Happy Git and GitHub for the useR

Jenny Bryan and the STAT 545 TAs

2016-07-09

Contents

What's going on here **9**

1 Why Git? Why GitHub? **11**

1.1 Why Git? . . . 11
1.2 Why GitHub? . . . 11
1.3 Is it going to hurt? . . . 12
1.4 What is the payoff? . . . 12
1.5 Who can do what? . . . 13
1.6 Special features of GitHub . . . 13
1.7 What's special about using R and with Git and GitHub? . . . 14
1.8 Audience and pre-reqs . . . 14
1.9 What this is NOT . . . 14

2 Contributors **15**

3 Workshops **17**

3.1 useR! 2016 Stanford . . . 17
3.2 CSAMA 2016: Statistical Data Analysis for Genome Biology . . . 18

I Installation **19**

4 Installation pain **21**

4.1 Success and operating systems . . . 21

5 Register a GitHub account **23**

5.1 Username advice . . . 23
5.2 Free private repos . . . 23
5.3 Pay for private repos . . . 24

6 Install or upgrade R and RStudio **25**

6.1 More about R and RStudio . . . 25

7 Install Git **27**

7.1 Git already installed? . . . 27

7.2 Windows . . . 27

7.3 Mac OS . . . 28

7.4 Linux . . . 29

8 Introduce yourself to Git **31**

8.1 More about `git config` . . . 31

9 Install a Git client **33**

9.1 What and why . . . 33

9.2 Recommended Git clients . . . 33

II Connect Git, GitHub, RStudio **35**

(PART) Connect Git, GitHub, RStudio **37**

10 Connect to GitHub **37**

10.1 Make a repo on GitHub . . . 37

10.2 Clone the repo to your local computer . . . 37

10.3 Make a local change, commit, and push . . . 38

10.4 Confirm the local change propagated to the GitHub remote . . . 39

10.5 Am I really going to type GitHub username and password on each push? . . . 39

10.6 Clean up . . . 40

11 Cache credentials for HTTPS **41**

11.1 Get a test repository . . . 41

11.2 Verify that your Git is new enough to have a credential helper . . . 42

11.3 Turn on the credential helper . . . 42

12 Set up keys for SSH **45**

12.1 SSH keys . . . 45

12.2 Check for existing keys . . . 45

12.3 Set up from RStudio . . . 45

12.4 Set up from the shell . . . 46

13 Connect RStudio to Git and GitHub **49**

13.1 Prerequisites . . . 49
13.2 Make a new repo on GitHub . . . 49
13.3 Clone the new GitHub repository to your computer via RStudio . . . 49
13.4 Make local changes, save, commit . . . 50
13.5 Push your local changes online to GitHub . . . 50
13.6 Confirm the local change propagated to the GitHub remote . . . 50
13.7 Were you challenged for GitHub username and password? . . . 51
13.8 Clean up . . . 51

14 Detect Git from RStudio **53**

14.1 Do you have a problem? . . . 53
14.2 Find Git yourself . . . 53
14.3 Tell RStudio where to find Git . . . 54

15 RStudio, Git, GitHub Hell **55**

15.1 I think I have installed Git but damn if I can find it . . . 55
15.2 Dysfunctional PATH . . . 55
15.3 Push/Pull buttons greyed out in RStudio . . . 56
15.4 I have no idea if my local repo and my remote repo are connected. . . . 56
15.5 Push fail at the RStudio level . . . 56
15.6 Push rejected, i.e. fail at the Git/GitHub level . . . 57
15.7 RStudio is not making certain files available for staging/committing . . . 57
15.8 I hear you have some Git repo inside your Git repo . . . 57

III Early Usage **59**

16 New project, GitHub first **61**

16.1 Make a repo on GitHub . . . 61
16.2 New RStudio Project via git clone . . . 61
16.3 Make local changes, save, commit . . . 62
16.4 Push your local changes to GitHub . . . 62
16.5 Confirm the local change propagated to the GitHub remote . . . 63
16.6 Make a change on GitHub . . . 63
16.7 Pull from GitHub . . . 63
16.8 The end . . . 63

17 Existing project, GitHub first **65**

17.1 Make a repo on GitHub . . . 65
17.2 New RStudio Project via git clone . . . 65
17.3 Bring your existing project over . . . 66
17.4 Stage and commit . . . 66
17.5 Push your local changes to GitHub . . . 66
17.6 Confirm the local change propagated to the GitHub remote . . . 66
17.7 The end . . . 67

18 Existing project, GitHub last **69**

18.1 Make or verify an RStudio Project . . . 69
18.2 Make or verify a Git repo . . . 69
18.3 Stage and commit . . . 69
18.4 Make a repo on GitHub . . . 70
18.5 Connect to GitHub . . . 70
18.6 Confirm the local change propagated to the GitHub remote . . . 70
18.7 The end . . . 71

19 Test drive R Markdown **73**

19.1 Hello World . . . 73
19.2 Push to GitHub . . . 74
19.3 Output format . . . 74
19.4 Push to GitHub . . . 75
19.5 Put your stamp on it . . . 75
19.6 Develop your report . . . 75
19.7 Publish your report . . . 76
19.8 HTML on GitHub . . . 76
19.9 Troubleshooting . . . 76

20 Render an R script **79**

21 Git commands **81**

IV Tutorial prompts **83**

22 Clone a repo **85**

23 Fork a repo **87**
23.1 Fork then clone . . . 87
23.2 Fork and pull . . . 87
23.3 Updating your fork . . . 87

24 Create a bingo card **91**

25 Burn it all down **93**

26 Resetting **95**

27 Search GitHub **97**
27.1 Basic resources . . . 97
27.2 Use case . . . 97

V More **99**

28 Notes **101**
28.1 Common workflow questions . . . 101
28.2 git stuff . . . 101
28.3 The repeated amend . . . 102
28.4 Disaster recovery . . . 102
28.5 Engage with R source on GitHub . . . 102
28.6 Workflow and psychology . . . 103
28.7 How the square bracket links work . . . 103

A The shell **105**
A.1 What is the shell? . . . 105
A.2 Starting the shell . . . 105
A.3 Using the shell . . . 105
A.4 Note for Windows users . . . 107

B Comic relief **109**

C References **111**

What's going on here

Still from Heaven King video

THIS IS AN EXPERIMENT!

Marshalling everything I have re: Git/GitHub + R/RStudio/Rmd in bookdown format, in anticipation of the tutorial at useR! 2016:

- repo that makes this site: https://github.com/jennybc/happy-git-with-r
- session in the useR! schedule
- full description on the main site
- slides for a related talk I've given a couple times

Chapter 1

Why Git? Why GitHub?

Why would a data analyst use hosted version control?

1.1 Why Git?

Git is a **version control system**. Its 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** – in a sane, highly structured way. If you have no idea what I'm talking about, think of it as the "Track Changes" features from Microsoft Word on steroids.

Git has been re-purposed by the data science community. In addition to using it for source code, we use it to manage the motley collection of files that make up typical data analytical projects, which often consist of data, figures, reports, and, yes, source code.

A solo data analyst, working on a single computer, will benefit from adopting version control. But not nearly enough to justify the pain of installation and workflow upheaval. There are much easier ways ways to get versioned back ups of your files, if that's all you're worried about.

In my opinion, **for new users**, the pros of Git only outweigh the cons when you factor in the overhead of communicating and collaborating with other people. Who among us does not need to do that? Your life is much easier if this is baked into your workflow, as opposed to being a separate process that you dread or neglect.

1.2 Why GitHub?

This is where hosting services like GitHub, Bitbucket, and GitLab come in. They provide a home for your Git-based projects on the internet. If you have no idea what I'm talking about, think of it as DropBox but much, much better. The remote host acts as a distribution channel or clearinghouse for your Git-managed project. It allows other people to see your stuff, sync up with you, and perhaps even make changes. These hosting providers improve upon traditional Unix Git servers with well-designed web-based interfaces.

Even for private solo projects, it's a good idea to push your work to a remote location for peace of mind. Why? Because it's fairly easy to screw up your local Git repository, especially when you're new at this. The good news is that often only the Git infrastructure is borked up. Your files are just fine! Which makes your Git pickle all the more frustrating. There are official Git solutions to these problems, but they might require expertise and patience you can't access at 3a.m. If you've recently pushed your work to GitHub, it's easy to grab a fresh copy, patch things up with the changes that only exist locally, and get on with your life.

We target GitHub – not Bitbucket or GitLab – for the sake of specificity. However, all the big-picture principles and even some mechanics will carry over to these alternative hosting platforms.

Don't get too caught up on public versus private at this point. There are many ways to get private repositories from the major providers for low or no cost. Just get started and figure out if and how Git/GitHub is going to work for you! If you outgrow this arrangement, you can throw some combination of technical savvy and money at the problem. You can either pay for a higher level of service or self-host one of these platforms.

1.3 Is it going to hurt?

You have to install Git, get local Git talking to GitHub, and make sure RStudio can talk to local Git (and, therefore, GitHub). This is one-time or once-per-computer pain.

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? 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 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.

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.

1.4 What is the payoff?

If someone needs to see your work or if you want them to try out your code, they can easily get it from GitHub. If they use Git, they can clone or fork your repository. If they don't use Git, they can still browse your project on GitHub like a normal website and even grab everything by downloading a zip archive.

If you need to collaborate on data analysis or code development, then all involved should use Git. Use GitHub as your clearinghouse: individuals work independently, then send work back to GitHub for reconciliation and transmission to the rest of the team.

If you care deeply about someone else's project, such as an R package you use heavily, you can track its development on GitHub. You can watch the repository to get notified of major activity. You can fork it to keep your own copy. You can modify your fork to add features or fix bugs and send them back to the owner as a proposed change.

The advantage of Git/GitHub is highlighted by comparing these two ways of collaborating on a document:

- **Edit, save, attach.** In this workflow, everyone has one (or more!) copies of the document and they circulate via email attachment. Which one is "master"? Is it even possible to say? How do different versions relate to each other? How should versions be reconciled? If you want to see the current best version, how do you get it? All of this usually gets sorted out by social contract and a fairly manual process.
- **Google Doc.** In this workflow, there is only one copy of the document and it lives in the cloud. Anyone can access the most recent version on demand. Anyone can edit or comment or propose a change and this is immediately available to everyone else. Anyone can see who's been editing the document and, if disaster strikes, can revert to a previous version. A great deal of ambiguity and annoying reconciliation work has been designed away.

Managing a project via Git/GitHub is much more like the Google Doc scenario and enjoys many of the same advantages. It is definitely more complicated than collaborating on a Google Doc, but this puts you in the right mindset.

1.5 Who can do what?

A public repository is readable by the world. The owner can grant higher levels of permission to others, such as the ability to push commits.

A private repository is invisible to the world. The owner can grant read, write (push), or admin access to others.

There is also a formal notion of an organization, which can be useful for managing repository permissions for entire teams of people.

1.6 Special features of GitHub

this is perhaps too detailed … full stop? or does it belong elsewhere?

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.

1.7 What's special about using R and with Git and GitHub?

- the active R package development community on GitHub
- workflows for R scripts and R Markdown files that make it easy to share source and rendered results on GitHub
- Git- and GitHub-related features of the RStudio IDE

1.8 Audience and pre-reqs

The target audience for this site is someone who analyzes data, probably with R, though much of the content may be useful to analysts using other languages. While R package development with Git(Hub) is absolutely in scope, it's not an explicit focus or requirement.

The site is aimed at intermediate to advanced R users, who are comfortable writing R scripts and managing R projects. You should have a good grasp of files and directories and be generally knowledgeable about where things live on your computer.

Although we will show alternatives for most Git operations, we will inevitably spend some time in the shell and we assume some prior experience. For example, you should know how to open up a shell, navigate to a certain directory, and list the files there. You should be comfortable using shell commands to view/move/rename files and to work with your command history.

1.9 What this is NOT

We aim to teach novices about Git on a strict "need to know" basis. Git was built to manage development of the Linux kernel, which is probably very different from what you do. Most people need a small subset of Git's functionality and that will be our focus. If you want a full-blown exposition of Git as a directed acyclic graph or a treatise on the Git-Flow branching strategy, you will be sad.

Chapter 2

Contributors

Jenny Bryan (twitter, GitHub) is a professor at the University of British Columbia. She's been using and teaching R (or S!) for 20 years, most recently in STAT 545 and Software Carpentry. Other aspects of her R life include work with rOpenSci, development of the `googlesheets` and `gapminder` packages, and being academic director for UBC's Master of Data Science.

The development and delivery of this material has benefited greatly from contributions by:

- Dean Attali (blog, GitHub, twitter) who recently earned his M.Sc. Bioinformatics at UBC. He's the developer of the shinyjs package and much more.
- Bernhard Konrad (GitHub, twitter) recently earned his Ph.D. in Applied Math at UBC, completed an Insight Data Science Fellowship, and is a software engineer at Google.
- Shaun Jackman (GitHub, twitter, sjackman.ca) is a Ph.D. in bioinformatics at UBC, the lead developer of Linuxbrew and a developer of Homebrew-Science and the genome sequence assembly software ABySS
- The STAT 545 TA group

Chapter 3

Workshops

These materials can be used for independent study, but I also use them to support

- in-person workshops (see below)
- STAT 545 at UBC
- UBC Master of Data Science

3.1 useR! 2016 Stanford

3.1.1 Logistics

Monday, June 27, 2016
8:15am - 9:00 TAs available to help complete system set up
9:00am - 12:00pm Workshop, with coffee break 10:15am - 10:30am
Lyons & Lodato 326 Galvez Street Stanford, CA 94305-6105 Google Maps
Sessions in the useR! schedule website: part 1, part 2
Full description on the main useR! website

3.1.2 Pre-tutorial set-up

It is vital that you attempt to set up your system in advance. You cannot show up at 9am with no preparation and keep up!

These are battle-tested instructions, so most will succeed. We believe in you! If you have trouble, you can open an issue here and we *might* be able to help in the days leading up to useR! (no promises). We will have TAs in the room starting at 8:15am and throughout the workshop.

Try this. Give it about 1 - 2 hours. If you hit a wall, we will help:

- Register a free GitHub account (chapter 5).
- Install or update R and RStudio (chapter 6).
- Install Git (chapter 7).
- Introduce yourself to Git (chapter 8).
- Prove local Git can talk to GitHub (chapter 10).
- Cache your username and password (chapter 11) or set up SSH keys (chapter 12.1) so you don't need to authenticate yourself to GitHub interactively *ad nauseum*.
- Prove RStudio can find local Git and, therefore, can talk to GitHub (chapter 13).

  - FYI: this is where our hands-on activities will start. We walk through a similar activity together, with narrative, and build from there.

Troubleshooting:

- If RStudio is having a hard time finding Git, see chapter 14.
- For all manner of problems, both installation and usage related, see chapter 19.9.

Optional reading on "big picture stuff":

- Read "Why Git? Why GitHub?" in chapter 1.

3.1.3 What we can do together

- Verify and complete the most difficult part: installation and configuration!
- Create a Git repository and connect the local repo to a GitHub remote, for new and existing projects.
- Run R code, via R Markdown or a script, and share a presentable report via GitHub.
- The intersection of GitHub and the R world:
  - R packages developed on GitHub and how to make use of Issues
  - METACRAN read-only mirror of all of CRAN + R-specific searching tips.
- Propose a change to someone else's project, i.e. "make a pull request".
- Discuss daily workflow:
  - How often should I commit? Which files should I commit?
  - Data files and the dilemmas they present.
  - Most common Git predicaments. How to avoid and recover.
  - How do groups of 1, 5, or 10 people structure their work with Git(Hub)?

3.2 CSAMA 2016: Statistical Data Analysis for Genome Biology

3.2.1 Logistics

http://www.huber.embl.de/csama2016/
July 10 - 15, 2016, Bressanone-Brixen, Italy
Monday July 11, Lab 2, 15:30 - 17:00 Reproducible research and R authoring with markdown and knitr
Thursday July 14, Lab 4, 14:00 - 15:30 Use of Git and GitHub with R, RStudio, and R Markdown

Part I

Installation

Chapter 4

Installation pain

Getting all the necessary software installed, configured, and playing nicely together is honestly half the battle here. Brace yourself for some pain. The upside is that you can give yourself a pat on the back once you get through this. And you WILL get through this.

You will find far more resources for how to *use Git* than for installation and configuration. Why? The experts …

- Have been doing this for years. It's simply not hard for them anymore.
- Probably use some flavor of Unix. They may secretly (or not so secretly) take pride in neither using nor knowing Windows.
- Get more satisfaction and reward for thinking and writing about Git concepts and workflows than Git installation.

In their defense, it's hard to write installation instructions. Failures can be specific to an individual OS or even individual computer. If you have some new problem and, especially, the corresponding solution, we'd love to hear from you!

4.1 Success and operating systems

Our installation instructions have been forged in the fires of STAT 545, STAT 540, and assorted workshops, over several years. We regularly hear from grateful souls on the internet who also have success.

Here's some recent data on the subset of STAT 545 students for which we recorded the operating system: half Mac, just under half Windows (various flavours), and a dash of Linux.

	2014	2015
Mac	16 (41%)	38 (52%)
Windows 7	9 (23%)	13 (18%)
Windows 8	12 (31%)	9 (12%)
Windows 10	0 (0%)	8 (11%)
Linux	2 (5%)	5 (7%)

Chapter 5

Register a GitHub account

Register an account with GitHub. It's free!

- https://github.com

5.1 Username advice

You will be able 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.
- 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` because my vector of GitHub usernames contained missing values. A variant of Little Bobby Tables.

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/

5.2 Free private repos

GitHub offers free unlimited private repositories for users and organizations in education, academic research, nonprofits, and charities.

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.
  - 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

5.3 Pay for private repos

Everyone else can pay for some private repos. A personal plan with unlimited private repos is $7 / month at the time of writing. See the current plans and pricing here:

- https://github.com/pricing

Chapter 6

Install or upgrade R and RStudio

Install pre-compiled binary of R for your OS: https://cloud.r-project.org

Install Preview version RStudio Desktop: https://www.rstudio.com/products/rstudio/download/preview/

Update your R packages: `update.packages(ask = FALSE, checkBuilt = TRUE)`

Read on for more detail or hand-holding.

6.1 More about 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.

For more guidance on installing or upgrading R and RStudio, go here:

STAT 545 page on Installing R and RStudio

Chapter 7

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.

7.1 Git already installed?

Go to the shell (Appendix A) and enter `which git` and `git --version`:

```
jenny@2015-mbp happy-git-with-r $ which git
/usr/bin/git

jenny@2015-mbp happy-git-with-r $ git --version
git version 2.7.4 (Apple Git-66)
```

If Git reports a path to an executable and a version, 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.

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

7.2 Windows

Option 1 (*recommended*): Install Git for Windows, previously 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. Use all the default settings during installation.

- This approach 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 Bash shell will be useful as you venture outside of R/RStudio.
- This also leaves you with a Git client, though not a very good one. So check out Git clients we recommend (chapter 9).

Option 2 (*NOT recommended*): The GitHub hosting site offers GitHub Desktop for Windows that provides Git itself, a client, and smooth integration with GitHub.

- Their Windows set-up instructions recommend this method of Git installation.
- Why don't we like it? We've seen GitHub Desktop for Windows lead to Git installation in suboptimal locations, such as `~/AppData/Local`, and in other places we could never find. If you were **only** going to interact with GitHub via this app, maybe that's OK, but that does not apply to you. Therefore, we recommend option 1 instead.

7.3 Mac OS

Option 1 (*highly recommended*): Install the Xcode command line tools (not all of Xcode), which includes Git.

Go to the shell and enter one of these commands to elicit an offer to install developer command line tools:

```
git --version
git config
```

Accept the offer! Click on "Install".

Another way to request this is via `xcode-select --install`. We just happen to find this Git-based trigger apropos.

Option 2 (*recommended*): Install Git from here: http://git-scm.com/downloads.

- This arguably sets you up the best for the future. It will certainly get you the latest version of Git of all approaches described here.
- The GitHub home for this project is here: https://github.com/timcharper/git_osx_installer.
  - At that link, there is a list of maintained builds for various combinations of Git and Mac OS version. If you're running 10.7 Lion and struggling, we've had success in September 2015 with binaries found here: https://www.wandisco.com/git/download.

Option 3 (*recommended*): If you anticipate getting heavily into scientific computing, you're going to be installing and updating lots of software. You should check out homebrew, "the missing package manager for OS X". Among many other things, it can install Git for you. Once you have Homebrew installed, do this in the shell:

```
brew install git
```

Option 4 (*NOT recommended*): The GitHub hosting site offers GitHub Desktop for Mac that provides *the option* to install Git itself, a client, and smooth integration with GitHub..

- Their Mac set-up instructions recommend this method of Git installation.
- We don't like GitHub Desktop as a Git client, so this is a very cumbersome way to install Git. Consider this option a last resort.

7.4 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 8

Introduce yourself to Git

In the shell (chapter A):

```
git config --global user.name 'Jennifer Bryan'
git config --global user.email 'jenny@stat.ubc.ca'
git config --global --list
```

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

8.1 More about `git config`

From RStudio, go to *Tools* > *Shell* and tell `git` your name and **GitHub email** by typing (use your own name and email):

- `git config --global user.name 'Jennifer Bryan'`
  - This does **NOT** have to be your GitHub username, although it can be. Another good option is your actual first name and last name. Your commits will be labelled with this name, so this should be informative to potential collaborators.
- `git config --global user.email 'jenny@stat.ubc.ca'`
  - This **must** be the email that you used to sign up for GitHub.

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

Chapter 9

Install a Git client

This is optional but **highly recommended**.

Unless specified, it is not required for live workshops and will not be explicitly taught, though you might see us using one of these clients.

9.1 What and why

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 they also help you interact with GitHub or other remotes.

Git and your Git client are not the same thing, just like R and RStudio are not the same thing. A Git client and the RStudio IDE are not necessary to use to Git or R, respectively, but they make the experience more pleasant because they reduce the amount of command line bullshittery.

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

Fair warning: for some things, you will have to use the command line. But the more powerful your Git client is, the less often this happens.

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.*

9.2 Recommended Git clients

- SourceTree is a free, powerful Git(Hub) client that I highly recommend. It was my first Git client and is still my favorite for nontrivial Git tasks. Available for Mac and Windows. If I'm teaching you in a course or workshop, you might see me using this.
- GitKraken is quite new on the scene and is free. I would probably start here if I was starting today. Why? Because it works across all three OSes my students use: Windows, Mac, and Linux. I hear very good reviews, especially from long-suffering Linux users who haven't had any great options until now.

- GitUp is a free, open source client for Mac OS. I've heard really 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'm so used to SourceTree. YMMV.
- GitHub also offers a free Git(Hub) client for Windows and Mac. We do NOT recommend it for Windows and have serious reservations even for Mac OS. What do we object to?
  - The degree of hand-holding offered by GitHub's clients borders on hand-*cuffs*.
  - The Windows client sometimes leaves the Git executable so well hidden that we can't find it. This is a deal-killer, because that means RStudio can't find it either.
  - These clients wrap Git functionality so thoroughly that we've had students make some destructive mistakes. For example, we've seen a "sync" operation that resulted in the loss of local uncommitted changes. Exactly which Git operations, in what order, are implied by "sync", is not entirely clear. We prefer clients that expose Git more explicitly.
  - We've heard similar negative reviews from other instructors. It's not just us.
- Others that I have heard positive reviews for:
  - SmartGit
  - git-cola
  - magit, for Emacs nerds
- Browse even more Git(Hub) clients.

Part II

Connect Git, GitHub, RStudio

Chapter 10

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 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.

10.1 Make a repo on GitHub

Go to 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, we will delete this)
Public
YES Initialize this repository with a README

Click big green button "Create repository."

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

10.2 Clone the repo to your local computer

Go to the shell.

Take charge of – or at least notice! – what directory you're in. `pwd` to display working directory. `cd` to move around. 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 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
less 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 $ less 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)
```

10.3 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")
```

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
```

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
```

10.4 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 13). But first ...

10.5 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. Do one of the follow to eliminate this annoyance:

- Credential caching for HTTPS access, chapter 11.
- Set up SSH keys, chapter 12.1.

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

10.6 Clean up

Local When you're ready to clean up, you 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 11

Cache credentials for HTTPS

If you plan to push/pull using HTTPS, you want Git to cache your credentials (username, password) (or you should set up SSH keys, chapter 12.1), so you don't need to enter them over and over again. You'll need to set this up on each computer you want to connect to GitHub from.

I do not explain all the shell and Git commands in detail. This is a black box diagnostic / configuration exercise.

11.1 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 10) from your local computer, that test repo will be perfect.

If you have just verified that you can work with GitHub from RStudio (chapter 13), 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 local is tracking remote. In a shell with 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 has your GitHub master branch (`origin/master`) as upstream remote. Gibberish? Just check that your output looks similar to mine:

```
master b8e03e3 [origin/master] line added locally
```

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

In a shell, do:

```
git --version
```

and verify your version is 1.7.10 or newer. If not, update Git (chapter 7) or use SSH keys (chapter 12.1).

11.3 Turn on the credential helper

11.3.0.1 Windows

In the shell, enter:

```
git config --global credential.helper wincred
```

11.3.0.2 Windows, plan B

If that doesn't seem to work, install an external credential helper.

- Download the git-credential-winstore.exe application.
- Run it! It should work if Git is in your `PATH` environment variable. If not, go to the directory where you downloaded the application and run the following:

  ```
  git-credential-winstore -i "C:\Program Files (x86)\Git\bin\git.exe"
  ```

11.3.0.3 Mac

Find out if the credential helper is already installed. In the shell, enter:

```
git credential-osxkeychain
```

And look for this output:

```
usage: git credential-osxkeychain <get|store|erase>
```

If you don't get this output, it means you need a more recent version of Git, either via command line developer tools or Homebrew. Go back to the Mac section of chapter (7).

Once you've confirmed you have the credential helper, enter:

```
git config --global credential.helper osxkeychain
```

11.3.0.4 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.

11.3.1 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 12

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 11), so you don't have to authenticate yourself interactively with GitHub over and over again. You'll need to set this up on each computer you want to connect to GitHub from.

12.1 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 string of characters: a public and a private key. You can place the public key on any server, and then unlock it by connecting to it with a client that already has the private key. 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 and Digital Ocean.

12.2 Check for existing keys

Go to the shell (appendix A).

List existing keys (at least, those in the default location):

```
ls -al ~/.ssh
```

If you are told `.ssh` doesn't exist, you don't have SSH keys! Keep reading to create them.

If you see a pair of files like `id_rsa.pub` and `id_rsa`, you have a key pair already. You can skip to the section about adding a key to the ssh-agent.

12.3 Set up from RStudio

Instructions for setting up SSH keys from RStudio are given in the Git and GitHub chapter of Wickham's R packages book. Look at the end of the section on initial set up:

- http://r-pkgs.had.co.nz/git.html#git-init

12.4 Set up from the shell

12.4.1 Create SSH key pair

Create the key pair by entering this, but substitute the email address **associated with your GitHub account**:

```
$ ssh-keygen -t rsa -b 4096 -C "jenny@stat.ubc.ca"
```

Accept the proposal to save the key in the default location, i.e., 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. If you take it, you will want to configure something called the ssh-agent to manage this for you (more below).

So either enter a passphrase (and store in your favorite password manager!) or decline by leaving this empty.

```
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 "jenny@stat.ubc.ca"
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 jenny@stat.ubc.ca
The key's randomart image is:
+---[RSA 4096]----+
|      o+   . .   |
|     .=.o . +    |
|     ..= + +     |
|      .+* E      |
|     .= So =     |
|    .  +. = +    |
|   o.. = ..* .   |
|  o ++=.o =o.    |
| ..o.++o.=+.     |
+----[SHA256]-----+
```

12.4.2 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.

Make sure ssh-agent is enabled:

```
jenny@2015-mbp ~ $ eval "$(ssh-agent -s)"
Agent pid 95727
```

Add your key. If you set a passphrase, you'll be challenged for it here. Give it.

```
jenny@2015-mbp ~ $ ssh-add ~/.ssh/id_rsa
Enter passphrase for /Users/jenny/.ssh/id_rsa:
Identity added: /Users/jenny/.ssh/id_rsa (/Users/jenny/.ssh/id_rsa)
```

12.4.3 Provide public key to GitHub

Copy the public key onto your clipboard. 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`****.*

In the top right corner of any page on GitHub, click your profile photo, then click Settings.

In the user settings sidebar, click SSH and GPG keys.

Click New SSH key.

In the "Title" field, add a descriptive label for the new key. For example, if you're using a personal Mac, you might call this key "Personal MacBook Air".

Paste your key into the "Key" field.

Click Add SSH key.

Confirm the action by entering your GitHub password

Chapter 13

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.

13.1 Prerequisites

We assume the following:

- You've registered a free GitHub account (chapter 5).
- You've installed/updated R and RStudio (chapter 6).
- You've installed Git (chapter 7).
- You've introduced yourself to Git (chapter 8).
- You've confirmed that you can push to / pull from GitHub from the command line (chapter 10).

13.2 Make a new repo on GitHub

Go to 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, we will delete this)
Public
YES Initialize this repository with a README

Click 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.

13.3 Clone the new GitHub repository to your computer via RStudio

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`.
  - Do you NOT see an option to get the Project from Version Control? Go to chapter 14 for tips on how to help RStudio find Git.
- Take charge of – or at least notice! – the local directory for the Project. 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".

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.

13.4 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".

13.5 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
```

13.6 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 ...

13.7 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.

Pick one:

- Credential caching for HTTPS access, chapter 11.
- Set up SSH keys, chapter 12.1.

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

13.8 Clean up

Local When you're ready to clean up, you 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 14

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 often "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.

14.1 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.

14.2 Find Git yourself

RStudio can only act as a GUI front-end for Git if Git has been successfully installed (chapter 7) **AND RStudio can find it**.

A basic test for successful installation of Git is to simply enter `git` in the 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 Bit executable lives, try this in a shell:

- `which git` (Mac, Linux)
- `where git` (most versions of Windows)

14.3 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 … the Git executable. It should read something like:

- `/usr/bin/git` (Mac, Linux)
- `C:/Program Files (x86)/Git/bin/git.exe` (Windows)

  Here is a screenshot of someone doing this on a Windows machine.

  - **WARNING**: On Windows, do **NOT** use `C:/Program Files (x86)/Git/cmd/git.exe`. `bin` in the path is GOOD YES! `cmd` in the path is BAD NO!
  - **WARNING**: On Windows, do **NOT** use `git-bash.exe`. Something that ends in `git.exe` is GOOD YES! `git-bash.exe` is BAD NO!

- At times, we've had trouble navigating to the necessary directory on Mac OS, once we'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 type or paste any path you want.

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 our help or install Git another way.
- 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 19.9.
- Get our help.

Chapter 15

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!

15.1 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:

- `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.

15.2 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 for instructions on what to put in your `.bash_profile` in order to add a directory to `PATH`.

15.3 Push/Pull buttons greyed out in RStudio

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

```
jenny@2015-mbp myrepo $ git remote -v
origin  https://github.com/jennybc/myrepo (fetch)
origin  https://github.com/jennybc/myrepo (push)

jenny@2015-mbp myrepo $ git branch -vv
* master b8e03e3 [origin/master] line added locally
```

We want to see that fetch and push are set to remote URLs that point to the remote repo. We also want to see that your local master branch has your GitHub master branch as upstream remote.

If you discover you still need to set a remote, go to the shell and get into the working directory of the RStudio Project and Git repo of interest.

- Initiate the "upstream" or "tracking" relationship by adding a remote. Substitute the HTTPS URL for **your GitHub repo**.

  ```
  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
  ```

- Cement the tracking relationship between your GitHub repository and the local repo by pushing and setting the "upstream" remote:

  ```
  git push -u origin master
  ```

15.4 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."

15.5 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
```

15.6 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.

15.7 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. 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 9) will be able to cope with these situations. But I make no promises. You should also try Git operations from the command line.

15.8 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 is is via 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.

Part III

Early Usage

Chapter 16

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 13). We're doing it again, *with feeling*.

16.1 Make a repo on GitHub

Do this once per new project.

Go to 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 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

- **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 now the "upstream" remote for your local repo. 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 on the command line or in another Git client.

16.3 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"

16.4 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
```

16.5 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".

16.6 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 big green button "Commit changes."

16.7 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.

16.8 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.

Chapter 17

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.

17.1 Make a repo on GitHub

Go to 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.

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 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.

17.2 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.

17.3 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.

17.4 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"

17.5 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
```

17.6 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".

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.

Chapter 18

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 16 and 17).

When might this come up? Maybe if it's an existing project that is also a Git repo with a history you care about? Then you have to do this properly.

This is less desirable for a novice because it involves command line Git. You can't stay within RStudio.

18.1 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:

- Create a new RStudio project: *File > New Project > Existing Directory.*
- Yes: "Open in new session".

If it's already an RStudio Project, launch it.

18.2 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:

Tools > Project Options ... > Git/SVN Under Version control system, select "Git". Confirm New Git Repository? Yes!

Project should re-launch in RStudio and you should now have a Git pane.

18.3 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 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".
- Click "Commit"

18.4 Make a repo on GitHub

Go to 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.

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

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

Click 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.

18.5 Connect to GitHub

Initiate the "upstream" or "tracking" relationship by adding a remote. Go to *Tools* > *shell* and do this, substituting the HTTPS or SSH URL for **your GitHub repo**, according to your setup:

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

- Push and cement the tracking relationship between your GitHub repository and the local repo by pushing and setting the "upstream" remote:

```
git push -u origin master
```

18.6 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".

18.7 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.

Chapter 19

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

19.1 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" 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.

19.2 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.

19.3 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: "2016-07-09"
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: "2016-07-09"
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.

19.4 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.

19.5 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.

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.

19.6 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.

19.7 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.

19.8 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 https://rawgit.com or http://htmlpreview.github.io. Expect some pain with HTML files inside private repos. 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 about how to make a GitHub repo function as a website.

19.9 Troubleshooting

Make sure RStudio and the `rmarkdown` package (and its dependencies) are up-to-date. In case of catastrophic failure to render R Markdown, consider that your software may be too old. R Markdown has been developing rapidly (written 2015-09), so you need a very current version of RStudio and `rmarkdown` to enjoy all the goodies we describe in this course.

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 stuff, it may be preventing the installation or usage of the most recent goodies (see above). Comment the whole file out or rename it something else and relaunch or even re-install RStudio.

Insert a chunk in your `.Rmd` document so that it renders even when there are errors. Some errors are easier to diagnose if you can execute specific R statements during rendering and leave more evidence behind for forensic examination. Put this chunk:

near the top of your R Markdown document if you want to soldier on through errors, i.e. turn `foo.Rmd` into `foo.md` and/or `foo.html` no matter what. This is also helpful if you are writing a tutorial and want to demo code that throws an error. You might want to keep this as an RStudio snippet for easy insertion.

Tolerate errors in one specific chunk. If it's undesirable to globally accept errors, you can still do this for a specific chunk like so:

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. 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. But not today.

Chapter 20

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.

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.

Hands-on experiment: start with an R Markdown file, such as the one you made in your Rmd test drive (chapter 19). Or the boilerplate Rmd RStudio makes with *File > New File > R Markdown …*:

- Save the file as `foo.R`, as opposed to `foo.Rmd`.
- 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.
- Change the syntax of R chunk headers like so:

  Before: ```` ```{r setup, include = FALSE} ````
  After: `#+ r setup, include = FALSE`
- Delete the 3 backticks that end each chunk.

If you're having syntax struggles, here's another exercise:

- Create a very simple R script. Literally, you could summarize and make a plot from the iris data. Make it boring.
- Click on the "notebook" icon in RStudio to "Compile Report". You'll get a very nice HTML report.
- Copy the YAML from the Rmd test drive (chapter 19), but prepend each line with `#'`.

All the workflow tips from the Rmd test drive (chapter 19) transfer: 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.

Chapter 21

Git commands

Here's a start on the various Git commands that have been largely going on under the hood. We've tried to pick workflows that have RStudio doing this for us. But all of this can be done from the command line.

Unless you use the GitHub API, 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
```

Part IV

Tutorial prompts

Chapter 22

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 analyzing 2,000 screenplays is here: https://github.com/brandles/scripts

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 16.

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 23.

Chapter 23

Fork a repo

In the previous prompt (chapter 22), we *cloned* someone else's repo to get a copy that we could keep up to date, by pulling new commits into our local copy.

What if you suspect you might want to propose a change to a repository? Then you should **fork** it, instead of clone it.

23.1 Fork then clone

On GitHub, navigate to a repo of interest.

In the upper right hand corner, click Fork.

This creates a copy of the repo in your GitHub account and takes you there in the browser.

Use the usual workflow (chapter 22) to clone this to your local machine.

23.2 Fork and pull

Make your changes locally, commit, then push back to your fork.

When/if you are ready to propose a change, you place a pull request from your fork on GitHub to the original repo owned by someone else.

Here's a sketch of how this looks:

23.3 Updating your fork

It is harder to keep a fork updated than a clone. Why? Because the primary remote associated with your local is your fork. But you need to get the new commits from the repository you originally forked.

You need to have more than one *remote* associated with your local repo.

If you're nervous about command line Git, you update your fork on GitHub purely via the browser. Then pull to get the commits into your local repo:

- Updating a fork directly from GitHub

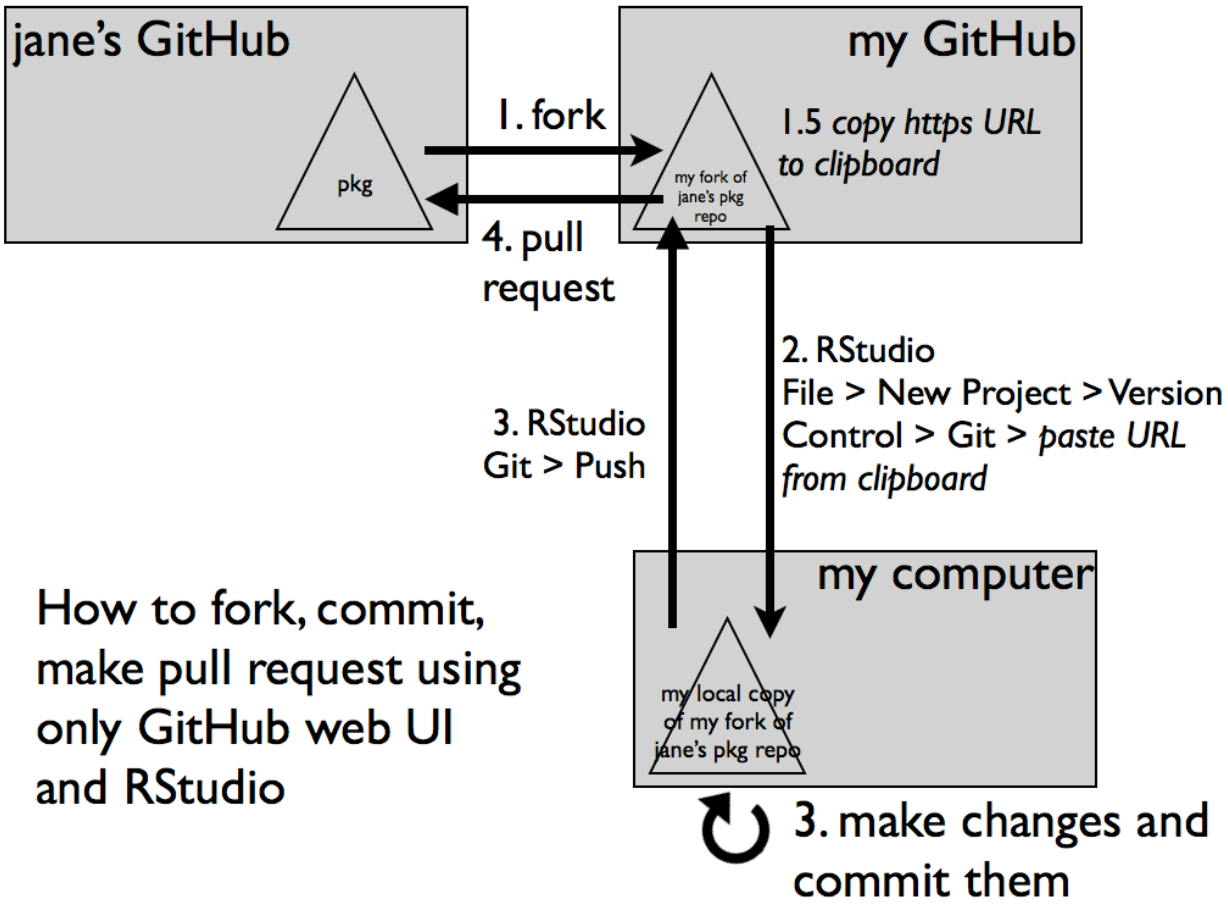

jane's GitHub
pkg
my GitHub
1. fork
1.5 copy https URL
to clipboard
my fork of
jane's pkg
repo
4. pull
request
2. RStudio
File > New Project > Version
Control > Git > paste URL
from clipboard
3. RStudio
Git > Push
my computer
my local copy
of my fork of
jane's pkg repo
3. make changes and
commit them
How to fork, commit,
make pull request using
only GitHub web UI
and RStudio

Figure 23.1:

In the long run, you will want to learn how to configure a second remote for your local repo. Pull the new commits into your local repo, then push them to your own fork:

- GitHub Help: Syncing a fork

Here's a handwritten sketch devised with Bill Mills in a bar, but that will have to do for now!

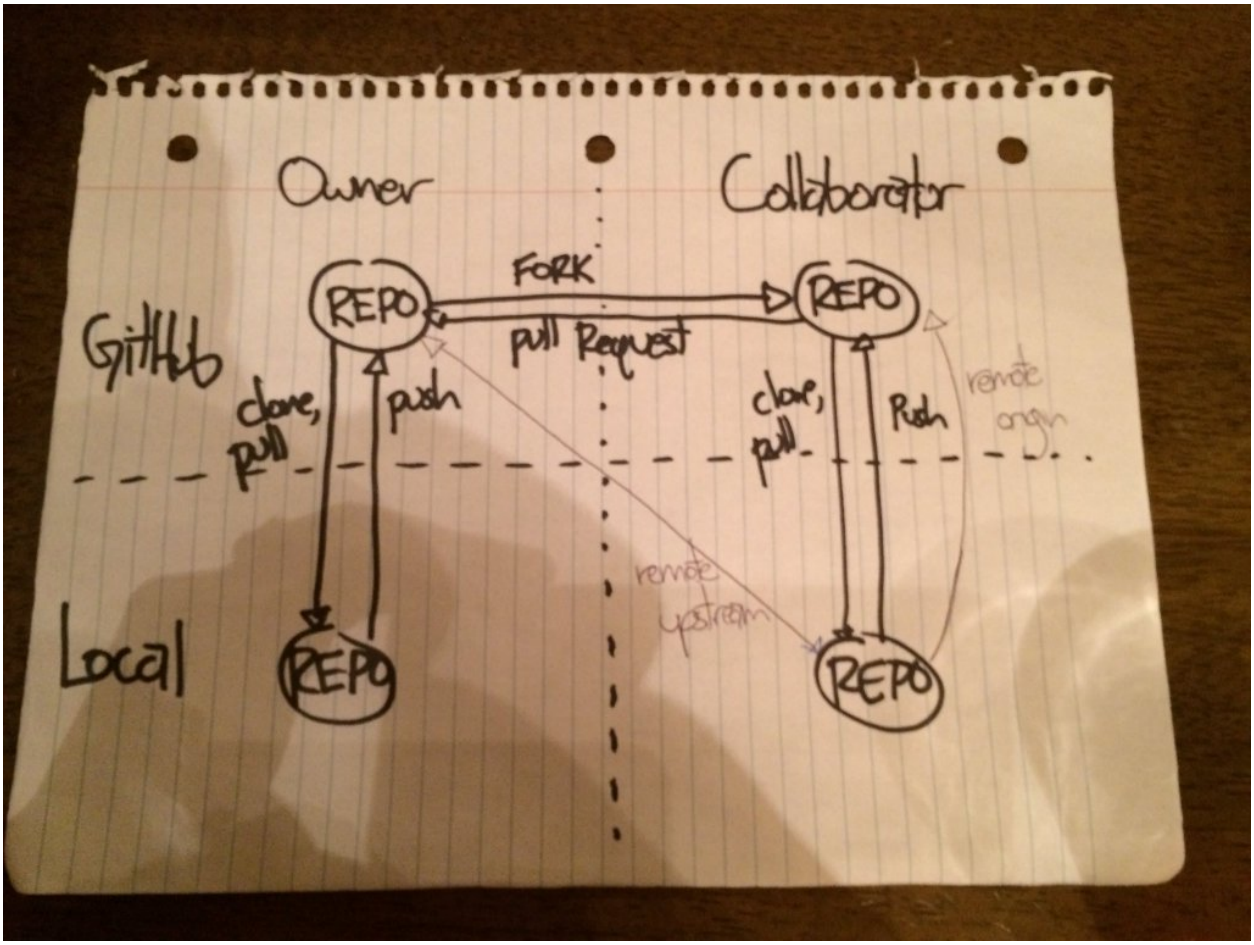


Figure 23.2:

Chapter 24

Create a bingo card

Here's a specific suggestion for practicing "fork and pull".

The general workflow is laid out in chapter 23.

Jenny and Dean have a repository that makes bingo cards with R:

- https://github.com/jennybc/bingo
- Read the README to learn more about it!

Your mission:

- Maybe find a partner? Or a couple of partners?
- Fork the `bingo` repo.
- Clone it to someone's local machine.
- Create a new bingo card by making a file of possible squares.
  - Follow the instructions in https://github.com/jennybc/bingo/blob/master/CONTRIBUTING.md to see how to contribute a new card.
  - Protip: It's easy to be very funny, but create a very difficult bingo card. Remember to include some easy stuff so people have a chance to bingo.
- If you're feeling virtuous, run the tests and check the package. Ask us for help! Or live dangerously and skip this.
- Commit!
- Push your changes back to your copy of the repo on GitHub.
- Make a pull request back to the main `bingo` repo.
- If you card is appropriate, we'll merge your request and it will become part of the package and available via the Shiny app.

Special inspiration for useR:

- Make useR-specific conference bingo.
- See this issue thread for lots of square ideas!
  - https://github.com/jennybc/bingo/issues/4

Chapter 25

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 26

Resetting

Practice recovering from mistakes.

Use a repository you've create earlier in 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 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/blog

Chapter 27

Search GitHub

27.1 Basic resources

GitHub searching

- https://github.com/search/advanced
- https://help.github.com/articles/searching-code/
- https://help.github.com/articles/search-syntax/

Read-only mirror of R source by Winston Chang:

- https://github.com/wch/r-source

Read-only mirror of all packages on CRAN by Gábor Csárdi:

- https://github.com/cran
- http://cran.github.io

27.2 Use case

What if a function in a package has no examples? Or is poorly exampled? Wouldn't it be nice to find functioning instances of it "in the wild"?

Via Twitter, Noam Ross taught me a clever way to do such searches on GitHub. Put this into the GitHub search box to see how packages on CRAN use the `llply()` function from `plyr`:

```
"llply" user:cran language:R
```

Or just click here.

Another example that recently came up on r-package-devel:

How to see lots of examples of roxygen templates?

This search finds >1400 examples of roxygen templates in the wild:

https://github.com/search?utf8= &q=man-roxygen+in%3Apath&type=Code&ref=searchresults

Part V

More

Chapter 28

Notes

Notes for future

28.1 Common workflow questions

28.1.1 Common predicaments and how to recover/avoid

https://twitter.com/JennyBryan/status/743457387730735104

28.1.2 Keep something out of Git

List it in `.gitignore`.

28.1.3 I didn't mean to commit that

Committing things you didn't mean to (too big, secret). How to undo.

28.2 git stuff

Git explainers, heavy on the diagrams

https://twitter.com/JennyBryan/status/743548245645791232

A Visual Git Reference
http://marklodato.github.io/visual-git-guide/index-en.html

A successful Git branching model
http://nvie.com/posts/a-successful-git-branching-model/

A successful Git branching model considered harmful
https://barro.github.io/2016/02/a-succesful-git-branching-model-considered-harmful/

Git Tutorials from Atlassian https://www.atlassian.com/git/tutorials/

Software Carpentry Git Novice Lesson
http://swcarpentry.github.io/git-novice/

Michael Freeman slides on Git collaboration
http://slides.com/michaelfreeman/git-collaboration#/

GitHub Training materials
https://services.github.com/kit/

Git for Ages 4 and Up
https://www.youtube.com/watch?v=3m7BgIvC-uQ

Learn Git Branching
http://learngitbranching.js.org

28.3 The repeated amend

A way to commit often, without exposing your WIP on GitHub or without creating a very cluttered history.

Make changes. Reach a decent stopping point. Test, check, if a package … Render if an analysis …. Nothing broken?

Commit. **Don't push.**

Make more progress. Keep testing or checking or rendering. Inspect diffs to watch what's changing.

Are things broken? Use an appropriate reset to fall back.

Are things improving? Commit but **amend** the previous commit.

Keep going like this until you've built up a commit you can be proud of.

Now push.

It is important to not push amended commits unless you really know what you're doing and you can be quite sure that no one else has pulled your work.

28.4 Disaster recovery

http://stackoverflow.com/questions?sort=votes

Break it down:

- Is something wrong with my filesystem/files?
- Is my git repo messed up?
- How can I keep this from happening again?

Rebase avoidance techniques.

Headless state. Rebase hell.

What to do when you can't, e.g., switch branches. Stashing and WIP commits.

28.5 Engage with R source on GitHub

Browsing

Searching

- My gist, re: the cran user: https://gist.github.com/jennybc/4a1bf4e9e1bb3a0a9b56

- Recent search for roxygen template usage in the wild: https://github.com/search?utf8= &q= man-roxygen+in:path&type=Code&ref=searchresults

Being a useful useR

- stay informed re: development
- use issues for bug reports, feature requests
- make pull requests

28.6 Workflow and psychology

Stress of working in the open

Workflows for group of 1, 2, 5, 10

- Fork and Pull vs Shared Repository
  - https://help.github.com/articles/types-of-collaborative-development-models/
  - https://help.github.com/articles/using-pull-requests/

28.7 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.
- BAD. You can see contributors in 2.

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.
  - The header text, e.g., `My header` be used verbatim as an **implicit header reference**. See 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
  - `[Success and operating systems]` 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
  - Explicit identifier: `[hello git! I'm Jenny](#hello-git)` hello git! I'm Jenny
  - Automatic identifier: `[Any text you want](#recommended-git-clients)` Any text you want

Appendix 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.

A.1 What is the shell?

The `shell` (or `bash` or terminal or Command Prompt on Windows) is a program on your computer whose job is to run other programs, rather than do calculations itself. The `shell` is a very old program and in a time before the mouse it was the only way to interact with a computer. It is still extremely popular among programmers because it is very fast, concise, and is particularly powerful at automating repetitive tasks.

Here we use the `shell` for quite modest goals: to navigate the file system, confirm the present working directory, configure Git, and configure Git remotes.

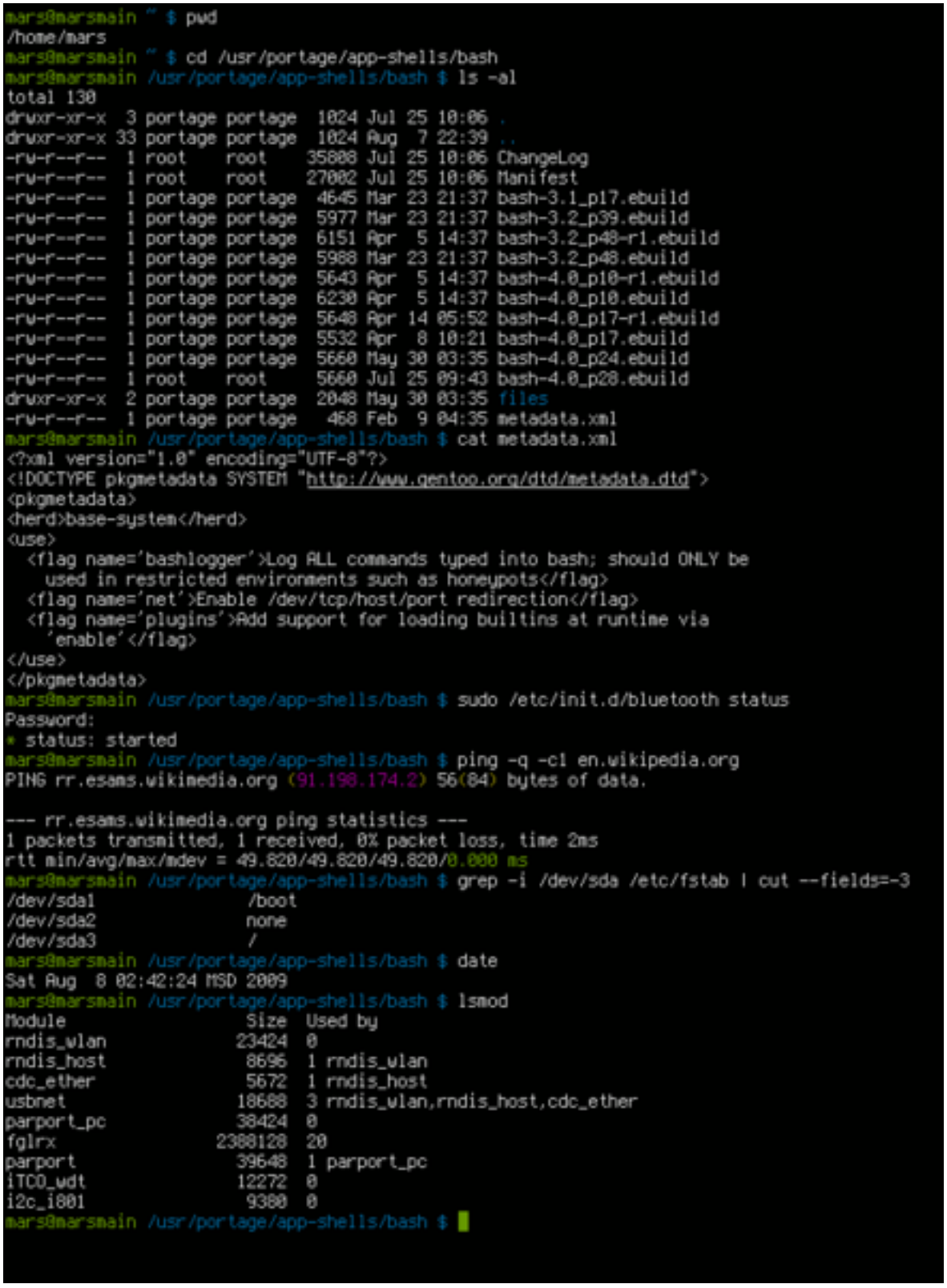
A.2 Starting the shell

In RStudio, go to *Tools > Shell*. This should take you to the shell (on Mac: Terminal, on Windows: GitBash or equivalent). It should be a simple blinking cursor, waiting for input and look similar to this (white text on black background, or black text on white background):

A.3 Using the shell

The most basic commands are listed below:

- `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` (**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` (**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 ..`


mars@marsmain ~ $ pwd
/home/mars
mars@marsmain ~ $ cd /usr/portage/app-shells/bash
mars@marsmain /usr/portage/app-shells/bash $ ls -al
total 130
drwxr-xr-x 3 portage portage 1024 Jul 25 10:06 .
drwxr-xr-x 33 portage portage 1024 Aug 7 22:39 ..
-rw-r--r-- 1 root   root   35808 Jul 25 10:06 ChangeLog
-rw-r--r-- 1 root   root   27002 Jul 25 10:06 Manifest
-rw-r--r-- 1 portage portage 4645 Mar 23 21:37 bash-3.1_p17.ebuild
-rw-r--r-- 1 portage portage 5977 Mar 23 21:37 bash-3.2_p39.ebuild
-rw-r--r-- 1 portage portage 6151 Apr 5 14:37 bash-3.2_p48-r1.ebuild
-rw-r--r-- 1 portage portage 5988 Mar 23 21:37 bash-3.2_p48.ebuild
-rw-r--r-- 1 portage portage 5643 Apr 5 14:37 bash-4.0_p10-r1.ebuild
-rw-r--r-- 1 portage portage 6230 Apr 5 14:37 bash-4.0_p10.ebuild
-rw-r--r-- 1 portage portage 5648 Apr 14 05:52 bash-4.0_p17-r1.ebuild
-rw-r--r-- 1 portage portage 5532 Apr 8 10:21 bash-4.0_p17.ebuild
-rw-r--r-- 1 portage portage 5660 May 30 03:35 bash-4.0_p24.ebuild
-rw-r--r-- 1 root   root   5660 Jul 25 09:43 bash-4.0_p28.ebuild
drwxr-xr-x 2 portage portage 2048 May 30 03:35 files
-rw-r--r-- 1 portage portage   468 Feb 9 04:35 metadata.xml
mars@marsmain /usr/portage/app-shells/bash $ cat metadata.xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE pkgmetadata SYSTEM "http://www.gentoo.org/dtd/metadata.dtd">
<pkgmetadata>
<herd>base-system</herd>
<use>
  <flag name='bashlogger'>Log ALL commands typed into bash; should ONLY be
    used in restricted environments such as honeypots</flag>
  <flag name='net'>Enable /dev/tcp/host/port redirection</flag>
  <flag name='plugins'>Add support for loading builtins at runtime via
    'enable'</flag>
</use>
</pkgmetadata>
mars@marsmain /usr/portage/app-shells/bash $ sudo /etc/init.d/bluetooth status
Password:
* status: started
mars@marsmain /usr/portage/app-shells/bash $ ping -q -c1 en.wikipedia.org
PING rr.esams.wikimedia.org (91.198.174.2) 56(84) bytes of data.

--- rr.esams.wikimedia.org ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 2ms
rtt min/avg/max/mdev = 49.820/49.820/49.820/0.000 ms
mars@marsmain /usr/portage/app-shells/bash $ grep -i /dev/sda /etc/fstab | cut --fields=-3
/dev/sda1   /boot
/dev/sda2   none
/dev/sda3   /
mars@marsmain /usr/portage/app-shells/bash $ date
Sat Aug 8 02:42:24 MSD 2009
mars@marsmain /usr/portage/app-shells/bash $ lsmod
Module   Size Used by
rndis_wlan   23424 0
rndis_host   8696 1 rndis_wlan
cdc_ether   5672 1 rndis_host
usbnet   18688 3 rndis_wlan,rndis_host,cdc_ether
parport_pc   38424 0
fglrx   2388128 20
parport   39648 1 parport_pc
iTCO_wdt   12272 0
i2c_i801   9380 0
mars@marsmain /usr/portage/app-shells/bash $

Figure A.1:

- go to your "home" directory: `cd ~` 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.
- *we should add more*

A.4 Note for Windows users

On Windows, the program that runs the shell is called *Command Prompt* or *cmd.exe*. Unfortunately, the default Windows shell does not support all the commands that other operating systems do. This is where GitBash comes in handy: it installs a light version of a shell that does support all the above commands. When you access the shell through RStudio, RStudio actually tries to open GitBash if it can find it, but it will open the default Windows Command Prompt if GitBash is not found.

If you get an error message such as `'pwd' is not recognized as an internal or external command, operable program or batch file.` from any of the previous commands, that means that RStudio could not find GitBash. The most likely cause of this is that you did not install git using the recommended method from this book, so try re-installing git.

Appendix B

Comic relief

It's not you, it's Git!

If you're not crying already, these fictional-but-realistic Git man pages should do the trick:

- git-man-page-generator
- And, of course, the underlying source is also available on GitHub:
  - https://github.com/Lokaltog/git-man-page-generator

If you can tolerate adult and often offensive language, you might enjoy:

- http://www.commitlogsfromlastnight.com

Your commits will look more glorious scrolling by Star Wars style:

- http://starlogs.net
- http://starlogs.net/#jennybc/googlesheets
- Do this for any repo: `http://starlogs.net/#USER/REPO`

Appendix C

References

C.0.1 Resources

We practice what we preach! This site is created with Git and R markdown, using the `bookdown` package. Go ahead and peek behind the scenes.

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 is to (command line) Git as swirl is to R. Learn by doing, in small bites.
- The book Pro Git is fantastic and comprehensive.
- 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.
- GitHub's own training materials may be helpful. They also point to many other resources
- Find a powerful Git client (chapter 9) if you'd like to minimize your usage of Git from the command line.
- Hadley Wickham's book R Packages has an excellent chapter on the use of Git, GitHub, and RStudio in R package development. He covers more advanced usage, such as commit best practices, issues, branching, and pull requests.

Bibliography