[17.4 Make and connect a GitHub repo, option 1 61](#_Toc31972653)

[17.5 Make and connect a GitHub repo, option 2 61](#_Toc31972654)

[17.5.1 Make a new repo on GitHub 61](#_Toc31972655)

[17.5.2 Connect local repo to GitHub repo 61](#_Toc31972656)

[17.5.3 In RStudio 62](#_Toc31972657)

[17.5.4 In the shell 62](#_Toc31972658)

[17.6 Confirm the local files propagated to the GitHub remote 62](#_Toc31972659)

[17.7 The end 62](#_Toc31972660)

[Chapter 18 Test drive R Markdown 63](#_Toc31972661)

[18.1 Hello World 63](#_Toc31972662)

[18.2 Push to GitHub 64](#_Toc31972663)

[18.3 Output format 64](#_Toc31972664)

[18.4 Push to GitHub 66](#_Toc31972665)

[18.5 Put your stamp on it 66](#_Toc31972666)

[18.6 Develop your report 67](#_Toc31972667)

[18.7 Publish your report 67](#_Toc31972668)

[18.8 HTML on GitHub 68](#_Toc31972669)

[18.9 Troubleshooting 68](#_Toc31972670)

[Chapter 19 Render an R script 70](#_Toc31972671)

[19.1 Morph R Markdown into a renderable R script 70](#_Toc31972672)

[19.2 Write a render-ready R script 71](#_Toc31972673)

[Some Git basics 72](#_Toc31972674)

[Chapter 20 Repo, commit, diff, tag 73](#_Toc31972675)

[20.1 Repos or repositories 73](#_Toc31972676)

[20.2 Commits, diffs, and tags 74](#_Toc31972677)

[Chapter 21 Git commands 76](#_Toc31972678)

[Chapter 22 Branches 78](#_Toc31972679)

[22.1 Create a new branch 78](#_Toc31972680)

[22.2 Switching branches 79](#_Toc31972681)

[22.3 Merging a branch 79](#_Toc31972682)

[22.4 Dealing with conflicts 79](#_Toc31972683)

[22.4.1 Bailing out 81](#_Toc31972684)

[Chapter 23 Remotes 81](#_Toc31972685)

[23.1 Listing what remotes exist 81](#_Toc31972686)

[23.2 Adding a new remote 81](#_Toc31972687)

[23.3 Fetching data from remotes 82](#_Toc31972688)

[23.4 Pushing to remotes 82](#_Toc31972689)

[23.5 Renaming and changing remotes 83](#_Toc31972690)

[23.6 Upstream tracking branches 83](#_Toc31972691)

[Useful Git patterns for real life 84](#_Toc31972692)

[Chapter 24 The Repeated Amend 84](#_Toc31972693)

[24.1 Rock climbing analogy 84](#_Toc31972694)

[24.2 Workflow sketch 85](#_Toc31972695)

[24.2.1 Initial condition 85](#_Toc31972696)

[24.2.2 Get to work 86](#_Toc31972697)

[24.3 What if I need to fall back? 88](#_Toc31972698)

[24.4 Why don’t we push intermediate progress? 88](#_Toc31972699)

[24.5 Um, what if I did push? 89](#_Toc31972700)

[Chapter 25 Dealing with push rejection 90](#_Toc31972701)

[25.1 She who pushes first wins! 91](#_Toc31972702)

[25.2 Stay in touch 91](#_Toc31972703)

[25.3 Use branches 91](#_Toc31972704)

[Chapter 26 Pull, but you have local work 92](#_Toc31972705)

[26.1 Local work is uncommitted 93](#_Toc31972706)

[26.1.1 Happy simple cases 93](#_Toc31972707)

[26.1.2 git stash works, sometimes 93](#_Toc31972708)

[26.1.3 git stash with conflicts 95](#_Toc31972709)

[26.2 Local work is committed 97](#_Toc31972710)

[26.2.1 Pull (fetch and merge) 97](#_Toc31972711)

[26.2.2 Pull and rebase 98](#_Toc31972712)

[26.3 Other approaches 99](#_Toc31972713)

[**26.3.1 Use a temporary branch for local work** 99](#_Toc31972714)

[26.4 Some local work is committed, some is not 100](#_Toc31972715)

[Chapter 27 Time travel: See the past 100](#_Toc31972716)

[27.1 Hyperlinks are awesome! 100](#_Toc31972717)

[27.2 Browse commit history and specific commits 101](#_Toc31972718)

[27.2.1 Use hyperlinks yourself! 102](#_Toc31972719)

[27.3 File driven inquiries 102](#_Toc31972720)

[27.3.1 Blame 102](#_Toc31972721)

[27.3.2 History 103](#_Toc31972722)

[27.3.3 Hyperlink to specific lines at a specific state 103](#_Toc31972723)

[27.4 Search 103](#_Toc31972724)

[27.4.1 Issue search 104](#_Toc31972725)

[Chapter 28 Fork and clone 104](#_Toc31972726)

[28.1 Initial workflow 104](#_Toc31972727)

[28.2 usethis::create\_from\_github("OWNER/REPO") 106](#_Toc31972728)

[28.3 Engage with the new repo 106](#_Toc31972729)

[28.4 Don’t mess with master 107](#_Toc31972730)

[28.5 The original repo as a remote 107](#_Toc31972731)

[28.5.1 No, you can’t do this via GitHub 109](#_Toc31972732)

[Chapter 29 Get upstream changes for a fork 109](#_Toc31972733)

[29.1 No, you can’t do this via GitHub 110](#_Toc31972734)

[29.2 Initial conditions 110](#_Toc31972735)

[29.3 List your remotes 111](#_Toc31972736)

[29.4 Add the upstream remote 113](#_Toc31972737)

[29.4.1 Command line Git 113](#_Toc31972738)

[29.4.2 RStudio 114](#_Toc31972739)

[29.5 Verify your upstream remote 114](#_Toc31972740)

[29.6 Pull changes from upstream 115](#_Toc31972741)

[29.7 Push these changes to origin/master 116](#_Toc31972742)

[29.8 Um, what if I did touch master? 116](#_Toc31972743)

[Chapter 30 Explore and extend a pull request 118](#_Toc31972744)

[30.1 Terminology 118](#_Toc31972745)

[30.2 Official GitHub advice, Version 1 118](#_Toc31972746)

[30.3 Official GitHub advice, Version 2 119](#_Toc31972747)

[30.4 Official GitHub advice, Version 3 120](#_Toc31972748)

[30.5 My under-development workflow 120](#_Toc31972749)

[Chapter 31 Make a GitHub repo browsable 121](#_Toc31972750)

[31.1 Be savvy about your files 121](#_Toc31972751)

[31.2 Get over your hang ups re: committing derived products 122](#_Toc31972752)

[31.3 Markdown 122](#_Toc31972753)

[31.4 R Markdown 123](#_Toc31972754)

[31.5 README.md 124](#_Toc31972755)

[31.6 Finding stuff 125](#_Toc31972756)

[31.7 HTML 125](#_Toc31972757)

[31.8 Source code 126](#_Toc31972758)

[31.9 Delimited files 126](#_Toc31972759)

[31.10 PNGs 127](#_Toc31972760)

[31.11 Other document formats 127](#_Toc31972761)

[31.12 Linking to a ZIP archive of your repo 128](#_Toc31972762)

[31.13 Links and embedded figures 129](#_Toc31972763)

[31.14 Let people correct you on the internet 129](#_Toc31972764)

[Chapter 32 Clone a repo 129](#_Toc31972765)

[Chapter 34 Burn it all down 130](#_Toc31972766)

[Chapter 35 Resetting 131](#_Toc31972767)

[Chapter 39 Bookdown cheat sheet 132](#_Toc31972768)

[39.1 Heading blah blah 132](#_Toc31972769)

[39.2 About labelling things 132](#_Toc31972770)

[39.3 Cross-references 132](#_Toc31972771)

[39.4 Figures, tables, citations 133](#_Toc31972772)

[39.5 How the square bracket links work 135](#_Toc31972773)

[A The shell 136](#_Toc31972774)

[A The shell 139](#_Toc31972775)

[A.1 What is the shell? 141](#_Toc31972776)

[A.2 Starting the shell 142](#_Toc31972777)

[A.2.1 From within RStudio 142](#_Toc31972778)

[A.2.2 Outside of RStudio 142](#_Toc31972779)

[A.2.2.2 Windows 142](#_Toc31972780)

[A.3 Windows is special … and not in a good way 142](#_Toc31972781)

[A.3.1 Git Bash 143](#_Toc31972782)

[A.3.1.1 Accessing Git Bash outside of RStudio 145](#_Toc31972783)

[A.3.2 Command prompt 147](#_Toc31972784)

[A.3.3 PowerShell 147](#_Toc31972785)

[A.3.4 Bash via Windows Services for Linux 148](#_Toc31972786)

[A.3.4.1 Windows bottom line 149](#_Toc31972787)

[A.4 Basic shell commands 149](#_Toc31972788)

[B GitHub Personal Access Tokens 150](#_Toc31972789)

[B.1 GitHub API 151](#_Toc31972790)

[B.2 How do you authenticate yourself? 152](#_Toc31972791)

[B.3 Step-by-step 152](#_Toc31972792)

[B.3.1 Create a PAT 152](#_Toc31972793)

[B.3.2 Add your PAT to .Renviron 152](#_Toc31972794)

[B.4 A note about security 154](#_Toc31972795)

[B.5 Why .Renviron instead of bash config? 154](#_Toc31972796)

[D Resources 154](#_Toc31972797)

[E References 155](#_Toc31972798)

# Set-Up

For new or existing projects, you will:

* Dedicate a directory (a.k.a “folder”) to it.
* Make it an RStudio Project.
* Make it a Git repository.
* Go about your usual business. But instead of only *saving* individual files, periodically you make a **commit**, which takes a multi-file snapshot of the entire project.
  + Have you ever versioned a file [by adding your initials or the date](http://www.phdcomics.com/comics/archive.php?comicid=1531)? That is effectively a **commit**, albeit only for a single file: it is a version that is significant to you and that you might want to inspect or revert to later.
* Push commits to GitHub periodically.
  + This is like sharing a document with colleagues on DropBox or sending it out as an email attachment. It signals you’re ready to make your work visible to others and invite comment or edits.

This is a change to your normal, daily workflow. It feels weird at first but quickly becomes second nature. FWIW, [STAT 545](http://stat545.com/) students are required to submit all coursework via GitHub. This is a major topic in class and office hours for the first two weeks. Then we practically never discuss it again.

More bad news. The [STAT 545](http://stat545.com/) pain is short-lived because students primarily work in their own repositories. Do you use GitHub to work with other people or to coordinate your own work from multiple computers? If so, after you recover from the initial setup, Git will crush you again with **merge conflicts**. And this is not one-time pain, this could be a dull ache for a long time. The best remedy is prevention, but also understanding how to back out of tricky situations and tackle them on your own terms.

The rest of this site is dedicated to walking you through the necessary setup and creating your first few Git projects. We conclude with prompts that guide you through some of the more advanced usage that makes all of this initial pain worthwhile.

# Features of GitHub

In addition to a well-designed user interface, GitHub offers two especially important features:

* **Issues.** Remember how we’re high-jacking software development tools? Well, this is the bug tracker. It’s a list of things … bugs, feature requests, to dos, whatever.
  + Issues are tightly integrated with email and therefore allow you to copy/embed important conversations in the associated repo.
  + Issues can be assigned to people (e.g., to dos) and tagged (“bug” or “progress-report”).
  + Issues are tightly integrated with commits and therefore allow you to record *that the changes in this commit solve that problem which was discussed in that issue*.
  + As a new user of GitHub, one of the most productive things you can do is to use GitHub issues to provide a clear bug report or feature request for a package you use.
* **Pull requests.** Git allows a project to have multiple, independent branches of development, with the notion that some should eventually be merged back into the main development branch. These are technical Git terms but hopefully also make sense on their own. A pull request is a formal proposal that says: “Here are some changes I would like to make.” It might be linked to a specific issue: “Related to #14.” or “Fixes #56”. GitHub facilitates and preserves the discussion of the proposal, holistically and line-by-line.

## set-up

Optional reading on the big picture motivation: [Why Git? Why GitHub?](https://happygitwithr.com/big-picture.html#big-picture)

**It is vital that you attempt to set up your system in advance. You cannot show up at the workshop with no preparation and keep up!**

Try this. Best case scenario is about 1 - 2 hours. If you hit a wall, we will help:

* [Register a free GitHub account](https://happygitwithr.com/github-acct.html#github-acct).
* [Install or update R and RStudio](https://happygitwithr.com/install-r-rstudio.html#install-r-rstudio).
* [Install Git](https://happygitwithr.com/install-git.html#install-git).
* [Introduce yourself to Git](https://happygitwithr.com/hello-git.html#hello-git).
* [Prove local Git can talk to GitHub](https://happygitwithr.com/push-pull-github.html#push-pull-github).
* [Cache your username and password](https://happygitwithr.com/credential-caching.html#credential-caching) or [set up SSH keys](https://happygitwithr.com/ssh-keys.html#ssh-keys) so you don’t need to authenticate yourself to GitHub interactively *ad nauseum*.
* [Create and save a GitHub Personal Access Token (PAT)](https://happygitwithr.com/github-pat.html#github-pat).
* [Prove RStudio can find local Git](https://happygitwithr.com/rstudio-git-github.html#rstudio-git-github) and, therefore, can talk to GitHub.
  + FYI: this is where our hands-on activities usually start. We walk through a similar activity together, with narrative, and build from there.
* Contemplate if you’d like to [install an optional Git client](https://happygitwithr.com/git-client.html#git-client), now or in future.

Troubleshooting:

* Sometimes RStudio [needs a little help finding Git](https://happygitwithr.com/rstudio-see-git.html#rstudio-see-git).
* General troubleshooting: [RStudio, Git, GitHub Hell](https://happygitwithr.com/troubleshooting.html#troubleshooting).

These are battle-tested instructions, so most will succeed. We believe in you! If you have trouble, reach out for help and stick with it. Where to get help:

* If you are enrolled in an upcoming workshop, find it below to get specifics on pre-workshop support.
* We *might* be able to respond to a GitHub issue [here](https://github.com/jennybc/happy-git-with-r/issues).
* If there is a clear R/RStudio angle, post on [https://community.rstudio.com](https://community.rstudio.com/).
* General advice: search with Google and on [https://stackoverflow.com](https://stackoverflow.com/), see also [https://github.community](https://github.community/).

## Username advice

You will be able to upgrade to a paid level of service, apply discounts, join organizations, etc. in the future, so don’t fret about any of that now. **Except your username. You might want to give that some thought.**

A few tips, which sadly tend to contradict each other:

* Incorporate your actual name! People like to know who they’re dealing with. Also makes your username easier for people to guess or remember.
* Reuse your username from other contexts, e.g., Twitter or Slack. But, of course, someone with no GitHub activity will probably be squatting on that.
* Pick a username you will be comfortable revealing to your future boss.
* Shorter is better than longer.
* Be as unique as possible in as few characters as possible. In some settings GitHub auto-completes or suggests usernames.
* Make it timeless. Don’t highlight your current university, employer, or place of residence, e.g. JennyFromTheBlock.
* Avoid words laden with special meaning in programming. In my first inept efforts to script around the GitHub API, I assigned lots of issues to [the guy with username NA](https://github.com/na) because my vector of GitHub usernames contained missing values. A variant of [Little Bobby Tables](https://xkcd.com/327/).
* Avoid the use of upper vs. lower case to separate words. We highly recommend all lowercase. GitHub treats usernames in a case insensitive way, but using all lowercase is kinder to people doing downstream regular expression work with usernames, in various languages. A better strategy for word separation is to use a hyphen - or underscore \_.

You can change your username later, but better to get this right the first time.

* <https://help.github.com/articles/changing-your-github-username/>
* <https://help.github.com/articles/what-happens-when-i-change-my-username/>

## 4.2 Free private repos

GitHub offers free unlimited private repositories for all users. These free private repositories support up to three external collaborators, making them a perfect place for your personal projects, for job applications, and testing things out before making your project open source.

Go ahead and register your free account NOW and then pursue any special offer that applies to you:

* Students, faculty, and educational/research staff: [GitHub Education](https://education.github.com/).
  + GitHub “Organizations” can be extremely useful for courses or research/lab groups, where you need some coordination across a set of repos and users.
* Official nonprofit organizations and charities: [GitHub for Good](https://github.com/nonprofit)

## Install or upgrade R and RStudio

1. Install a pre-compiled binary of R for your OS from here:  
   [https://cloud.r-project.org](https://cloud.r-project.org/)  
   Already have R installed? **Hold on: This is a great time to make sure your R installation is current.** Check your current version like so:

* R.version.string

#> [1] "R version 3.6.2 (2017-01-27)"

* Install RStudio Desktop.  
  Already have RStudio? **Hold on: This is a great time to upgrade to the latest Preview version.** Download it here:  
  <https://www.rstudio.com/products/rstudio/download/preview/>
* Update your R packages:

update.packages(ask = FALSE, checkBuilt = TRUE)

## 5.1 How to think about upgrading R and RStudio

**Get current, people.** You don’t want to adopt new things on day one. But at some point, running old versions of software adds unnecessary difficulty.

In live workshops, there is a limit to how much we can help with ancient versions of R or RStudio. Also, frankly, there is a limit to our motivation. By definition, these problems are going away and we’d rather focus on edge cases with current versions, which affect lots of people.

Is your R version “old”? It is good to be on the current minor version. For example, when the released version of R is 3.5.2, you want to be on 3.5.something. The previous minor version, referred to as “r-oldrel”, would be 3.4.something in this example. This probably won’t cause trouble. Once you are 2 minor versions behind (3.3 or earlier in our example), you will start to suffer. In particular, you can no longer install pre-built binary add-on packages from CRAN.

Is your RStudio “old”? Unless you have a specific reason to prefer the released version, try the Preview. RStudio is fairly conservative with official releases, so the Preview version is used by many people for their daily work. This allows you to enjoy the latest goodies much sooner. The Preview version updates much more frequently (and in smaller increments) than the released version. This is something you might update once every month or so.

# Chapter 6 Install Git

You need Git, so you can use it at the command line and so RStudio can call it.

If there’s any chance it’s installed already, verify that, rejoice, and skip this step.

Otherwise, find installation instructions below for your operating system.

## 6.1 Git already installed?

Go to the shell (Appendix [A](https://happygitwithr.com/shell.html#shell)). Enter which git to request the path to your Git executable:

which git

## /usr/bin/git

and git --version to see its version:

git --version

## git version 2.21.0

If you are successful, that’s great! You have Git already. No need to install! Move on.

If, instead, you see something more like git: command not found, keep reading.

macOS users might get an immediate offer to install command line developer tools. Yes, you should accept! Click “Install” and read more below.

## 6.2 Windows

**Option 1** (*highly recommended*): Install [Git for Windows](https://git-for-windows.github.io/), also known as msysgit or “Git Bash”, to get Git in addition to some other useful tools, such as the Bash shell. Yes, all those names are totally confusing, but you might encounter them elsewhere and I want you to be well-informed.

We like this because Git for Windows leaves the Git executable in a conventional location, which will help you and other programs, e.g. RStudio, find it and use it. This also supports a transition to more expert use, because the “Git Bash” shell will be useful as you venture outside of R/RStudio.

* **NOTE:** When asked about “Adjusting your PATH environment”, make sure to select “Git from the command line and also from 3rd-party software”. Otherwise, we believe it is good to accept the defaults.
* Note that RStudio for Windows prefers for Git to be installed below C:/Program Files and this appears to be the default. This implies, for example, that the Git executable on my Windows system is found at C:/Program Files/Git/bin/git.exe. Unless you have specific reasons to otherwise, follow this convention.

This also leaves you with a Git client, though not a very good one. So check out Git clients we recommend (chapter [8](https://happygitwithr.com/git-client.html#git-client)).

FYI, this appears to be equivalent to what you would download from here: <https://git-scm.com/download/>.

**Option 2** (*recommended*): Install [Git for Windows](https://git-for-windows.github.io/) via the [Chocolatey](https://chocolatey.org/) package manager. If this means anything to you, Chocolatey is like [apt-get](https://en.wikipedia.org/wiki/APT_(Debian)) or [Homebrew](https://brew.sh/), but for Windows instead of Debian/Ubuntu Linux or macOS. As far as I can tell, using Chocolatey to install Git for Windows gives the same result as installing it yourself (option 1).

This obviously requires that you already have [Chocolatey](https://chocolatey.org/) installed or that you are up for installing it. It is not hard and the [instructions are here](https://chocolatey.org/install). This may be worthwhile if it seems likely you will be installing more open source software in the future.

After you install Chocolatey, in a shell (Appendix [A](https://happygitwithr.com/shell.html#shell)), do:

choco install git.install

This installs the most current [Git (Install) X.Y.Z](https://chocolatey.org/packages/git.install) Chocolatey package. At the time of writing, that is “Git (Install) 2.20.1”, but that version number will increment over time.

If you [search Chocolatey packages](https://chocolatey.org/packages) yourself, you might see two packages that install Git – “Git (Install) 2.20.1” and “Git 2.20.1”, at the time of writing. I believe “Git (Install) 2.20.1” is technically the more correct, but I also think it doesn’t really matter which one you use. A rather confusing explanation is found [here](https://chocolatey.org/faq#what-is-the-difference-between-packages-no-suffix-as-compared-to-install-portable). Don’t worry too much about whether you do choco install git.install or choco install git.

## 6.4 Linux

Install Git via your distro’s package manager.

Ubuntu or Debian Linux:

sudo apt-get install git

Fedora or RedHat Linux:

sudo yum install git

A comprehensive list for various Linux and Unix package managers:

<https://git-scm.com/download/linux>

# Chapter 7 Introduce yourself to Git

In the shell (Appendix [A](https://happygitwithr.com/shell.html#shell)):

git config --global user.name 'Jane Doe'

git config --global user.email 'jane@example.com'

git config --global --list

substituting your name and **the email associated with your GitHub account**.

The [usethis package](https://usethis.r-lib.org/) offers an alternative approach. You can set your Git user name and email from within R:

## install if needed (do this exactly once):

## install.packages("usethis")

library(usethis)

use\_git\_config(user.name = "Jane Doe", user.email = "jane@example.org")

## 7.1 More about git config

An easy way to get into a shell from RStudio is *Tools > Terminal* or *Tools > Shell*. More about the shell in the Appendix [A](https://happygitwithr.com/shell.html#shell).

Special Windows gotchas: If you are struggling on Windows, consider there are different types of shell and you might be in the wrong one. You want to be in a “Git Bash” shell, as opposed to Power Shell or the legacy cmd.exe command prompt. Read more in [the Appendix](https://happygitwithr.com/shell.html#windows-shell-hell). This might also be a reason to do this configuration via the usethis package in R.

What user name should you give to Git? This does not have to be your GitHub user name, although it can be. Another good option is your actual first name and last name. If you commit from different machines, sometimes people work that info into the user name. Your commits will be labelled with this user name, so make it informative to potential collaborators and future you.

What email should you give to Git? This **must** be the email associated with your GitHub account.

These commands return nothing. You can check that Git understood what you typed by looking at the output of git config --global --list.

## 7.1.1 Configure the Git editor

Another Git option that many people eventually configure is the editor. At some point, you will fail to give Git what it wants in terms of a commit message and it will kick you into an editor. This can be distressing, if it’s not your editor of choice and you don’t even know how to save and quit. You can enforce your will with something along these lines:

git config --global core.editor "emacs"

Substitute your preferred editor for "emacs" here. Software Carpentry’s Git lesson has a comprehensive listing of the exact git config command needed for [many combinations of OS and editor](http://swcarpentry.github.io/git-novice/02-setup/).

Chapter 8 Install a Git clien**t**

This is optional but **highly recommended**.

Learning to use version control can be rough at first. I found the use of a GUI – as opposed to the command line – extremely helpful when I was getting started. I call this sort of helper application a Git client. It’s really a Git(Hub) client because it also helps you interact with GitHub or other remotes.

A Git client is not required for live workshops and will not be explicitly taught, though you might see us using one of these clients.

## 8.1 What is a Git client? Why would you want one?

“Git” is really just a collection of individual commands you execute in the shell (Appendix [A](https://happygitwithr.com/shell.html#shell)). This interface is not appealing for everyone. Some may prefer to do Git operations via a client with a graphical interface.

Git and your Git client are not the same thing, just like R and RStudio are not the same thing. A Git client and an [integrated development environment](https://en.wikipedia.org/wiki/Integrated_development_environment), such as RStudio, are not necessary to use Git or R, respectively. But they make the experience more pleasant because they reduce the amount of [command line bullshittery](http://www.pgbovine.net/command-line-bullshittery.htm) and provide a richer visual representation of the current state.

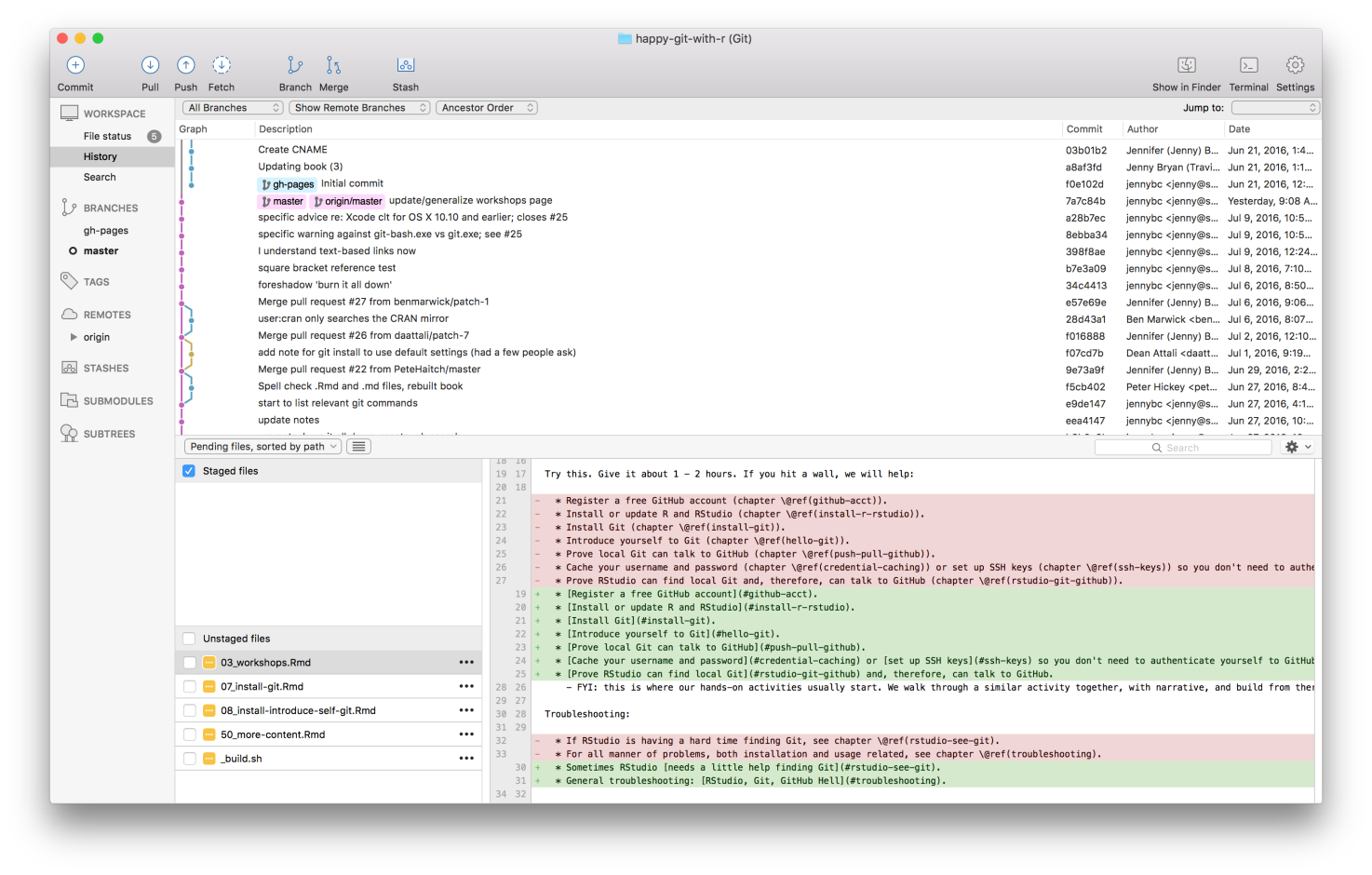
RStudio offers a very basic Git client via its Git pane. I use this often for simple operations, but you probably want another, more powerful one as well.

Fair warning: for some tasks, you must use the command line. But the more powerful your Git client is, the less often this happens. The visual overview given by your Git client can also be invaluable for understanding the current state of things, even when preparing calls to command line Git.

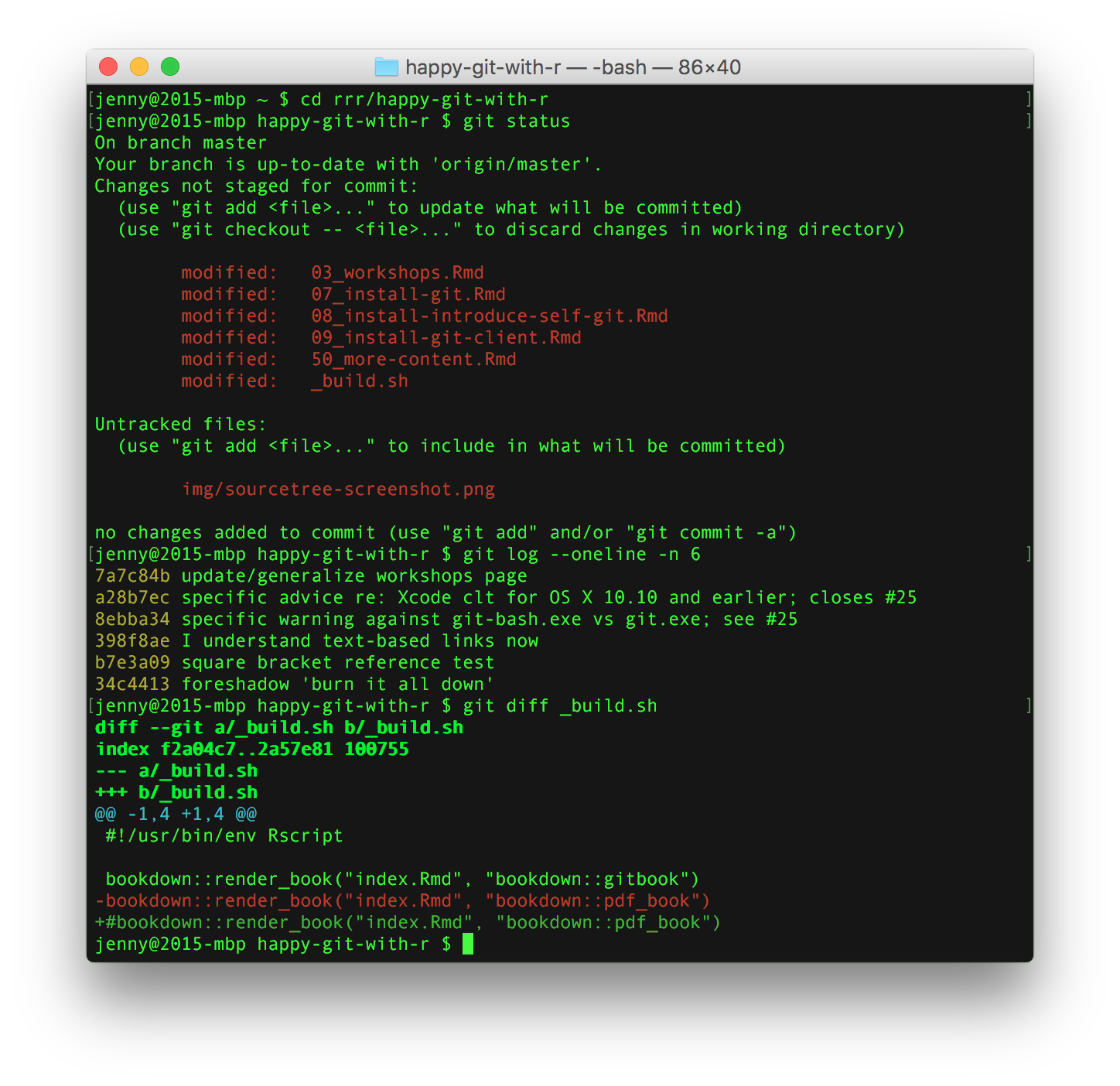
Fantastic news: because all of the clients are just forming and executing Git commands on your behalf, you don’t have to pick one. You can literally do one operation from the command line, do another from RStudio, and another from SourceTree, one after the other, and it just works. *Very rarely, both clients will scan the repo at the same time and you’ll get an error message about* .git/index.lock*. Try the operation again at least once before doing any further troubleshooting.*

## 8.2 A picture is worth a thousand words

Here’s a screenshot of SourceTree (see below) open to the repository for this site. You get a nice graphical overview of the recent commit history, branches, and diffs, as well as a GUI that facilitates the most common Git operations.



In contrast, here’s a shell session where I’ve used command line Git to access some of the same information.



## 8.3 No one is giving out Git Nerd merit badges

Work with Git in whatever way makes you most effective. Feel free to revisit your approach over time or to use different approaches for different tasks. No one can tell whether you use the command line or a GUI when they look at your Git history or your GitHub repo.

I sometimes encounter people who feel it’s “better” to use command line Git, but for very ill-defined reasons. These people may feel like they *should* work in the shell, even if it leads to Git-avoidance, frequent mistakes, or limiting themselves to a small set of ~3 Git commands. This is counterproductive.

I had two false starts with Git, where I failed to get proficient enough, quickly enough to truly incorporate version control into my daily work. I found a visual Git client invaluable – SourceTree, in my case. It made me willing to use Git multiple times per day, for a sustained period of time. This helped me build the mental model necessary for more advanced Git operations like rebasing, cherry-picking, and resetting.

If your Git life happens on your own computer, there is no reason to deny yourself a GUI if that’s what you like. If you prefer working in the shell or if you frequently log into a remote server, then it makes sense to prioritize building Git skills at the command line. Do whatever works for you, but don’t do anything for the sake of purity or heroism.

## 8.4 Recommended Git clients

* [GitKraken](https://www.gitkraken.com/) is a free, powerful Git(Hub) client that is my current favorite. It’s especially exciting because it works on Windows, macOS, and Linux. This is great news, especially for long-suffering Linux users who have previously had very few options.

* [SourceTree](https://www.sourcetreeapp.com/) is another free client that I highly recommend, at least on Windows.[1](https://happygitwithr.com/git-client.html#fn1) It was my first and most beloved Git client, but I eventually had to give it up on macOS, due to a [long-standing bug re: leaking file handles](http://openradar.appspot.com/radar?id=1387401) that they will clearly never fix. I still use SourceTree on Windows.
* GitHub offers a free Git(Hub) client, [GitHub Desktop](https://desktop.github.com/), for Windows and macOS. Although we previously discouraged its use, GitHub’s client has since gotten a thorough makeover that eliminates several of our concerns, so we’re cautiously optimistic. GitHub Desktop is aimed at beginners who want the most useful features of Git front and center. The flipside is that it may not support some of the more advanced workflows exposed by the clients above. If you choose GitHub Desktop and it works well for you, we’d love to hear more in an issue, so we can continue to refine this recommendation.
* [GitUp](https://gitup.co/) is a free, open source client for macOS. I’ve heard good things about it and like what I read on the website. However, I tried to make myself use it for a day and went sort of nuts, possibly because I was already so used to other clients. YMMV.
* Others that I have heard positive reviews for:
  + [SmartGit](http://www.syntevo.com/smartgit/)
  + [git-cola](https://git-cola.github.io/)
  + [magit](https://magit.vc/), for Emacs nerds
* Browse [even more Git(Hub) clients](http://git-scm.com/downloads/guis).

1. During installation and registration, you’ll need to create a free Atlassian Bitbucket account and link that to a free Atlassian Bitbucket Cloud account. Also, feel free to uncheck the checkbox about installing Mercurial (another version control system), unless you feel you need it.[↩](https://happygitwithr.com/git-client.html#fnref1)

## Can you hear me now?

The next few chapters walk through some basic operations to confirm you have installed the necessary software and that the necessary connections are being made, between tools on your computer and between your computer and GitHub.

This has a lot of overlap with some basic workflows we revisit later, but the second time around (or in a live workshop), we’ll spend more time explaining what’s happening and why.

In [Connect to GitHub](https://happygitwithr.com/push-pull-github.html#push-pull-github) we use Git in the shell to make sure you can clone a repo from GitHub and establish two-way communications.

In [Cache credentials for HTTPS](https://happygitwithr.com/credential-caching.html#credential-caching) and [Set up keys for SSH](https://happygitwithr.com/ssh-keys.html#ssh-keys) we present two approaches for persistently authenticating yourself with GitHub, so you don’t need to provide credentials *ad nauseam*.

In [Connect RStudio to Git and GitHub](https://happygitwithr.com/rstudio-git-github.html#rstudio-git-github) we confirm that RStudio can work with your Git installation to perform local operations and communicate with GitHub.

Hopefully you won’t need it, but this part concludes with two troubleshooting chapters: [Detect Git from RStudio](https://happygitwithr.com/rstudio-see-git.html#rstudio-see-git) and [RStudio, Git, GitHub Hell](https://happygitwithr.com/troubleshooting.html#troubleshooting).

# Chapter 9 Connect to GitHub

Objective: make sure that you can pull from and push to GitHub from your computer.

I do not explain all the shell (Appendix [A](https://happygitwithr.com/shell.html#shell)) and Git commands in detail. This is a black box diagnostic / configuration exercise. In later chapters and in live workshops, we revisit these operations with much more narrative.

## 9.1 Make a repo on GitHub

Go to [https://github.com](https://github.com/) and make sure you are logged in.

Click green “New repository” button. Or, if you are on your own profile page, click on “Repositories”, then click the green “New” button.

How to fill this in:

* Repository name: myrepo (or whatever you wish, we’ll delete this soon anyway).
* Description: “testing my setup” (or whatever, but some text is good for the README).
* Public.
* YES Initialize this repository with a README.

For everything else, just accept the default.

Click big green button “Create repository.”

Copy the HTTPS clone URL to your clipboard via the green “Clone or Download” button.

## 9.2 Clone the repo to your local computer

Go to the shell (Appendix [A](https://happygitwithr.com/shell.html#shell)).

Take charge of – or at least notice! – what directory you’re in. pwd displays the working directory. cd is the command to change directory. Personally, I would do this sort of thing in ~/tmp.

Clone myrepo from GitHub to your computer. This URL should have **your GitHub username** and the name of **your practice repo**. If your shell (Appendix [A](https://happygitwithr.com/shell.html#shell)) cooperates, you should be able to paste the whole https://.... bit that we copied above. But some shells are not (immediately) clipboard aware. In that sad case, you must type it. **Accurately.**

git clone https://github.com/YOUR-USERNAME/YOUR-REPOSITORY.git

This should look something like this:

jenny@2015-mbp tmp $ git clone https://github.com/jennybc/myrepo.git

Cloning into 'myrepo'...

remote: Counting objects: 3, done.

remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0

Unpacking objects: 100% (3/3), done.

Checking connectivity... done.

Make this new repo your working directory, list its files, display the README, and get some information on its connection to GitHub:

cd myrepo

ls

head README.md

git remote show origin

This should look something like this:

jenny@2015-mbp ~ $ cd myrepo

jenny@2015-mbp myrepo $ ls

README.md

jenny@2015-mbp myrepo $ head README.md

# myrepo

tutorial development

jenny@2015-mbp myrepo $ git remote show origin

\* remote origin

Fetch URL: https://github.com/jennybc/myrepo.git

Push URL: https://github.com/jennybc/myrepo.git

HEAD branch: master

Remote branch:

master tracked

Local branch configured for 'git pull':

master merges with remote master

Local ref configured for 'git push':

master pushes to master (up to date)

## 9.3 Make a local change, commit, and push

Add a line to README and verify that Git notices the change:

echo "A line I wrote on my local computer" >> README.md

git status

This should look something like this:

jenny@2015-mbp myrepo $ echo "A line I wrote on my local computer" >> README.md

jenny@2015-mbp myrepo $ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: README.md

no changes added to commit (use "git add" and/or "git commit -a")

Stage (“add”) and commit this change and push to your remote repo on GitHub. If you’re a new GitHub user, you will be challenged for your GitHub username and password. Provide them!

git add -A

git commit -m "A commit from my local computer"

git push

The -m "blah blah blah" piece is very important! Git requires a commit message for every commit, so if you forget the -m flag, Git will prompt you for a commit message anyway. And you might not like [the editor that Git chooses](https://happygitwithr.com/hello-git.html#git-editor). It is good practice to write meaningful commit messages, so that, in the future, potential collaborators (and your future self) will understand the progression of a project.

This should look something like this:

jenny@2015-mbp myrepo $ git add -A

jenny@2015-mbp myrepo $ git commit -m "A commit from my local computer"

[master de669ba] A commit from my local computer

1 file changed, 1 insertion(+)

jenny@2015-mbp myrepo $ git push

Counting objects: 3, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 311 bytes | 0 bytes/s, done.

Total 3 (delta 0), reused 0 (delta 0)

To https://github.com/jennybc/myrepo.git

b4112c5..de669ba master -> master

### 9.3.1 Windows and line endings

On Windows, you might see a message about LF will be replaced by CRLF. This is normal and does not require any action on your part. Windows handles line endings differently from other operating systems, but the default setup for Git for Windows is appropriate for most people and situations.

Here’s a command to reveal the current line ending configuration and some typical output **on Windows**:

$ git config --show-origin --get core.autocrlf

file:"C:\\ProgramData/Git/config" true

If your value shows as false, you can set it to true with this command:

$ git config --global core.autocrlf true

true is the current default setting for core.autocrlf for [Git for Windows](https://happygitwithr.com/install-git.html#install-git-windows), our recommended method for installing Git on Windows. The need to set this explicitly in your global user config suggests you should consider reinstalling or updating Git for Windows.

## 9.4 Confirm the local change propagated to the GitHub remote

Go back to the browser. I assume we’re still viewing your new GitHub repo.

Refresh.

You should see the new “A line I wrote on my local computer” in the README.

If you click on “commits,” you should see one with the message “A commit from my local computer.”

If you have made it this far, you are ready to graduate to using Git and GitHub with RStudio (chapter [12](https://happygitwithr.com/rstudio-git-github.html#rstudio-git-github)). But first …

## 9.5 Am I really going to type GitHub username and password on each push?

It is likely that your first push, above, leads to a challenge for your GitHub username and password. This will drive you crazy in the long-run and make you reluctant to push. You want to eliminate this annoyance.

Luckily, if you’ve installed Git one of the ways recommended by Happy Git, it is likely that Git is already using a credential helper provided by your operating system! If so, your GitHub credentials were cached when you successfully pushed above. This setup applies across repos, i.e. it’s not limited to our current test repo.

I suggest you make another local change to README.md, stage (i.e. “add”) it, commit it, and push, using the commands shown above. If this “just works” and shows up on GitHub, rejoice. You are ready to work with GitHub via HTTPS without constantly re-entering your credentials. You are ready to delete this toy repo.

If you are challenged for your username and password *again*, do one of the following:

* Cache credentials for HTTPS access, chapter [10](https://happygitwithr.com/credential-caching.html#credential-caching).
* Set up SSH keys, chapter [11](https://happygitwithr.com/ssh-keys.html#ssh-keys).

Now is the perfect time to do this, since you have a functioning test repo.

## 9.6 Clean up

**Local** When you’re ready to clean up, you can delete the local repo any way you like. It’s just a regular directory on your computer.

Here’s how to do that in the shell, if current working directory is myrepo:

cd ..

rm -rf myrepo/

**GitHub** In the browser, go to your repo’s landing page on GitHub. Click on “Settings”.

Scroll down, click on “delete repository,” and do as it asks.

# Chapter 10 Cache credentials for HTTPS

If you plan to push/pull using HTTPS, you want to cache your credentials (username, password), so you don’t need to enter them over and over again. Alternatively, you could set up SSH keys, chapter [11](https://happygitwithr.com/ssh-keys.html#ssh-keys)). I suggest you set up one of these methods of authentication on each computer you want to connect to GitHub from.

I find HTTPS easier to get working quickly and **strongly recommend** it when you first start working with Git/GitHub. [It is what GitHub recommends](https://stackoverflow.com/a/11041782/2825349), presumably for exactly the same reasons. I started with HTTPS, but eventually switched over to SSH. Either is fine, and you can change your mind later.

## 10.1 You might not need to do anything!

As of January 2019, if you install Git using the [methods recommended here in Happy Git](https://happygitwithr.com/install-git.html#install-git), it is possible that Git will use a credential helper provided by your operating system. That is, you may not need to do anything special in order to cache your GitHub username and password.

Specifically, if you are on macOS or Windows, don’t do anything described here until you have actual proof that it’s necessary, i.e. that you have experienced repeated challenges for your username and password when using HTTPS.

### 10.1.1 Windows

In my most recent experiments, Git for Windows is configured to store credentials via the Windows Credentials Manager. This meant that I only needed to enter my GitHub username and password once to get setup for HTTPS.

Here’s a command to reveal the configuration and the output I see in a fresh installation of Git for Windows:

$ git config --show-origin --get credential.helper

file:C:/Program Files/Git/mingw64/etc/gitconfig manager

### 10.1.2 macOS

*I’m not currently in a position to say if credential caching “just works” on macOS in a fresh Git installations. Have you recently experienced this?* [*Let me know in an issue*](https://github.com/jennybc/happy-git-with-r/issues)*! If you need to turn it on explicitly, instructions for that are given below.*

## 10.2 Special consideration re: two-factor authentication

If you are using [two-factor authentication](https://help.github.com/articles/about-two-factor-authentication/) (a.k.a. “2FA”) with GitHub, you will not be able to use your GitHub password via HTTPS. 2FA users should probably be using [SSH](https://happygitwithr.com/ssh-keys.html#ssh-keys) anyway, but I digress.

If you really want to use HTTPS, you must enter a [personal access token](https://help.github.com/articles/creating-a-personal-access-token-for-the-command-line/) in lieu of your password. This is the same type of PAT you could use to access the [GitHub API](https://developer.github.com/v3/) and that many R users define in .Renviron as the environment variable GITHUB\_PAT.

## 10.3 Get a test repository

You need a functioning test Git repository. One that exists locally and remotely on GitHub, with the local repo tracking the remote.

If you have just verified that you can interact with GitHub (chapter [9](https://happygitwithr.com/push-pull-github.html#push-pull-github)) from your local computer, that test repo will be perfect.

If you have just verified that you can work with GitHub from RStudio (chapter [12](https://happygitwithr.com/rstudio-git-github.html#rstudio-git-github)), that test repo will also be perfect.

You may proceed when

* You have a test repo.
* You know where it lives on your local computer. Example:
  + /home/jenny/tmp/myrepo
* You know where it lives on GitHub. Example:
  + https://github.com/jennybc/myrepo
* You know the GitHub repo is setup as a remote. In a shell (Appendix [A](https://happygitwithr.com/shell.html#shell)) working directory set to the local Git repo, enter:

git remote -v

Output like this confirms that fetch and push are set to remote URLs that point to your GitHub repo:

origin https://github.com/jennybc/myrepo (fetch)

origin https://github.com/jennybc/myrepo (push)

Now enter:

git branch -vv

Here we confirm that the local master branch is tracking your GitHub master branch (origin/master). Gibberish? Just check that your output looks similar to mine:

master b8e03e3 [origin/master] line added locally

## 10.4 Verify that your Git is new enough to have a credential helper

In a shell (Appendix [A](https://happygitwithr.com/shell.html#shell)), do:

git --version

and verify your version is 1.7.10 or newer. If not, update Git (chapter [6](https://happygitwithr.com/install-git.html#install-git)) or use SSH keys (chapter [11](https://happygitwithr.com/ssh-keys.html#ssh-keys)).

**1**0.5 Turn on the credential helper

**10.5.0.1 Windows, plan A**

The preferred setting for credential.helper is now manager, which configures Git to use Git Credential Manager for Windows, which ships with Git for Windows. This may already be configured, but this command sets it explicitly:

git config --global credential.helper manager

**10.5.0.2 Windows, plan B**

If for some reason, Git Credential Manager does not work, you may have success with an older method, wincred:

git config --global credential.helper wincred

This is, however, considered deprecated and suggests you should install the current Git for Windows. So only use this as a temporary solution to get unstuck, until you can upgrade.

### 10.5.0.4 Linux

In the shell, enter:

git config --global credential.helper 'cache --timeout=10000000'

to store your password for ten million seconds or around 16 weeks, enough for a semester.

### 10.5.1 Trigger a username / password challenge

Change a file in your local repo and commit it. Do that however you wish. Here are shell commands that will work:

echo "adding a line" >> README.md

git add -A

git commit -m "A commit from my local computer"

Now push!

git push -u origin master

One last time you will be asked for your username and password, which hopefully will be cached.

Now push AGAIN.

git push

You should NOT be asked for your username and password, instead you should see Everything up-to-date.

Rejoice and close the shell.

# Chapter 11 Set up keys for SSH

If you plan to push/pull using SSH, you need to set up SSH keys. You want to do this (or cache your username and password, chapter [10](https://happygitwithr.com/credential-caching.html#credential-caching)), so you don’t have to authenticate yourself interactively with GitHub over and over again. I suggest you set up one of these methods of authentication on each computer you want to connect to GitHub from.

## 11.1 SSH keys

SSH keys provide a more secure way of logging into a server than using a password alone. While a password can eventually be cracked with a brute force attack, SSH keys are nearly impossible to decipher by brute force alone. Generating a key pair provides you with two long strings of characters: a public and a private key. You can place the public key on any server (like GitHub!), and then unlock it by connecting to it with a client that already has the private key (your computer!). When the two match up, the system unlocks without the need for a password. You can increase security even more by protecting the private key with a passphrase.

Adapted from instructions provided by [GitHub](https://help.github.com/categories/ssh/) and [Digital Ocean](https://www.digitalocean.com/community/tutorials/how-to-set-up-ssh-keys--2).

## 11.2 SSH outline and advice

High level overview of what must happen:

* Create a public-private SSH key pair. Literally, 2 special files, in a special place. Optionally, encrypt the private key with a passphrase (best practice).
* Add the private key to your ssh-agent. If you protected it with a passphrase, you may have additional configuration.
* Add your public key to your GitHub profile.

Advice:

* If you are new to programming and the shell, you’ll probably find HTTPS easier at first (chapter [10](https://happygitwithr.com/credential-caching.html#credential-caching)). You can always switch to SSH later. You can use one method from computer A and the other from computer B.
* You should swap out your SSH keys periodically. Something like once a year.
* It’s best practice to protect your private key with a passphrase. This can make setup and usage harder, so if you’re not up for that (yet), either don’t use a passphrase or seriously consider using HTTPS instead.
* Don’t do weird gymnastics in order to have only one key pair, re-used over multiple computers. You should probably have one key per computer (I do this). Some people even have one key per computer, per service (I do not do this).
* It is normal to associate multiple public keys with your GitHub account. For example, one public key for each computer you connect with.

## 11.3 Do you already have keys?

You can check this from RStudio or from the shell.

Global advice: if you do have existing keys, but have no clue where they came from or why you created them, you should seriously consider creating a new SSH key pair. It’s up to you to figure out whether/how to delete the old ones. But don’t let that keep you from creating new keys and moving forward.

### 11.3.1 From RStudio

Go to *Tools > Global Options…> Git/SVN*. If you see something like ~/.ssh/id\_rsa in the SSH RSA Key box, you definitely have existing keys. Caveat: RStudio only looks for a key pair named id\_rsa and id\_rsa.pub. This makes sense, because it’s the default and very common. But SSH keys *can* have other names. If you want to be completely certain, you should also check in the shell.

### 11.3.2 From the shell

Go to the shell (appendix [A](https://happygitwithr.com/shell.html#shell)).

List existing keys:

ls -al ~/.ssh/

If you are told ~/.ssh/ doesn’t exist, you don’t have SSH keys!

If you see a pair of files like id\_rsa.pub and id\_rsa, you have a key pair already. The typical pattern is id\_FOO.pub (the public key) and id\_FOO (the private key). If you’re happy to stick with your existing keys, skip to the sections about adding a key to the ssh-agent and GitHub.

## 11.4 Create an SSH key pair

### 11.4.1 Option 1: Set up from RStudio

Go to *Tools > Global Options…> Git/SVN > Create RSA Key…*.

RStudio prompts you for a passphrase. It is optional, but also a best practice. Configuring your system for smooth operation with a passphrase-protected key introduces more moving parts. If you’re completely new at all this, skip the passphrase (or use HTTPS!) and implement it next time, when you are more comfortable with system configuration. I did not use a passphrase at first, but I do now, and record it in a password manager.

Click “Create” and RStudio will generate an SSH key pair, stored in the files ~/.ssh/id\_rsa and ~/.ssh/id\_rsa.pub.

### 11.4.2 Option 2: Set up from the shell

Create the key pair like so, but substitute a comment that means something to you, especially if you’ll have multiple SSH keys in your life. Consider the email associated with your GitHub account or the name of your computer, e.g. you@example.com or 2018-mbp.

$ ssh-keygen -t rsa -b 4096 -C "USEFUL-COMMENT"

Accept the proposal to save the key in the default location. Just press Enter here:

Enter file in which to save the key (/Users/jenny/.ssh/id\_rsa):

You have the option to protect the key with a passphrase. It is optional, but also a best practice. Configuring your system for smooth operation with a passphrase-protected key introduces more moving parts. If you’re completely new at all this, skip the passphrase and implement it next time, when you are more comfortable with system configuration. I did not use a passphrase at first, but I do now, and record it in a password manager.

Enter passphrase (empty for no passphrase):

The process should complete now and should have looked like this:

jenny@2015-mbp ~ $ ssh-keygen -t rsa -b 4096 -C "USEFUL-COMMENT"

Generating public/private rsa key pair.

Enter file in which to save the key (/Users/jenny/.ssh/id\_rsa):

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in /Users/jenny/.ssh/id\_rsa.

Your public key has been saved in /Users/jenny/.ssh/id\_rsa.pub.

The key fingerprint is:

SHA256:ki0TNHm8qIvpH7/c0qQmdv2xxhYHCwlpn3+rVhKVeDo USEFUL-COMMENT

The key's randomart image is:

+---[RSA 4096]----+

| o+ . . |

| .=.o . + |

| ..= + + |

| .+\* E |

| .= So = |

| . +. = + |

| o.. = ..\* . |

| o ++=.o =o. |

| ..o.++o.=+. |

+----[SHA256]-----+

### 11.4.3 Add key to ssh-agent

Tell your ssh-agent about the key and, especially, set it up to manage the passphrase, if you chose to set one.

Things get a little OS-specific around here. When in doubt, consult [GitHub’s instructions for SSH](https://help.github.com/articles/connecting-to-github-with-ssh/), which is kept current for Mac, Windows, and Linux.

### 11.4.3.2 Windows

In a Git Bash shell, make sure ssh-agent is running:

$ eval $(ssh-agent -s)

Agent pid 59566

Add your key.

$ ssh-add ~/.ssh/id\_rsa

### 11.4.3.3 Linux

In a shell, make sure ssh-agent is running:

$ eval "$(ssh-agent -s)"

Agent pid 59566

Add your key.

ssh-add ~/.ssh/id\_rsa

## 11.5 Provide public key to GitHub

Now we store a copy of your public key on GitHub.

### 11.5.1 RStudio to clipboard

Go to *Tools > Global Options…> Git/SVN*. If your key pair has the usual name, id\_rsa.pub and id\_rsa, RStudio will see it and offer to “View public key”. Do that and accept the offer to copy to your clipboard. If your key pair is named differently, use another method.

### 11.5.2 Shell to clipboard

Copy the public key onto your clipboard. For example, open ~/.ssh/id\_rsa.pub in an editor and copy the contents to your clipboard. Or do one of the following at the command line:

* Mac OS: pbcopy < ~/.ssh/id\_rsa.pub
* Windows: clip < ~/.ssh/id\_rsa.pub
* Linux: xclip -sel clip < ~/.ssh/id\_rsa.pub

*Linux: if needed, install via* apt-get *or* yum*. For example,* sudo apt-get install xclip*.*

### 11.5.3 On GitHub

Make sure you’re signed into GitHub. Click on your profile pic in upper right corner and go *Settings*, then *SSH and GPG keys*. Click “New SSH key”. Paste your public key in the “Key” box. Give it an informative title, presumably related to the comment you used above, during key creation. For example, you might use 2018-mbp to record the year and computer. Click “Add SSH key”.

In theory, we’re done! You can use [ssh -T git@github.com](https://help.github.com/articles/testing-your-ssh-connection/) to test your connection to GitHub. If you’re not sure what to make of the output, see the link for details. Of course, the best test is to work through the realistic usage examples elsewhere in this guide.

## 11.6 Troubleshooting

### 11.6.1 HTTPS vs SSH

If you think you have SSH set up correctly and yet you are still challenged for credentials, consider this: for the repo in question, have you possibly set up GitHub, probably called origin, as an HTTPS remote, instead of SSH?

How to see the remote URL(s) associated with the current repo in the shell:

git remote -v

An SSH remote will look like this:

git@github.com:USERNAME/REPOSITORY.git

whereas an HTTPS remote will look like this:

https://github.com/USERNAME/REPOSITORY.git

You can toggle between these with git remote set-url:

* <https://help.github.com/articles/changing-a-remote-s-url/>

### 11.6.2 git2r – or some other tool – can’t find SSH keys on Windows

Have you seen this error message?

Error in .local(object, ...) :

Error in 'git2r\_push': error authenticating: failed connecting agent

We’ve seen it when working with Git/GitHub from R via the [git2r](https://cran.r-project.org/web/packages/git2r/index.html) package, which is used under the hood by many R packages, such as devtools, ghit, and usethis.

git2r uses the libgit2 library, not the Git you installed. This means you can have SSH keys configured properly for Git work in a Git Bash shell and from RStudio and still have problems with git2r! Ugh.

The root cause is confusion about the location of .ssh/ on Windows. R’s idea of your home directory on Windows often differs from the default location of config files for Git and ssh, such as .ssh/. On \*nix systems, these generally coincide and there’s no problem.

Two important directories on Windows are the user’s HOME and USERPROFILE. R usually associates ~ with HOME, but Git and ssh often consult USERPROFILE for their config files. On my Windows 10 VM, I see:

normalizePath("~")

#> [1] "C:\\Users\\JennyVM\\Documents"

as.list(Sys.getenv(

c("HOME", "USERPROFILE")

))

#> $HOME

#> [1] "C:/Users/JennyVM/Documents"

#>

#> $USERPROFILE

#> [1] "C:\\Users\\JennyVM"

list.files(

Sys.getenv("USERPROFILE"),

pattern = "ssh|git",

include.dirs = TRUE,

all.files = TRUE

)

#> [1] ".gitconfig" ".ssh"

Two workarounds:

* Tell git2r explicitly where to find your public and private key. Example using usethis::use\_github():
* cred <- git2r::cred\_ssh\_key(

publickey = "~/../.ssh/id\_rsa.pub",

privatekey = "~/../.ssh/id\_rsa"

)

use\_github(credentials = cred)

* [Create a symbolic link](https://www.howtogeek.com/howto/16226/complete-guide-to-symbolic-links-symlinks-on-windows-or-linux/) so that .ssh/ in R’s home directory points to your actual .ssh/ directory. Example contributed by Ian Lyttle on Windows 7 using Command Prompt:
* MKLINK /D "C:\Users\username\Documents\.ssh" "C:\Users\username\.ssh"

Finally, if git2r seems unable to get your SSH passphrase from ssh-agent, install the getPass package:

install.packages("getPass")

and git2r should launch a popup where you can enter your passphrase. Thanks to Ian Lyttle for this tip.

This link provides a great explanation of the uncertainty about where .ssh/ and user’s .gitconfig are located on Windows: [git on Windows - location of configuration files](https://www.onwebsecurity.com/configuration/git-on-windows-location-of-global-configuration-file.html). Bottom line: locate where your main tool expects and create symbolic links to help other tools find this stuff.

### 11.6.3 Other

Other things to double-check:

* Did you add the SSH to your ssh-agent?
* Did you configure Mac OS Sierra or High Sierra to persistently store your passphrase in the keychain?
* Did you add the public key to GitHub?

# Chapter 12 Connect RStudio to Git and GitHub

Here we verify that RStudio can issue Git commands on your behalf. Assuming that you’ve gotten local Git to talk to GitHub, this means you’ll also be able to pull from and push to GitHub from RStudio.

In later chapters and in live workshops, we revisit these operations with much more explanation.

If you succeed here, your set up is DONE.

## 12.1 Prerequisites

We assume the following:

* You’ve registered a free GitHub account (chapter [4](https://happygitwithr.com/github-acct.html#github-acct)).
* You’ve installed/updated R and RStudio (chapter [5](https://happygitwithr.com/install-r-rstudio.html#install-r-rstudio)).
* You’ve installed Git (chapter [6](https://happygitwithr.com/install-git.html#install-git)).
* You’ve introduced yourself to Git (chapter [7](https://happygitwithr.com/hello-git.html#hello-git)).
* You’ve confirmed that you can push to / pull from GitHub from the command line (chapter [9](https://happygitwithr.com/push-pull-github.html#push-pull-github)).

## 12.2 Make a repo on GitHub

Go to [https://github.com](https://github.com/) and make sure you are logged in.

Click the green “New repository” button. Or, if you are on your own profile page, click on “Repositories”, then click the green “New” button.

How to fill this in:

* Repository name: myrepo (or whatever you wish, we’ll delete this soon anyway).
* Description: “testing my setup” (or whatever, but some text is good for the README).
* Public.
* YES Initialize this repository with a README.

For everything else, just accept the default.

Click the big green button “Create repository.”

Copy the HTTPS clone URL to your clipboard via the green “Clone or Download” button.

## 12.3 Clone the new GitHub repository to your computer via RStudio

In RStudio, start a new Project:

* *File > New Project > Version Control > Git*. In “Repository URL”, paste the URL of your new GitHub repository. It will be something like this https://github.com/jennybc/myrepo.git.
  + Do you NOT see an option to get the Project from Version Control? Restart RStudio and try again. Still no luck? Go to chapter [13](https://happygitwithr.com/rstudio-see-git.html#rstudio-see-git) for tips on how to help RStudio find Git.
* Accept the default project directory name, e.g. myrepo, which coincides with the GitHub repo name.
* Take charge of – or at least notice! – where the Project will be saved locally. A common rookie mistake is to have no idea where you are saving files or what your working directory is. Pay attention. Be intentional. Personally, I would do this in ~/tmp.
* I suggest you check “Open in new session”, as that’s what you’ll usually do in real life.
* Click “Create Project”.

You should find yourself in a new local RStudio Project that represents the new test repo we just created on GitHub. This should download the README.md file from GitHub. Look in RStudio’s file browser pane for the README.md file.

## 12.4 Make local changes, save, commit

From RStudio, modify the README.md file, e.g., by adding the line “This is a line from RStudio”. Save your changes.

Commit these changes to your local repo. How?

From RStudio:

* Click the “Git” tab in upper right pane.
* Check “Staged” box for README.md.
* If you’re not already in the Git pop-up, click “Commit”.
* Type a message in “Commit message”, such as “Commit from RStudio”.
* Click “Commit”.

## 12.5 Push your local changes online to GitHub

Click the green “Push” button to send your local changes to GitHub. If you are challenged for username and password, provide them (but see below). You should see some message along these lines.

[master dc671f0] blah

3 files changed, 22 insertions(+)

create mode 100644 .gitignore

create mode 100644 myrepo.Rproj

## 12.6 Confirm the local change propagated to the GitHub remote

Go back to the browser. I assume we’re still viewing your new GitHub repo.

Refresh.

You should see the new “This is a line from RStudio” in the README.

If you click on “commits”, you should see one with the message “Commit from RStudio”.

If you have made it this far, you are DONE with set up. But first …

## 12.7 Were you challenged for GitHub username and password?

If you somehow haven’t done so yet, now is the perfect time to make sure you don’t need to keep providing username and password on each push.

First, make another small change locally and push again, to make sure we’ve given your system every opportunity to cache your credentials and retrieve them from the cache. It might “just work”.

Are you still challenged? Pick one:

* Credential caching for HTTPS access, chapter [10](https://happygitwithr.com/credential-caching.html#credential-caching).
* Set up SSH keys, chapter [11](https://happygitwithr.com/ssh-keys.html#ssh-keys).

Now is the perfect time to do this, since you have a functioning test repo.

## 12.8 Clean up

**Local** When you’re ready to clean up, you can delete the local repo any way you like. It’s just a regular directory on your computer.

**GitHub** In the browser, go to your repo’s landing page on GitHub. Click on “Settings”.

Scroll down, click on “delete repository,” and do as it asks.

# Chapter 13 Detect Git from RStudio

If you want RStudio to help with your Git and GitHub work, it must be able to find the Git executable.

This usually “just works”, so this page is aimed at people who have reason to suspect they have a problem.

This is something you set up once-per-computer.

## 13.1 Do you have a problem?

Let’s check if RStudio can find the Git executable.

* *File > New Project…* Do you see an option to create from Version Control? If yes, good.
* Select *New Directory* > *Empty Project*. Do you see a checkbox “Create a git repository”? If yes, good, CHECK IT.
* Give this disposable test project a name and click *Create Project*. Do you see a “Git” tab in the upper right pane, the same one that has “Environment” and “History”? If yes, good.

If all looks good, you can delete this project. Looks like RStudio and Git are talking to each other.

Keep reading if things don’t go so well or you want to know more.

## 13.2 Find Git yourself

RStudio can only act as a GUI front-end for Git if Git has been successfully installed (chapter [6](https://happygitwithr.com/install-git.html#install-git)) **AND RStudio can find it**.

A basic test for successful installation of Git is to simply enter git in the shell (Appendix [A](https://happygitwithr.com/shell.html#shell)). If you get a complaint about Git not being found, it means installation was unsuccessful or that it is not being found, i.e. it is not on your PATH.

If you are not sure where the Git executable lives, try this in a shell:

* which git (Mac, Linux, Git Bash shell on Windows)
* where git (Windows command prompt, i.e. cmd.exe)

## 13.3 Tell RStudio where to find Git

If Git appears to be installed and findable, launch RStudio. Quit and re-launch RStudio if there’s **any doubt in your mind** about whether you opened RStudio before or after installing Git. Don’t make me stop this car and restart RStudio for you in office hours. DO IT.

From RStudio, go to *Tools > Global Options > Git/SVN* and make sure that the box *Git executable* points to your Git executable.

On macOS and Linux, the path usually looks something like this:

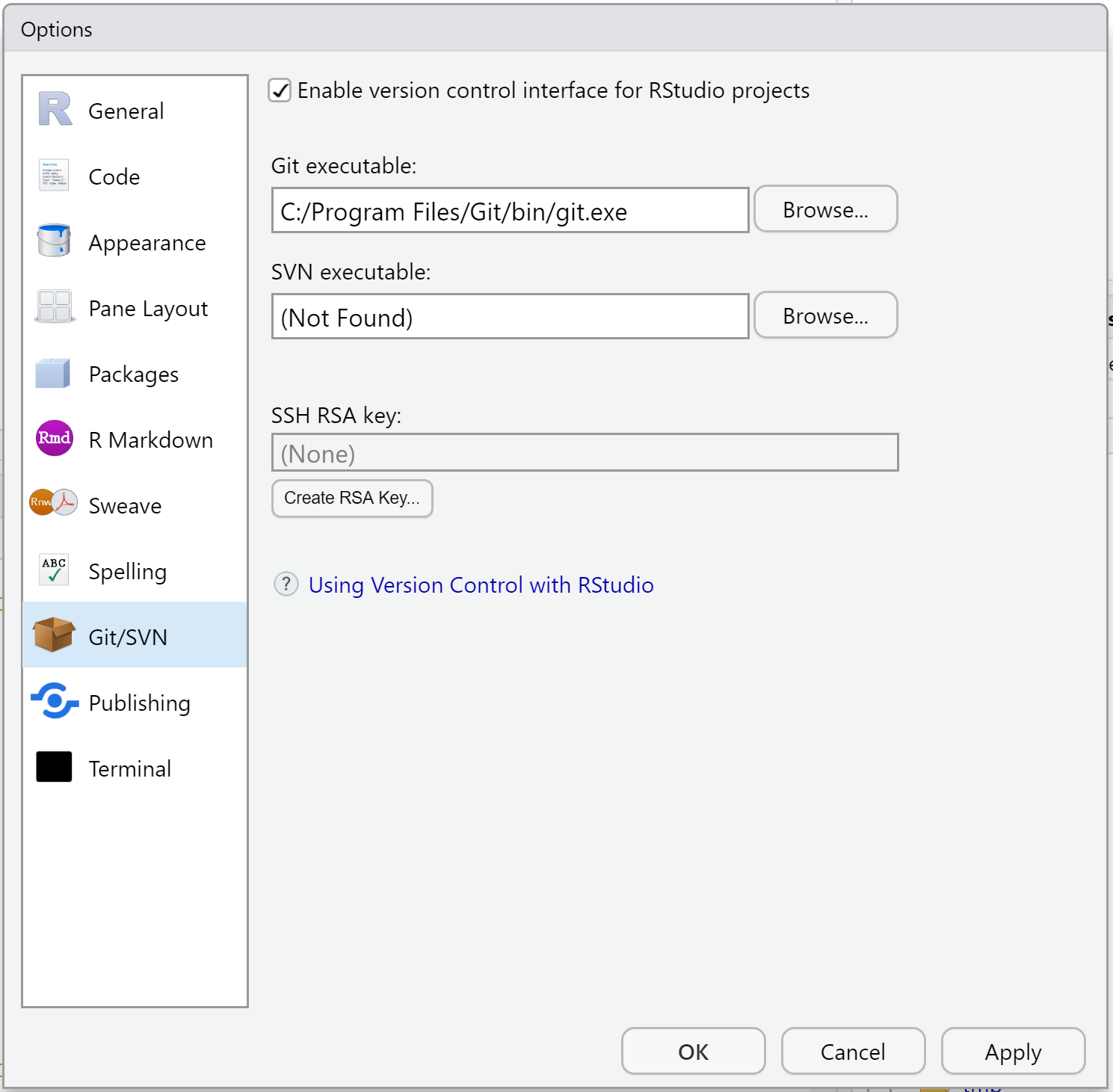
/usr/bin/git

If you need to set this on macOS, it can sometimes be hard to navigate to the necessary directory, once you’ve clicked “Browse” and are working with a Finder-type window. The keyboard shortcut “command + shift + g” will summon “Go To Folder”, where you will be able to type or paste any path you want.

On Windows, this path should look something like this:

C:/Program Files/Git/bin/git.exe

and here is a screenshot on Windows:



**WARNING**: On Windows, do **NOT** use C:/Program Files/Git/cmd/git.exe. bin in the path is GOOD YES! cmd in the path is BAD NO!

**WARNING**: On Windows, do **NOT** set this to git-bash.exe. Something that ends in git.exe is GOOD YES! git-bash.exe is BAD NO!

**Restart RStudio if you make any changes.** Don’t make me stop this car again and restart RStudio for you in office hours. DO IT.

Do the steps at the top of the page to see if RStudio and Git are communicating now.

No joy?

* I’ve seen this help: With your Project open, go to Tools > Project Options.... If available, click on “Git/SVN” and select “Git” in the Version control system dropdown menu. Answer “yes” to the “Confirm New Git Repository” pop up. Answer “yes” to the “Confirm Restart RStudio” pop up.
* If you installed Git via GitHub for Windows, it is possible the Git executable is really well hidden. Get help or use one of [our recommended methods of installing Git](https://happygitwithr.com/install-git.html#install-git).
* Your PATH is probably not set up correctly and/or you should re-install Git and control/notice where it’s going. Read more in [14](https://happygitwithr.com/troubleshooting.html#troubleshooting).
* Get our help.

# Chapter 14 RStudio, Git, GitHub Hell

Problems we have seen and possible solutions.

If you experience some new problem and, especially, find the corresponding solution, [we’d love to hear from you!](https://github.com/jennybc/happy-git-with-r/issues)

## 14.1 I think I have installed Git but damn if I can find it

When you install Git, try to control or record where it is being installed! Make a mental or physical note of these things.

You may be able to find Git after the fact with these commands in the shell (Appendix [A](https://happygitwithr.com/shell.html#shell)):

* which git (Mac, Linux, or anything running a bash shell)
* where git (Windows, when not in a bash shell)

It is not entirely crazy to just re-install Git, using a method that leaves it in a more conventional location, and to pay very close attention to where it’s being installed. Live and learn.

## 14.2 RStudio Git pane disappears on Mac OS

Sometimes the RStudio Git pane disappears on a system where it was previously working. This usually happens to people who installed Git by installing the Xcode command line tools. It is usually a sign that you need to re-agree to the Xcode license agreement. This is necessary after a Mac OS upgrade, re-installing Xcode, or even quiet Xcode upgrades that happen in normal system maintenance.

In the shell, you could execute git status and you might see a message along these lines:

Agreeing to the Xcode/iOS license requires admin privileges, please run “sudo xcodebuild -license” and then retry this command.

If you get such clear instructions, by all means do what it says, i.e. run sudo xcodebuild -license, to re-agree to the license.

In any case, you need to tickle the Xcode command line tools to prompt you for whatever it needs. Here are other commands that, depending on the situation, might trigger the necessary prompts:

xcode-select --install

or

git config --global --list

Then **restart RStudio**.

## 14.3 Dysfunctional PATH

I’m pretty sure that most cases of RStudio *not* automatically detecting the Git executable stem from problems with PATH. This is the set of directories where your computer will look for executables, such as Git (today) or make (later in this course). Certain methods of Git installation, especially on Windows and/or older OSes, have a higher tendency to put Git in an unconventional location or to fail to add the relevant directory to PATH.

How to see your PATH?

In the shell:

echo $PATH

Take a good hard look at this. See the point above about finding your Git executable or re-installing it while you are **wide awake**. Is the host directory in your PATH? No? **Fix that.**

Go [here](http://www.troubleshooters.com/linux/prepostpath.htm) for instructions on what to put in your .bash\_profile in order to add a directory to PATH.

## 14.4 Push/Pull buttons greyed out in RStudio

Are you sure your local repository is associated with a remote repository, e.g. a GitHub repo? In a shell with working directory set to the local Git repo, enter this command:

jenny@2015-mbp myrepo $ git remote -v

origin https://github.com/jennybc/myrepo (fetch)

origin https://github.com/jennybc/myrepo (push)

We want to see that fetch and push are set to remote URLs that point to the remote repo. Note also that the GitHub repo is a remote named origin, as far as your local repo is concerned. This is typical and, though I think github is a vastly superior name, origin is such a strong convention that I follow it.

If you discover you still need to set a remote, get the HTTPS or SSH URL, as appropriate, for your GitHub repo. This is easy to get onto your clipboard from the repo’s GitHub page. Do this in the shell:

git remote add origin https://github.com/jennybc/myrepo.git

Download all the files from the online GitHub repository and deal with any conflicts.

git pull origin master

Call git remote -v again. Once you can prove that your GitHub remote is set properly, you can move on to the next step.

Are you sure the current branch is *tracking* a branch on the remote? In that same shell, in your repo, do this:

jenny@2015-mbp myrepo $ git branch -vv

\* master b8e03e3 [origin/master] line added locally

The above shows successful confirmation that the local master branch is tracking origin/master, i.e. the master branch on GitHub. If you don’t see the [origin/master] bit, that is a problem. By the way, git branch -r is another handy way to examine your remote-tracking branches. (If you’re working with a branch other than master, adjust everything accordingly.)

When connecting a local repo to a new GitHub repo, a lot of people remember to add the GitHub remote, but forget to also cement this tracking relationship for any relevant branches.

If you discover your local master branch is not yet tracking master on GitHub, fix that like so:

git push --set-upstream origin master

This is equivalent to git push -u origin master but conveys more about what you are doing.

Call git branch -vv or git branch -r again to confirm that the master branch on GitHub is the upstream or tracking branch for the local master branch.

## 14.5 I have no idea if my local repo and my remote repo are connected.

See the above section on “Push/Pull buttons greyed out in RStudio.”

## 14.6 Push fail at the RStudio level

Do you get this error in RStudio?

error: unable to read askpass response from 'rpostback-askpass'

Open the shell: *Tools > Shell*.

git push -u origin master

## 14.7 Push rejected, i.e. fail at the Git/GitHub level

You might have changes on the remote AND on your local repo. Just because you don’t remember making any edits in the browser doesn’t mean you didn’t. Humor me.

Pull first. Resolve any conflicts. Then try your push again.

## 14.8 RStudio is not making certain files available for staging/committing

Do you have a space in your directory or file names? [A space in a file name is a space in your soul.](https://twitter.com/aaronquinlan/status/711593127551733761) Get rid of it.

Is your Git repo / RStudio Project inside a folder that … eventually rolls up to Google Drive, DropBox, Microsoft OneDrive, or a network drive? If yes, I recommend you move the repo / Project into a plain old directory that lives directly on your computer and that is not managed by, e.g., Google Drive.

If you cannot deal with the two root causes identified above, then it is possible that a more powerful Git client (chapter [8](https://happygitwithr.com/git-client.html#git-client)) will be able to cope with these situations. But I make no promises. You should also try Git operations from the command line.

## 14.9 I hear you have some Git repo inside your Git repo

Do not create a Git repository inside another Git repository. Just don’t.

If you have a genuine need for this, which is really rare, the proper way to do it is via [submodules](http://git-scm.com/book/en/v2/Git-Tools-Submodules).

In STAT 545, we certainly do not need to do this and when we’ve seen it, it’s been a mistake. This has resulted in the unexpected and complete loss of the inner Git repository. To be sure, there was more going on here (cough, GitHub Desktop client), but non-standard usage of Git repos makes it much easier to make costly mistakes.

# Get started with GitHub

Now that we’ve verified your Git/GitHub/RStudio setup, we can demo the workflows you’ll use to get your work onto GitHub:

* [New project, GitHub first](https://happygitwithr.com/new-github-first.html#new-github-first) is the easiest way to get a working project.
* [Existing project, GitHub first](https://happygitwithr.com/existing-github-first.html#existing-github-first) is a deeply pragmatic way to get pre-existing work onto GitHub.
* [Existing project, GitHub last](https://happygitwithr.com/existing-github-last.html#existing-github-last) is the more proper way to connect existing local work to a remote on GitHub, especially if there’s already a Git history.

This part concludes with two R-specific workflows that show off how well [R Markdown (the format)](https://rmarkdown.rstudio.com/) and [rmarkdown (the package)](https://cran.r-project.org/package=rmarkdown) work with GitHub:

* [Test drive R Markdown](https://happygitwithr.com/rmd-test-drive.html#rmd-test-drive)
* [Render an R script](https://happygitwithr.com/r-test-drive.html#r-test-drive)

# Chapter 15 New project, GitHub first

We create a new Project, with the preferred “GitHub first, then RStudio” sequence. Why do we prefer this? Because this method of copying the Project from GitHub to your computer also sets up the local Git repository for immediate pulling and pushing. Under the hood, we are doing git clone.

You’ve actually done this before during set up (chapter [12](https://happygitwithr.com/rstudio-git-github.html#rstudio-git-github)). We’re doing it again, *with feeling*.

The workflow is pretty similar for other repository managers like GitLab or Bitbucket. We will specify below when you may need to do something differently.

## 15.1 Make a repo on GitHub

**Do this once per new project.**

Go to [https://github.com](https://github.com/) and make sure you are logged in.

Click green “New repository” button. Or, if you are on your own profile page, click on “Repositories”, then click the green “New” button.

* Repository name: myrepo (or whatever you wish)
* Public
* YES Initialize this repository with a README

Click the big green button “Create repository.”

Copy the HTTPS clone URL to your clipboard via the green “Clone or Download” button. Or copy the SSH URL if you chose to set up SSH keys.

## 15.2 New RStudio Project via git clone

In RStudio, start a new Project:

* *File > New Project > Version Control > Git*. In the “repository URL” paste the URL of your new GitHub repository. It will be something like this https://github.com/jennybc/myrepo.git.
* Be intentional about where you create this Project.
* Suggest you “Open in new session”.
* Click “Create Project” to create a new directory, which will be all of these things:
  + a directory or “folder” on your computer
  + a Git repository, linked to a remote GitHub repository
  + an RStudio Project
* **In the absence of other constraints, I suggest that all of your R projects have exactly this set-up.**

This should download the README.md file that we created on GitHub in the previous step. Look in RStudio’s file browser pane for the README.md file.

There’s a big advantage to the “GitHub first, then RStudio” workflow: the remote GitHub repo is added as a remote for your local repo and your local master branch is now tracking master on GitHub. This is a technical but important point about Git. The practical implication is that you are now set up to push and pull. No need to fanny around setting up Git remotes and tracking branches on the command line.

## 15.2.1 Optional: peek under the hood

Completely optional activity: use command line Git to see what we’re talking about above, i.e. the remote and tracking branch setup.

git remote -vor git remote --verbose shows the remotes you have setup. Here’s how that looks for someone using SSH with GitHub and calling it origin (a convention I hate but am resigned to):

$ git remote -v

origin git@github.com:jennybc/myrepo.git (fetch)

origin git@github.com:jennybc/myrepo.git (push)

git branch -vv prints info about the current branch. In particular, we can see that local master is tracking the master branch on origin, a.k.a. origin/master.

$ git branch -vv

\* master 7c98308 [origin/master] Initial commit

git clone, which RStudio did for us, sets all of this up automatically. This is why “GitHub first, then RStudio” is the preferred way to start projects early in your Git/GitHub life.

## 15.3 Make local changes, save, commit

**Do this every time you finish a valuable chunk of work, probably many times a day.**

From RStudio, modify the README.md file, e.g., by adding the line “This is a line from RStudio”. Save your changes.

Commit these changes to your local repo. How?

* Click the “Git” tab in upper right pane
* Check “Staged” box for any files whose existence or modifications you want to commit.
  + To see more detail on what’s changed in file since the last commit, click on “Diff” for a Git pop-up
* If you’re not already in the Git pop-up, click “Commit”
* Type a message in “Commit message”, such as “Commit from RStudio”.
* Click “Commit”

## 15.4 Push your local changes to GitHub

**Do this a few times a day, but possibly less often than you commit.**

You have new work in your local Git repository, but the changes are not online yet.

This will seem counterintuitive, but first let’s stop and pull from GitHub.

Why? Establish this habit for the future! If you make changes to the repo in the browser or from another machine or (one day) a collaborator has pushed, you will be happier if you pull those changes in before you attempt to push.

Click the blue “Pull” button in the “Git” tab in RStudio. I doubt anything will happen, i.e. you’ll get the message “Already up-to-date.” This is just to establish a habit.

Click the green “Push” button to send your local changes to GitHub. You should see some message along these lines.

[master dc671f0] blah

3 files changed, 22 insertions(+)

create mode 100644 .gitignore

create mode 100644 myrepo.Rproj

## 15.5 Confirm the local change propagated to the GitHub remote

Go back to the browser. I assume we’re still viewing your new GitHub repo.

Refresh.

You should see the new “This is a line from RStudio” in the README.

If you click on “commits,” you should see one with the message “Commit from RStudio”.

## 15.6 Make a change on GitHub

Click on README.md in the file listing on GitHub.

In the upper right corner, click on the pencil for “Edit this file”.

Add a line to this file, such as “Line added from GitHub.”

Edit the commit message in “Commit changes” or accept the default.

Click the big green button “Commit changes.”

## 15.7 Pull from GitHub

Back in RStudio locally …

Inspect your README.md. It should NOT have the line “Line added from GitHub”. It should be as you left it. Verify that.

Click the blue Pull button.

Look at README.md again. You should now see the new line there.

## 15.8 The end

Now just … repeat. Do work somewhere. Commit it. Push it or pull it\* depending on where you did it, but get local and remote “synced up”. Repeat.

\* Note that in general (and especially in future when collaborating with other developers) you will usually need to pull changes from the remote (GitHub) before pushing the local changes you have made. For this reason, it’s a good idea to try and get into the habit of pulling before you attempt to push.

# Chapter 16 Existing project, GitHub first

A novice-friendly workflow for bringing an existing R project into the RStudio and Git/GitHub universe.

We do this in a slightly awkward way, in order to avoid using Git at the command line. You won’t want to work this way forever, but it’s perfectly fine as you’re getting started!

We assume you’ve got your existing R project isolated in a directory on your computer. If that’s not already true, make it so. Create a directory and marshal all the existing data and R scripts there. It doesn’t really matter where you do this, but note where the project currently lives.

## 16.1 Make a repo on GitHub

Go to [https://github.com](https://github.com/) and make sure you are logged in.

Click the green “New repository” button. Or, if you are on your own profile page, click on “Repositories”, then click the green “New” button.

Pick a repository name that actually reminds you what the project is about! But try to be concise.

Public  
YES Initialize this repository with a README

Click the big green button “Create repository.”

Copy the HTTPS clone URL to your clipboard via the green “Clone or Download” button. Or copy the SSH URL if you chose to set up SSH keys.

## 16.2 New RStudio Project via git clone

In RStudio, start a new Project:

* *File > New Project > Version Control > Git*. In the “repository URL” paste the URL of your new GitHub repository. It will be something like this https://github.com/jennybc/myrepo.git.
* Be intentional about where you create this Project.
* Suggest you “Open in new session”.
* Click “Create Project” to create a new directory, which will be all of these things:
  + a directory or “folder” on your computer
  + a Git repository, linked to a remote GitHub repository
  + an RStudio Project

This should download the README.md file that we created on GitHub in the previous step. Look in RStudio’s file browser pane for the README.md file.

## 16.3 Bring your existing project over

Using your favorite method of moving or copying files, copy the files that constitute your existing project into the directory for this new project.

In RStudio, consult the Git pane and the file browser.

* Are you seeing all the files? They should be here if your move/copy was successful.
* Are they showing up in the Git pane with questions marks? They should be appearing as new untracked files.

## 16.4 Stage and commit

Commit your files to this repo. How?

* Click the “Git” tab in upper right pane
* Check “Staged” box for all files you want to commit.
  + Default: stage it.
  + When to reconsider: this will all go to GitHub. So do consider if that is appropriate for each file. **You can absolutely keep a file locally, without committing it to the Git repo and sending to GitHub**. Just let it sit there in your Git pane, without being staged. No harm will be done. If this is a long-term situation, list the file in .gitignore.
* If you’re not already in the Git pop-up, click “Commit”
* Type a message in “Commit message”, such as “init”.
* Click “Commit”

## 16.5 Push your local changes to GitHub

Click the green “Push” button to send your local changes to GitHub. You should see some message along these lines.

[master dc671f0] blah

3 files changed, 22 insertions(+)

create mode 100644 .gitignore

create mode 100644 myrepo.Rproj

## 16.6 Confirm the local change propagated to the GitHub remote

Go back to the browser. I assume we’re still viewing your new GitHub repo.

Refresh.

You should see all the project files you committed there.

If you click on “commits,” you should see one with the message “init”.

## 16.7 The end

Now just … repeat. Do work somewhere. Commit it. Push it or pull it\* depending on where you did it, but get local and remote “synced up”. Repeat.

\* Note that in general (and especially in future when collaborating with other developers) you will usually need to pull changes from the remote (GitHub) before pushing the local changes you have made. For this reason, it’s a good idea to try and get into the habit of pulling before you attempt to push.

# Chapter 17 Existing project, GitHub last

An explicit workflow for connecting an existing local R project to GitHub, when for some reason you cannot or don’t want to do a “GitHub first” workflow (see chapters [15](https://happygitwithr.com/new-github-first.html#new-github-first) and [16](https://happygitwithr.com/existing-github-first.html#existing-github-first)).

When does this come up? Example: it’s an existing project that is already a Git repo with a history you care about. Then you have to do this properly.

This is less desirable for a novice because there are more opportunities to get confused and make a mistake. RStudio and the [usethis](https://cran.r-project.org/package=usethis) package are gradually making this easier, so you can avoid working with Git in the shell (Appendix [A](https://happygitwithr.com/shell.html#shell)). But command line Git remains a viable option. All are covered below.

## 17.1 Make or verify an RStudio Project

We assume you’ve got your existing R project isolated in a directory on your computer.

If it’s not already an RStudio Project, make it so:

* If you use the [usethis](https://cran.r-project.org/package=usethis) package, set the existing directory as the current project with usethis::create\_project().
* Within RStudio you can do: *File > New Project > Existing Directory* and, if you wish, “Open in new session”.

If your project is already an RStudio Project, launch it.

## 17.2 Make or verify a Git repo

You should be in RStudio now, in your project.

Is it already a Git repository? The presence of the Git pane should tip you off. If yes, you’re done.

If not, you have several options:

* If you have the [usethis](https://cran.r-project.org/package=usethis) package, call usethis::use\_git().
* In RStudio, go to *Tools > Project Options … > Git/SVN*. Under “Version control system”, select “Git”. Confirm New Git Repository? Yes!
* In the shell, in Project’s directory, do git init.

If you used RStudio or usethis, the Project should re-launch in RStudio. Do that yourself if you did git init. RStudio should now have a Git pane.

## 17.3 Stage and commit

If your local project was already a Git repo and was up-to-date, move on. Otherwise, you probably need to stage and commit.

* Click the “Git” tab in upper right pane
* Check “Staged” box for all files you want to commit.
  + Default: stage everything
  + When to do otherwise: this will all go to GitHub. So consider if that is appropriate for each file. **You can absolutely keep a file locally, without committing it to the Git repo and sending to GitHub**. Just let it sit there in your Git pane, without being staged. No harm will be done. If this is a long-term situation, list the file in .gitignore.
* If you’re not already in the Git pop-up, click “Commit”
* Type a message in “Commit message”.
* Click “Commit”

## 17.4 Make and connect a GitHub repo, option 1

If you use the [usethis](https://cran.r-project.org/package=usethis) package AND you have configured a GitHub Personal Access Token (PAT) (see [the appendix](https://happygitwithr.com/github-pat.html#github-pat) for how to set this up.), do this in R:

usethis::use\_github()

This will create a new repo on GitHub, add it as a remote, set up a tracking branch, and open it in your browser. Read the use\_github() help to learn about its arguments and advice on configuring a PAT. This is extremely handy for a variety of workflows that call the GitHub API. Consider setting this up if you use usethis, devtools, or gh regularly.

## 17.5 Make and connect a GitHub repo, option 2

### 17.5.1 Make a new repo on GitHub

Go to [https://github.com](https://github.com/) and make sure you are logged in.

Click the green “New repository” button. Or, if you are on your own profile page, click on “Repositories”, then click the green “New” button.

Pick a repository name – it should probably match the name of your local Project and directory. Why confuse yourself?

Public or private, as appropriate  
DO NOT initialize this repository with a README.

Click the big green button “Create repository.”

Copy the HTTPS clone URL to your clipboard via the green “Clone or Download” button. Or copy the SSH URL if you chose to set up SSH keys.

### 17.5.2 Connect local repo to GitHub repo

You can do this in the shell or, with recent versions of RStudio, via RStudio.

### 17.5.3 In RStudio

Click on the “two purple boxes and a white square” in the Git pane. Click “Add remote”. Paste the URL here and pick a remote name, almost certainly origin. Now “Add”.

We should be back in the “New Branch” dialog (if not, click on the “two purple boxes and a white square” in the Git pane again). I assume you’re on the master branch want it to track master on GitHub. Enter master as the branch name and make sure “Sync branch with remote” is checked. Click “Create” (yes, even though the branch already exists). In the next dialog, choose “overwrite”.

### 17.5.4 In the shell

In a shell, do this, substituting your URL:

git remote add origin https://github.com/jennybc/myrepo.git

Push and cement the tracking relationship between your local master branch and master on GitHub:

git push --set-upstream origin master

## 17.6 Confirm the local files propagated to the GitHub remote

Go back to the browser. I assume we’re still viewing your new GitHub repo.

Refresh.

You should see all the project files you committed there.

If this project already had a Git history, it should be reflected on GitHub.

## 17.7 The end

Now just … repeat. Do work somewhere. Commit it. Push it or pull it depending on where you did it, but get local and remote “synced up”. Repeat.

Note that in general (and especially in future when collaborating with other developers) you will usually need to pull changes from the remote (GitHub) before pushing the local changes you have made. For this reason, it’s a good idea to try and get into the habit of pulling before you attempt to push.

# Chapter 18 Test drive R Markdown

We will author an R Markdown document and render it to HTML. We discuss how to keep the intermediate Markdown file, the figures, and what to commit to Git and push to GitHub. If GitHub is the primary venue, we render directly to GitHub-flavored markdown and never create HTML.

Here is the official R Markdown documentation: [http://rmarkdown.rstudio.com](http://rmarkdown.rstudio.com/)

## 18.1 Hello World

We’ll practice with RStudio’s boilerplate R Markdown document.

Launch RStudio in a Project that is a Git repo that is connected to a GitHub repo.

We are modelling “walk before you run” here. It is best to increase complexity in small increments. We test our system’s ability to render the [“hello world”](http://en.wikipedia.org/wiki/%22Hello,_world!%22_program) of R Markdown documents before we muddy the waters with our own, probably buggy, documents.

Do this: *File > New File > R Markdown …*

* Give it an informative title. This will appear in the document but does not necessarily have anything to do with the file’s name. But the title and filename should be related! Why confuse yourself? The title is for human eyeballs, so it can contain spaces and punctuation. The filename is for humans and computers, so it should have similar words in it but no spaces and no punctuation.
* Accept the default Author or edit if you wish.
* Accept the default output format of HTML.
* Click OK.

Save this document to a reasonable filename and location. The filename should end in .Rmd or .rmd. Save in the top-level of this RStudio project and Git repository, that is also current working directory. Trust me on this and do this for a while.

You might want to commit here. So you can see what’s about to change …

Click on “Knit HTML” or do *File > Knit Document*. RStudio should display a preview of the resulting HTML. Also look at the file browser. You should see the R Markdown document, i.e. foo.Rmd AND the resulting HTML foo.html.

Congratulations, you’ve just made your first reproducible report with R Markdown.

You might want to commit here.

## 18.2 Push to GitHub

Push the current state to GitHub.

Go visit it in the browser.

Do you see the new files? An R Markdown document and the associated HTML? Visit both in the browser. Verify this:

* Rmd is quite readable. But the output is obviously not there.
* HTML is ugly.

## 18.3 Output format

Do you really want HTML? Do you only want HTML? If so, you can skip this step!

The magical process that turns your R Markdown to HTML is like so: foo.Rmd --> foo.md --> foo.html. Note the intermediate markdown, foo.md. By default RStudio discards this, but you might want to hold on to that markdown.

Why? GitHub gives very special treatment to markdown files. They are rendered in an almost HTML-like way. This is great because it preserves all the charms of plain text but gives you a pseudo-webpage for free when you visit the file in the browser. In contrast, HTML is rendered as plain text on GitHub and you’ll have to take special measures to see it the way you want.

In many cases, you *only want the markdown*. In that case, we switch the output format to github\_document. This means render will be foo.Rmd --> foo.md, where foo.md is GitHub-flavored markdown. If you still want the HTML *but also the intermediate markdown*, there’s a way to request that too.

**Output format** is one of the many things we can control in the YAML frontmatter – the text at the top of your file between leading and trailing lines of ---.

You can make some changes via the RStudio IDE: click on the “gear” in the top bar of the source editor, near the “Knit HTML” button. Select “Output options” and go to the Advanced tab and check “Keep markdown source file.” Your YAML should now look more like this:

---

title: "Something fascinating"

author: "Jenny Bryan"

date: "2020-01-15"

output:

html\_document:

keep\_md: true

---

You should have gained the line keep\_md: true. You can also simply edit the file yourself to achieve this.

In fact this hand-edit is necessary if you want to keep only markdown and get GitHub-flavored markdown. In that case, make your YAML look like this:

---

title: "Something fascinating"

author: "Jenny Bryan"

date: "2020-01-15"

output: github\_document

---

Save!

You might want to commit here.

Render via “Knit HTML” button.

Now revisit the file browser. In addition to foo.Rmd, you should now see foo.md. If there are R chunks that make figures, the usage of markdown output formats will also cause those figure files to be left behind in a sensibly named sub-directory, foo\_files.

If you commit and push foo.md and everything inside foo\_files, then anyone with permission to view your GitHub repo can see a decent-looking version of your report.

If your output format is html\_document, you should still see foo.html. If your output format is github\_document and you see foo.html, that’s leftover from earlier experiments. Delete that. It will only confuse you later.

You might want to commit here.

## 18.4 Push to GitHub

Push the current state to GitHub.

Go visit it in the browser.

Do you see the modifications and new file(s)? Your Rmd should be modified (the YAML frontmatter changed). And you should have gained at least, the associated markdown file.

* Visit the markdown file and compare to our previous HTML.
* Do you see how the markdown is much more useful on GitHub? Internalize that.

## 18.5 Put your stamp on it

Select everything but the YAML frontmatter and … delete it!

Write a single English sentence.

Insert an empty R chunk, via the “Chunk” menu in upper right of source editor or with corresponding keyboard shortcut.

```{r}

## insert your brilliant WORKING code here

```

Insert 1 to 3 lines of functioning code that begin the task at hand. “Walk through” and run those lines using the “Run” button or the corresponding keyboard shortcut. You MUST make sure your code actually works!

Satisfied? Save!

You might want to commit here.

Now render the whole document via “Knit HTML.” Voilà!

You might want to commit here.

## 18.6 Develop your report

In this incremental manner, develop your report. Add code to this chunk. Refine it. Add new chunks. Go crazy! But keep running the code “manually” to make sure it works.

If it doesn’t work with you babysitting it, I can guarantee you it will fail, in a more spectacular and cryptic way, when run at arms-length via “Knit HTML” or rmarkdown::render().

Clean out your workspace and restart R and re-run everything periodically, if things get weird. There are lots of chunk menu items and keyboard shortcuts to accelerate this workflow. Render the whole document often to catch errors when they’re easy to pinpoint and fix. Save often and commit every time you reach a point that you’d like as a “fall back” position.

You’ll develop your own mojo soon, but this should give you your first successful R Markdown experience.

## 18.7 Publish your report

If you’ve been making HTML, you can put that up on the web somewhere, email to your collaborator, whatever.

No matter what, technically you can publish this report merely by pushing a rendered version to GitHub. However, certain practices make this effort at publishing more satisfying for your audience.

Here are two behaviors I find very frustrating:

* “Here is my code. Behold.” This is when someone only pushes their source, i.e. R Markdown or R code AND they want other people to look at their “product”. The implicit assumption is that target audience will download code and run it. Sometimes the potential payoff simply does not justify this effort.
* “Here is my HTML. Behold.” This is when someone doesn’t bother to edit the default output format and accepts HTML only. What am I supposed to do with HTML on GitHub?

Creating, committing, and pushing markdown is a very functional, lighweight publishing strategy. Use output: github\_document or keep\_md: true if output is html\_document. In both cases, it is critical to also commit and push everything inside foo\_files. Now people can visit and consume your work like any other webpage.

This is (sort of) another example of keeping things machine- and human-readable, which is bliss. By making foo.Rmd available, others can see and run your **actual code**. By sharing foo.md and/or foo.html, others can casually browse your end product and decide if they even want to bother.

## 18.8 HTML on GitHub

HTML files, such as foo.html, are not immediately useful on GitHub (though your local versions are easily viewable). Visit one and you’ll see the raw HTML. Yuck. But there are ways to get a preview: such as [http://htmlpreview.github.io](http://htmlpreview.github.io/). Expect much pain with HTML files inside private repos (hence the recommendations above to emphasize markdown). When it becomes vital for the whole world to see proper HTML in its full glory, it’s time to use a more sophisticated web publishing strategy.

I have more [general ideas](https://happygitwithr.com/workflows-browsability.html#workflows-browsability) about how to make a GitHub repo function as a website.

## 18.9 Troubleshooting

**Make sure RStudio and the** rmarkdown **package (and its dependencies) are up-to-date.** In case of catastrophic failure to render the boilerplate R Markdown document, consider that your software may be too old. Details on the system used to render this document and how to check your setup:

* rmarkdown version 2.0. Use packageVersion("rmarkdown") to check yours.
* R version 3.6.2 (2017-01-27). Use R.version.string to check yours.
* RStudio IDE 1.2.1555. Use *RStudio > About RStudio* or RStudio.Version()$version to check yours.

**Get rid of your** .Rprofile, at least temporarily. I have found that a “mature” .Rprofile that has accumulated haphazardly over the years can cause trouble. Specifically, if you’ve got anything in there relating to knitr, markdown, rmarkdown and RStudio itself, it may be preventing the installation or usage of the most recent goodies. Comment the whole file out or rename it something else and relaunch or even re-install RStudio.

**“I have ignored your advice and dumped a bunch of code in at once. Now my Rmd does not render.”** If you can’t figure out what’s wrong by reading the error messages, pick one:

* Back out of these changes, get back to a functional state (possibly with no code), and restore them gradually. Run your code interactively to make sure it works. Render the entire document frequently. Commit after each successful addition! When you re-introduce the broken code, now it will be part of a small change and the root problem will be much easier to pinpoint and fix.
* Tell knitr to soldier on, even in the presence of errors. Some problems are easier to diagnose if you can execute specific R statements during rendering and leave more evidence behind for forensic examination.
  + Insert this chunk near the top of your .Rmd document:
  + ```{r setup, include = FALSE, cache = FALSE}
  + knitr::opts\_chunk$set(error = TRUE)

```

* + If it’s undesirable to globally accept errors, you can still do this for a specific chunk like so:
  + ```{r wing-and-a-prayer, error = TRUE}
  + ## your sketchy code goes here ;)

```

* Adapt the [“git bisect” strategy](http://webchick.net/node/99):
  + Put knitr::knit\_exit() somewhere early in your .Rmd document, either in inline R code or in a chunk. Keep moving it earlier until things work. Now move it down in the document. Eventually you’ll be able to narrow down the location of your broken code well enough to find the line(s) and fix it.

**Check your working directory.** It’s going to break your heart as you learn how often your mistakes are really mundane and basic. Ask me how I know. When things go wrong consider:

* What is the working directory?
* Is that file I want to read/write actually where I think it is?

Drop these commands into R chunks to check the above:

* getwd() will display working directory at **run time**. If you monkeyed around with working directory with, e.g., the mouse, maybe it’s set to one place for your interactive development and another when “Knit HTML” takes over?
* list.files() will list the files in working directory. Is the file you want even there?

**Don’t try to change working directory within an R Markdown document**. Just don’t. See [knitr FAQ #5](https://yihui.name/knitr/faq/). That is all.

**Don’t be in a hurry to create a complicated sub-directory structure.** RStudio/knitr/rmarkdown (which bring you the “Knit HTML” button) are rather opinionated about the working directory being set to the .Rmd file’s location and about all files living together in one big happy directory. This can all be worked around. For example, I [recommend the here package](https://github.com/jennybc/here_here#readme) for building file paths, once you require sub-directories. But don’t do this until you really need it.

# Chapter 19 Render an R script

An underappreciated fact is that much of what you can do with R Markdown, you can also do with an R script.

If you’re in analysis mode and want a report as a side effect, write an R script. If you’re writing a report with a lot of R code in it, write Rmd. In either case, render to markdown and/or HTML to communicate with other human beings.

* In R markdown, prose is top-level and code is tucked into chunks.
* In R scripts, code is top-level and prose is tucked into comments. You will use #' to request that certain comments appear as top-level prose in the rendered output.

You will continue to specify things like the output format via YAML at the top of the file. This will need to be commented with #'.

## 19.1 Morph R Markdown into a renderable R script

Get yourself a working R Markdown file, such as the one you made in your [Rmd test drive](https://happygitwithr.com/rmd-test-drive.html#rmd-test-drive). Or use the boilerplate Rmd document RStudio makes with *File > New File > R Markdown …*.

Save the file as foo.R, as opposed to foo.Rmd. Yes, for a brief moment, you will have R Markdown saved as an R script, but that won’t be true for long.

Transform the Rmd to R:

* Anything that’s not R code? Like the YAML and the English prose? Protect it with roxygen-style comments: start each line with #'.
* Anything that’s R code? Let it exist “as is” as top-level code. That means you’ll need to change the syntax of R chunk headers like so:

Before: ```{r setup, include = FALSE}  
After: #+ r setup, include = FALSE

Replace the leading backticks and opening curly brace with #+.  
Delete the trailing curly brace.  
Delete the 3 backticks that end each chunk.

Render the R script through one of these methods:

* Click on the “notebook” icon in RStudio to “Compile Report”.
* In RStudio, do *File > Knit Document*.
* In R, do rmarkdown::render("foo.R").

You’ll get a markdown and/or HTML report, just as with R Markdown.

If you’re having trouble making all the necessary changes and you’re frustrated, see below for an example you can copy and paste.

All the workflow tips from the [Rmd test drive](https://happygitwithr.com/rmd-test-drive.html#rmd-test-drive) apply here: when you script an analysis, render it to markdown, commit the .R, the .md, any associated figures, and push to GitHub. Collaborators can see your code but also browse the result without running it. This makes the current state of your analysis accessible to someone who does not even run R or who wants to take a quick look at things from a cell phone or while on vacation.

## 19.2 Write a render-ready R script

Instead of morphing an R Markdown file, let’s create a render-ready R script directly.

Create a new R script and copy/paste this code into it.

#' Here's some prose in a very special comment. Let's summarize the built-in

#' dataset `VADeaths`.

## here is a regular code comment, that will remain as such

summary(VADeaths)

#' Here's some more prose. I can use usual markdown syntax to make things

#' \*\*bold\*\* or \*italics\*. Let's use an example from the `dotchart()` help to

#' make a Cleveland dot plot from the `VADeaths` data. I even bother to name

#' this chunk, so the resulting PNG has a decent name.

#+ dotchart

dotchart(VADeaths, main = "Death Rates in Virginia - 1940")

Render the R script through one of these methods:

* Click on the “notebook” icon in RStudio to “Compile Report”.
* In RStudio, do *File > Knit Document*.
* In R, do rmarkdown::render("YOURSCRIPT.R").

Revel in your attractive looking report with almost zero effort! Seriously, all you had to do was think about when to use special comments #' in order to promote that to nicely rendered text.

Drawing on the workflow tips in [Rmd test drive](https://happygitwithr.com/rmd-test-drive.html#rmd-test-drive), let’s add some YAML frontmatter, properly commented with #', and request the github\_document. Here’s the whole script again:

#' ---

#' title: "R scripts can be rendered!"

#' author: "Jenny Bryan"

#' date: "April 1, 2014"

#' output: github\_document

#' ---

#'

#' Here's some prose in a very special comment. Let's summarize the built-in

#' dataset `VADeaths`.

## here is a regular code comment, that will remain as such

summary(VADeaths)

#' Here's some more prose. I can use usual markdown syntax to make things

#' \*\*bold\*\* or \*italics\*. Let's use an example from the `dotchart()` help to

#' make a Cleveland dot plot from the `VADeaths` data. I even bother to name

#' this chunk, so the resulting PNG has a decent name.

#+ dotchart

dotchart(VADeaths, main = "Death Rates in Virginia - 1940")

Behind the scenes here we have used rmarkdown::render() to render this script and you can go [visit it on GitHub](https://github.com/jennybc/happy-git-with-r/blob/master/render-r-script-demo.md).

# Some Git basics

We’ve told you shockingly little about Git so far! This is by design.

We find that actual usage, in the course of your work, is the most effective way to build up a useful mental model for Git. In live workshops, we strive to introduce the most important basic ideas in the context of our guided activities. Self-learners can achieve the same by working through the “batteries included” guides earlier in the previous sections.

However, building on this early success, now is the perfect time to explicitly define some Git vocabulary. We also want to help you link Git concepts to data science tasks and projects.

This part collects anything we’ve written about core Git concepts. It is a work in progress and is conceived as a complement to the many excellent [external resources for Git](https://happygitwithr.com/resources.html#resources), which we have no desire to re-invent.

# Chapter 20 Repo, commit, diff, tag

## 20.1 Repos or repositories

Git is a version control system whose original purpose was to help groups of developers work collaboratively on big software projects. Git manages the evolution of a set of files – called a **repository** or **repo** – in a highly structured way. Historically, these files would have consisted of source code and the instructions for how to build an application from its source.

Git has been re-purposed by the data science community (Ram [2013](https://happygitwithr.com/references.html#ref-Ram2013); Bartlett [2016](https://happygitwithr.com/references.html#ref-git-for-humans); Perez-Riverol et al. [2016](https://happygitwithr.com/references.html#ref-ten-simple-rules-git)). We use it to manage the motley collection of files that make up typical data analytical projects, which consist of data, figures, reports, and, yes, some source code.

For new or existing projects, we recommand that you:

* Dedicate a local directory or folder to it.
* Make it an RStudio Project. *Optional but recommended; obviously only applies to projects involving R and users of RStudio.*
* Make it a Git repository.

This setup happens once per project and can happen at project inception or at any later point. Chances are your existing projects each already live in a dedicated directory. Making such a directory an RStudio Project and Git repository boils down to allowing those applications to leave notes for themselves in hidden files or directories. The project is still a regular directory on your computer, that you can locate, name, move, and generally interact with as you wish. You don’t have to handle it with special gloves!

The daily workflow is probably not dramatically different from what you do currently. You work in the usual way, writing R scripts or authoring reports in LaTeX or R Markdown. But instead of only *saving* individual files, periodically you make a **commit**, which takes a snapshot of all the files in the entire project. If you have ever versioned a file [by adding your initials or the date](http://www.phdcomics.com/comics/archive.php?comicid=1531), you have effectively made a commit, albeit only for a single file. It is a version that is significant to you and that you might want to inspect or revert to later. Periodically, you push commits to GitHub. This is like sharing a document with colleagues on DropBox or sending it out as an email attachment. By pushing to GitHub, you make your work and all your accumulated progress accessible to others.

This is a moderate change to your normal, daily workflow. It feels weird at first, but quickly becomes second nature. In [STAT 545](http://stat545.com/) students are required to submit all coursework via GitHub, starting in week one. Most have never seen Git before and do not identify as programmers. It is a major topic in class and office hours for the first two weeks. Then we practically never discuss it again.

## 20.2 Commits, diffs, and tags

We now connect the fundamental concepts of Git to the data science workflow:

* repository
* commit
* diff

Recall that a repository or repo is just a directory of files that Git manages holistically. A commit functions like a snapshot of all the files in the repo, at a specific moment. Under the hood, that is not exactly how Git implements things. Although mental models don’t have to be accurate in order to be useful, in this case it helps to align the two.

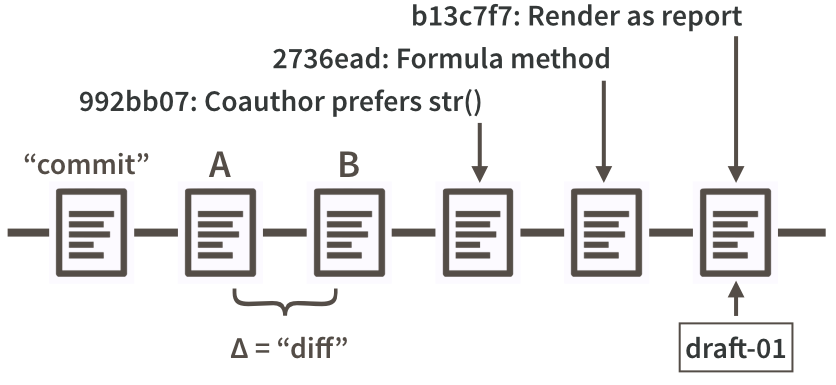
Figure 20.1: Partial commit history for our iris example, highlighting diffs, commit messages, SHAs, and tags.

Figure [20.1](https://happygitwithr.com/git-basics.html#fig:commit-diff-sha-tag) is a look at a fictional analysis of the iris data, focusing on the evolution of a script, iris.R. Consider version A of this file and a modified version, version B. Assume that version A was part of one Git commit and version B was part of the next commit. The set of differences between A and B is called a “diff” and Git users contemplate diffs a lot. Diff inspection is how you re-explain to yourself how version A differs from version B. Diff inspection is not limited to adjacent commits. You can inspect the diffs between any two commits.

In fact, Git’s notion of any specific version of iris.R is as an accumulation of diffs. If you go back far enough, you find the commit where the file was created in the first place. Every later version is stored by Git as that initial version, plus all the intervening diffs in the history that affect the file. We’ll set these internal details aside now, but understanding the importance of these deltas will make Git’s operations less baffling in the long run.

So, by looking at diffs, it’s easy to see how two snapshots differ, but what about the why?

Every time you make a commit you must also write a short **commit message**. Ideally, this conveys the motivation for the change. Remember, the diff will show the content. When you revisit a project after a break or need to digest recent changes made by a colleague, looking at the **history**, by reading commit messages and skimming through diffs, is an extremely efficient way to get up to speed. Figure [20.1](https://happygitwithr.com/git-basics.html#fig:commit-diff-sha-tag) shows the messages associated with the last three commits.

Every commit needs some sort of nickname, so you can identify it. Git does this automatically, assigning each commit what is called a SHA, a seemingly random string of 40 letters and numbers (it is not, in fact, random but is a SHA-1 checksum hash of the commit). Though you will be exposed to these, you don’t have to handle them directly very often and, when you do, usually the first 7 characters suffice. The commit messages in Figure [20.1](https://happygitwithr.com/git-basics.html#fig:commit-diff-sha-tag) are prefixed by such truncated SHAs. You can also designate certain snapshots as special with a **tag**, which is a name of your choosing. In a software project, it is typical to tag a release with its version, e.g., “v1.0.3”. For a manuscript or analytical project, you might tag the version submitted to a journal or transmitted to external collaborators. Figure [20.1](https://happygitwithr.com/git-basics.html#fig:commit-diff-sha-tag) shows a tag, “draft-01”, associated with the last commit.

# Chapter 21 Git commands

A collection of some of the Git commands that have been largely going on under the hood. We’ve emphasized early workflows that are possible in RStudio. But all of this and much more can be done from the command line. This list is here mostly so we can consult it during live workshops if needed.

*Unless you use the* [*GitHub API*](https://developer.github.com/v3/)*, most of the GitHub bits really have to be done from the browser.*

New local git repo from a repo on GitHub:

git clone https://github.com/jennybc/happy-git-with-r.git

Check the remote was cloned successfully:

git remote --verbose

Stage local changes, commit:

git add foo.txt

git commit --message "A commit message"

Check on the state of the Git world:

git status

git log

git log --oneline

Compare versions:

git diff

Add a remote to existing local repo:

git remote add origin https://github.com/jennybc/happy-git-with-r

git remote --verbose

git remote show origin

Push local master to GitHub master and have local master track master on GitHub:

git push --set-upstream origin master

## shorter form

git push -u origin master

## you only need to set upstream tracking once!

Regular push:

git push

## the above usually implies (and certainly does in our tutorial)

git push origin master

## git push [remote-name] [branch-name]

Pull commits from GitHub:

git pull

Pull commits and don’t let it put you in a merge conflict pickle:

git pull --ff-only

Fetch commits

git fetch

Switch to a branch

git checkout [branch-name]

Checking remote and branch tracking

git remote -v

git branch -vv

# Chapter 22 Branches

Branching means that you take a detour from the main stream of development and do work without changing the main stream. It allows one or many people to work in parallel without overwriting each other’s work.

Branching in git is very lightweight, which means creating a branch and switching between branches is nearly instantaneous. This means git encourages workflows which create small branches for exploration or new features, often merging them back together quickly.

## 22.1 Create a new branch

You can create a new branch with git branch, then checkout the branch with git checkout. To distinguish it from the main stream of development, presumably on master, we’ll call this a “feature branch”.

git branch issue-5

git checkout issue-5

You can also use the shortcut git checkout -b issue-5 to create and checkout the branch all at once.

Once you have switched to a branch, you can commit to it as usual.

## 22.2 Switching branches

You use git checkout to switch between branches.

But what do you do if you are working on a branch and need to switch, but the work on the current branch is not complete? One option is the [Git stash](https://git-scm.com/book/en/v2/ch00/_git_stashing), but generally a better option is to safeguard the current state with a temporary commit. Here I use “WIP” as the commit message to indicate work in progress.

git commit --all -m "WIP"

git checkout master

Then when you come back to the branch and continue your work, you need to undo the temporary commit by [resetting](https://happygitwithr.com/reset.html#reset) your state. Specifically, we want a mixed reset. This is “working directory safe”, i.e. it does not affect the state of any files. But it does peel off the temporary WIP commit. Below, the reference HEAD^ says to roll the commit state back to the parent of the current commit (HEAD).

git checkout issue-5

git reset HEAD^

If this is difficult to remember, or to roll the commit state back to a different previous state, the reference can also be given as the SHA of a specific commit, which you can see via git log.

## 22.3 Merging a branch

Once you have done your work and committed it to the feature branch, you can switch back to master and merge the feature branch.

git checkout master

git merge issue-5

## 22.4 Dealing with conflicts

Most of the time, the merge will go smoothly. However if both the branches you are merging changed the same part of the same file you will get a merge conflict.

git merge issue-5

# Auto-merging index.html

# CONFLICT (content): Merge conflict in index.html

# Automatic merge failed; fix conflicts and then commit the result.

The first thing to do is **NOT PANIC**. Merge conflicts are not the end of the world and most are relatively small and straightforward to resolve.

The first step to solving a merge conflict is determining which files are in conflict, which you can do with git status:

git status

# On branch master

# You have unmerged paths.

# (fix conflicts and run "git commit")

#

# Unmerged paths:

# (use "git add <file>..." to mark resolution)

#

# both modified: index.html

#

# no changes added to commit (use "git add" and/or "git commit -a")

So this shows only index.html is unmerged and needs to be resolved. We can then open the file to see what lines are in conflict.

<<<<<<< HEAD:index.html

<div id="footer">contact : email.support@github.com</div>

=======

<div id="footer">

please contact us at support@github.com

</div>

>>>>>>> issue-5:index.html

In this conflict, the lines between <<<<<< HEAD:index.html and ====== are the content from the branch you are currently on. The lines between ======= and >>>>>>> issue-5:index.html are from the feature branch we are merging.

To resolve the conflict, edit this section until it reflects the state you want in the merged result. Pick one version or the other or create a hybrid. Also remove the conflict markers <<<<<<, ====== and >>>>>>.

<div id="footer">

please contact us at email.support@github.com

</div>

Now run git add index.html and git commit to finalize the merge. CONFLICTS RESOLVED.

### 22.4.1 Bailing out

If, during the merge, you get confused about the state of things or make a mistake, use git merge --abort to abort the merge and go back to the state prior to running git merge. Then you can try to complete the merge again.

Git Basic Branching and Merging:

<https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging>

# Chapter 23 Remotes

Remote repositories are versions of your project that are hosted on the Internet or another network. A single project can have 1, 2 or even hundreds of remotes. You pull others changes from remotes and push your changes to remotes.

## 23.1 Listing what remotes exist

git remote lists the names of available remotes, but usually it is more useful to see what URLs each note corresponds to (with -v).

git remote -v

## origin https://github.com/jennybc/happy-git-with-r.git (fetch)

## origin https://github.com/jennybc/happy-git-with-r.git (push)

## 23.2 Adding a new remote

git clone automatically adds a new remote, so often you do not need to do this manually initially. However, after the initial clone, it is often useful to add additional remotes.

Use git remote add to add a new remote

git remote add happygit https://github.com/jennybc/happy-git-with-r.git

Note: when you add a remote you give it a nickname (here happygit), which you can use in git commands in place of the entire URL.

git fetch happygit

Sidebar on nicknames: there is a strong convention to use origin as the nickname of your main remote. At this point, it is common for the main remote of a repo to be hosted on GitHub (or GitLab or Bitbucket). It is tempting to use a more descriptive nickname (such as github), but you might find that following convention is worth it. It makes your setup easier for others to understand and for you to transfer information that you read in documentation, on Stack Overflow, or in blogs.

A common reason to add a second remote is when you have done a “fork and clone” of a repo and your personal copy is set up as the origin remote. Eventually you will want to pull changes from the original repository. It is common to use upstream as the nickname for this remote.

## 23.3 Fetching data from remotes

To get new data from a remote use git fetch <remote\_name>. This retrieves the data locally, but importantly it does *not* change the state of your repository or your files in any way. To incorporate the data into your repository, you need to merge or rebase your project with the remote project.

# Fetch the data

git fetch happygit

# Now merge it with our local master

git merge happygit/master master

# git pull is a shortcut which does the above in one command

git pull happygit master

For more detail on git pull workflows, see [26](https://happygitwithr.com/pull-tricky.html#pull-tricky).

## 23.4 Pushing to remotes

Use git push <remote> <branch> to push your local changes to the <branch> branch on the <remote> remote.

# push my local changes to the origin remote's master branch

git push origin master

# push my local changes to the happygit remote's test branch

git push happygit test

## 23.5 Renaming and changing remotes

git remote rename can be used to rename a remote

git remote rename happygit hg

git remote set-url can be used to change the URL for a remote. This is sometimes useful if you initially set up a remote using https, but now want to use the SSH URL instead (or vise versa).

git remote set-url happygit git@github.com:jennybc/happy-git-with-r.git

One fairly common workflow is you initially cloned a repository on GitHub locally (without forking it), but now want to create your own fork and push changes to it. As described earlier, it is common to call the main repository upstream and to call your fork origin. So, in this case, you need to first rename the existing remote (from origin to upstream). Then add your fork as a new remote, with the name origin.

git remote rename origin upstream

git remote add origin git@github.com:jimhester/happy-git-with-r.git

## 23.6 Upstream tracking branches

It is possible to set the branch on the remote each of your local remotes corresponds to. git clone sets this up automatically, so for your own master branch this is not something you will run into. However by default if you create a new branch and try to push to it you will see something like this.

git checkout -b mybranch

git push

# fatal: The current branch foo has no upstream branch.

# To push the current branch and set the remote as upstream, use

#

# git push --set-upstream origin foo

You can do as the error message says and explicitly set the upstream branch with --set-upstream. However I would recommend instead changing the default behavior of push to automatically set the upstream branch to the branch with the same name on the remote.

You can do this by changing the git push.default option to current.

git config --global push.default current

See also Working with Remotes:

<https://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes>

# Useful Git patterns for real life

*much of this originates as live workshop materials; we are gradually creating nice prose versions*

Git patterns that come up frequently in real work:

* Commit early and often. Push less often. [The Repeated Amend](https://happygitwithr.com/repeated-amend.html#repeated-amend).
* [Help, my push was rejected!](https://happygitwithr.com/push-rejected.html#push-rejected)
* [Integrating remote and local work](https://happygitwithr.com/pull-tricky.html#pull-tricky). Probably so you can push again.
* Burn it all down.
* Time travel:
  + [“I just need to see the past”.](https://happygitwithr.com/time-travel-see-past.html#time-travel-see-past) Browse and search on GitHub.
  + “I need to visit the past”. Create a checkout a branch.
  + “I want to return to the past”. git revert, git reset
  + “I had a great cookie last October”. git cherry pick, git checkout REF -- path

Play well with others:

* [Fork and clone](https://happygitwithr.com/fork-and-clone.html#fork-and-clone).
* [Get upstream changes for a fork](https://happygitwithr.com/upstream-changes.html#upstream-changes).
* Disposable fork.
* [Make your repo rewarding to browse on GitHub](https://happygitwithr.com/workflows-browsability.html#workflows-browsability).
* [Explore and extend a pull request](https://happygitwithr.com/pr-extend.html#pr-extend)

# Chapter 24 The Repeated Amend

One of the principal joys of version control is the freedom to experiment without fear. If you make a mess of things, you can always go back to a happier version of your project. We describe several methods of such time travel in *link to come*. But you must have a good commit to fall back to!

## 24.1 Rock climbing analogy

Using a Git commit is like using anchors and other protection when climbing. If you’re crossing a dangerous rock face you want to make sure you’ve used protection to catch you if you fall. Commits play a similar role: if you make a mistake, you can’t fall past the previous commit. Coding without commits is like free-climbing: you can travel much faster in the short-term, but in the long-term the chances of catastrophic failure are high! Like rock climbing protection, you want to be judicious in your use of commits. Committing too frequently will slow your progress; use more commits when you’re in uncertain or dangerous territory. Commits are also helpful to others, because they show your journey, not just the destination.

[R Packages, Hadley Wickham](http://r-pkgs.had.co.nz/git.html#git-commit) (Wickham ([2015](https://happygitwithr.com/references.html#ref-r-pkgs-book)))

Let’s talk about this:

use more commits when you’re in uncertain or dangerous territory

When I’m doing something tricky, I often proceed towards my goal in small increments, checking that everything still works along the way. Yes it works? Make a commit. This is my new worst case scenario. Keep going.

What’s not to love?

This can lead to an awful lot of tiny commits. This is absolutely fine and nothing to be ashamed of. But one day you may start to care about the utility and aesthetics of your Git history.

The Repeated Amend is a pattern where, instead of cluttering your history with lots of tiny commits, you build up a “good” commit gradually, by amending.

*Yes, there are other ways to do this, e.g. via squashing and interactive rebase, but I think amending is the best way to get started.*

## 24.2 Workflow sketch

### 24.2.1 Initial condition

Start with your project in a functional state:

* R package? Run your tests or R CMD check.
* Data analysis? Re-run your script or re-render your .Rmd with the new chunk.
* Website or book? Make sure the project still compiles.
* You get the idea.

Make sure your “working tree is clean” and you are synced up with your GitHub remote. git status should show something like:

On branch master

Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean

### 24.2.2 Get to work

Imagine we start at commit C, with previous commit B and, before that, A:

... -- A -- B -- C

Make a small step towards your goal. Re-check that your project “works”.

Stage those changes and make a commit with the message “WIP”, meaning “work in progress”. Do this in RStudio or in the shell (Appendix [A](https://happygitwithr.com/shell.html#shell)):

git commit -m "WIP"

The message can be anything, but “WIP” is a common convention. If you use it, whenever you return to a project where the most recent commit message is “WIP”, you’ll know that you were probably in the middle of something. If you push a “WIP” commit, on purpose or by mistake, it signals to other people that more commits might be coming.

Your history now looks like this:

A -- B -- C -- WIP\*

**Don’t push!** The \* above signifies a commit that exists only in your local repo, not (yet) on GitHub.

Do a bit more work. Re-check that your project is still in a functional state. Commit again but this time **amend** your previous commit. RStudio offers a check box for “Amend previous commit” or in the shell:

git commit --amend --no-edit

The --no-edit part retains the current commit message of “WIP”.

**Don’t push!** Your history now looks like this:

A -- B -- C -- WIP\*

but the changes associated with the WIP\* commit now represent your last two commits, i.e. all the accumulated changes since state C.

Keep going like this.

Let’s say you’ve finally achieved your goal. One last time, check that your project is functional and in a state you’re willing to share with others.

Commit, amending again, but with a real commit message this time. Think of this as commit D. Push. Do this in RStudio or the shell:

git commit --amend -m "Implement awesome feature; closes #43"

git push

Your history – and that on GitHub – look like this:

A -- B -- C -- D

As far as the world knows, you implemented the feature in one fell swoop. But you got to work on the task incrementally, with the peace of mind that you could never truly break things.

## 24.3 What if I need to fall back?

Imagine you’re in the middle of a Repeated Amend workflow:

A -- B -- C -- WIP\*

and you make some changes that break your project, e.g. tests start failing. These bad changes are not yet committed, but they are saved. You want to fall back to the last good state, represented by WIP\*.

In Git lingo, you want to do a **hard reset** to the WIP\* state. With the command line:

git reset --hard

which is implicitly the same as

git reset --hard HEAD

which says: “reset my files to their state at the most recent commit”.

This is also possible in RStudio. Click on “Diff” or “Commit”. Select a file with changes you do not want. Use “Discard All” to discard all changes in that file. Use “Discard chunk” to discard changes in that file, selectively. Repeat this procedure for each affected file until you are back to an acceptable state. Carry on.

If you committed a bad state, go to *link to come* for more reset scenarios.

## 24.4 Why don’t we push intermediate progress?

Amending a commit is an example of what’s called “rewriting Git history”.

Rewriting history that has already been pushed to GitHub – and therefore potentially pulled by someone else – is a controversial practice. Like most controversial practices, lots of people still indulge in it, as do I.

But there is the very real possibility that you create headaches for yourself and others, so in Happy Git we must recommend that you abstain. Once you’ve pushed something, consider it written in stone and move on.

## 24.5 Um, what if I did push?

I told you not to!

But OK here we are.

Let’s imagine you pushed this state to GitHub by mistake:

A -- B -- C -- WIP (85bf30a)

and proceeded to git commit --amend again locally, leading to this state:

A -- B -- C -- WIP\* (6e884e6)

I’m deliberately showing two histories that sort of look the same, in terms of commit messages. But the last SHA reveals they are actually different.

You are in a pickle now, as you can’t do a simple push or pull. A push will be rejected and a pull will probably lead to a merge that you don’t want.

You have two choices:

* Hard reset your local repo to C (git reset --hard HEAD^) and pull from GitHub. Do this if you have collaborators who may have pulled that state. GitHub and local history now show this:
* A -- B -- C -- WIP (85bf30a)
* Force push to GitHub (git push --force). Do this if you have no collaborators or you have reason to believe they have not pulled. GitHub and local history now show this:
* A -- B -- C -- WIP\* (6e884e6)

Now you are synced up again.

# Chapter 25 Dealing with push rejection

Problem: You want to push changes to GitHub, but you are rejected like so:

$ git push

To https://github.com/YOU/REPO.git

! [rejected] master -> master (fetch first)

error: failed to push some refs to 'https://github.com/YOU/REPO.git'

hint: Updates were rejected because the remote contains work that you do

hint: not have locally. This is usually caused by another repository pushing

hint: to the same ref. You may want to first integrate the remote changes

hint: (e.g., 'git pull ...') before pushing again.

hint: See the 'Note about fast-forwards' in 'git push --help' for details.

This means that your local Git history and that on the GitHub remote are not compatible, i.e. they have diverged.

I suggest that you use git status, your [Git client](https://happygitwithr.com/git-client.html#git-client), or visit your GitHub remote in the browser to get more information about the situation, i.e. to get a sense of this work that you do not have.

In the abstract, this is the state on GitHub:

A -- B -- C (on GitHub)

And this is your local state:

A -- B -- D (what you have)

You can’t cause some sort of merge to happen to the GitHub copy when you push.

Instead, you’ve got pull the commit C and somehow integrate it into your D-containing history. Then you will be able to push again.

This is covered in the workflow [Pull, but you have local work](https://happygitwithr.com/pull-tricky.html#pull-tricky).

But before you behold the full horror of that, this is a great time to reflect on what we can learn from this situation.

## 25.1 She who pushes first wins!

You may have noticed that you – the author of D – are faffing around with Git more than the person who committed and pushed C, i.e. your collaborator.

There is a lesson to be learned here!

If you had pushed D first, you’d be relaxing and they’d be figuring out how to integrate C into their history in order to push. So push your work often. Don’t go dark and work “offline” for long stretches of time.

Obviously, you should push work to master because it’s “ready” to share (or at least “ready enough”), not to avoid Git merges.

There is a truly legitimate point here: It is better for the overall health of a project to be committing, pushing, and integrating more often, not less. This does not eliminate the need to integrate different lines of work, but it makes each integration smaller, less burdensome, and less prone to error.

## 25.2 Stay in touch

Another take away is this: the sooner you know about C, the better. Pull (or fetch) often.

Let’s think about your commit D. Maybe it was built up over a couple of days via the [Repeated Amend pattern](https://happygitwithr.com/repeated-amend.html#repeated-amend). Maybe C was sitting there on GitHub the whole time or appeared very early in your process.

Consider that it might be easier to integrate C into your work D sooner rather than later. Sometimes this is not true, but more often it is.

In general, it pays off to be proactively aware of what others are doing (e.g. to pull or fetch often) than to always be in reactive mode, learning about your collaborator’s work only when your push is rejected.

## 25.3 Use branches

Finally, your early experiences collaborating with others and yourself in master will give you a visceral understanding of why most Git users eventually start to use [branches](https://happygitwithr.com/git-branches.html#git-branches).

Branches afford explicit workflows for integrating different lines of work on your own terms. This is much nicer than trying to do a tricky merge or rebase in a frustrated panic, because you need to push your work to GitHub at the end of the day.

# Chapter 26 Pull, but you have local work

Problem: You want to pull changes from upstream, but you have done some new work locally since the last time you pulled. This often comes up because [what you actually want to do is *push*](https://happygitwithr.com/push-rejected.html#push-rejected), but Git won’t let you until you first incorporate the upstream changes.

For the sake of simplicity, assume we’re dealing with the master branch and the remote is called origin.

Recent commit history of origin/master:

A--B--C

Recent commit history of the local master branch:

A--B--D

or maybe

A--B--(uncommitted changes)

Your goal: get commit C into your local branch, while retaining the work in commit D or your uncommitted changes.

* Local state is A--B--(uncommitted changes): You could use git stash. Or you could just make a commit to simplify your life (see next bullet).
* Local state is A--B--D: You can get to A--B--C--D or A--B--(something that includes C and D).
* Local state is A--B--D--(uncommitted changes): You could just make a commit – a new one or amend D – to simplify your life (see previous bullet).

We prioritize simple approaches that are good for early Git use, but mention nicer long-term alternatives.

## 26.1 Local work is uncommitted

Remote state is A--B--C.  
Local state is A--B--(uncommitted changes).

### 26.1.1 Happy simple cases

There are two happy scenarios, in which git pull will “just work”:

* You’ve introduced completely new files that don’t exist in the remote branch and, therefore, cannot possibly have conflicting changes. You’re in luck! You can just git pull.
* The files affected by your local work have ZERO overlap with the files affected by the changes you need to pull from the remote. You’re also in luck! You can just git pull.

Summary of these happy git pull scenarios:

Remote: A--B--C

Local before 'git pull': A--B--(uncommitted changes)

Local after 'git pull': A--B--C--(uncommitted changes)

What has actually happened here is that git pull resulted in a *fast-forward merge*, i.e. we placed commit C right on the end of your history. This would also be the case in the simpler situation where recent local history was just A--B, i.e. you had not added any local work since the last sync up with origin/master.

### 26.1.2 git stash works, sometimes

If your changes affect a file (foo.R in the example below) that has also been changed in commit C, you cannot git pull. It doesn’t hurt to try, but you will fail and it will look something like this:

jenny@2015-mbp ethel $ git pull

remote: Enumerating objects: 5, done.

remote: Counting objects: 100% (5/5), done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 1), reused 1 (delta 0), pack-reused 0

Unpacking objects: 100% (3/3), done.

From github.com:jennybc/ethel

db046b4..2d33a6f master -> origin/master

Updating db046b4..2d33a6f

error: Your local changes to the following files would be overwritten by merge:

foo.R

Please commit your changes or stash them before you merge.

Aborting

Now what? First, you must safeguard your local changes by either stashing or committing them. (I personally would choose to commit and execute a workflow described in [26.2](https://happygitwithr.com/pull-tricky.html#git-pull-with-local-commits).)

I am not a big fan of git stash; I think it’s usually better to take every possible chance to solidify your skills around core concepts and operations, e.g., make a commit, possibly in a branch. But if you want to use git stash, this opportunity is as good as it gets.

git stash is a way to temporarily store some changes to get them out of the way. Now you can do something else, without a lot of fuss. In our case, “do something else” is to get the upstream changes with a nice, simple git pull. Then you reapply and delete the stash and pick up where you left off.

For more details about stashing, I recommend

* The stashing coverage in the “Filesystem interactions” chapter of Git in Practice ([book website](https://gitinpractice.com/) or [read on GitHub](https://github.com/GitInPractice/GitInPractice#readme))
* [7.3 Git Tools - Stashing and Cleaning](https://git-scm.com/book/en/v2/Git-Tools-Stashing-and-Cleaning) in [Pro Git](https://git-scm.com/book/en/v2).

Here’s the best case scenario for “stash, pull, unstash” in the example above:

git stash save

git pull

git stash pop

And here’s the output from our example:

jenny@2015-mbp ethel $ git stash save

Saved working directory and index state WIP on master: db046b4 Merge branch 'master'of github.com:jennybc/ethel

jenny@2015-mbp ethel $ git pull

Updating db046b4..2d33a6f

Fast-forward

foo.R | 2 +-

1 file changed, 1 insertion(+), 1 deletion(-)

jenny@2015-mbp ethel $ git stash pop

Auto-merging foo.R

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: foo.R

no changes added to commit (use "git add" and/or "git commit -a")

Dropped refs/stash@{0} (012c4dcd3a4c3af6757c4c3ca99a9eaeff1eb2a4)

That is what success looks like. You’ve achieved this:

Remote: A--B--C

Local before: A--B--(uncommitted changes)

Local after: A--B--C--(uncommitted changes)

As above, we have just enjoyed a fast-forward merge, made possible by temporarily stashing then unstashing the uncommitted local changes.

### 26.1.3 git stash with conflicts

If your local changes have some overlap with changes you are pulling, you will, instead get a merge conflict from git stash pop. Now you have some remedial work to do. In this case, you have gained nothing by using git stash in the first place, which explains my general lack of enthusiasm for git stash.

Here how to execute the git stash workflow in our example, in the face of conflicts (based on [this Stack Overflow answer](https://stackoverflow.com/a/27382210/2825349)):

jenny@2015-mbp ethel $ git stash save

Saved working directory and index state WIP on master: 2d33a6f Back to 5

jenny@2015-mbp ethel $ git pull

Updating 2d33a6f..1eddf9e

Fast-forward

foo.R | 1 +

1 file changed, 1 insertion(+)

jenny@2015-mbp ethel $ git stash pop

Auto-merging foo.R

CONFLICT (content): Merge conflict in foo.R

At this point, you must resolve the merge conflict (*future link*). Literally, at each locus of conflict, pick one version or the other (upstream or stashed) or create a hybrid yourself. Remove the all the markers inserted to demarcate the conflicts. Save.

Since git stash pop did not go smoothly, we need to manually reset (*future link*) and delete the stash to finish.

jenny@2015-mbp ethel $ git reset

Unstaged changes after reset:

M foo.R

jenny@2015-mbp ethel $ git stash drop

Dropped refs/stash@{0} (7928db50288e9b4d934803b6b451a000fd7242ed)

Phew, we are done. We’ve achieved this:

Remote: A--B--C

Local before: A--B--(uncommitted changes)

Local after: A--B--C--(uncommitted changes\*)

The asterisk on uncommitted changes\* indicates that your uncommitted changes might now reflect adjustments made when you resolved the conflicts.

## 26.2 Local work is committed

Remote state is A--B--C.  
Local state is A--B--D.

### 26.2.1 Pull (fetch and merge)

The simplest option is to fetch the commits from upstream and merge them, which is what git pull does. This is a good option if you’re new to Git. It leads to a messier history, but when you are new, this is the least of your worries. Merge, be happy, and carry on.

Here is the best case, no-merge-conflicts version of git pull:

jenny@2015-mbp ethel $ git pull

< YOU WILL PROBABLY BE KICKED INTO AN EDITOR HERE RE: MERGE COMMIT MESSAGE! >

Merge made by the 'recursive' strategy.

README.md | 4 ++--

1 file changed, 2 insertions(+), 2 deletions(-)

Depending on your version of Git, your config, and your use of a GUI, you might be required to confirm/edit a commit message for the merge commit.

Or what if things don’t go this smoothly? If commit C (on the remote) and commit D (local) have changes to the same parts of one or more files, Git may not be able to automatically merge and you will get merge conflicts. It will look something like this:

jenny@2015-mbp ethel $ git pull

Auto-merging foo.R

CONFLICT (content): Merge conflict in foo.R

Automatic merge failed; fix conflicts and then commit the result.

You must resolve these conflicts (*future link*). Literally, at each locus of conflict, pick one version or the other (upstream or local) or create a hybrid yourself. Remove the all the markers inserted to demarcate the conflicts. Save.

Mark the affected file foo.R as resolved via git add and make an explicit git commit to finalize this merge.

jenny@2015-mbp ethel $ git add foo.R

jenny@2015-mbp ethel $ git commit

[master 20b297b] Merge branch 'master' of github.com:jennybc/ethel

Again, do not be surprised if, during git commit, you find yourself in an editor, confirming/editing the commit message for the merge commit.

We’ve achieved this:

Remote: A--B--C

Local before: A--B--D

Local after: A--B--D--(merge commit)

\\_C\_/

### 26.2.2 Pull and rebase

git pull --rebase creates a nicer history than git pull when integrating local and remote commits. It avoids a merge commit, so the history is less cluttered and is linear. It can make merge conflicts more onerous to resolve, which is why I still recommend git pull as the entry-level solution.

Here is the best case, no-merge-conflicts version of git pull --rebase:

jenny@2015-mbp ethel $ git pull --rebase

First, rewinding head to replay your work on top of it...

Applying: Take max

Notice that you were NOT kicked into an editor to fiddle with the commit message for the merge commit, because there is no merge commit! This is the beauty of rebasing.

We’ve achieved this:

Remote: A--B--C

Local before: A--B--D

Local after: A--B--C--D

It is as if we pulled the upstream work in commit C, then did the local work embodied in commit D. We have no cluttery merge commits and a linear history. Nice!

The bad news: As with plain vanilla git pull, it is still possible to get merge conflicts with git pull --rebase. If you have multiple local commits, you can even find yourself resolving conflicts over and over, as these commits are sequentially replayed. Hence this is a better fit for more experienced Git users and in situations where conflicts are unlikely (those tend to be correlated, actually).

At this point, if you try to do git pull --rebase and get bogged down in merge conflicts, I recommend git rebase --abort to back out. For now, just pursue a more straightforward strategy.

## 26.3 Other approaches

There are many more ways to handle this situation, which you can discover and explore as you gain experience and start to care more about the history. We sketch some ideas here.

**26.3.1 Use a temporary branch for local work**

Recall:  
Remote state is A--B--C.  
Local state is A--B--(uncommitted changes).

This is an alternative to the stash workflow that has the advantage of giving you practice with Git techniques that are more generally useful. It also leads to a nice history.

Create a new, temporary branch and commit your uncommitted changes there. Checkout master and git pull to get changes from upstream. You now need to recover the work from the commit in the temporary branch. Options:

* Merge the temporary branch into master.
* Cherry pick the commit from the temporary branch into master.

In either case, it is still possible you will need to deal with merge conflicts.

In either case, if you felt forced to commit before you were ready or to accept an ugly merge commit, you can either do a mixed reset to “uncommit” but keep the changes on master or keep amending until you are satisfied with the commit.

## 26.4 Some local work is committed, some is not

This is an awkward hybrid situation that can be handled with a combination of strategies seen above: make a pragmatic commit on master or a temporary branch. Integrate the upstream and local changes in master. If you aren’t happy with the final pragmatic commit (which only exists locally), reset or amend until you are.

# Chapter 27 Time travel: See the past

Sometimes you just need to **see** various files in your project as they were at some significant moment in the past. Examples:

* “I liked the color scheme of this plot better in last week’s draft”. “What’s up with that new funky outlier in figure 2?”
  + Here you’ll want to visit scripts and source data as they were the last time you generated visualizations to share with this colleague.
* “The build has been failing on Windows for two weeks.”
  + Here you’ll want to inspect package source at the “last known good” version and scrutinize subsequent commits.

All projects move through various states that you regard as “good” vs. “bad” or something in between. It can be useful to explore the past, when trying to get into a “good” state.

## 27.1 Hyperlinks are awesome!

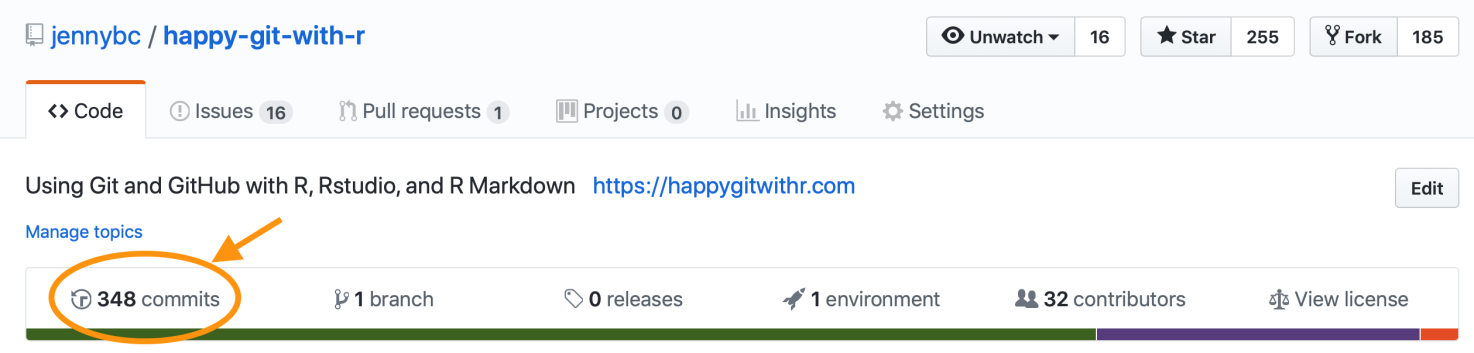
This is where GitHub (and GitLab or BitBucket) really shine. The ability to quickly explore different commits/states, switch between branches, inspect individual files, and see the discussion in linked issues is incredibly powerful.

Yes, technically, you can visit past states of your project using Git commands locally. But it is significantly more cumbersome. You generally have to checkout these previous states, which raises the prospect of getting comfortable in the “detached head” state and unintentionally making new commits on the wrong branch or on no branch at all.

GitHub’s hyperlink-rich presentation of your repo and its history is one of the top reasons to sync local work to a copy on GitHub, even if you keep it private. It can be much easier to hone in on a state or change of interest by clicking around or using GitHub’s search features. Also, because it is so clearly a remote and read-only action, there is no possibility of goofing up local state or committing new work to the wrong branch.

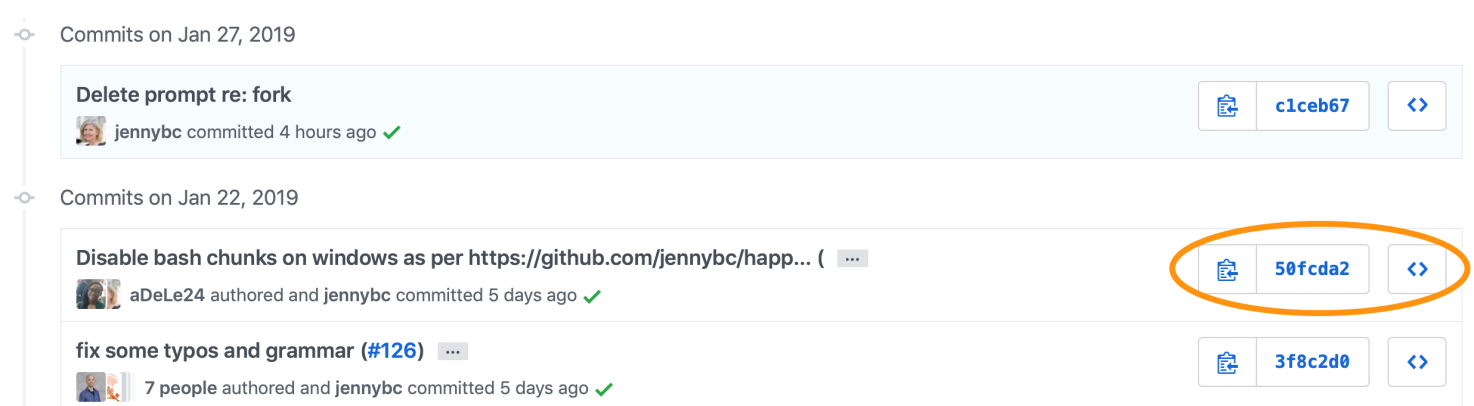
## 27.2 Browse commit history and specific commits

From your repo’s landing page, access commit history by clicking on “xyz commits”. This is like using git log locally, but much more rewarding. If you have a good [local Git client](https://happygitwithr.com/git-client.html#git-client), it probably also provides a graphical overview of history.

Figure 27.1: Link to commit listing on GitHub.

Once you’re viewing the history, notice three ways to access more info for each commit:

1. The clipboard icon copies the SHA-1 of the commit. This can be handy if you need to refer to this commit elsewhere, e.g. in an issue thread or a commit message or in a Git command you’re forming for local execution.
2. Click on the abbreviated SHA-1 itself in order to the view the diff associated with the commit.
3. Click on the double angle brackets <> to browse the state of the entire repo at that point in history.

Back out of any drilled down view by clicking on YOU/REPO to return to your repo’s landing page. This brings you back to the present state and top-level of your repo.

### 27.2.1 Use hyperlinks yourself!

Once you’ve identified a relevant commit, diff, or file state, you can copy the current URL from your browser and use it to enhance online discussion elsewhere, i.e. to bring other people to this exact view of the repo. The hyperlink-iness of repos hosted on GitHub can make online discussion of a project much more precise and efficient.

## 27.3 File driven inquiries

What if you’re interested in how a specific file came to be the way it is? First navigate to the file, then notice “Blame” and “History” in the upper right.

Figure 27.3: Visiting a specific file on GitHub.

### 27.3.1 Blame

The “blame” view of a file is related to what git blame does on the command line. It reveals who last touched each line of the file, how long ago, and the associated commit message. Click on the commit message to visit that commit. Or click the “stacked rectangles” icon to move further back in time, but staying in blame view. This is handy when doing forensics on a specific and small set of lines.

*add a screenshot (and update that above) but first pick a better example than Happy Git, i.e. one with more contributors / more interesting history*

### 27.3.2 History

The “history” view for a file is very much like the overall commit history described above, except it only includes commits that affect the file of interest. This can be handy when your inquiry is rather diffuse and you’re trying to digest the general story arc for a file.

### 27.3.3 Hyperlink to specific lines at a specific state

When viewing a file on GitHub, you can click on a line number to highlight it. Use “click … shift-click” to select a range of lines. Notice your browser’s URL shows something of this form:

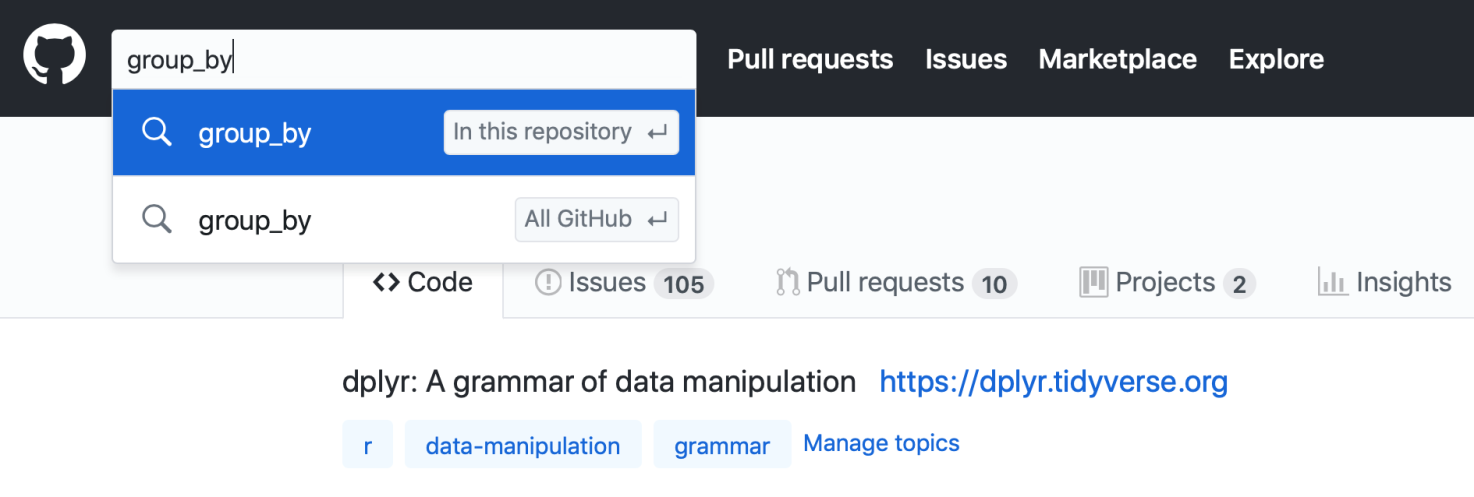
https://github.com/OWNER/REPO/blob/SHA/path/to/file.R#L27-L31

If the URL does not contain the SHA, type “y” to toggle into that form.

These file- and SHA-specific URLs are a great way to point people at particular lines of code in online conversations. It’s best practice to use the uglier links that contain the SHA, as they will stand the test of time.

## 27.4 Search

Search is always available in the upper-righthand corner of GitHub.

Figure 27.4: Typing into GitHub search bar.

Once you enter some text in the search box, a dropdown provides the choice to search in the current repo (the default) or all of GitHub. GitHub searches the contents of files (described as “Code”), commit messages, and issues. Take advantage of the search hits across these different domains. Again, this is a powerful way to zoom in on specific lines of code, revisit an interesting time in project history, or re-discover a conversation thread.

### 27.4.1 Issue search

If you want to search issues specifically, the search box on any repo’s Issues page is prepopulated with the filters is:issue and is:open.

# Chapter 28 Fork and clone

Use “fork and clone” to get a copy of someone else’s repo if there’s any chance you will want to propose a change to the owner, i.e. send a “pull request”. If you are waffling between “clone” and “fork and clone”, go with “fork and clone”.

## 28.1 Initial workflow

On [GitHub](https://github.com/), make sure you are signed in and navigate to the repo of interest. Think of this as OWNER/REPO, where OWNER is the user or organization who owns the repository named REPO.

In the upper right hand corner, click **Fork**.

This creates a copy of REPO in your GitHub account and takes you there in the browser. Now we are looking at YOU/REPO.

**Clone** YOU/REPO, which is your copy of the repo, a.k.a. your fork, to your local machine. You have two options:

* [Existing project, GitHub first](https://happygitwithr.com/existing-github-first.html#existing-github-first), an RStudio workflow we’ve used before.
  + Your fork YOU/REPO plays the role of the existing GitHub repo, in this case – not the original repo!
  + Make a conscious decision about the local destination directory and HTTPS vs SSH URL.
* Execute git clone https://github.com/YOU/REPO.git (or git clone git@github.com:YOU/REPO.git) in the shell (Appendix [A](https://happygitwithr.com/shell.html#shell)).
  + Clone your fork YOU/REPO– not the original repo!
  + cd to the desired parent directory first. Make a conscious decision about HTTPS vs SSH URL.

We’re doing this:

## C:\Users\Roy\AppData\Local\Temp\lu826050lcmy.tmp\lu826050lcno_tmp_4f618fc2d625e98b.png28.2 usethis::create\_from\_github("OWNER/REPO")

The [usethis package](https://usethis.r-lib.org/) has a convenience function, [create\_from\_github()](https://usethis.r-lib.org/reference/create_from_github.html), that can do “fork and clone”. In fact, it can go even further and [set the upstream remote](https://happygitwithr.com/upstream-changes.html#upstream-changes). However, create\_from\_github() requires that you have [configured a GitHub personal access token](https://happygitwithr.com/github-pat.html#github-pat). It hides lots of detail and can feel quite magical.

Due to these difference, we won’t dwell on create\_from\_github() here. But once you get tired of doing all of this “by hand”, check it out!

## 28.3 Engage with the new repo

If you did “fork and clone” via [Existing project, GitHub first](https://happygitwithr.com/existing-github-first.html#existing-github-first), you are probably in an RStudio Project for this new repo.

Regardless, get yourself into this project, whatever that means for you, using your usual method.

Explore the new repo in some suitable way. If it is a package, you could run the tests or check it. If it is a data analysis project, run a script or render an Rmd. Convince yourself that you have gotten the code.

## 28.4 Don’t mess with master

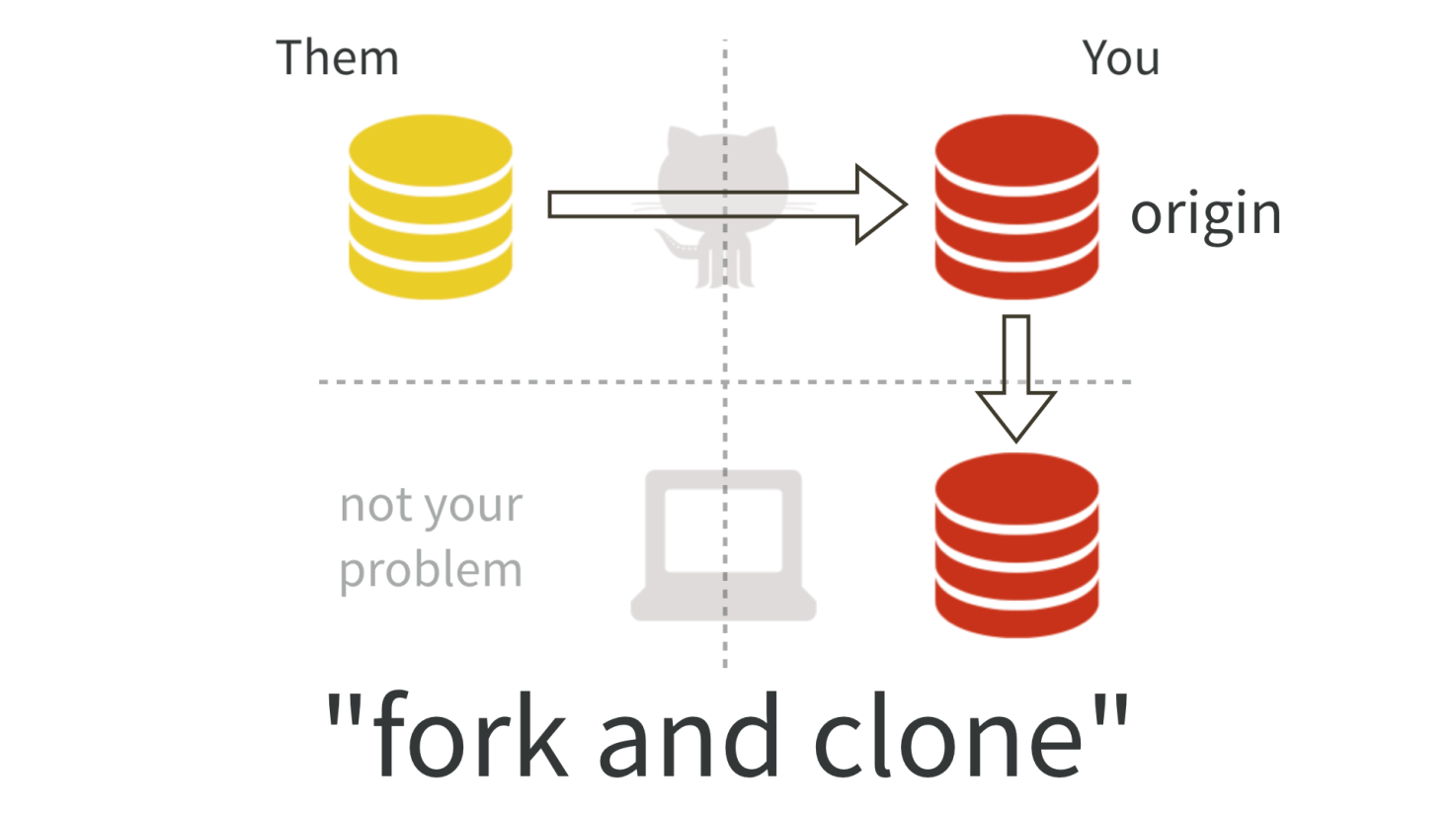
If you make any commits in your local repository, I **strongly recommend** that you work in [a new branch](https://happygitwithr.com/git-branches.html#git-branches), not master.

I **strongly recommend** that you do not make commits to master of a repo you have forked.

This will make your life much easier if you want to [pull upstream work](https://happygitwithr.com/upstream-changes.html#upstream-changes) into your copy. The OWNER of REPO will also be happier to receive your pull request from a non-master branch.

## 28.5 The original repo as a remote

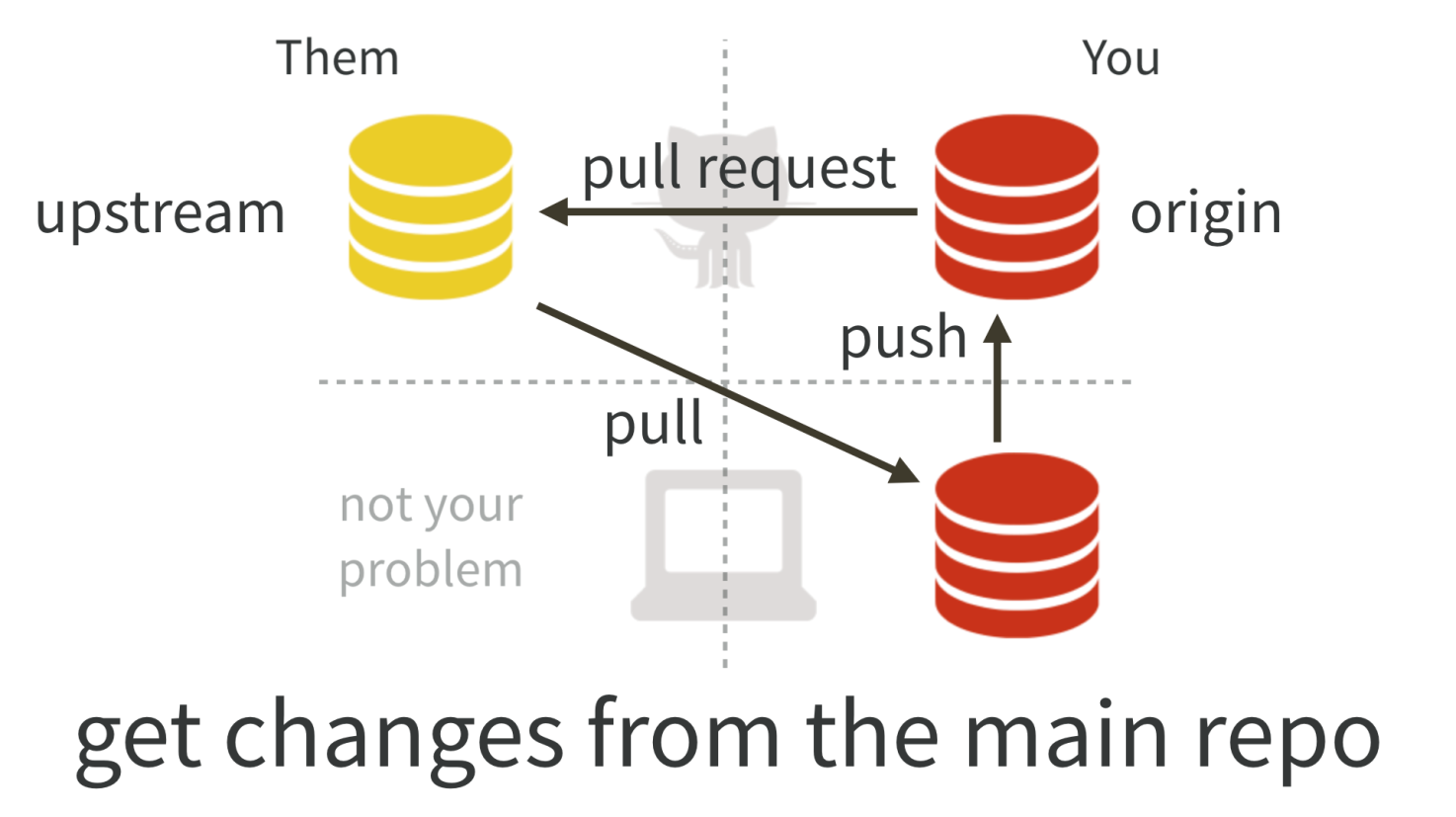
Remember we are here:

Here is the current situation in words:

* You have a fork YOU/REPO, which is a repo on GitHub.
* You have a local clone of your fork.
* Your fork YOU/REPO is the remote known as origin for your local repo.
* You are well positioned to make a pull request to OWNER/REPO.

But notice the lack of a direct connection between your local copy of this repo and the original OWNER/REPO. This is a problem.

As time goes on, the original repository OWNER/REPO will continue to evolve. You probably want the ability to keep your copy up-to-date. In Git lingo, you will need to get the “upstream changes”.

See the workflow [Get upstream changes for a fork](https://happygitwithr.com/upstream-changes.html#upstream-changes) for how to inspect your remotes, add OWNER/REPO as upstream if necessary, and pull changes, i.e. how to complete the “triangle” in the figure above.

### 28.5.1 No, you can’t do this via GitHub

You might hope that GitHub could automatically keep your fork YOU/REPO synced up with the original OWNER/REPO. Or that you could do this in the browser interface. Then you could pull those upstream changes into your local repo.

But you can’t.

There are some tantalizing, janky ways to sort of do parts of this. But they have fatal flaws that make them unsustainable. I believe you really do need to [add upstream as a second remote on your repo and pull from there](https://happygitwithr.com/upstream-changes.html#upstream-changes).

# Chapter 29 Get upstream changes for a fork

This workflow is relevant if you have done [fork and clone](https://happygitwithr.com/fork-and-clone.html#fork-and-clone) and now you need to pull subsequent changes from the original repo into your copy.

Sometimes you set this up right away, when you fork and clone, even though you don’t need it yet. Congratulations, you are planning for the future!

It’s also very typical to do this step a few days or months later. Maybe you’re taking an interest in someone else’s work for the second time and you want to make another pull request. Or you just want your copy to reflect their recent work. It is also totally normal to set this up upon first need.

Vocabulary: OWNER/REPO refers to the original GitHub repo, owned by OWNER, who is not you. YOU/REPO refers to your copy on GitHub, i.e. your fork.

## 29.1 No, you can’t do this via GitHub

You might hope that GitHub could automatically keep your fork YOU/REPO synced up with the original OWNER/REPO. Or that you could do this in the browser interface. Then you could pull those upstream changes into your local repo.

But you can’t.

There are some tantalizing, janky ways to sort of do parts of this. But they have fatal flaws that make them unsustainable. I believe you really do need to add OWNER/REPO as a second remote on your repo and pull from there.

## 29.2 Initial conditions

Get into the repo of interest, i.e. your local copy. For many of you, this means launching it as an RStudio Project. You’ll probably also want to open a terminal (Appendix [A](https://happygitwithr.com/shell.html#shell)) within RStudio for some Git work via *Tools > Terminal > New Terminal*.

Make sure you are on the master branch and your “working tree is clean”. git status should show something like:

On branch master

Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean

BTW I recommend that you [never make your own commits to the master branch of a fork](https://happygitwithr.com/fork-and-clone.html#dont-touch-master). However, if you have already done so, we are going to address your sorry situation below.

## 29.3 List your remotes

Let’s inspect [the current remotes](https://happygitwithr.com/git-remotes.html#git-remotes) for your local repo. In the shell (Appendix [A](https://happygitwithr.com/shell.html#shell)):

git remote -v

Most of you will see output along these lines (let’s call this BEFORE):

origin https://github.com/YOU/REPO.git (fetch)

origin https://github.com/YOU/REPO.git (push)

There is only one remote, named origin, corresponding to your fork on GitHub. This figure depicts a BEFORE scenario:

This is sad, because there is no direct connection between OWNER/REPO and your local copy of the repo.

The state we want to see is like this (let’s call this AFTER):

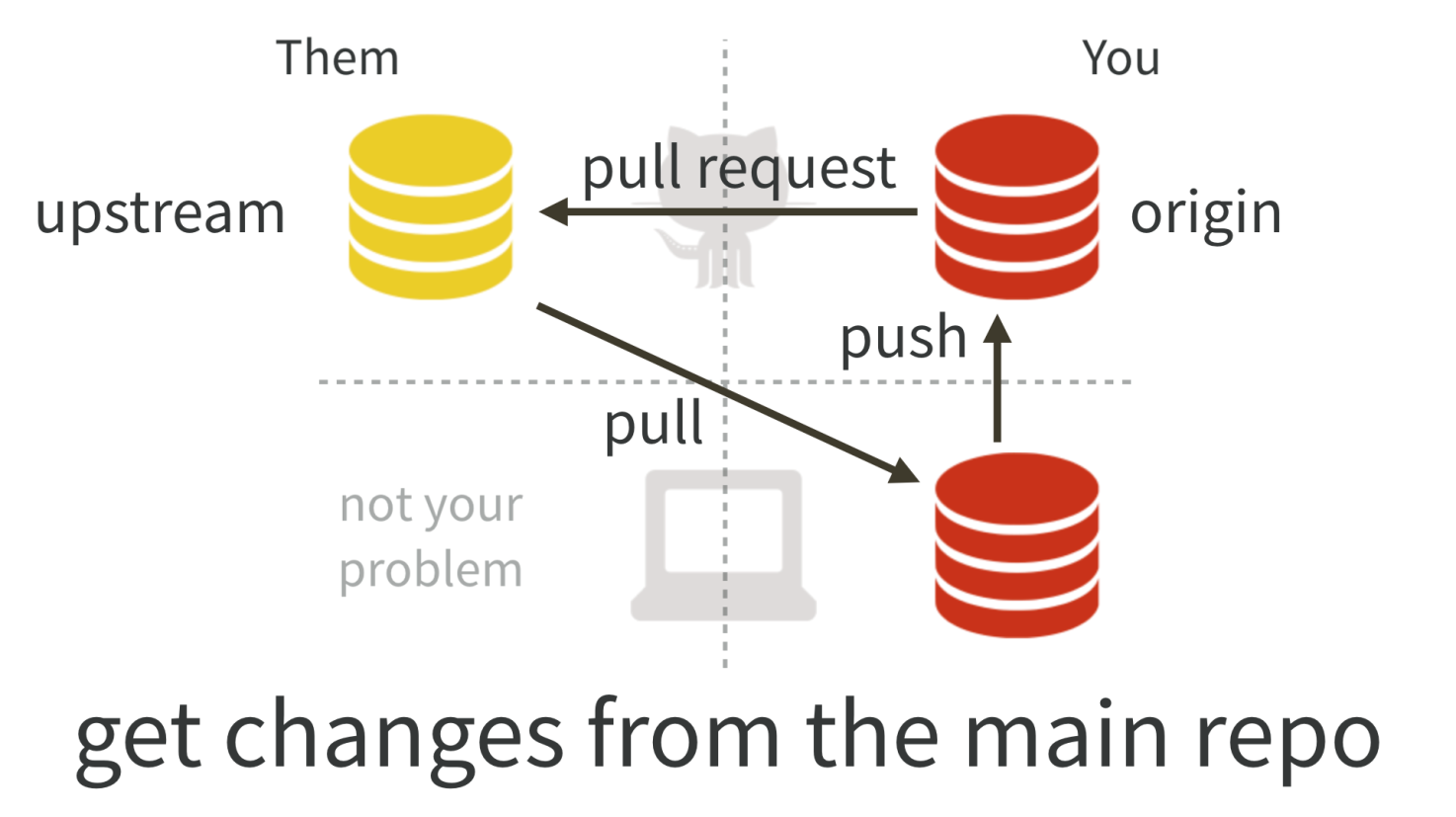
origin https://github.com/YOU/REPO.git (fetch)

origin https://github.com/YOU/REPO.git (push)

upstream https://github.com/OWNER/REPO.git (fetch)

upstream https://github.com/OWNER/REPO.git (push)

Notice the second remote, named upstream, corresponding to the original repo on GitHub. This figure depicts AFTER, the scenario we want to achieve:

Sidebar: If you used usethis::create\_from\_github("OWNER/REPO") for your original “fork and clone”, the upstream should already be set up. In that case, you can skip to the part where we pull from upstream.

## 29.4 Add the upstream remote

Let us add OWNER/REPO as the upstream remote.

On [GitHub](https://github.com/), make sure you are signed in and navigate to the original repo, OWNER/REPO. It is easy to get to from your fork, YOU/REPO, via “forked from” links near the top.

Use the big green “Clone or download” button to get the URL for OWNER/REPO on your clipboard. Be intentional about whether you copy the HTTPS or SSH URL.

### 29.4.1 Command line Git

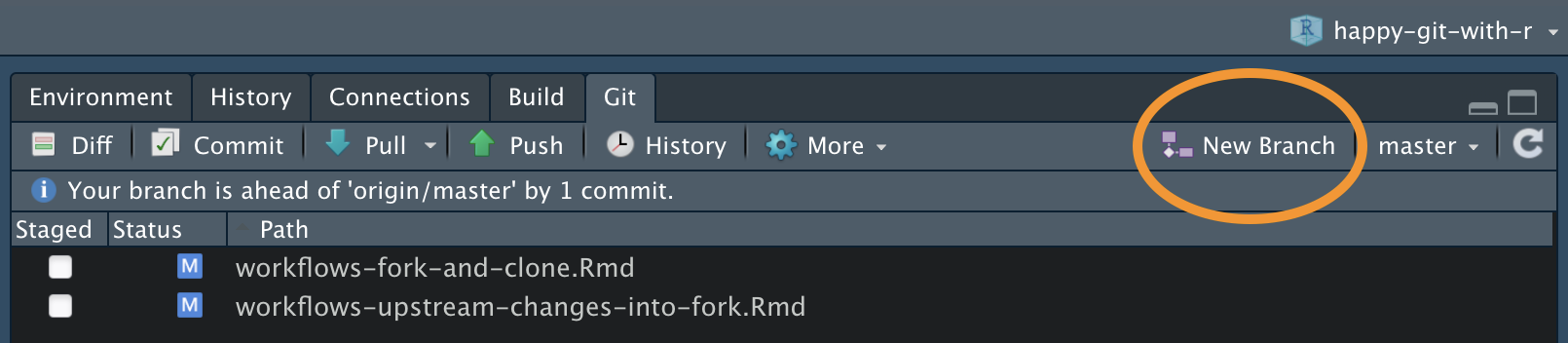
Use a command like this, but make an intentional choice about using an HTTPS vs SSH URL.

git remote add upstream https://github.com/OWNER/REPO.git

The nickname upstream can technically be whatever you want. There is a strong tradition of using upstream in this context and, even though I have better ideas, I believe it is best to conform. Every book, blog post, and Stack Overflow thread that you read will use upstream here. Save your psychic energy for other things.

### 29.4.2 RStudio

This feels a bit odd, but humor me. Click on “New Branch” in the Git pane.



This will reveal a button to “Add Remote”. Click it. Enter upstream as the remote name and paste the URL for OWNER/REPO that you got from GitHub. Click “Add”. Decline the opportunity to add a new branch by clicking “Cancel”.

## 29.5 Verify your upstream remote

Let’s inspect [the current remotes](https://happygitwithr.com/git-remotes.html#git-remotes) for your local repo AGAIN. In the shell:

git remote -v

Now you should see something like

origin https://github.com/YOU/REPO.git (fetch)

origin https://github.com/YOU/REPO.git (push)

upstream https://github.com/OWNER/REPO.git (fetch)

upstream https://github.com/OWNER/REPO.git (push)

Notice the second remote, named upstream, corresponding to the original repo on GitHub. We have gotten to this:

## C:\Users\Roy\AppData\Local\Temp\lu826050lcmy.tmp\lu826050lcno_tmp_782360aa50ab0847.png29.6 Pull changes from upstream

Now we can pull the changes that we don’t have from OWNER/REPO into our local copy.

git pull upstream master --ff-only

This says: “pull the changes from the remote known as upstream into the master branch of my local repo”. We are being explicit about the remote and the branch in this case, because (as our git remote -v commands reveal), upstream/master is **not** the default tracking branch for local master.

I **highly recommend** using the --ff-only flag in this case, so that you also say “if I have made my own commits to master, please force me to confront this problem NOW”. Here’s what it looks like if a fast-forward merge isn’t possible:

$ git pull upstream master --ff-only

From github.com:OWNER/REPO

\* branch master -> FETCH\_HEAD

fatal: Not possible to fast-forward, aborting.

See [Um, what if I did touch master?](https://happygitwithr.com/upstream-changes.html#touched-master) to get yourself back on the happy path.

## 29.7 Push these changes to origin/master

This is, frankly, totally optional and many people who are facile with Git do not bother.

If you take my advice to [never work in master of a fork](https://happygitwithr.com/fork-and-clone.html#dont-touch-master), then the state of the master branch in your fork YOU/REPO does not matter. You will never make a pull request from master.

If, however, your grasp of all these Git concepts is tenuous at best, it can be helpful to try to keep things simple and orderly and synced up.

Feel free to push the newly updated state of local master to your fork YOU/REPO and enjoy the satisfaction of being “caught up” with OWNER/REPO.

In the shell:

git push

Or use the green “Push” button in RStudio.

## 29.8 Um, what if I did touch master?

I told you not to!

But OK here we are.

Let’s imagine this is the state of the original repo OWNER/REPO:

... -- A -- B -- C -- D -- E -- F

and and this is the state of the master branch in your local copy:

... -- A -- B -- C -- X -- Y -- Z

The two histories agree, up to commit or state C, then they diverge.

If you want to preserve the work in commits X, Y, and Z, create a new branch right now, with tip at Z, via git checkout -b my-great-innovations (pick your own branch name!). Then checkout master via git checkout master.

I now assume you have either preserved the work in X, Y, and Z (with a branch) or have decided to let it go.

Do a hard reset of the master branch to C.

git reset --hard C

You will have to figure out how to convey C in Git-speak. Specify it relative to HEAD or provide the SHA. See *future link about resets* for more support.

The instructions above for pulling changes from upstream should now work. Your master branch should reflect the history of OWNER/REPO:

... -- A -- B -- C -- D -- E -- F

If you chose to create a branch with your work, you will also have that locally:

... -- A -- B -- C -- D -- E -- F (master)

\

-- X -- Y -- Z (my-great-innovations)

If you pushed your alternative history (with commits X, Y, and Z) to your fork YOU/REPO and you like keeping everything synced up, you will also need to force push master via git push --force, but we really really don’t like discussing force pushes in Happy Git. We only do so here, because we are talking about a fork, which is fairly easy to replace if things so sideways.

# Chapter 30 Explore and extend a pull request

Scenario: you maintain an R package on GitHub with pull requests (PRs) from external contributors e.g. Jane Doe, janedoe on GitHub. Sometimes you need to experiment with the PR in order to provide feedback or to decide whether or not to merge. Going further, sometimes you want to add a few commits and then merge. Or maybe there are just some merge conflicts that require your personal, local attention. Let’s also assume that you want the original PR author to get credit for their commits, i.e. you want to preserve history and provenance, not just diffs.

How do you checkout and possibly extend an external PR?

## 30.1 Terminology

Vocabulary I use throughout.

**fork branch** The name of the branch in the fork from which the PR was made. Best case scenario: informative name like fix-fluffy-bunny. Worst case scenario: PR is from master.

**local PR branch** The name of the local branch you’ll use to work with the PR. Best case scenario: can be same as fork branch. Worse case scenario: PR is from master, so you must make up a new name based on something about the PR, e.g. pr-666 or janedoe-master.

**PR parent** The SHA of the commit in the main repo that is the base for the PR.

**PR remote** The SSH or HTTPS URL for the fork from which the PR was made. Or the nickname of the remote, if you’ve bothered to set that up.

## 30.2 Official GitHub advice, Version 1

Every PR on GitHub has a link to “command line instructions” on how to merge the PR locally via command line Git. On this journey, there is a point at which you can pause and explore the PR locally.

Here are their steps with my vocabulary and some example commands:

* Create and check out the local PR branch, anticipating its relationship to the fork branch. Template of the Git command, plus an example of how it looks under both naming scenarios:
* # Template of the Git command
* git checkout -b LOCAL\_PR\_BRANCH master
* # How it looks under both naming scenarios
* git checkout -b fix-fluffy-bunny master

git checkout -b janedoe-master master

* Pull from the fork branch of the PR remote:
* # Template of the Git command
* git pull REMOTE FORK\_PR\_BRANCH
* # How it looks under both naming scenarios
* git pull https://github.com/janedoe/yourpackage.git fix-fluffy-bunny

git pull https://github.com/janedoe/yourpackage.git master

* Satisfy yourself that all is well and you want to merge.
* Checkout master:

git checkout master

* Merge the local PR branch into master with --no-ff, meaning “no fast forward merge”. This ensures you get a true merge commit, with two parents.
* # Template of the Git command
* git merge --no-ff LOCAL\_PR\_BRANCH
* # How it looks under both naming scenarios
* git merge --no-ff fix-fluffy-bunny

git merge --no-ff janedoe-master

* Push master to GitHub.

git push origin master

What’s not to like? The parent commit of the local PR branch will almost certainly not be the parent commit of the fork PR branch, where the external contributor did their work. This often means you get merge conflicts in git pull, which you’ll have to deal with ASAP. The older the PR, the more likely this is and the hairier the conflicts will be.

I would prefer to deal with the merge conflicts only *after* I’ve vetted the PR and to resolve the conflicts locally, not on GitHub. So I don’t use this exact workflow.

## 30.3 Official GitHub advice, Version 2

GitHub has another set of instructions: [Checking out pull requests locally](https://help.github.com/articles/checking-out-pull-requests-locally/)

It starts out by referring to the Version 1 instructions, but goes on to address an inactive pull request“, defined as a PR”whose owner has either stopped responding, or, more likely, has deleted their fork".

This workflow may NOT give the original PR author credit (next time it’s easy to test this, I’ll update with a definitive answer). I’ve never used it verbatim because I’ve never had this exact problem re: deleted fork.

## 30.4 Official GitHub advice, Version 3

GitHub has yet another set of instructions: [Committing changes to a pull request branch created from a fork](https://help.github.com/articles/committing-changes-to-a-pull-request-branch-created-from-a-fork/)

The page linked above explains all the pre-conditions, but the short version is that a maintainer can probably push new commits to a PR, effectively pushing commits to a fork. Strange, but true!

This set of instructions suggests that you clone the fork, checkout the branch from which the PR was made, make any commits you wish, and then push. Any new commits you make will appear in the PR. And then you could merge.

My main takeaway: maintainer can push to the branch of a fork associated with a PR.

## 30.5 My under-development workflow

*work in progress*

This combines ideas from the three above approaches, but with a few tweaks. I am sketching this up in R code, with the hope of putting this into a function and package at some point. This is a revision of an earlier approach, based on feedback from Jim Hester.

Example of a PR from the master branch (suboptimal but often happens) from fictional GitHub user abcde on usethis.

library(git2r)

## add the pull requester's fork as a named remote

remote\_add(name = "abcde", url = "git@github.com:abcde/usethis.git")

## fetch

fetch(name = "abcde")

## list remote branches and isolate the one I want

b <- branches(flags = "remote")

b <- b[["abcde/master"]]

## get the SHA of HEAD on this branch

sha <- branch\_target(b)

## create local branch

branch\_create(commit = lookup(sha = sha), name = "abcde-master")

## check it out

checkout(object = ".", branch = "abcde-master")

## set upstream tracking branch

branch\_set\_upstream(repository\_head(), name = "abcde/master")

## confirm upstream tracking branch

branch\_get\_upstream(repository\_head())

## make one or more commits here

## push to the branch in the fork and, therefore, into the PR

push()

# Chapter 31 Make a GitHub repo browsable

**The unreasonable effectiveness of GitHub browsability**. One of my favorite aspects of GitHub is the ability to inspect a repository’s files in a browser. Certain practices make browsing more rewarding and can postpone the day when you must create a proper website for a project. Perhaps indefinitely.

## 31.1 Be savvy about your files

Keep files in the plainest, web-friendliest form that is compatible with your main goals. Plain text is the very best. GitHub offers special handling for certain types of files:

* Markdown files, which may be destined for conversion into, e.g., HTML
* Markdown files named README.md
* HTML files, often the result of compiling Markdown files
* Source code, such as .R files
* Delimited files, such as CSVs and TSVs
* PNG files

## 31.2 Get over your hang ups re: committing derived products

Let’s acknowledge the discomfort some people feel about putting derived products under version control. Specifically, if you’ve got an R Markdown document foo.Rmd, it can be knit() to produce the intermediate product foo.md, which can be converted to the ultimate output foo.html. Which of those files are you “allowed” to put under version control? Source-is-real hardliners will say only foo.Rmd but pragmatists know this can be a serious bummer in real life. Just because I *can* rebuild everything from scratch, it doesn’t mean I *want* to.

The taboo of keeping derived products under version control originates from compilation of binary executables from source. Software built on a Mac would not work on Windows and so it made sense to keep these binaries out of the holy source code repository. Also, you could assume the people with access to the repository have the full development stack and relish opportunities to use it. None of these arguments really apply to the foo.Rmd --> foo.md --> foo.html workflow. We don’t have to blindly follow traditions from the compilation domain!

In fact, looking at the diffs for foo.md or foo-figure-01.png can be extremely informative. This is also true in larger data analytic projects after a make clean; make all operation. By looking at the diffs in the downstream products, you often catch unexpected changes. This can tip you off to changes in the underlying data and/or the behavior of packages you depend on.

This chapter explores cool things GitHub can do with various file types, if they happen to end up in your repo. I won’t ask you how they got there.

## 31.3 Markdown

You will quickly discover that GitHub renders Markdown files very nicely. By clicking on foo.md, you’ll get a decent preview of foo.html. Yay! You should read [GitHub’s own guide](https://guides.github.com/features/mastering-markdown/) on how to leverage automatic Markdown rendering.

Exploit this aggressively. Make Markdown your default format for narrative text files and use them liberally to embed notes to yourself and others in a repository hosted on Github. It’s an easy way to get pseudo-webpages inside a project “for free”. You may never even compile these files to HTML explicitly; in many cases, the HTML preview offered by GitHub is all you ever need.

## 31.4 R Markdown

What does this mean for R Markdown files? **Keep intermediate Markdown. Or only render to Markdown.** Commit both foo.Rmd and foo.md, even if you choose to .gitignore the final product, e.g. foo.html or foo.pdf or foo.docx. From [September 2014](https://github.com/github/markup/pull/343), GitHub renders R Markdown files nicely, like Markdown, and with proper syntax highlighting, which is great. But, of course, the code blocks just sit there un-executed, so my advice about keeping Markdown still holds.

If your target output format is not Markdown, you want [YAML frontmatter](https://gist.github.com/jennybc/402761e30b9be8023af9) that looks something like this for .Rmd:

---

title: "Something fascinating"

author: "Jenny Bryan"

date: "`r format(Sys.Date())`"

output:

html\_document:

keep\_md: TRUE

---

or like this for .R:

#' ---

#' title: "Something fascinating"

#' author: "Jenny Bryan"

#' date: "`r format(Sys.Date())`"

#' output:

#' html\_document:

#' keep\_md: TRUE

#' ---

The keep\_md: TRUE part says to keep the intermediate Markdown. In RStudio, when editing .Rmd, click on the gear next to “Knit HTML” for YAML authoring help.

Since 2016, rmarkdown offers a [custom output format for GitHub-flavored markdown, github\_document](http://rmarkdown.rstudio.com/github_document_format.html). Read about [R Markdown workflows](https://happygitwithr.com/rmd-test-drive.html#rmd-test-drive) for explicit examples of how to use this. If Markdown is your target output format, your [YAML can be even simpler](https://gist.github.com/jennybc/402761e30b9be8023af9) and look like this for .Rmd:

---

output: github\_document

---

or like this for .R:

#' ---

#' output: github\_document

#' ---

For a quick, stand-alone document that doesn’t fit neatly into a repository or project (yet), make it a [Gist](https://gist.github.com/). Example: Hadley Wickham’s [advice on what you need to do to become a data scientist](https://gist.github.com/hadley/820f09ded347c62c2864). Gists can contain multiple files, so you can still provide the R script or R Markdown source **and** the resulting Markdown, as I’ve done in this write-up of [Twitter-sourced tips for cross-tabulation](https://gist.github.com/jennybc/04b71bfaaf0f88d9d2eb). I’ve collected [YAML examples](https://gist.github.com/jennybc/402761e30b9be8023af9) for all the above scenarios in a gist.

## 31.5 README.md

You probably already know that GitHub renders README.md at the top-level of your repo as the *de facto* landing page. This is analogous to what happens when you point a web browser at a directory instead of a specific web page: if there is a file named index.html, that’s what the server will show you by default. On GitHub, files named README.md play exactly this role for directories in your repo.

Implication: for any logical group of files or mini project-within-your-project, create a sub-directory in your repository. And then create a README.md file to annotate these files, collect relevant links, etc. Now when you navigate to the sub-directory on GitHub the nicely rendered README.md will simply appear. The GitHub repo that backs the [gapminder](https://cran.r-project.org/package=gapminder) data package has a [README in the data-raw subdirectory](https://github.com/jennybc/gapminder/tree/master/data-raw#readme) that explains exactly how the package data is created. In fact, it is generated programmatically from [README.Rmd](https://github.com/jennybc/gapminder/blob/master/data-raw/README.Rmd).

Some repositories consist solely of README.md. Examples: Jeff Leek’s write-ups on [How to share data with a statistician](https://github.com/jtleek/datasharing) or [Developing R packages](https://github.com/jtleek/rpackages). I am becoming a bigger fan of README-only repos than gists because repo issues trigger notifications, whereas comments on gists do not.

If you’ve got a directory full of web-friendly figures, such as PNGs, you can use [code like this](https://gist.github.com/jennybc/0239f65633e09df7e5f4) to generate a README.md for a quick DIY gallery, as Karl Broman has done with [his FruitSnacks](https://github.com/kbroman/FruitSnacks/blob/master/PhotoGallery.md). I did same for all the [fantastic O RLY book covers](https://github.com/jennybc/orly-full-res#readme) made by The Practical Dev.

I have also used this device to share Keynote slides on GitHub (*mea culpa!*). Export them as PNGs images and throw ’em into a README gallery: slides on [file organization](https://github.com/Reproducible-Science-Curriculum/rr-organization1/tree/27883c8fc4cdd4dcc6a8232f1fe5c726e96708a0/slides/organization-slides) and some on [file naming](https://github.com/Reproducible-Science-Curriculum/rr-organization1/tree/27883c8fc4cdd4dcc6a8232f1fe5c726e96708a0/slides/naming-slides).

## 31.6 Finding stuff

OK these are pure GitHub tips but if you’ve made it this far, you’re obviously a keener.

* Press t to activate [the file finder](https://github.com/blog/793-introducing-the-file-finder) whenever you’re in a repo’s file and directory view. AWESOME, especially when there are files tucked into lots of subdirectories.
* Press y to [get a permanent link](https://help.github.com/articles/getting-permanent-links-to-files/) when you’re viewing a specific file. Watch what changes in the URL. This is important if you are about to *link* to a file or [to specific lines](http://stackoverflow.com/questions/23821235/how-to-link-to-specific-line-number-on-github). Otherwise your links will break easily in the future. If the file is deleted or renamed or if lines get inserted or deleted, your links will no longer point to what you intended. Use y to get links that include a specific commit in the URL.

## 31.7 HTML

If you have an HTML file in a GitHub repository, simply visiting the file shows the raw HTML. Here’s a nice ugly example:

* <https://github.com/STAT545-UBC/STAT545-UBC.github.io/blob/master/bit003_api-key-env-var.html>

No one wants to look at that. You can provide this URL to [rawgit.com](http://rawgit.com/) to serve this HTML more properly and get a decent preview.

You can form two different types of URLs with [rawgit.com](http://rawgit.com/):

* For sharing low-traffic, temporary examples or demos with small numbers of people, do this:
  + <https://rawgit.com/STAT545-UBC/STAT545-UBC.github.io/master/bit003_api-key-env-var.html>
  + Basically: replace https://github.com/ with https://rawgit.com/
* For use on production websites with any amount of traffic, do this:
  + <https://cdn.rawgit.com/STAT545-UBC/STAT545-UBC.github.io/master/bit003_api-key-env-var.html>
  + Basically: replace https://github.com/ with https://cdn.rawgit.com/

*2018-10-09 update: RawGit* [*announced*](https://rawgit.com/) *that it is in a sunset phase and will soon shut down. They recommended:* [*jsDelivr*](https://www.jsdelivr.com/rawgit)*,* [*GitHub Pages*](https://pages.github.com/)*,* [*CodeSandbox*](https://codesandbox.io/)*, and* [*unpkg*](https://unpkg.com/#/) *as alternatives.*

This sort of enhanced link might be one of the useful things to put in a README.md or other Markdown file in the repo.

You may also want to check out this [Chrome extension](https://chrome.google.com/webstore/detail/github-html-preview/cphnnfjainnhgejcpgboeeakfkgbkfek?hl=en) or [GitHub & BitBucket HTML Preview](https://htmlpreview.github.io/), though recently I’ve more success with [rawgit.com](http://rawgit.com/). (Neither work with private GitHub repos, which is all the more reason to keep intermediate markdown files for HTML, as described above.)

Sometimes including HTML files will cause GitHub to think that your R repository is HTML. Besides being slightly annoying, this can make it difficult for people to find your work if they are searching specifically for R repos. You can exclude these files or directories from GitHub’s language statistics by [adding a .gitattributes file](https://github.com/github/linguist#using-gitattributes) that marks them as ‘documentation’ rather than code. [See an example here](https://github.com/jennybc/googlesheets/blob/master/.gitattributes).

## 31.8 Source code

You will notice that GitHub does automatic syntax highlighting for source code. For example, notice the coloring of this [R script](https://github.com/jennybc/ggplot2-tutorial/blob/master/gapminder-ggplot2-stripplot.r). The file’s extension is the primary determinant for if/how syntax highlighting will be applied. You can see information on recognized languages, the default extensions and more at [github/linguist](https://github.com/github/linguist/blob/master/lib/linguist/languages.yml). You should be doing it anyway, but let this be another reason to follow convention in your use of file extensions.

Note you can click on “Raw” in this context as well, to get just the plain text and nothing but the plain text.

## 31.9 Delimited files

GitHub will nicely render tabular data in the form of .csv (comma-separated) and .tsv (tab-separated) files. You can read more in the [blog post](https://github.com/blog/1601-see-your-csvs) announcing this feature in August 2013 or in [this GitHub help page](https://help.github.com/articles/rendering-csv-and-tsv-data).

Advice: take advantage of this! If something in your repo can be naturally stored as delimited data, by all means, do so. Make the comma or tab your default delimiter and use the file suffixes GitHub is expecting. I have noticed that GitHub is more easily confused than R about things like quoting, so always inspect the GitHub-rendered .csv or .tsv file in the browser. You may need to do light cleaning to get the automagic rendering to work properly. Think of it as yet another way to learn about imperfections in your data.

Here’s an example of a tab delimited file on GitHub: [lotr\_clean.tsv](https://github.com/jennybc/lotr/blob/master/lotr_clean.tsv), originally found here (nope, IBM shut down manyeyes July 2015).

Note you can click on “Raw” in this context as well, to get just the plain text and nothing but the plain text.

## 31.10 PNGs

PNG is the “no brainer” format in which to store figures for the web. But many of us like a vector-based format, such as PDF, for general purpose figures. Bottom line: PNGs will drive you less crazy than PDFs on GitHub. To reduce the aggravation around viewing figures in the browser, make sure to have a PNG version in the repo.

Examples:

* [This PNG figure](https://github.com/jennybc/STAT545A/blob/master/hw06_scaffolds/01_justR/stripplot_wordsByRace_The_Fellowship_Of_The_Ring.png) just shows up in the browser
* A different figure [stored as PDF](https://github.com/jennybc/ggplot2-tutorial/blob/master/gapminder-country-colors.pdf) produces the dreaded, annoying “View Raw” speed bump. You’ll have to click through and, on my OS + browser, wait for the PDF to appear in an external PDF viewer. *2015-06-19 update: since I first wrote this GitHub has* [*elevated its treament of PDFs*](https://github.com/blog/1974-pdf-viewing) *so YAY. It’s slow but it works.*

Hopefully we are moving towards a world where you can have “web friendly” and “vector” at the same time, without undue headaches. As of [October 2014](https://github.com/blog/1902-svg-viewing-diffing), GitHub provides enhanced viewing and diffing of SVGs. So don’t read this advice as discouraging SVGs. Make them! But consider keeping a PNG around as emergency back up for now.

## 31.11 Other document formats

You may also have a document you want others to be able to browse and interact with, but it is not in the markdown format. Fortunately, the open-source Pandoc program, written by John MacFarlane, allows you to convert a range of formats into markdown, including the widely used .docx format.

When you click the Knit button in RStudio it is actually Pandoc which performs the final conversion to HTML or Microsoft Word (.docx) formats. If you are willing to use the command-line, you can perform the opposite conversion (eg .docx to .md), commonly retaining features such as headings, tables, equations and even figures.

As some boilerplate, running in Windows PowerShell pandoc --extract-media .\media -f docx .\example.docx -t markdown\_github -o example\_image.md converts a word document called example.docx to markdown, and extracts the images into a directory which corresponds to a filepath in the newly created example.md document. A full list of supported formats and example code for conversions are available at <https://pandoc.org/>.

You can also perform simple conversions to GitHub-flavored markdown from different markdown flavours (Pandoc supports markdown\_mmd, markdown\_php\_extra and markdown\_strict) from within RStudio. To do so you need to rename the file by changing the extension (eg from foo.md to foo.Rmd), then open the renamed file in RStudio and add the following text to the top of the document.

---

output: github\_document

---

You can then click on “Knit” then “Knit to github document” to perform the conversion. See [Output format](https://happygitwithr.com/workflows-browsability.html#%23%20Output%20format) for more details of controlling output formats with the YAML frontmatter.

## 31.12 Linking to a ZIP archive of your repo

The browsability of GitHub makes your work accessible to people who care about your content but who don’t (yet) use Git themselves. What if such a person wants all the files? Yes, there is a clickable “Download ZIP” button offered by GitHub. But what if you want a link to include in an email or other document? If you add /archive/master.zip *to the end* of the URL for your repo, you construct a link that will download a ZIP archive of your repository. Click here to try this out on a very small repo:

<https://github.com/jennybc/lotr/archive/master.zip>

Go look in your downloads folder!

## 31.13 Links and embedded figures

* To link to another page in your repo, just use a relative link: [admin](courseAdmin/) will link to the courseAdmin/ directory inside the current directory. [admin](/courseAdmin/) will link to the top-level courseAdmin/ directory from any where in the repo
* The same idea also works for images. ![](image.png) will include image.png located in the current directory

## 31.14 Let people correct you on the internet

They love that!

You can create a link that takes people directly to an editing interface in the browser. Behind the scenes, assuming the click-er is signed into GitHub but is not you, this will create a fork in their account and send you a pull request. When I click the link below, I am able to actually commit directly to master for this repo.

[CLICK HERE to suggest an edit to this page!](https://github.com/jennybc/happy-git-with-r/edit/master/workflows-make-github-repo-browsable.Rmd)

Here’s what that link looks like in the Markdown source:

[CLICK HERE to suggest an edit to this page!](https://github.com/jennybc/happy-git-with-r/edit/master/workflows-make-github-repo-browsable.Rmd)

and here it is with placeholders:

[INVITATION TO EDIT](<URL to your repo>/edit/master/<path to target source file>)

AFAIK, to do that in a slick automatic way across an entire repo/site, you need to be using Jekyll or some other automated system. But you could easily handcode such links on a small scale.

# Chapter 32 Clone a repo

Clone someone else’s repository on GitHub where you just want a copy. But you also want to track its evolution. That is what differentiates a GitHub clone from, say, simply downloading the ZIP archive at a specific point in time.

Pick a GitHub repository that interests you. Inspiration:

* an R package you care about
* a data analytic project you find interesting
  + Example: The GitHub repo that underpins [Polygraphing’s blog post](http://polygraph.cool/films/) analyzing 2,000 screenplays is here: <https://github.com/brandles/scripts>
  + Example: FiveThirtyEight shared the data and code behind their [Gun Deaths in America](http://fivethirtyeight.com/gun-deaths/) project on GitHub: <https://github.com/fivethirtyeight/guns-data>. Have a look around their other repos as well.

Create a new RStudio Project from this GitHub repo. Refresh your memory of how to do that by re-visiting our “GitHub first” workflow in chapter [15](https://happygitwithr.com/new-github-first.html#new-github-first).

Once you have the code locally, try to run some of it. Try to understand how it works.

Do you want to make a change? Fine do that!

Do you want to send changes back to the original author? Now you have firsthand knowledge of when you should *fork instead of clone*. See chapter [**??**](https://happygitwithr.com/clone.html#fork).

# Chapter 34 Burn it all down

This is a highly inelegant, but effective technique for disaster recovery.

It has been immortalized in an xkcd comic, so it must be ok:

* <https://xkcd.com/1597/>
* <http://explainxkcd.com/wiki/index.php/1597:_Git>

Basic idea:

* Commit early and often.
* Push to a remote, like GitHub, often.
* The state of things on GitHub is your new “worst case scenario”.
* If you really screw things up locally, copy all the files (or the ones that have changed) to a safe place.
  + Usually your files are JUST FINE. But it is easy to goof up the Git infrastructure when you’re new at this. And it can be hard to get that straightened out on your own.
* Rename the existing local repo as a temporary measure, i.e. before you do something radical, like delete it.
* Clone the repo from GitHub to your local machine. You are back to a happy state.
* Copy all relevant files back over from your safe space. The ones whose updated state you need to commit.
* Stage and commit. Push.
* Carry on with your life.

Practice this before you need it, so you see how it works.

# Chapter 35 Resetting

Practice recovering from mistakes.

Use a repository you’ve created earlier in the tutorial for this. It only needs to be local, i.e. this does not involve GitHub.

If it’s not your most recent commit, seriously consider just letting that go. Just. Let. It. Go.

So you want to undo the last commit?

If “YES UNDO IT COMPLETELY”: git reset --hard HEAD^. You will lose any changes that were not reflected in the commit-before-last!

If “YES undo the commit, but leave the files in that state (but unstaged)”: git reset HEAD^. Your files will stay the same but the commit will be undone and nothing will be staged.

If “YES go right back to the moment before I committed”: git reset --soft HEAD^. Your files will stay the same but the commit will be undone. Even your staged changes will be restored.

**If you just want to fiddle with the most recent commit or its message, you can amend it. You can do this from RStudio!**

* Make the change you want and amend the commit.
* Do you only want to change the commit message?
  + Make another small change. Surely you have a typo somewhere? Amend the commit, which gives you the chance to edit the message

To amend from the command line, using an editor to create the message:

git commit --amend

To amend from the command line, providing the new message:

git commit --amend -m "New commit message"

Git Reset Demystified:

<https://git-scm.com/book/en/v2/Git-Tools-Reset-Demystified>

# Chapter 39 Bookdown cheat sheet

Here’s where I park *little* *examples* **for myself** about bookdown mechanics that I keep forgetting.

The bookdown book: <https://bookdown.org/yihui/bookdown/>

## 39.1 Heading blah blah

## 39.2 About labelling things

You can label chapter and section titles using {#label} after them, e.g., we can reference Section [39.2](https://happygitwithr.com/bookdown-cheat-sheet.html#id-example). If you do not manually label them, there will be automatic labels anyway, e.g., this reference to the unlabelled heading [39.1](https://happygitwithr.com/bookdown-cheat-sheet.html#heading-blah-blah) uses the automatically generated label \@ref(heading-blah-blah).

## 39.3 Cross-references

Add an explicit label by adding {#label} to the end of the section header. If you know you’re going to refer to something, this is probably a good idea.

To refer to in a chapter- or section-number-y way, use \@ref(label).

* \@ref(install-git) example: In chapter [6](https://happygitwithr.com/install-git.html#install-git) we explain how to install Git.

If you are happy with the section header as the link text, use it inside a single set of square brackets:

* [A picture is worth a thousand words]: example “A picture is worth a thousand words” via [A picture is worth a thousand words](https://happygitwithr.com/git-client.html#a-picture-is-worth-a-thousand-words)

There are two ways to specify custom link text:

* [link text][Section header text], e.g., “pic = 1000 words” via [pic = 1000 words](https://happygitwithr.com/git-client.html#a-picture-is-worth-a-thousand-words)
* [link text](#label), e.g., “RStudio, meet Git” via [RStudio, meet Git](https://happygitwithr.com/rstudio-see-git.html#rstudio-see-git)

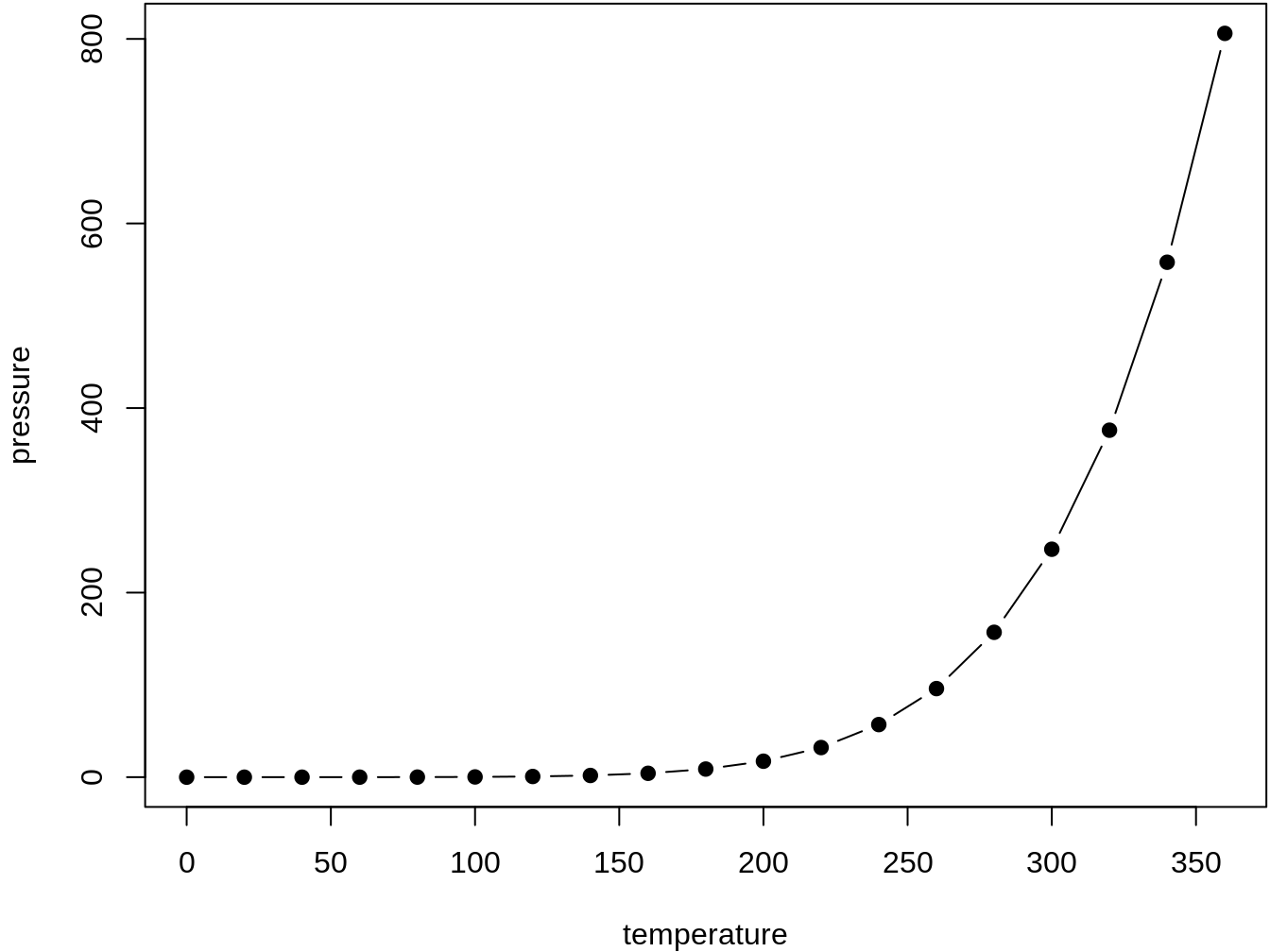
The Pandoc documentation provides more details on automatic section IDs and implicit header references.

## 39.4 Figures, tables, citations

Figures and tables with captions will be placed in figure and table environments, respectively.

par(mar = c(4, 4, .1, .1))

plot(pressure, type = 'b', pch = 19)

Figure 39.1: Here is a nice figure!

Reference a figure by its code chunk label with the fig: prefix, e.g., see Figure [39.1](https://happygitwithr.com/bookdown-cheat-sheet.html#fig:nice-fig). Similarly, you can reference tables generated from knitr::kable(), e.g., see Table [39.1](https://happygitwithr.com/bookdown-cheat-sheet.html#tab:nice-tab).

knitr::kable(

head(iris, 20), caption = 'Here is a nice table!',

booktabs = TRUE

)

Table 39.1: Here is a nice table!

| **Sepal.Length** | **Sepal.Width** | **Petal.Length** | **Petal.Width** | **Species** |
| --- | --- | --- | --- | --- |
| 5.1 | 3.5 | 1.4 | 0.2 | setosa |
| 4.9 | 3.0 | 1.4 | 0.2 | setosa |
| 4.7 | 3.2 | 1.3 | 0.2 | setosa |
| 4.6 | 3.1 | 1.5 | 0.2 | setosa |
| 5.0 | 3.6 | 1.4 | 0.2 | setosa |
| 5.4 | 3.9 | 1.7 | 0.4 | setosa |
| 4.6 | 3.4 | 1.4 | 0.3 | setosa |
| 5.0 | 3.4 | 1.5 | 0.2 | setosa |
| 4.4 | 2.9 | 1.4 | 0.2 | setosa |
| 4.9 | 3.1 | 1.5 | 0.1 | setosa |
| 5.4 | 3.7 | 1.5 | 0.2 | setosa |
| 4.8 | 3.4 | 1.6 | 0.2 | setosa |
| 4.8 | 3.0 | 1.4 | 0.1 | setosa |
| 4.3 | 3.0 | 1.1 | 0.1 | setosa |
| 5.8 | 4.0 | 1.2 | 0.2 | setosa |
| 5.7 | 4.4 | 1.5 | 0.4 | setosa |
| 5.4 | 3.9 | 1.3 | 0.4 | setosa |
| 5.1 | 3.5 | 1.4 | 0.3 | setosa |
| 5.7 | 3.8 | 1.7 | 0.3 | setosa |
| 5.1 | 3.8 | 1.5 | 0.3 | setosa |

You can write citations, too. For example, we are using the **bookdown** package (Xie [2020](https://happygitwithr.com/references.html#ref-R-bookdown)) in this sample book, which was built on top of R Markdown and **knitr** (Xie [2015](https://happygitwithr.com/references.html#ref-knitr-book)).

## 39.5 How the square bracket links work

Context: you prefer to link with text, not a chapter or section number.

* GOOD! Here’s a link to [Contributors](https://happygitwithr.com/contrib.html#contrib).
* BAD. You can see contributors in [2](https://happygitwithr.com/contrib.html#contrib).

Facts and vocabulary

* Each chapter is a file. These files should begin with the chapter title using a level-one header, e.g., # Chapter Title.
* A chapter can be made up of sections, indicated by lower-level headers, e.g., ## A section within the chapter.
* There are three ways to address a section when creating links within your book:
  + **Explicit identifier**: In # My header {#foo} the explicit identifier is foo.
  + **Automatically generated identifier**: my-header is the auto-identifier for # My header. Pandoc creates auto-identifiers according to rules laid out in [Extension: auto\_identifiers](http://pandoc.org/README.html#extension-auto_identifiers).
  + The header text, e.g., My header be used verbatim as an **implicit header reference**. See [Extension: implicit\_header\_references](http://pandoc.org/README.html#extension-implicit_header_references) for more.
* All 3 forms can be used to create cross-references but you build the links differently.
* Advantage of explicit identification: You are less likely to update the section header and then forget to make matching edits to references elsewhere in the book.

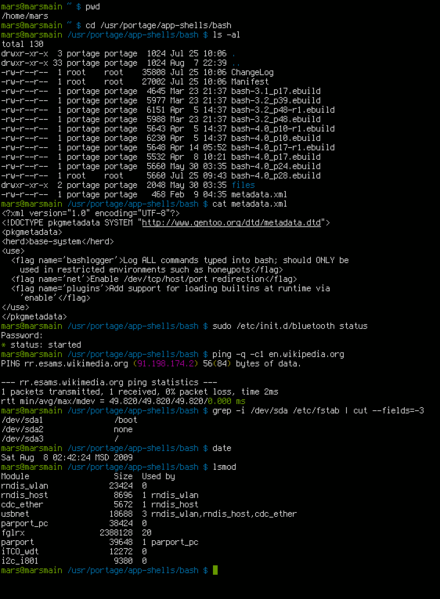
How to make text-based links using explicit identifiers, automatic identifiers, and implicit references:

* Use implicit reference alone to get a link where the text is exactly the section header:
  + [Introduce yourself to Git] [Introduce yourself to Git](https://happygitwithr.com/hello-git.html#hello-git)
  + [Success and operating systems] [Success and operating systems](https://happygitwithr.com/install-intro.html#success-and-operating-systems)
* You can provide custom text for the link with all 3 methods of addressing a section:
  + Implicit header reference: [link text][Recommended Git clients] [link text](https://happygitwithr.com/git-client.html#recommended-git-clients)
  + Explicit identifier: [hello git! I'm Jenny](#hello-git) [hello git! I’m Jenny](https://happygitwithr.com/hello-git.html#hello-git)
  + Automatic identifier: [Any text you want](#recommended-git-clients) [Any text you want](https://happygitwithr.com/git-client.html#recommended-git-clients)

# A The shell

Even if you do most of your Git operations via a client, such as RStudio or GitKraken, you must sometimes work in the shell. As you get more comfortable with Git, you might prefer to do more and more via the command line. You might also need to use Git or file system operations on a server that lacks your usual Git client. For all these reasons, it is a good idea to learn your way around the shell.

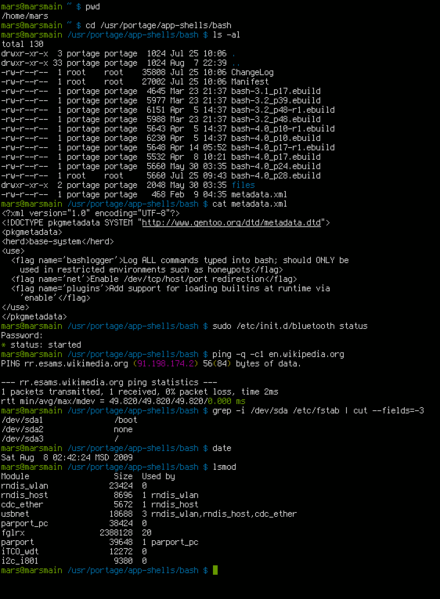
Here’s a typical look for a shell. You’ll see a simple blinking cursor, waiting for input:

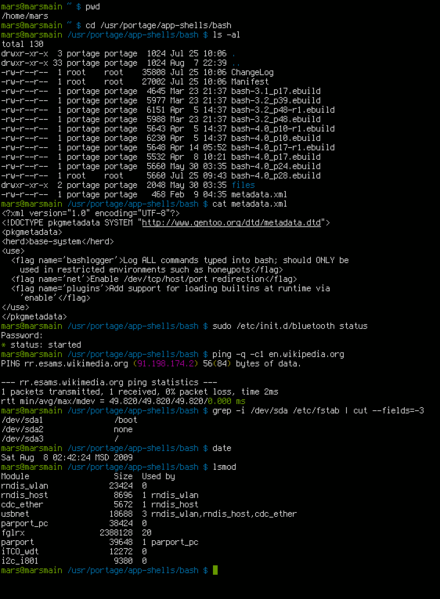


# A The shell

Even if you do most of your Git operations via a client, such as RStudio or GitKraken, you must sometimes work in the shell. As you get more comfortable with Git, you might prefer to do more and more via the command line. You might also need to use Git or file system operations on a server that lacks your usual Git client. For all these reasons, it is a good idea to learn your way around the shell.

Here’s a typical look for a shell. You’ll see a simple blinking cursor, waiting for input:





## A.1 What is the shell?

The shell is a program on your computer whose job is to run other programs. Pseudo-synonyms are “terminal”, “command line”, and “console”. There’s a whole StackExchange thread on the differences ([What is the difference between Terminal, Console, Shell, and Command Line?](https://askubuntu.com/questions/506510/what-is-the-difference-between-terminal-console-shell-and-command-line)), but I don’t find it to be terribly enlightening. Your mileage may vary.

Many programmers spend lots of time in a shell, as opposed to in GUIs, because it is very fast, concise, and ubiquitous in their relevant computing environments. This is how all work was done before we got the mouse and GUIs.

The most common shell is bash and it gets thrown around as a proxy for “shell” sometimes, just like “Coke” and “Kleenex” are proxies for cola and tissues.

In Happy Git, sometimes we demo the use of a shell for certain tasks, like navigating the file system and doing Git operations, when we don’t want to or can’t use RStudio. Providing shell commands is also less ambiguous and less perishable than describing human interactions with a GUI.

# A.2 Starting the shell

## A.2.1 From within RStudio

You can launch a shell from RStudio. This is often handy, because RStudio makes every effort to put you in a sane working directory, i.e. in the current project.

There are two ways:

* *Tools > Terminal* launches a shell within RStudio, graphically and process-wise. I believe this is usually what you want.
* *Tools > Shell …* launches a shell external to RStudio.

## A.2.2 Outside of RStudio

### A.2.2.2 Windows

We defer this until the next section, due to the more complex shell situation on Windows.

## A.3 Windows is special … and not in a good way

Windows is not the ideal platform for scientific computing and software development. A lot of the functionality is going to feel janky and strapped on. Because it is.

There are no fewer than 4 possible shells you can end up in. Unless you know better, you almost certainly want to be in a Git Bash shell, especially here in Happy Git.

Windows users will want to understand the different types of shell, how to launch them, and how to tell which one you’re in.

## A.3.1 Git Bash

*TL;DR how to tell if you’re in a Git Bash shell? Do this:*

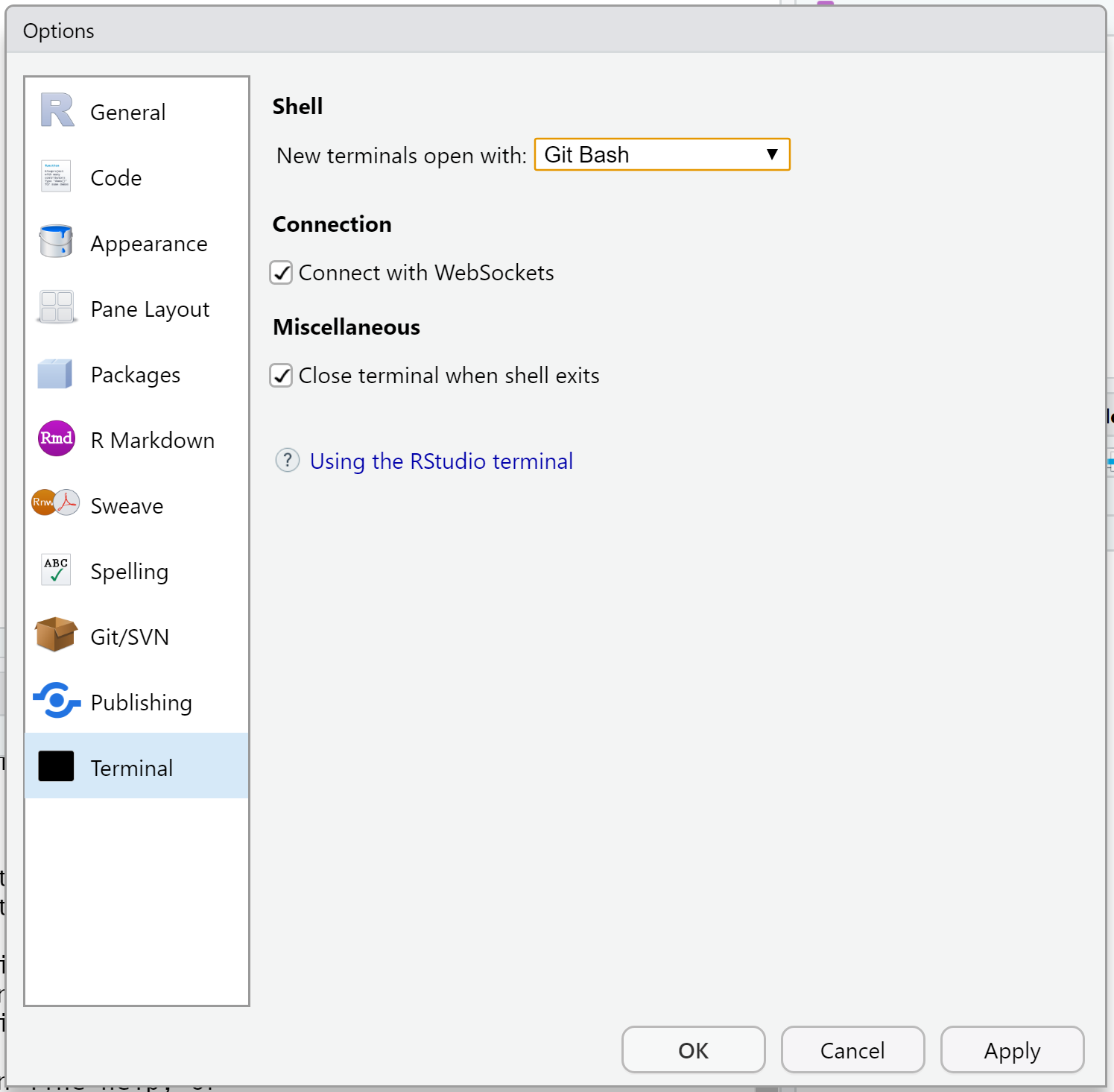
$ echo $SHELL

/usr/bin/bash

Git Bash is a bash shell that ships with Git for Windows, which is [the Happy Git way to install Git on Windows](https://happygitwithr.com/install-git.html#install-git-windows). Therefore, you will not have Git Bash on your system until you install Git for Windows.

Git Bash is always the Windows shell we are targeting in Happy Git instructions.

RStudio should automatically detect the presence of Git Bash. You can inspect and influence this directly via *Tools > Global Options > Terminal*. Unless you have good reason to do otherwise, you want to see “Git Bash” in the “New terminals open with …” dropdown menu.

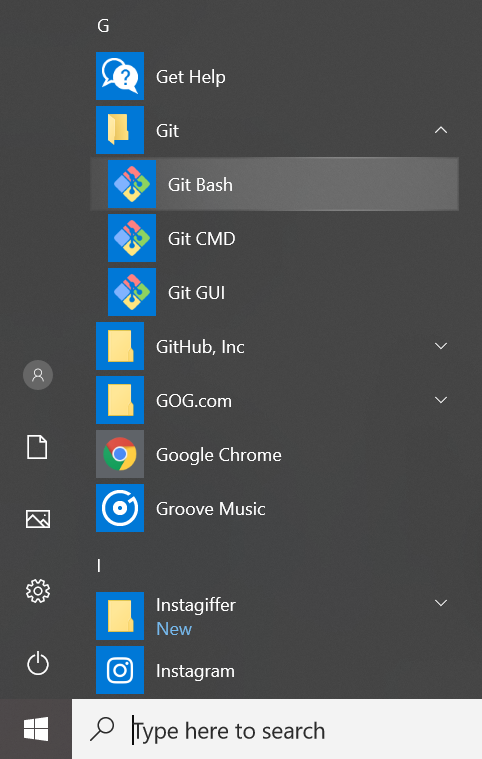


Troubleshooting tips:

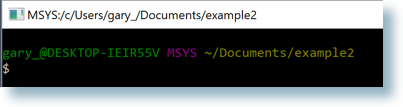
* Restart RStudio. You need to restart all instances of RStudio after installing Git for Windows (+ Git Bash), in order for RStudio to auto-detect Git Bash.
* Update RStudio. The shell handling in RStudio has improved dramatically over time, so older versions might not behave as described here.

### A.3.1.1 Accessing Git Bash outside of RStudio

Sometimes you want to run Git Bash outside of RStudio. Here’s the easiest way: click the “Git” menu in the Windows menu and select “Git Bash”.

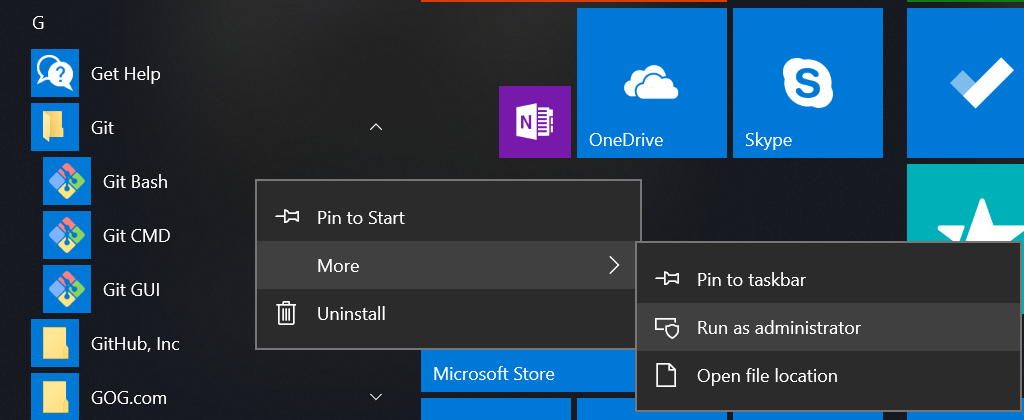


A Git Bash shell running outside of RStudio looks something like this:



Notice MSYS in the title bar. You might also see MINGW64.

Sometimes you need to run Git Bash as administrator, e.g. to run with higher privileges. Easiest way: click the “Git” menu in the Windows menu and *right-click* on “Git Bash”. This reveals a submenu. Select “more” and then “Run as administrator”.



## A.3.2 Command prompt

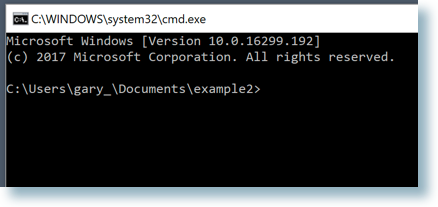
*TL;DR how to tell if you’re in Command Prompt? Do this:*

C:\Users\jenny>echo %COMSPEC%

C:\WINDOWS\system32\cmd.exe

This is the native Windows command line interpreter. It’s rarely what you want, especially for the work described in Happy Git.

A Command Prompt session running outside of RStudio looks something like this:



Notice the cmd.exe in the title bar, although it is not *always* present. You might also see “Command Prompt”.

If you get an error message such as 'pwd' is not recognized as an internal or external command, operable program or batch file. from a shell command, that suggests you have somehow launched into cmd.exe when you did not mean to.

## A.3.3 PowerShell

*TL;DR how to tell if you’re in PowerShell? Do this:*

PS C:\Users\jenny> Get-ChildItem Env:ComSpec

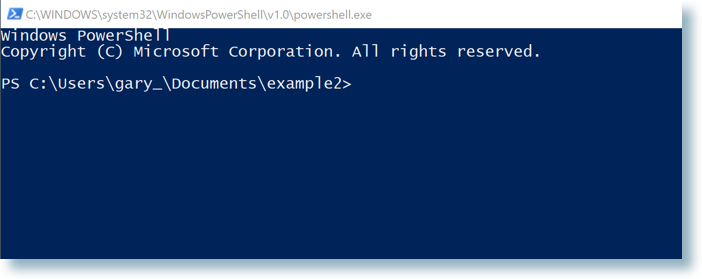
Name Value

---- -----

ComSpec C:\WINDOWS\system32\cmd.exe

PowerShell is yet another Windows shell, a more modern successor to Command Prompt. It’s also rarely what you want, especially for the work described in Happy Git.

A PowerShell session running outside of RStudio looks something like this:

Notice the powershell.exe in the title bar.

## A.3.4 Bash via Windows Services for Linux

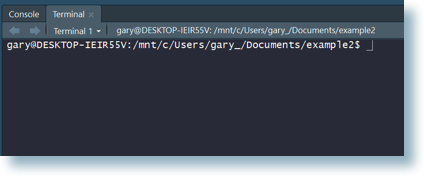
*TL;DR how to tell if you’re in Bash via WSL? Do this:*

$ echo $SHELL

/bin/bash

In 2016, Microsoft launched the Windows Subsystem for Linux (WSL), “a new Windows 10 feature that enables you to run native Linux command-line tools directly on Windows”. Overall, this is a fantastic development. However, at the time of writing (January 2019), you will only have this if you’re running Windows 10 64-bit and have chosen to [install the optional WSL system component](https://docs.microsoft.com/en-us/windows/wsl/install-win10). Therefore, I expect only keeners to have this and, in that case, you probably don’t need this chapter.

A WSL bash shell running outside of RStudio looks something like this:



FYI Microsoft also refers to WSL as Bash on Ubuntu on Windows.

### A.3.4.1 Windows bottom line

When in doubt, you probably want to be in a Git Bash shell.

## A.4 Basic shell commands

The most basic commands are listed below:

* [pwd](https://en.wikipedia.org/wiki/Pwd) (**p**rint **w**orking **d**irectory). Shows directory or “folder” you are currently operating in. This is not necessarily the same as the R working directory you get from getwd().
* [ls](https://en.wikipedia.org/wiki/Ls) (**l**i**s**t files). Shows the files in the current working directory. This is equivalent to looking at the files in your Finder/Explorer/File Manager. Use ls -a to also list hidden files, such as .Rhistory and .git.
* [cd](https://en.wikipedia.org/wiki/Cd_(command)) (**c**hange **d**irectory). Allows you to navigate through your directories by changing the shell’s working directory. You can navigate like so:
  + go to subdirectory foo of current working directory: cd foo
  + go to parent of current working directory: cd ..
  + go to your “home” directory: [cd ~](http://tilde.club/~ford/tildepoint.jpg) or simply cd
  + go to directory using absolute path, works regardless of your current working directory: cd /home/my\_username/Desktop. Windows uses a slightly different syntax with the slashes between the folder names reversed, \, e.g. cd C:\Users\MY\_USERNAME\Desktop.
    - Pro tip 1: Dragging and dropping a file or folder into the terminal window will paste the absolute path into the window.
    - Pro tip 2: Use the tab key to autocomplete unambiguous directory and file names. Hit tab twice to see all ambiguous options.
* Use arrow-up and arrow-down to repeat previous commands. Or search for previous commands with CTRL + r.

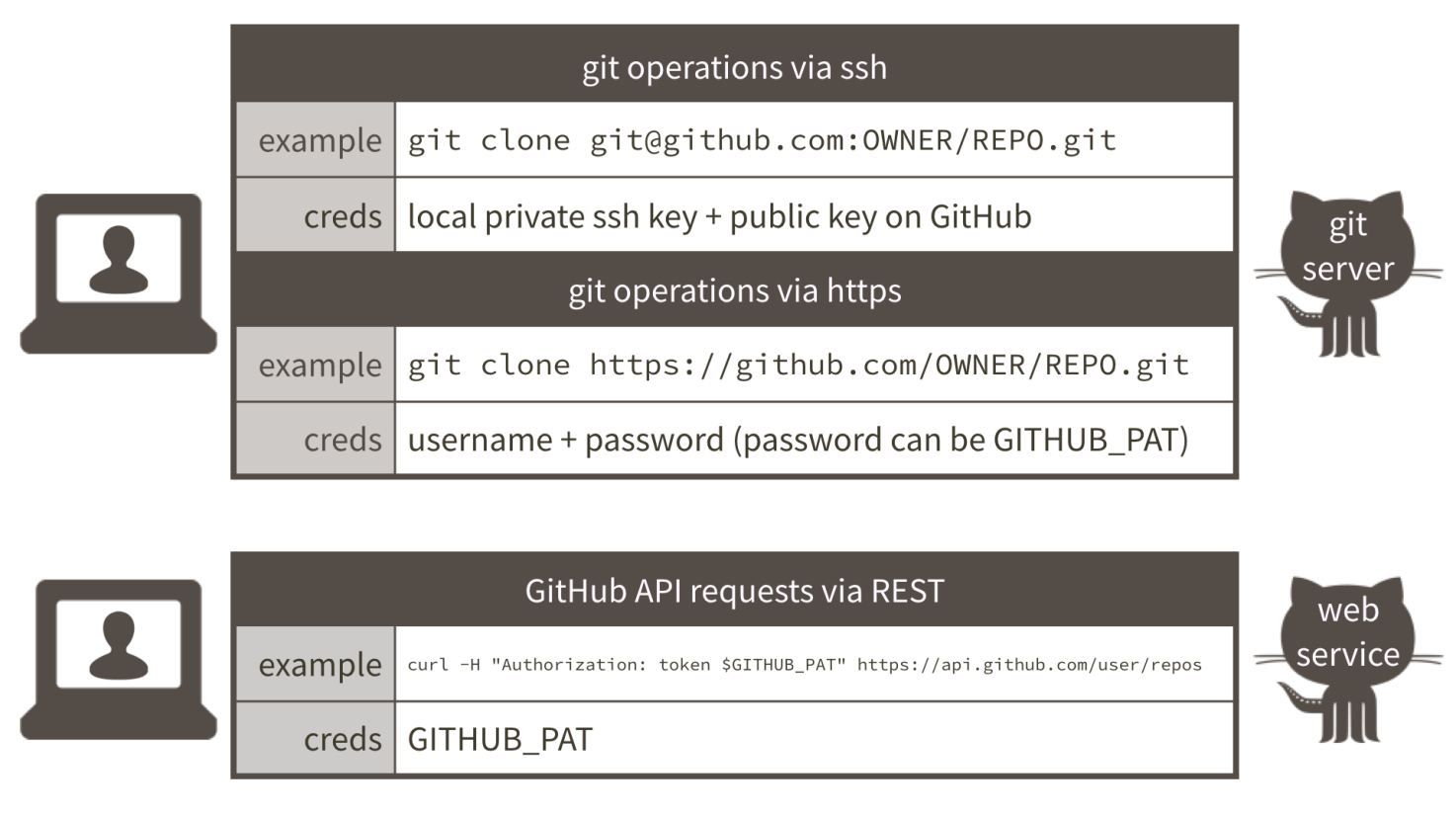
A few Git commands:

* git status is the most used git command and informs you of your current branch, any changes or untracked files, and whether you are in sync with your remotes.
* git remote -v lists all remotes. Very useful for making sure git knows about your remote and that the remote address is correct.
* git remote add origin GITHUB\_URL adds the remote GITHUB\_URL with nickname origin.
* git remote set-url origin GITHUB\_URL changes the remote url of origin to GITHUB\_URL. This way you can fix typos in the remote url.
* *Feel free to suggest other commands that deserve listing in a* [*GitHub issue*](https://github.com/jennybc/happy-git-with-r/issues)*.*

# B GitHub Personal Access Tokens

If you Git push or pull via the command line, you can avoid the constant authentication challenge by [caching your username and password](https://happygitwithr.com/credential-caching.html#credential-caching) for HTTPS access or by [setting up SSH keys](https://happygitwithr.com/ssh-keys.html#ssh-keys). This includes any Git operations done by RStudio on your behalf.

But for some operations you must call the [GitHub API](https://developer.github.com/v3/) (**A**pplication **P**rogramming **I**nterface). And this requires that you [authenticate yourself](https://developer.github.com/v3/#authentication) in a different way.

Figure B.1: Ways to authenticate to GitHub’s git server and API

## B.1 GitHub API

The first reason most people need GitHub API access is because they want to use the [devtools](https://cran.r-project.org/web/packages/devtools/index.html) package. It is popular for the devtools::install\_github() function to install directly from GitHub. This is in contrast to installing a package from CRAN via install.packages().

Why install from GitHub instead of CRAN?

* The package is developed on GitHub and you prefer the development version because it has some features or bug fixes not yet on CRAN.
* You contribute to or use R packages that are in private repositories on GitHub (or GitHub Enterprise). CRAN is not an option.
* You want to use helper functions from usethis to manage your GitHub projects, including:
  + usethis::create\_from\_github(fork = TRUE) to fork and clone a GitHub repository.
  + The usethis::pr\_\*() family of functions (pr\_init(), pr\_fetch(), pr\_pull(), pr\_push(), etc.) to create, update, and locally review pull requests.

Other reasons to call the GitHub API include doing almost anything you might do in the browser, especially if you need to do it on a grand scale, i.e., create repositories or users or fetch data. I recommend the [gh package](https://github.com/gaborcsardi/gh) for this. Which you must install from GitHub!

## B.2 How do you authenticate yourself?

With access via API, what plays the role of the HTTPS username/password or SSH keys? For the GitHub API, we use a *personal access token* (PAT), which is a convention followed by many other APIs.

You will need to:

* Obtain a PAT. This is done in the browser or with usethis::browse\_github\_pat().
* Store it so packages like devtools can find it and use it on your behalf. The path of least resistance is to define it in your .Renviron file as the environment variable named GITHUB\_PAT.

## B.3 Step-by-step

### B.3.1 Create a PAT

The easiest way to create a GitHub PAT is with usethis::browse\_github\_pat(). You will be sent to the form to create a PAT, with reasonable default scopes pre-populated.

Alternatively, you can create a PAT by logging in to your GitHub account. Go to <https://github.com/settings/tokens>. “Generate new token”. Give it a nickname that reminds you of the intended purpose, e.g., “devtools”.

Pick scopes that confer the privileges you need. When in doubt, check the repo scope, which is required for typical devtools operations. There used to be a set of default scopes, but that no longer appears to be true.

After you click “Generate token”, the token will be displayed. It’s a string of 40 random letters and digits. This is the last time you will see it SO COPY IT TO THE CLIPBOARD. Leave this window open until you’re done. If you somehow goof this up, just generate a new one and try again.

### B.3.2 Add your PAT to .Renviron

Put your PAT in your .Renviron file. Have a line that looks like this:

GITHUB\_PAT=8c70fd8419398999c9ac5bacf3192882193cadf2

but with *your* PAT instead of mine. Don’t worry, I’ve revoked this one!

.Renviron is a hidden file that lives in your home directory. Not sure where that is? The easiest way to find and edit .Renviron is with a function from the **usethis** package. In R, do:

usethis::edit\_r\_environ()

Your .Renviron file should pop up in your editor. Add your GITHUB\_PAT as above, save and close it.

Put a line break at the end! If you’re using an editor that shows line numbers, there should be two lines, where the second one is empty.

If questioned, YES you do want to use a filename that begins with a dot .. Note that, by default, most dotfiles are hidden in the RStudio file browser, but .Renviron should always be visible.

Restart R (*Session > Restart R* in the RStudio menu bar), as environment variables are loaded from .Renviron only [at the start of an R session](http://stat.ethz.ch/R-manual/R-patched/library/base/html/Startup.html). Check that the PAT is now available like so:

usethis::git\_sitrep()

You should see the following line in the output:

\* Personal access token: '<found in env var>'

Now commands you run from the devtools package, which consults GITHUB\_PAT by default, will be able to access private GitHub repositories to which you have access, and you can install them with devtools::install\_github('username/reponame').

## B.4 A note about security

Many R packages that access web services, such as [those maintained by rOpenSci](http://ropensci.org/packages/), give the option of storing API “tokens” or “keys” as environment variables in your .Renviron file. These tokens and this file, should *never be committed to GitHub, stored in Dropbox, or on any other online service*. They should not appear in your code or code comments. This is one reason why this file is stored *outside* your project folder. These tokens can be used to access your accounts [and wreak all kinds of havoc](https://securosis.com/blog/my-500-cloud-security-screwup). If you mistakenly share your tokens, log into the relevant site via the browser, revoke the compromised token, and generate a new one.

## B.5 Why .Renviron instead of bash config?

**FAQ**: Why define this environment variable via .Renviron instead of in .bash\_profile or .bashrc?\*

Because there are many combinations of OS and ways of running R where the .Renviron approach “just works” and the bash stuff does not. When R is a child process of, say, Emacs or RStudio, you can’t always count on environment variables being passed to R. Put them in an R-specific start-up file and save yourself some grief.

# D Resources

We practice what we preach! This site is created with Git and R markdown, using the [bookdown](https://github.com/rstudio/bookdown/) package. Go ahead and [peek behind the scenes](https://github.com/jennybc/happy-git-with-r).

Long-term, you should understand more about what you are doing. Rote clicking in RStudio may be a short-term survival method but won’t work for long.

* [trygit](https://try.github.io/levels/1/challenges/1) is to (command line) Git as [swirl](http://swirlstats.com/) is to R. Learn by doing, in small bites.
* The book [Pro Git](http://git-scm.com/book) is fantastic and comprehensive.
* [Git in Practice](https://www.manning.com/books/git-in-practice) by Mike McQuaid is an more approachable book, probably better than Pro Git for most people starting out. Ancillary materials [on GitHub](https://github.com/MikeMcQuaid/GitInPractice).
* [Git for Humans](https://speakerdeck.com/alicebartlett/git-for-humans) is a great set of slides by [Alice Bartlett](https://alicebartlett.co.uk/), originally delivered in 2016 at UX Brighton.
* [GitHub’s own training materials](http://training.github.com/kit/) may be helpful. They also point to [many other resources](https://help.github.com/articles/what-are-other-good-resources-for-learning-git-and-github)
* Find a powerful Git client (chapter [8](https://happygitwithr.com/git-client.html#git-client)) if you’d like to minimize your usage of Git from the command line.
* Hadley Wickham’s book [R Packages](http://r-pkgs.had.co.nz/) has an excellent [chapter on the use of Git, GitHub, and RStudio](http://r-pkgs.had.co.nz/git.html) in R package development. He covers more advanced usage, such as commit best practices, issues, branching, and pull requests.
* Ten Simple Rules for Taking Advantage of Git and GitHub <http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1004947>
* RStudio’s guide [Version Control with Git and SVN](https://support.rstudio.com/hc/en-us/articles/200532077-Version-Control-with-Git-and-SVN)
* The book [*Team Geek*](http://shop.oreilly.com/product/0636920018025.do) has insightful advice for the human and collaborative aspects of version control. It proposes Git strategies suited to different characteristics of teams.

# E References

Bartlett, Alice. 2016. “Git for Humans.” Financial Times, London; Talk at UX Brighton. <https://speakerdeck.com/alicebartlett/git-for-humans>.

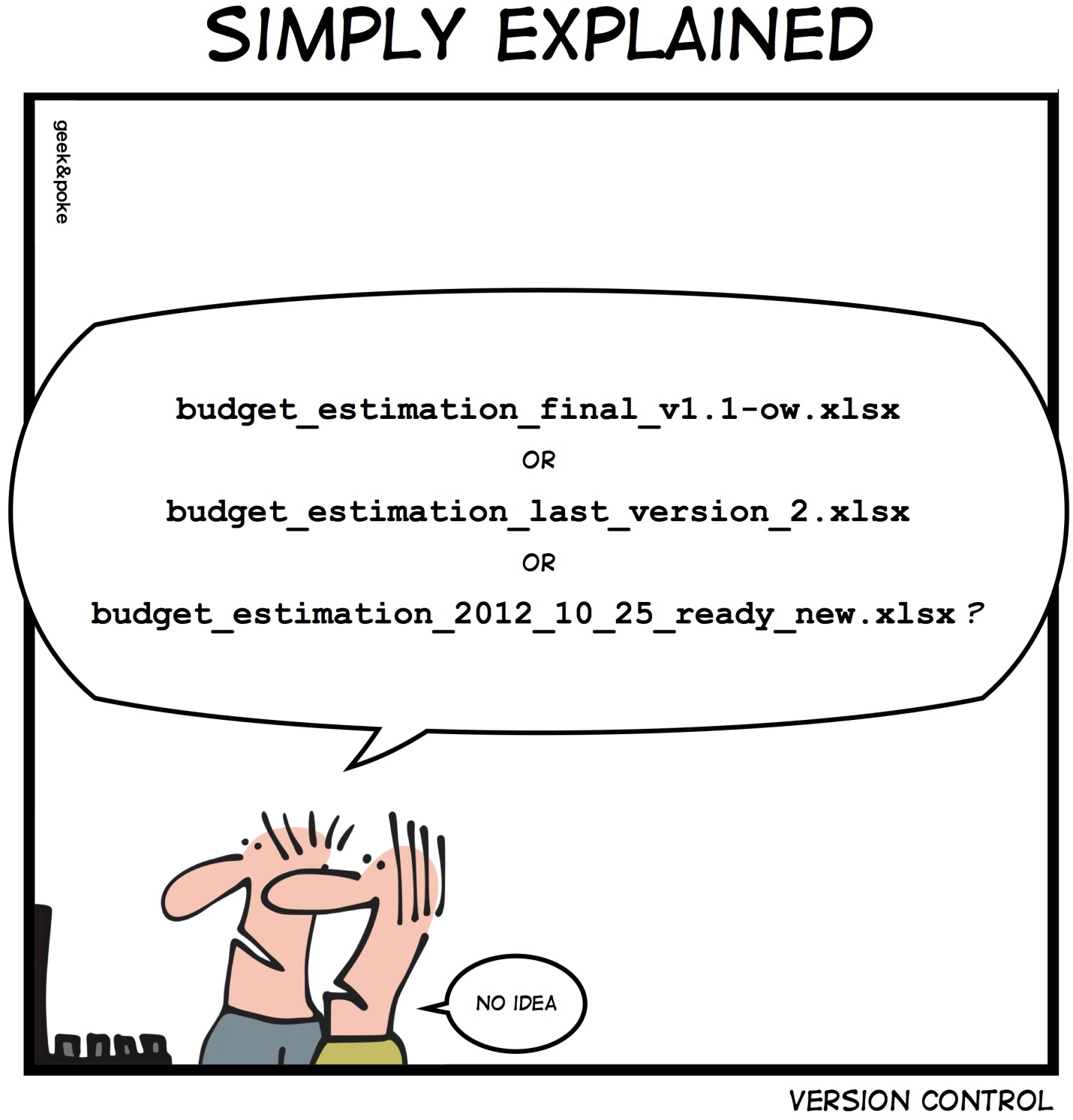
Perez-Riverol, Yasset, Laurent Gatto, Rui Wang, Timo Sachsenberg, Julian Uszkoreit, Felipe da Veiga Leprevost, Christian Fufezan, et al. 2016. “Ten Simple Rules for Taking Advantage of Git and Github.” *PLOS Computational Biology* 12 (7). Public Library of Science: 1–11. <https://doi.org/10.1371/journal.pcbi.1004947>.

Ram, Karthik. 2013. “Git Can Facilitate Greater Reproducibility and Increased Transparency in Science.” *Source Code for Biology and Medicine* 8 (1): 7. <https://doi.org/10.1186/1751-0473-8-7>.

Wickham, Hadley. 2015. *R Packages*. 1st ed. O’Reilly Media, Inc.

Xie, Yihui. 2015. *Dynamic Documents with R and Knitr*. 2nd ed. Boca Raton, Florida: Chapman; Hall/CRC. <http://yihui.name/knitr/>.

———. 2020. *Bookdown: Authoring Books and Technical Documents with R Markdown*. <https://github.com/rstudio/bookdown>.



This exercise is divided in two parts:

1. [Getting set up with Git and RStudio](https://aberdeenstudygroup.github.io/studyGroup/lessons/SG-T1-GitHubVersionControl/VersionControl/#1.)
   * 1.1. [Configure Git](https://aberdeenstudygroup.github.io/studyGroup/lessons/SG-T1-GitHubVersionControl/VersionControl/#1.1.)
2. [Version control with RStudio and Git](https://aberdeenstudygroup.github.io/studyGroup/lessons/SG-T1-GitHubVersionControl/VersionControl/#2.)
   * 2.1. [Setting up a version controlled project in RStudio](https://aberdeenstudygroup.github.io/studyGroup/lessons/SG-T1-GitHubVersionControl/VersionControl/#2.1.)
   * 2.2. [Making changes](https://aberdeenstudygroup.github.io/studyGroup/lessons/SG-T1-GitHubVersionControl/VersionControl/#2.2.)
   * 2.3. [Setting up a remote repository on GitHub](https://aberdeenstudygroup.github.io/studyGroup/lessons/SG-T1-GitHubVersionControl/VersionControl/#2.3.)
   * 2.4. [Subsequent workflow](https://aberdeenstudygroup.github.io/studyGroup/lessons/SG-T1-GitHubVersionControl/VersionControl/#2.4.)
   * 2.4.1. [Pull and push changes](https://aberdeenstudygroup.github.io/studyGroup/lessons/SG-T1-GitHubVersionControl/VersionControl/#2.4.1.)
   * 2.4.2. [Create a new branch](https://aberdeenstudygroup.github.io/studyGroup/lessons/SG-T1-GitHubVersionControl/VersionControl/#2.4.2.)
   * 2.4.3. [Make and push changes to a branch](https://aberdeenstudygroup.github.io/studyGroup/lessons/SG-T1-GitHubVersionControl/VersionControl/#2.4.3.)
   * 2.4.4. [Pull request and merge changes](https://aberdeenstudygroup.github.io/studyGroup/lessons/SG-T1-GitHubVersionControl/VersionControl/#2.4.4.)
3. [Summary](https://aberdeenstudygroup.github.io/studyGroup/lessons/SG-T1-GitHubVersionControl/VersionControl/#3.)

## 1. Getting set up with Git and RStudio

By now you should have a GitHub account ([sign up here](https://github.com/) and [get setup here](https://help.github.com/articles/signing-up-for-a-new-github-account/)) and Git installed on your laptop ([Download Git here](https://git-scm.com/downloads)). Let us know if you don’t or you will not be able to go trough the rest of this tutorial. In order to connect RStudio with GitHub we need to configure Git, which is the version control software that GitHub is built on. First let’s make sure that we have actually installed Git.

**Windows**

* Go to the Start menu
* In the Search or Run line type cmd and press enter

**Mac**

* Go to Applications -> Utilities -> Terminal

**Linux**

* Open a terminal (Ctrl+Alt+T often does it)

Type git version. You should see a short message containing some version information (Let us know if you don’t).

#### 1.1. Configure Git

After installing git, you need to tell it who you are. Open a terminal window or command prompt (see above) and type the following:

git config --global user.email "you@youremail.com"

git config --global user.name "Your Name"

On succesful completion, you should see no output from these commands. You can also configure git to use your preferred editor for commit messages, e.g. on a **Mac** or **Linux**:

git config --global core.editor nano

or on **Windows**:

git config --global core.editor notepad

It’s a good idea to follow this step since the default editor selected by git is quite difficult to use!

If the above commands in the terminal/command prompt seem like magic, or you just want to learn more about using git, try their excellent [online guides that let you learn to write git commands directly in your web browser](https://try.github.io/levels/1/challenges/1). This is also an excellent place to learn more about what we will discuss below, without worrying about messing up any actual files.

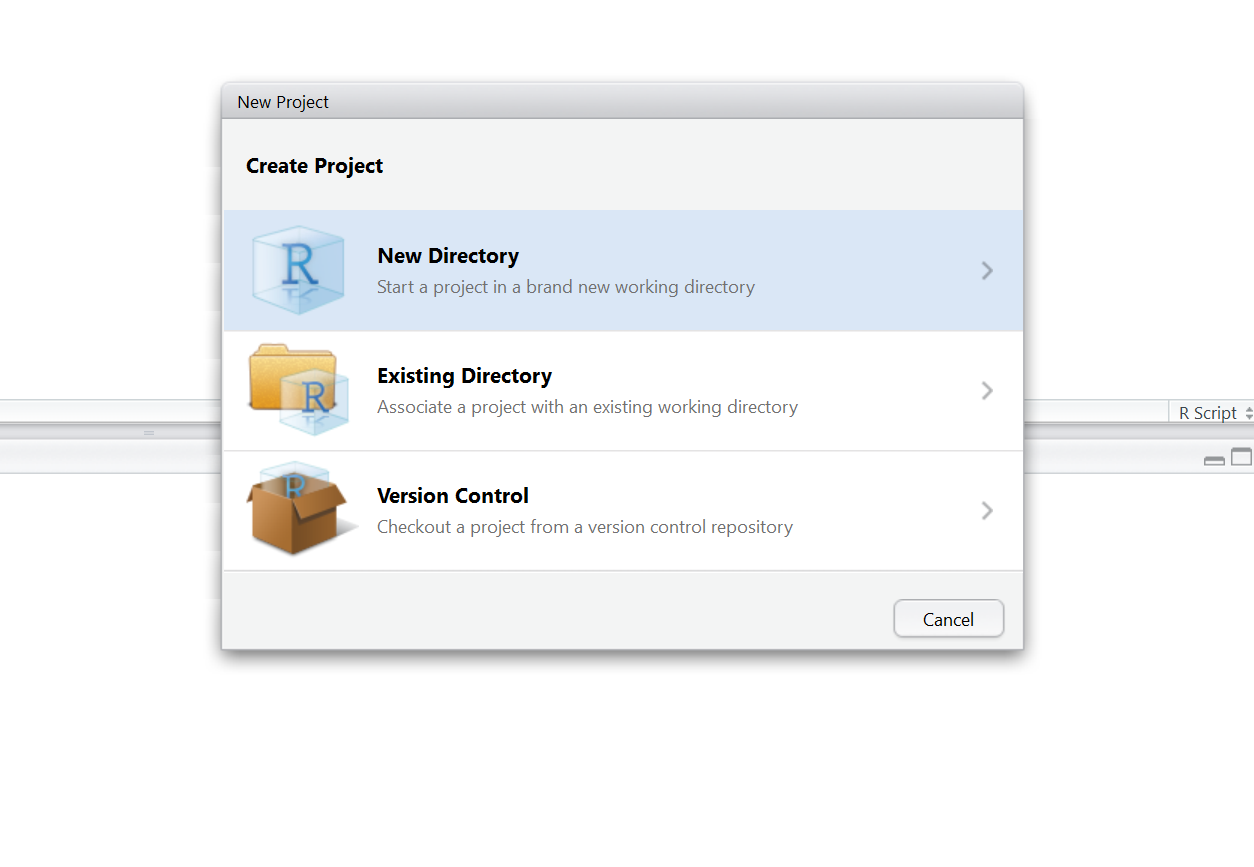
## 2. Version control with RStudio and Git

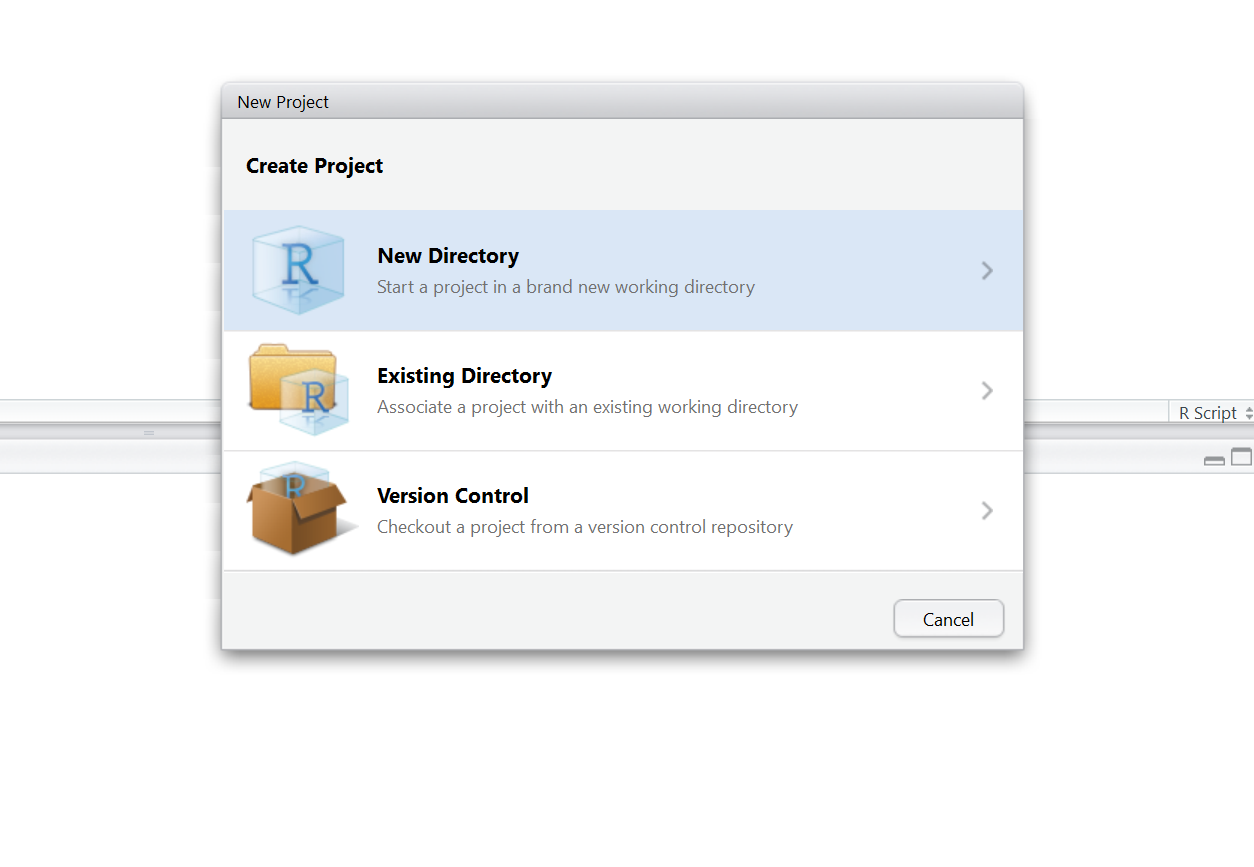
### 2.1. Setting up a version controlled project in RStudio

RStudio provides an interface for version control using git. We’ll look at setting up and managing a version controlled project through RStudio.

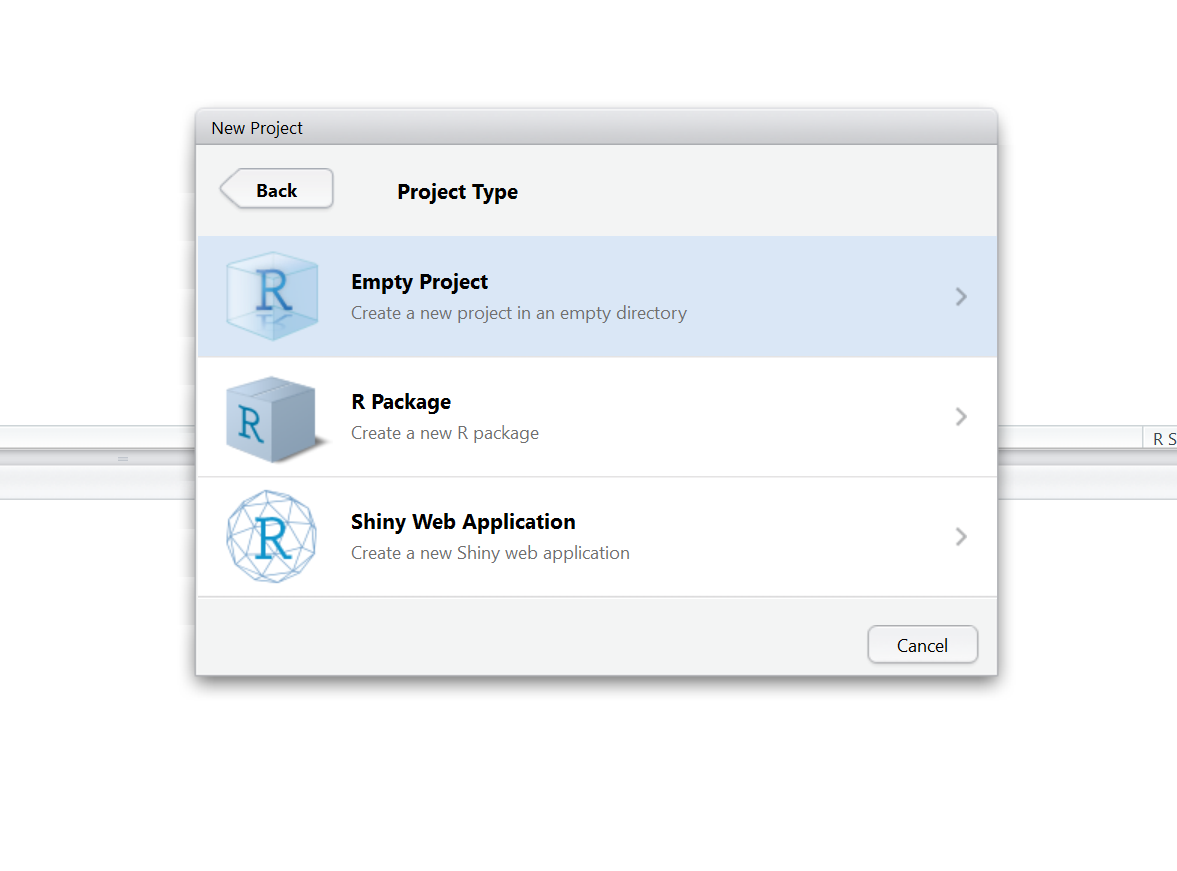
Create a new project in RStudio as follows:

**File** -> **New Project** -> **New Directory**

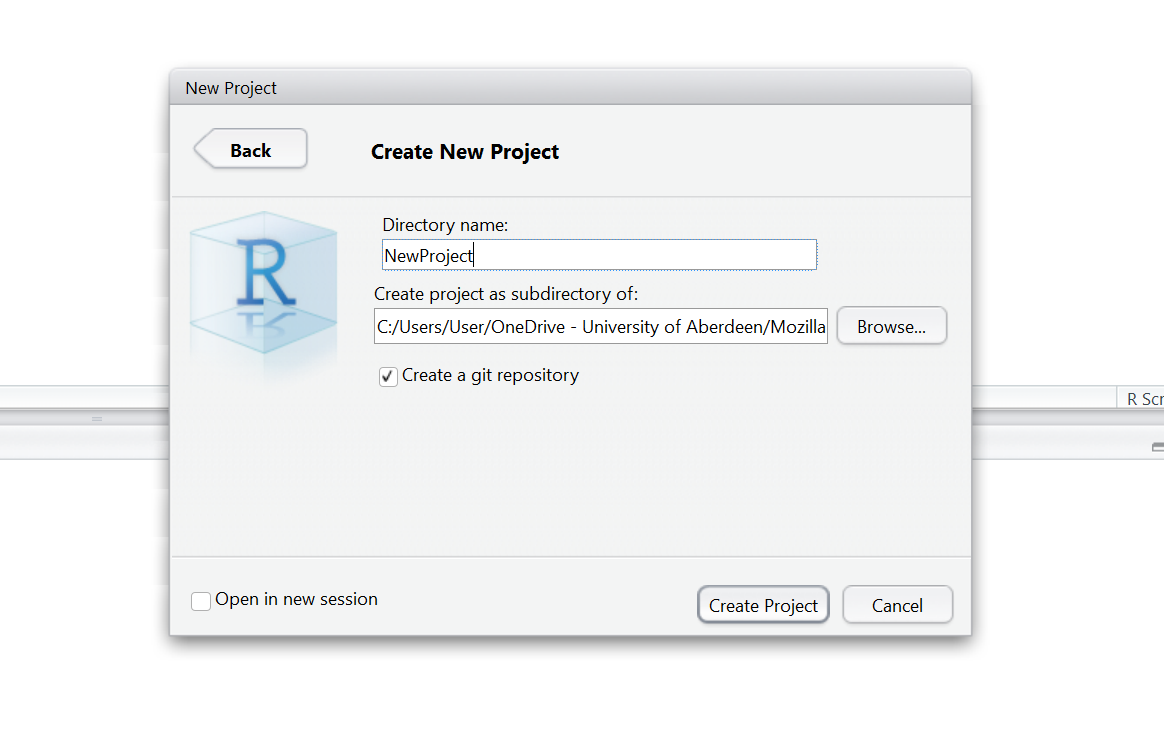




Then click on **Empty Project**

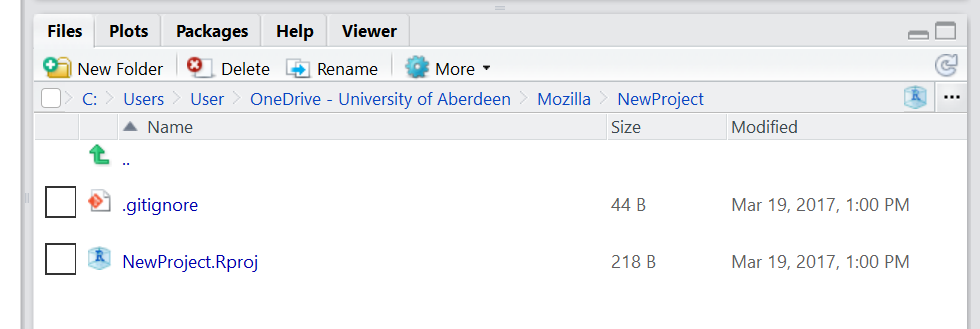


In the **Create New Project** screen, give your project a name,navigate to the directory you want to save your project in and make sure that **Create a git repository** is checked. Then click on **Create Project**.



Two files are created in the otherwise empty project:

* **.gitignore** – Specifies files that should be ignored by the version control system.
* **NewProject.Rproj** – Configuration information for the RStudio project.



Once your project is set up, you can start work on your code, perhaps by creating an R script that reads in your data and carries out some preliminary analysis. For the purposes of this tutorial, start by creating a simple script containing these few lines of R.

First open a new R script as follows:

**File** -> **New File** -> **R Script**

Then, in the new empty R script, type the following lines of code, and click on the **Run** button in the top right corner of you script.

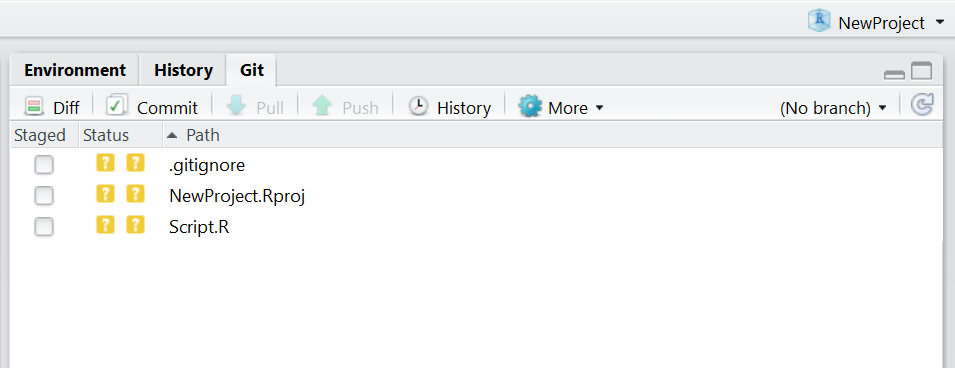
data(cars)

plot(cars)

The first line of code loads the dataset cars into R and the second line plots the data.

Now you can save your script by clicking on the disk icon in the top left corner of your script. RStudio will automatically save your script in your project directory, so just give it a name and click **Save**.

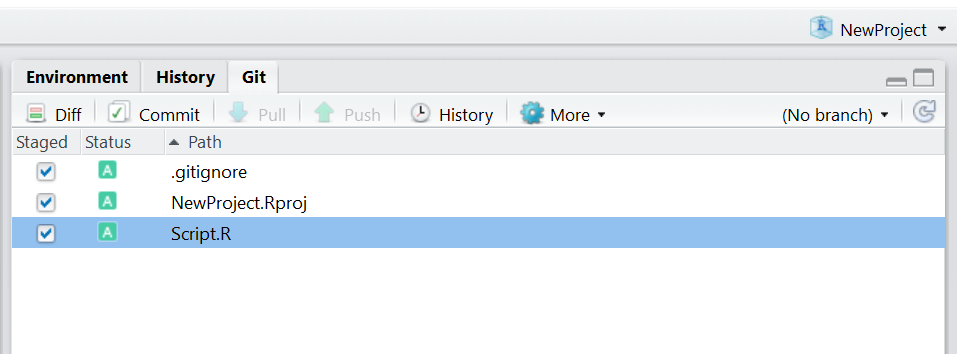
After you’ve created this file in your project, you need to get it under version control. By default, the git version control pane is in the top right hand corner of RStudio. Find it and click on the **Git** tab. You’ll see a list of the files in your project directory.



If you hover the mouse pointer over the yellow ? marks in RStudio, you’ll see a tooltip telling you that the files are **untracked**. This means that git is not tracking versions of these files.

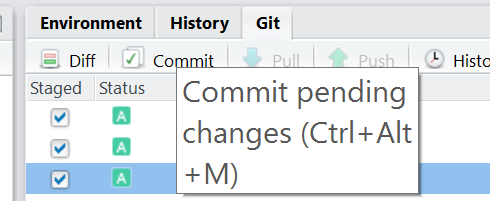
Git will only keep track of modifications to files that it has been asked to manage, so you need to tell git which files to track.

Tick the **Staged** checkbox for each file.



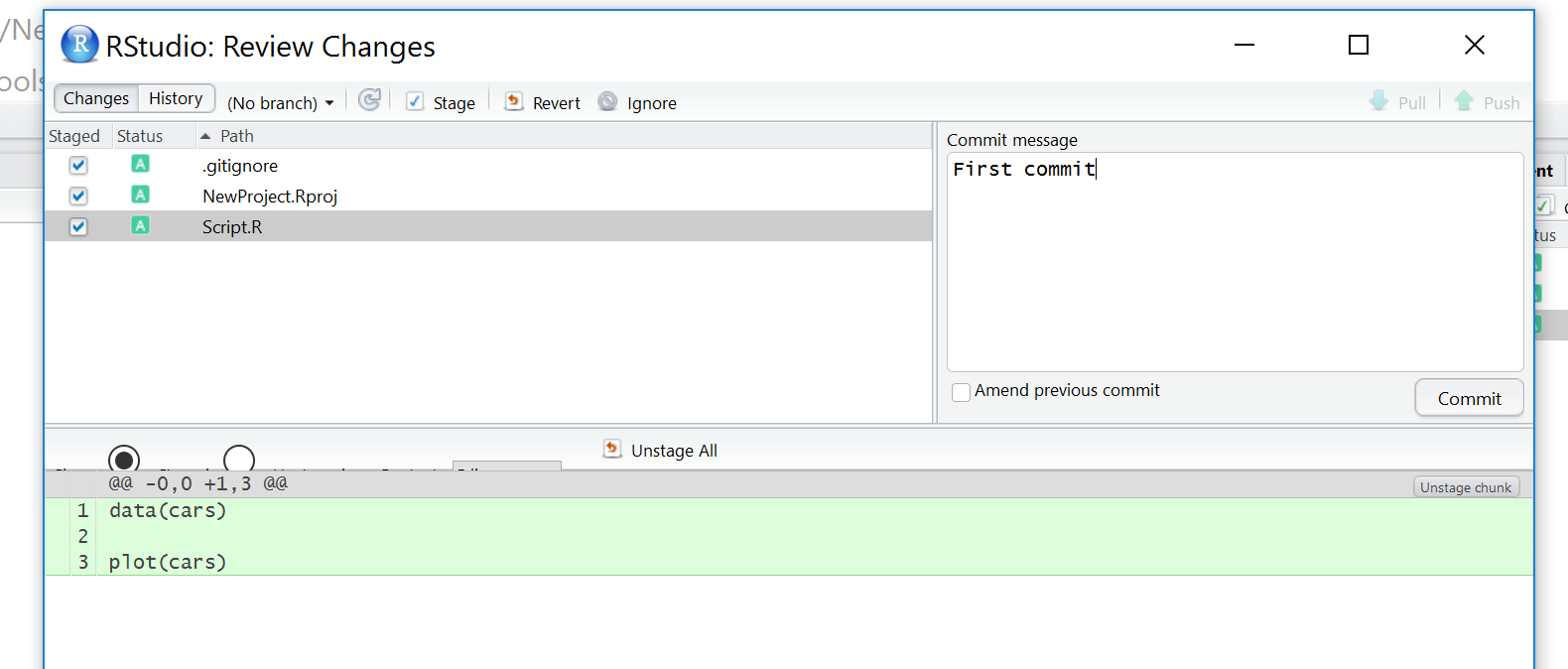
All of your files are now staged, ready for you to record those changes as a **commit**. Each commit creates a snapshot of the project directory, allowing you to record the state of your project through time.

Click on **Commit**.

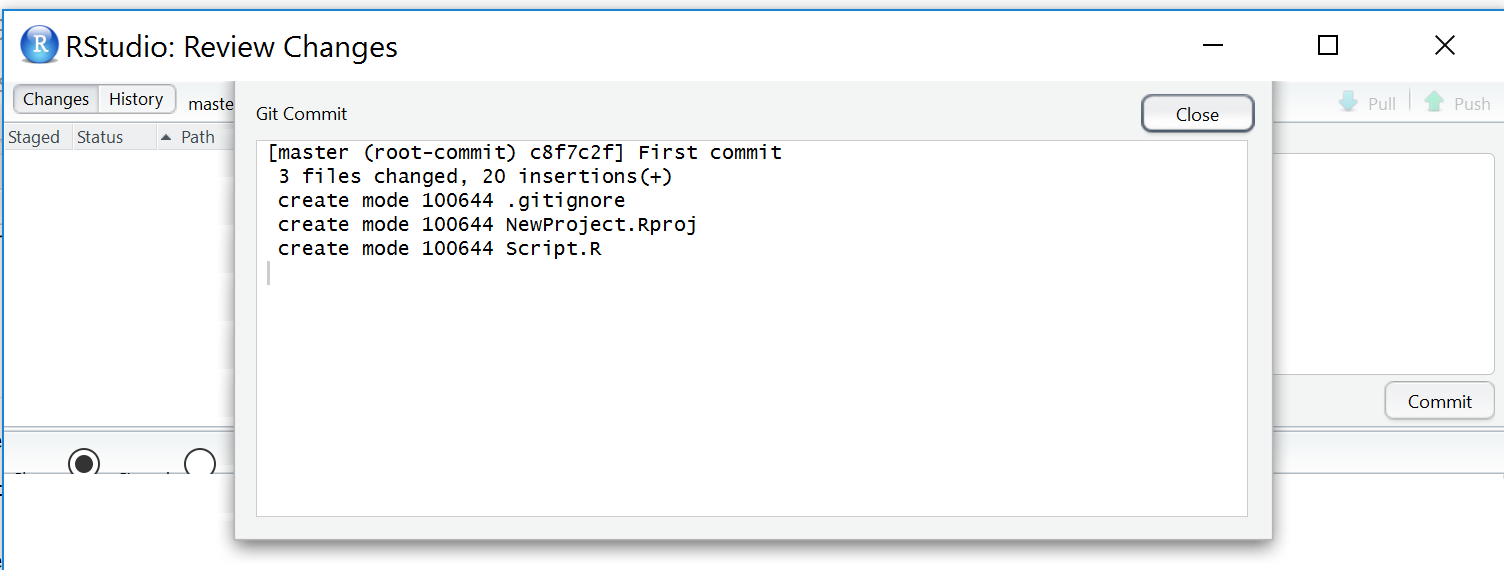


This will open a pane showing details of the commit. You need to specify a **commit message** in the right hand panel. Commit messages should explain **why you have made a change** and not *what* you have changed. Git keeps track of *what* has changed (see use of **Diff** below) and so it is best not to use commit messages for this purpose. The commit message lets your future self know the current state of your mind **why** you made this change or it will inform collaborators *why* the change was made.

Since this is the first commit, you can follow tradition and use a brief message such as **First commit**, but for subsequent commits it’s worth providing a more descriptive message. Once you’ve typed a commit message, click on **Commit**.



A message displaying the output from git will be shown.



### 2.2. Making changes

Now we want to make some changes to our R script. For example we could change the colour of the dots in the plots from black (the default) to red. Change the line in your script that makes to plot to:

plot(cars, col="red")

Save the script.

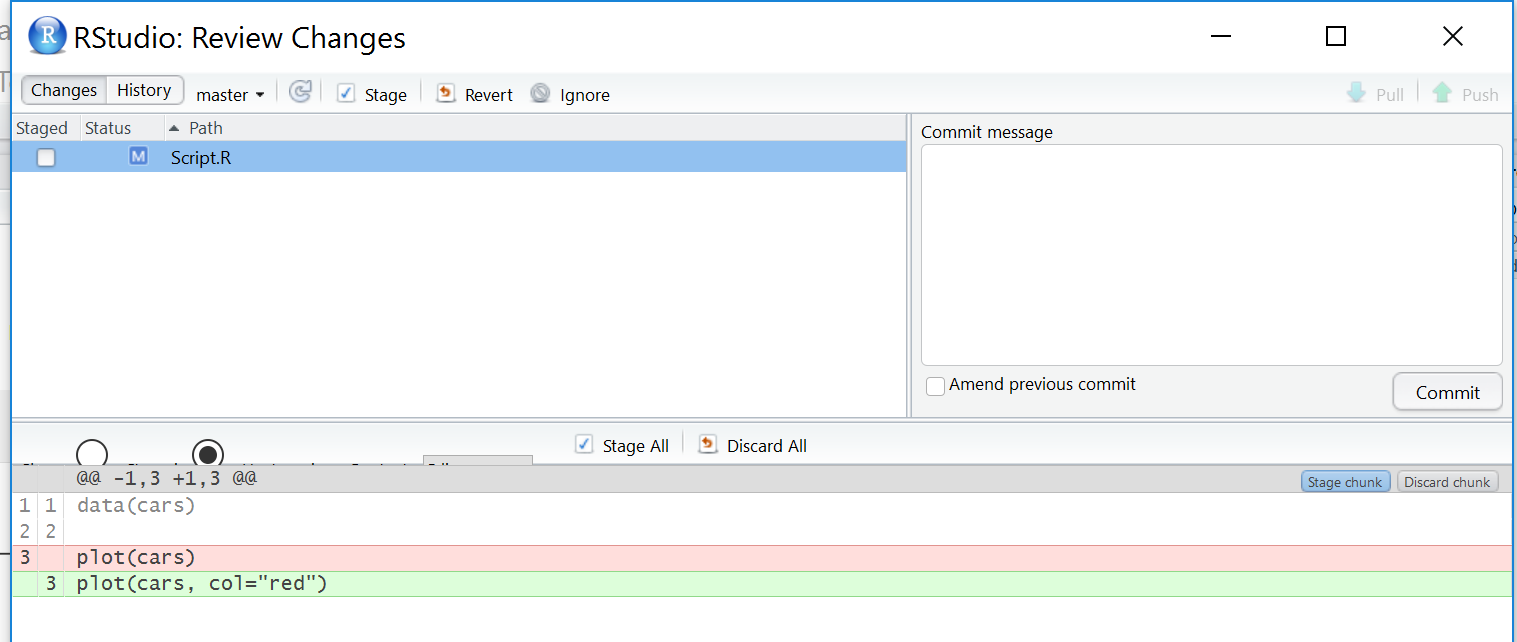
Now, click on the **Git** tab in RStudio. You’ll notice that there’s only one file listed, since RStudio only lists files that have changed in its git interface.





Click on the **Staged** checkbox to stage your change. Now click on **Diff** (next to **Commit**) to display a summary of what’s changed. Lines that have been added are green. Lines that have been removed are red.

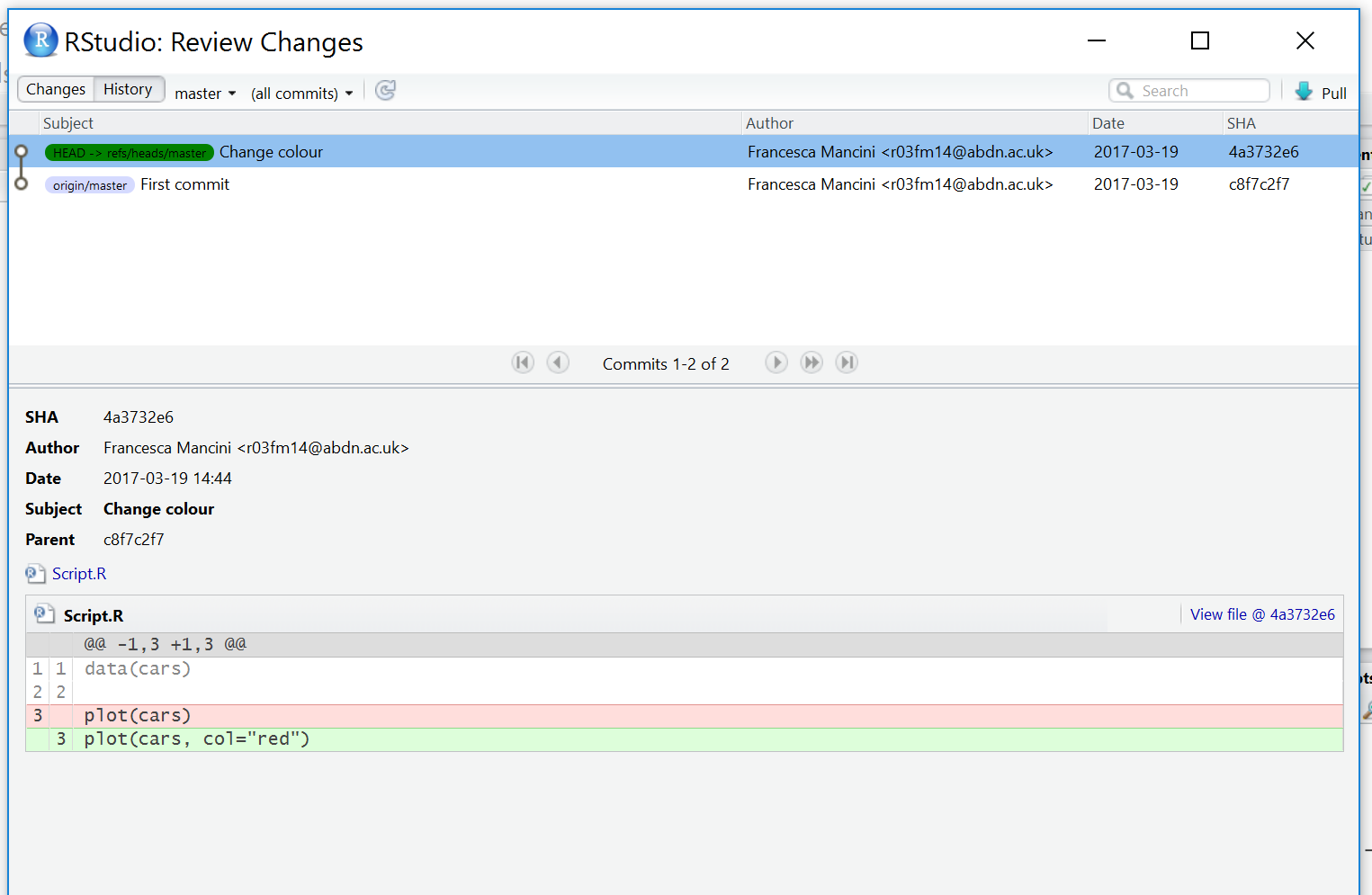
(Note that from git’s point of view, a modification to a line is actually two operations: the removal of the original line followed by the creation of a new line.)



Add a commit message and click on **Commit**.



In the **Git** tab of RStudio, click on **Diff** and then **History**. You’ll see that you can look through every commit you’ve ever made. This can be extremely useful for finding bugs or looking back to how your model looked a few months ago.

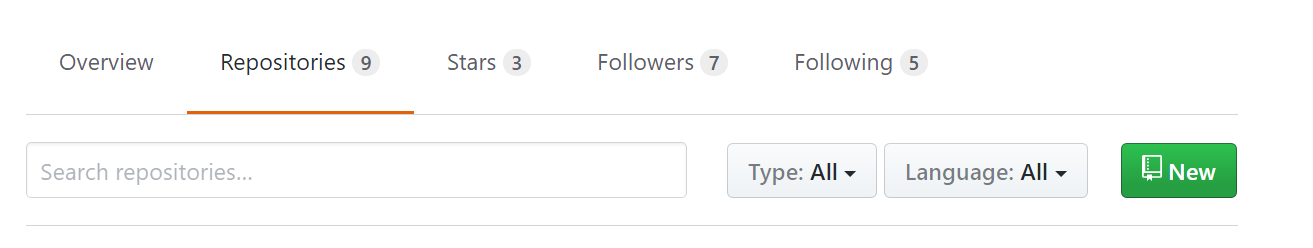


### 2.3. Setting up a remote repository on GitHub

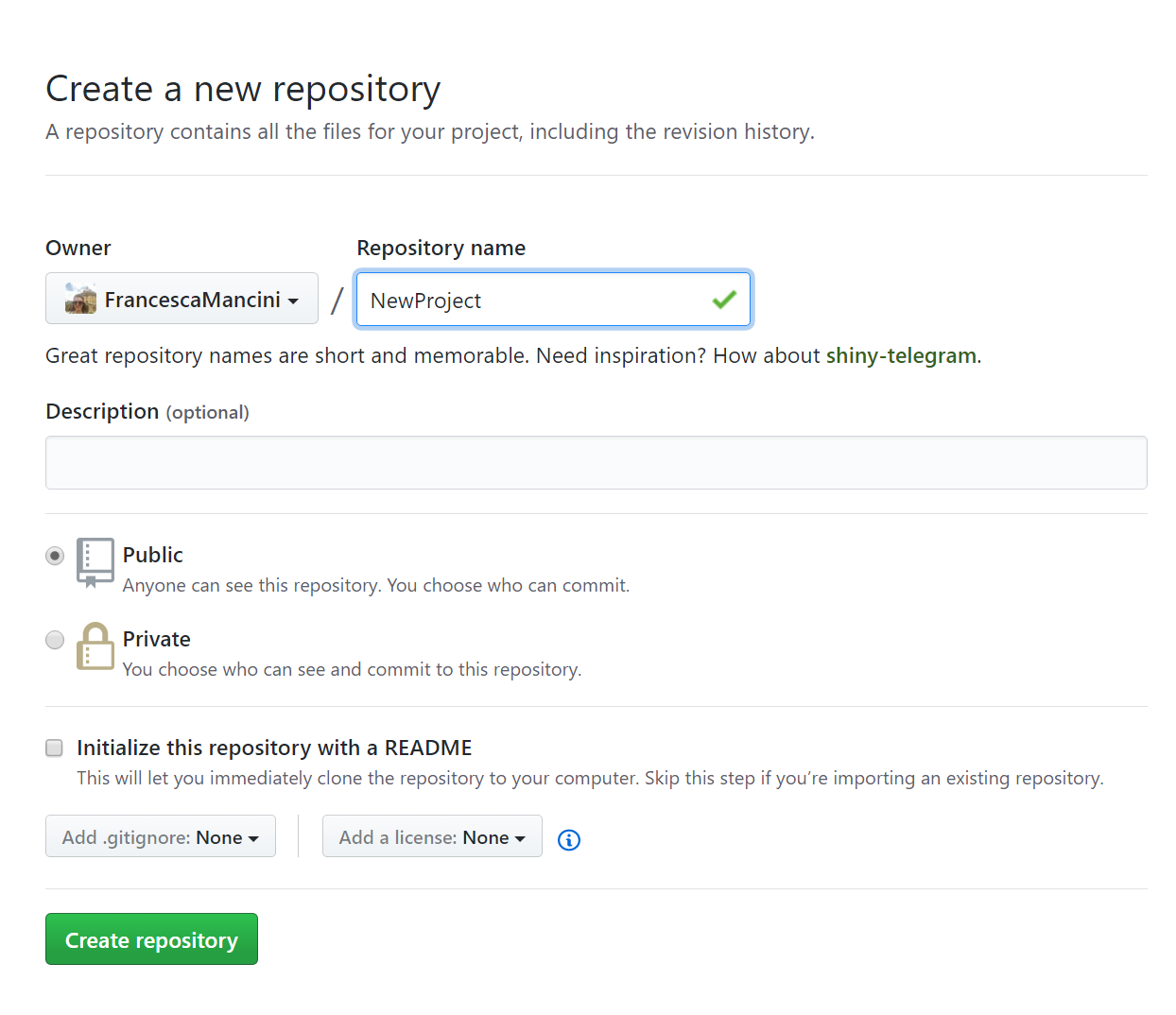
Your project is fully version controlled so you have access to a detailed history of every change you’ve ever made to it. This is a great first step but all of this only exists on your own computer at the moment.

Likely you’ll want to connect your local repository to a remote repository using a hosting service such as GitHub. Syncing your local repository to a remote means that your code is backed up. Also, the remote repository acts as a central store for your project, allowing you to share your work or to access it from another computer.

First, you need to create a GitHub repository to use as a remote. Log into GitHub and go to your profile page. On the **Repositories** tab, click **New**



At the **Create a new repository** screen, give your repository a name and click **Create Repository**

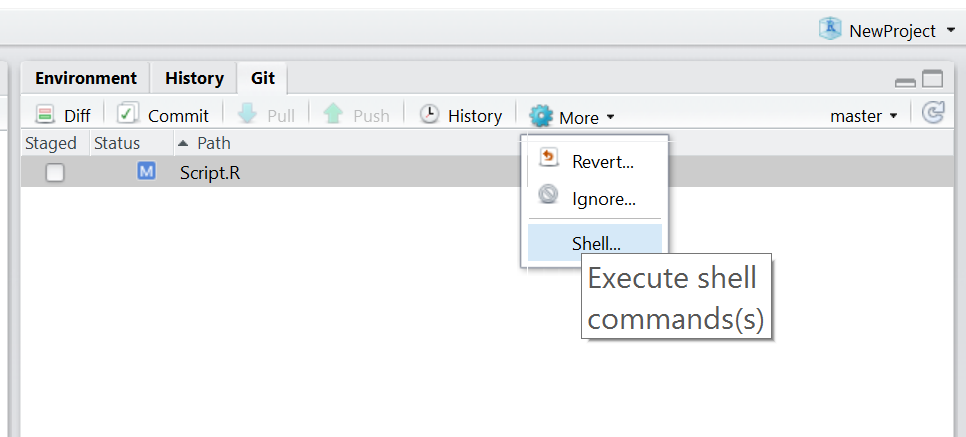


The next screen gives sets of git commands for use in various circumstances. Find the section labelled **…or push an existing repository from the command line** and copy the commands to the clipboard.

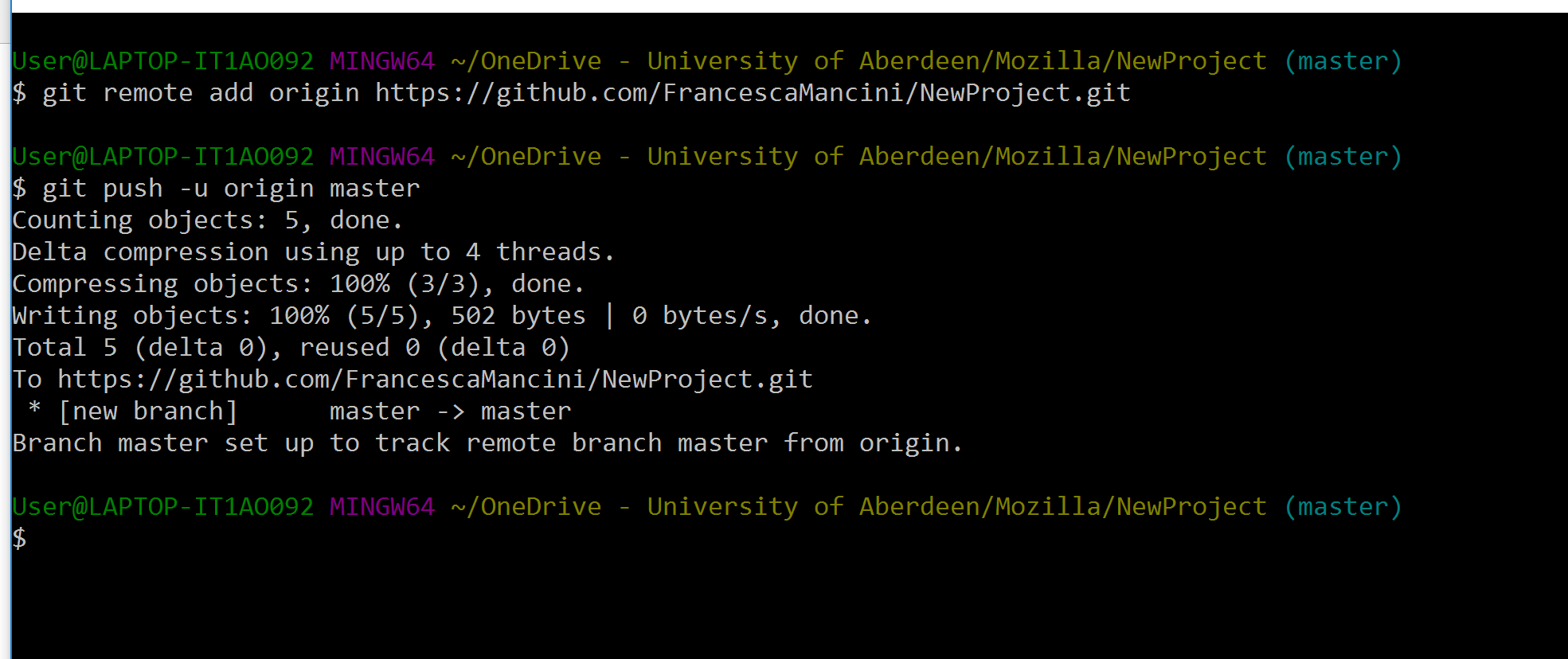


These commands tell git to set the remote repository for your local repository.

In RStudio, navigate to the **Git** tab and click on **More** -> **Shell…**.

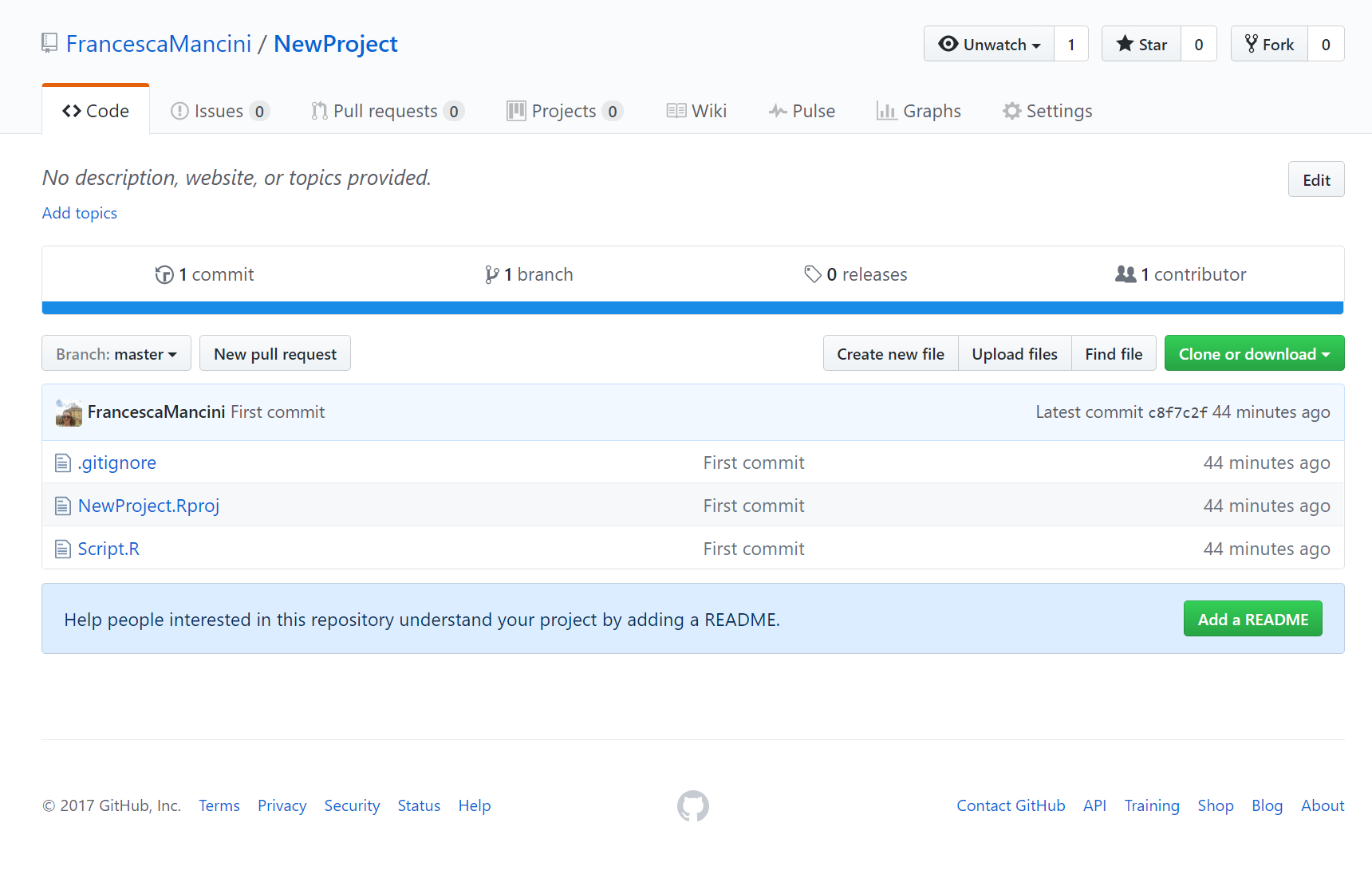


Paste the git commands into the shell and press **Enter** to execute them.



Confirm that the project has been uploaded to your GitHub repository.

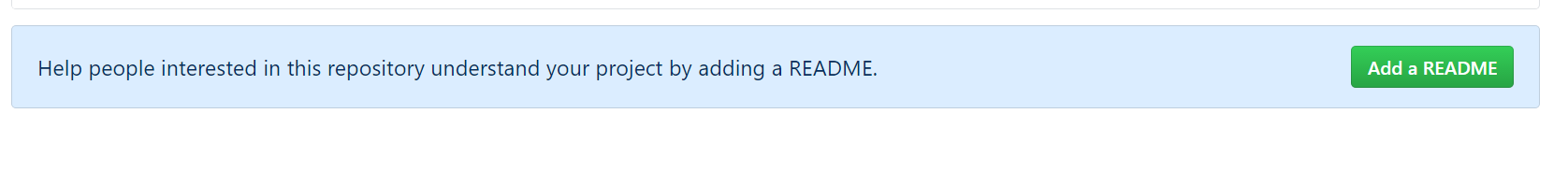
It should look like this:



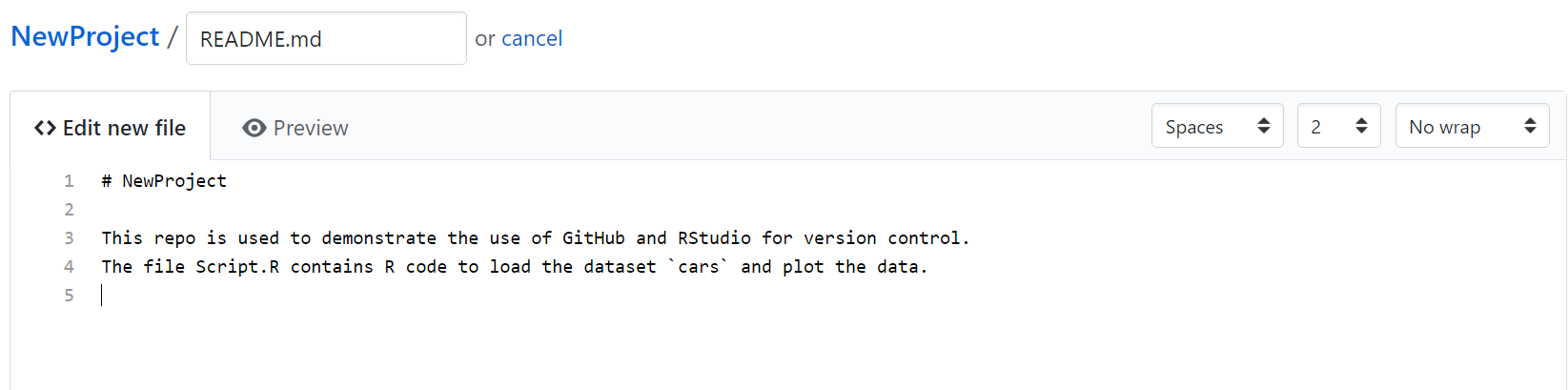
It is good practice to always add a **README.md** file to your repository. A README.md file describes your project and all the files that are in the repo. A good README.md file makes it easy for others (or the future you!) to use your code and reproduce your project.

You can create a README.md file from RSTudio or in GitHub. Today we will be using the second option.

From the page of your repository in GitHub click on **Add a README**.

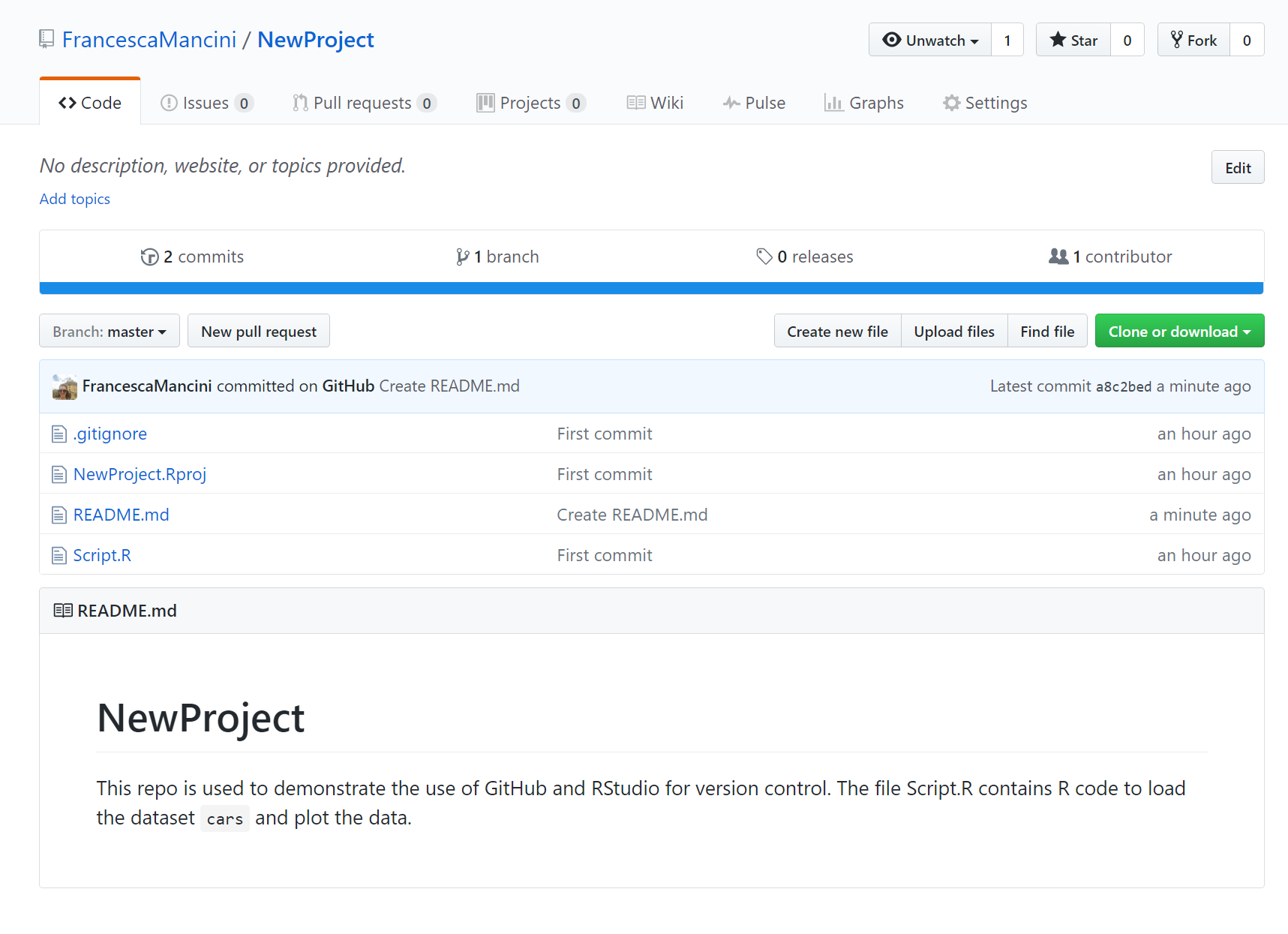


Now write a short description of your project, something like this.



At the bottom of the page, you can now change the commit message and add a more detailed description of your changes if you like. For the purpose of this exercise the commit message “Create README.md” is good enough. Click on **Commit new file**.

Your repo should now look like this:



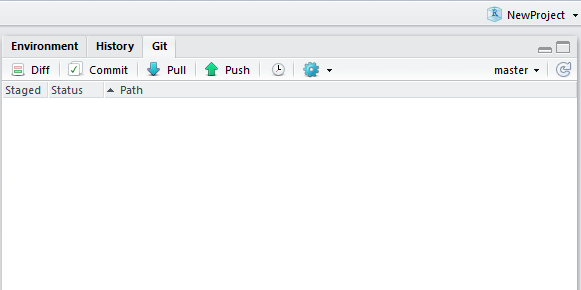
### 2.4. Subsequent workflow

Once a local repository has been associated with GitHub, it’s not necessary to use the command line for subsequent uploads. However, you will want to always keep the local (e.g., your personal computer) and remote (e.g., GitHub) in sync with one another. To do this you will **Pull** and **Push** changes from one place/repository to the next.

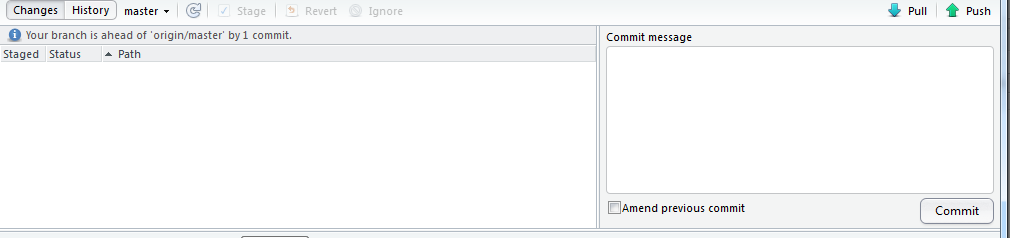
#### 2.4.1. Pull and push changes

When you start working on your local repository (e.g., files on your own computer), it is best to start by **Pulling** any new changes off of your remote repository (e.g., GitHub). Similarly, when you are finished working on them in your local repository (e.g., your computer) it is good to then **Push** them to the remote repository (e.g., GitHub). We will walk through this basic workflow that enables you to keep your local files in sync across multiple computers or when files may change because of someone else working on the project.

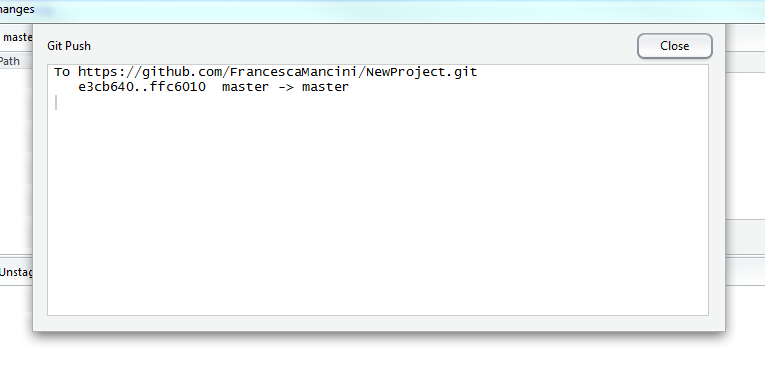
To start working on a project, **Pull** any new changes to update your local copy.



Now you can make changes on your own computer. When you are finished, to **Push** these changes to the remote repository (e.g., GitHub).



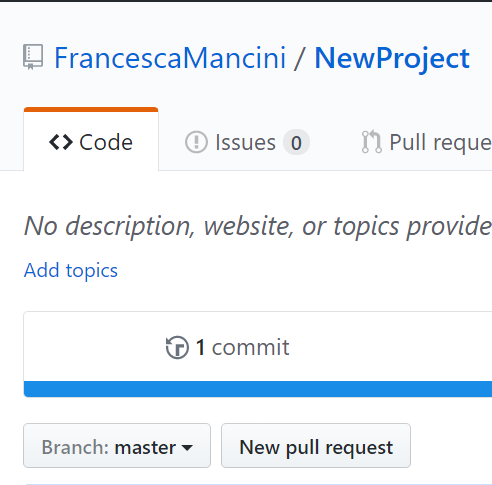
and you should see



#### 2.4.2. Create a new branch

Now we need to create a **Branch**. **Branching** is the way to work on different versions of a repository at one time.

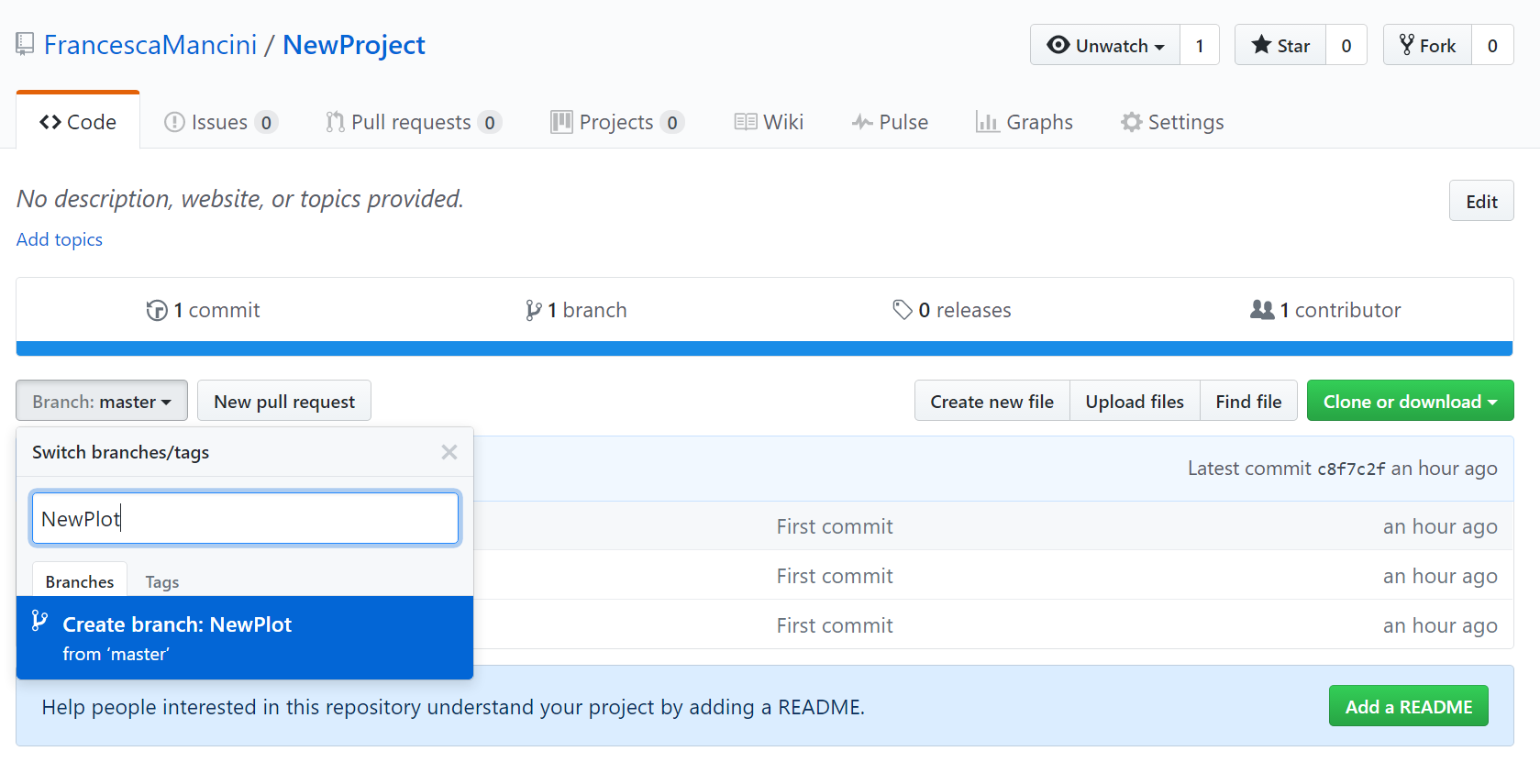
By default your repository has one branch named master which is considered to be the definitive branch.



We use branches to experiment and make edits before committing them to master.

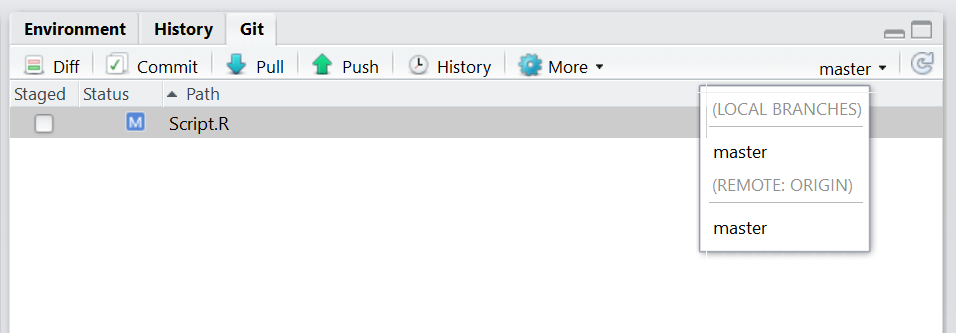
When you create a branch off the master branch, you’re making a copy, or snapshot, of master as it was at that point in time. This is particualrly useful when you are working on a collaborative project. For example, if someone else made changes to the master branch while you were working on your branch, you could pull in those updates.

Go to your repository and click the drop down at the top of the file list that says **branch: master**. Type a branch name into the new branch text box. Select the blue **Create branch** box or hit “Enter” on your keyboard.



Now you have two branches, master and NewPlot. They look exactly the same, but not for long! Next we’ll add our changes to the new branch.

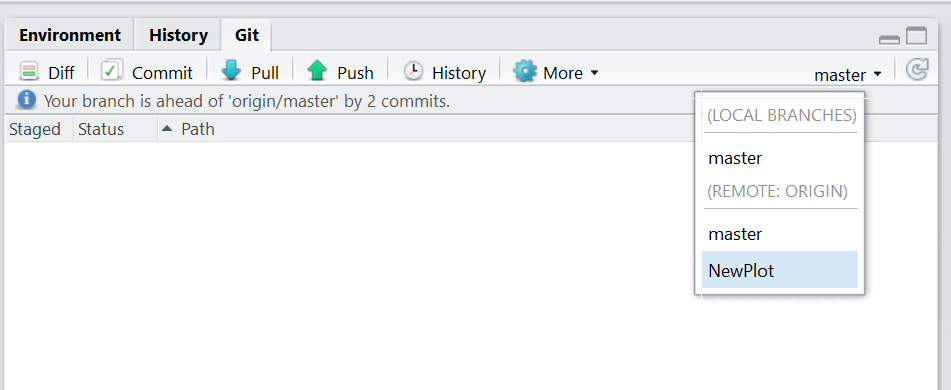
Go back to your project in RStudio. In the **Git** tab, if you hover the mouse pointer over the **master** drop menu you will see a tooltip that says **Switch branch**. However, if you click on **master** and open the drop menu you will notice that the only branch present is master.



This is because we need to **Pull** the changes that we made to our online repo into our local copy of the project on our computer. In the **Git** tab click on **Pull**. You will receive a message that tells you about all the changes that you have made in your repo, in this case adding a README.md file and creating a new branch.



If you click on **master** again you will now see the new branch that you created. Click on it to switch from master to NewPlot so we can continue working on the new branch.



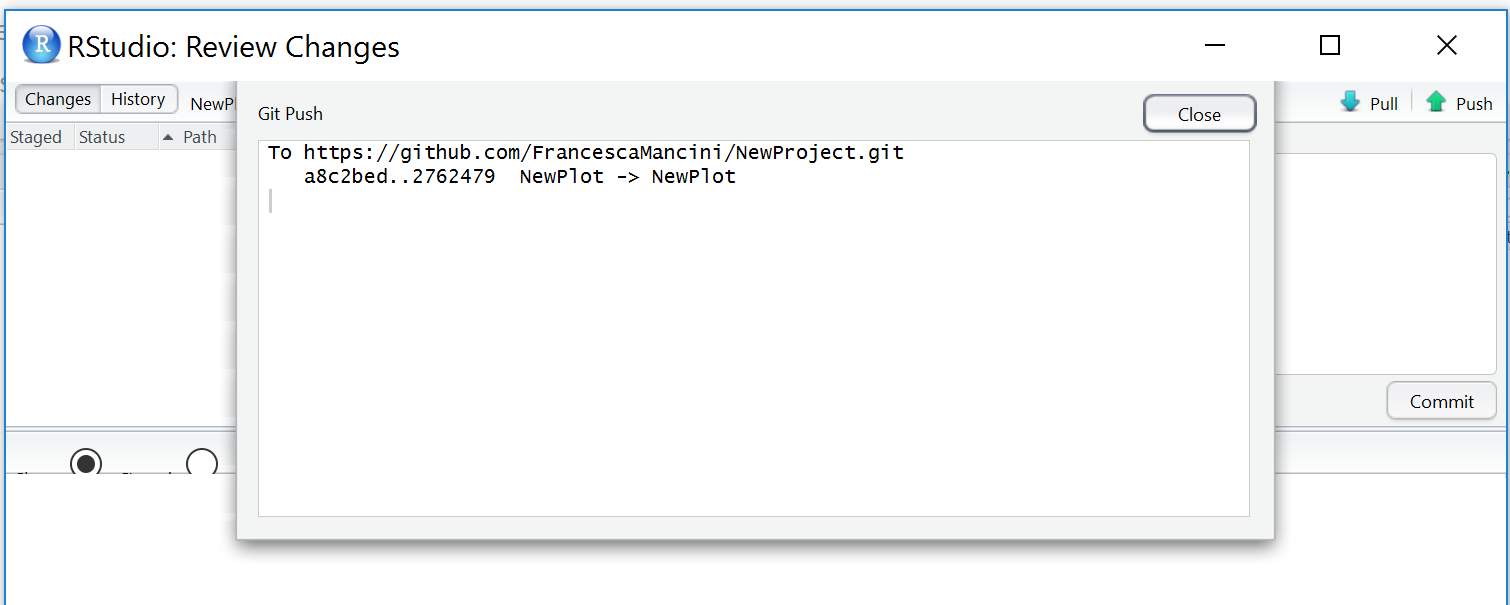
#### 2.4.3. Make and push changes to a branch

Now we can add a new plot to our script. Type the following code and then click **Run**.

boxplot(cars$speed)

Save your edited R script, click on the **Staged** box like you did before and click on **Commit**. In the **Review changes** page, write a commit message and click on **Commit**. Before you close the window, click on **Push**.

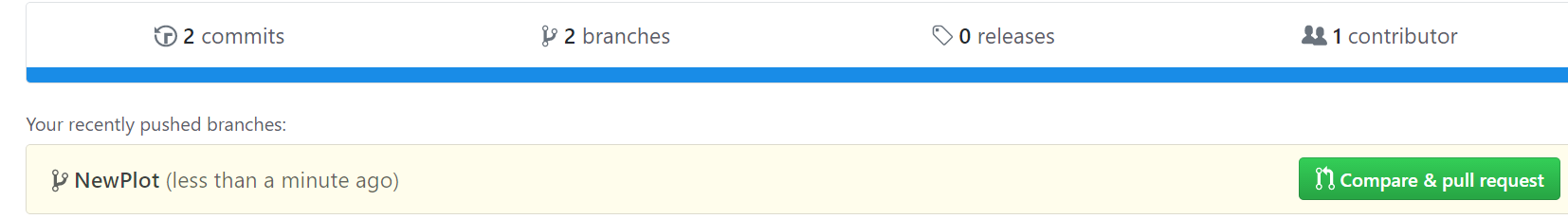




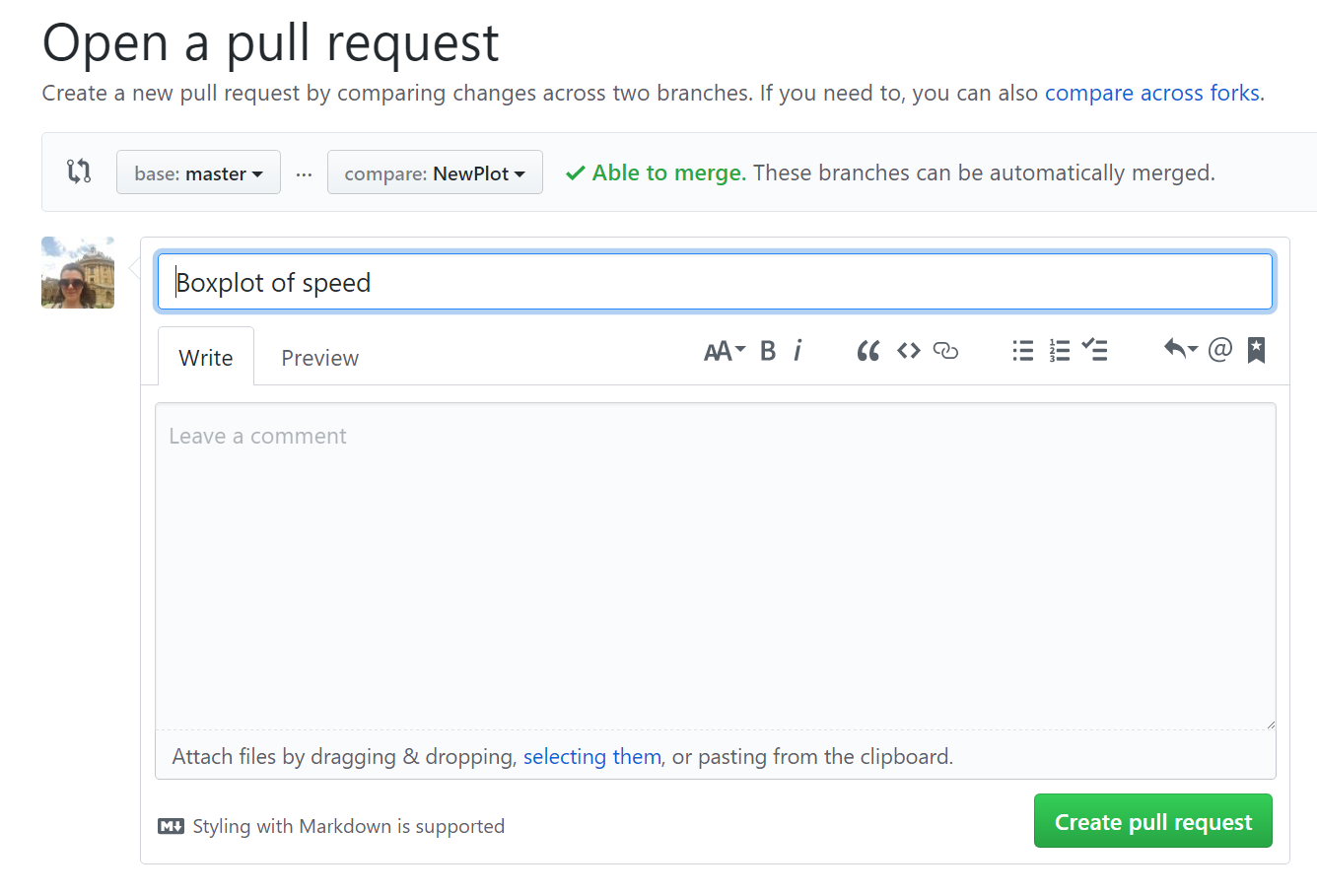
#### 2.4.4. Pull request and merge changes

Now go back to your repo on GitHub. You will notice that at the top of all you files there is a new section that says: **Your recently pushed branches:**. The changes you made to your script have been pushed to your online copy of the project and are now in the branch NewPlot. In order to merge them into your master branch we need to create a pull request.

Click on the button that says \*Compare and pull request\*\*.



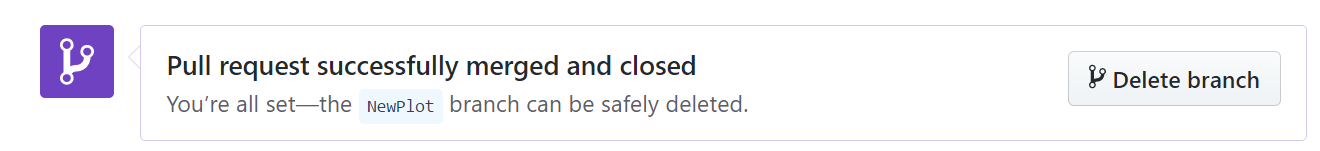
In the **Open a pull request** page click on **Create pull request**.



GitHub has now compared the two versions of your project that are in the two branches, master and NewPlot and it is telling us that there are no conflicts between the two versions.



Now it’s time to bring your changes together – merging your NewPlot branch into the master branch. Click the green **Merge pull request** button to merge the changes into master. Click **Confirm merge**. Go ahead and delete the branch, since its changes have been incorporated, with the **Delete branch** button in the purple box.



### 3. Summary

**Congratulations!**

You now know how to set up a version controlled project with RStudio and GitHub and your computer will never be full of v1.R, v2.R etc again!

You have seen that the standard workflow loop for version control in RStudio is very quick and simple:

* Create a new branch
* Make your change
* Stage your change
* Commit
* Push to GitHub
* Create a pull request
* Merge your changes

As well as this basic workflow, you’ve learned how to look at changes using Diff and to inspect your project’s history.

We hope you have enjoyed this tutorial and that you are now convinced that version control can make the management of your project a much easier task.

If you have any questions about completing this tutorial, please contact us on aberdeen-study-group@googlegroups.com.