Data Science And AI for Economists

Lecture 1: Toolkits: Introduction to Git and Github

Zhaopeng Qu
Business School, Nanjing University
September 03 2025

Roadmap

Today we will cover the following topics:

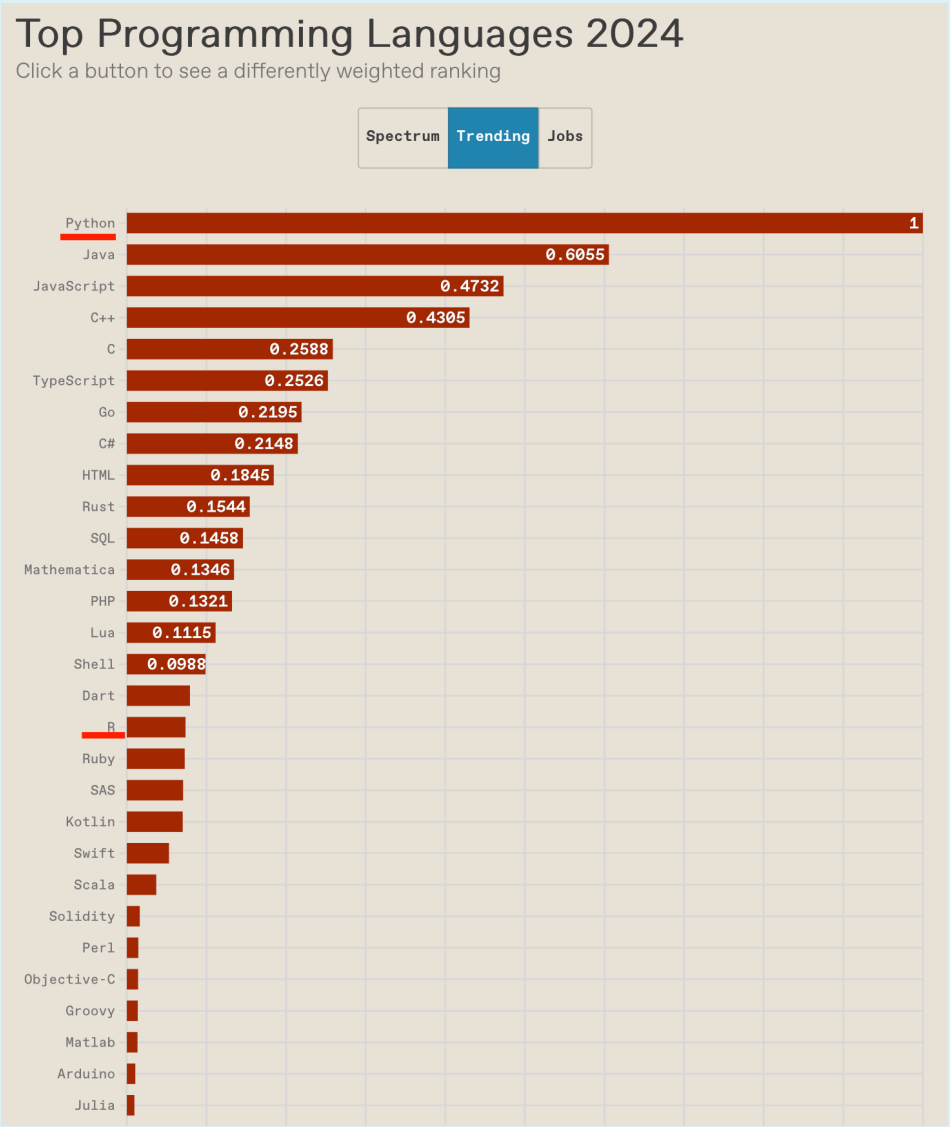
- Programming Language Selection:
  - Install and Setup for R
  - Install and Setup for RStudio
- Version Control, Git and GitHub
  - Why do we need it?
  - How to use it?

Programming Languages

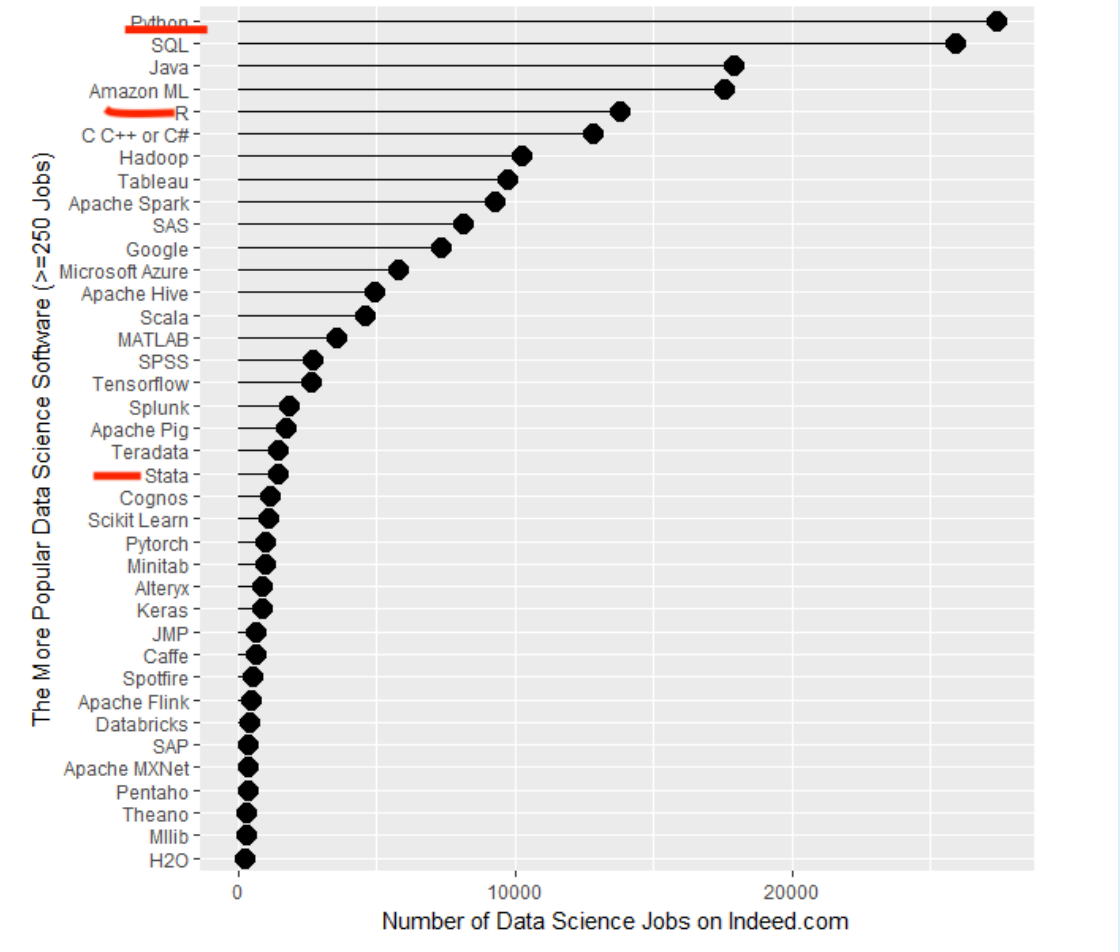
Top Programming Languages by IEEE in 2019-2024

Rank	Language	Type	Score
1	Python		100.0
2	Java		96.3
3	C		94.4
4	C++		87.5
5	R		81.5
6	JavaScript		79.4
7	C#		74.5
8	Matlab		70.6
9	Swift		69.1
10	Go		68.0

2019



Popular Softwares by **Jobs** in 2019



2019

Popular Softwares by Jobs in 2024

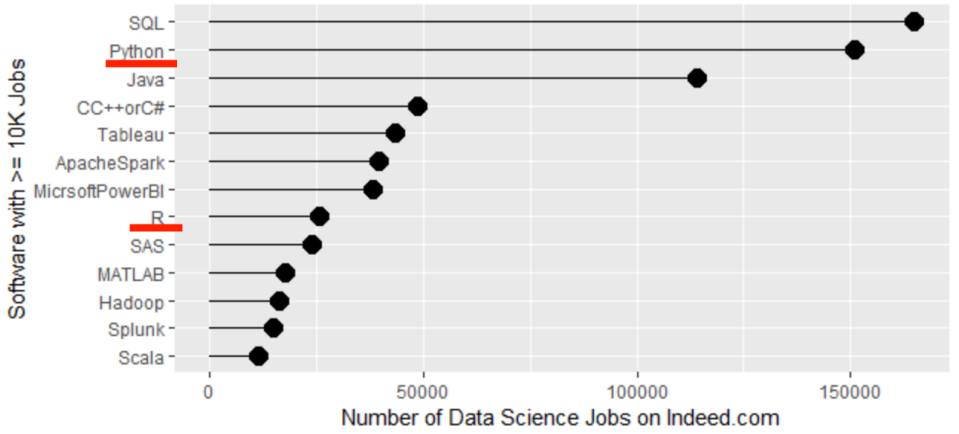
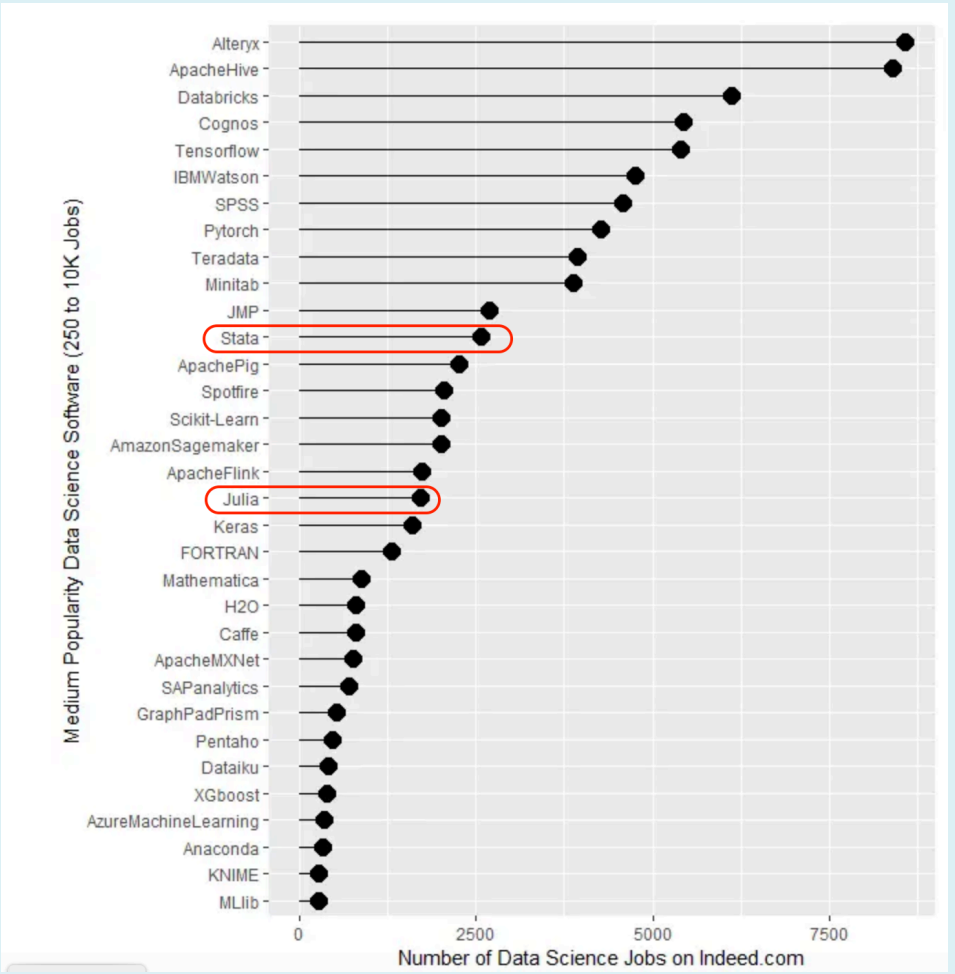


Figure 1a. Number of data science jobs for the more popular software (>= 10,000 jobs).

2024



2024

Languages: Python, R, Julia and Stata

Python:

- Pros:
  - General-purpose: web development, data analysis, machine learning, etc.
  - Rich libraries: numpy, pandas, scikit-learn, tensorflow, etc.
  - Community: Stack Overflow, GitHub, etc.
- Cons:
  - Speed: slower than C/C++/Fortran
  - Memory: not good at memory management
  - Syntax: not as clean as R
  - IDE: not as good as RStudio

Julia

- Pros:
  - Speed: as fast as C/C++/Fortran
  - Syntax: as clean as Python
  - Memory: good at memory management
  - Community: Stack Overflow, GitHub, etc.
- Cons:
  - IDE: not as good as RStudio
  - Libraries: not as rich as Python and R

Languages: Python, R, Julia and Stata

R:

- Pros:
  - Statistical language: designed for data analysis
  - Rich libraries: dplyr, ggplot2, tidyr, etc.
  - Community: Stack Overflow, GitHub, etc.
  - IDE: RStudio
- Cons:
  - Speed: slower than C/C++/Fortran
  - Memory: not good at memory management
  - Syntax: not as clean as Python
  - General-purpose: not as good as Python

Stata

- Pros:
  - Statistical language: designed for statistics and econometrics
  - Community: Stata Forum, etc. mailist
  - IDE: Stata
- Cons:
  - Speed: slower than C/C++/Fortran
  - Memory: not good at memory management
  - Syntax: the most clean even better than python
  - General-purpose: not as good as Python

Which programming language should you learn?

- Of course, more is better than less. But we'd better focus on **one or two languages** at the beginning.
- It depends on what you already know and what you want to do.
  - For Data Science & AI, Python and R are the most popular languages.
  - For Econometrics, R and Stata are the most popular languages.
  - For Macroeconomics and High Performance Computing, Julia(C++) is the best choice.
- For this course, we will focus on **R** and **Python**.
- Today we will focus on **R** and **RStudio**.

Installing and Setting Up R

Installing R and RStudio

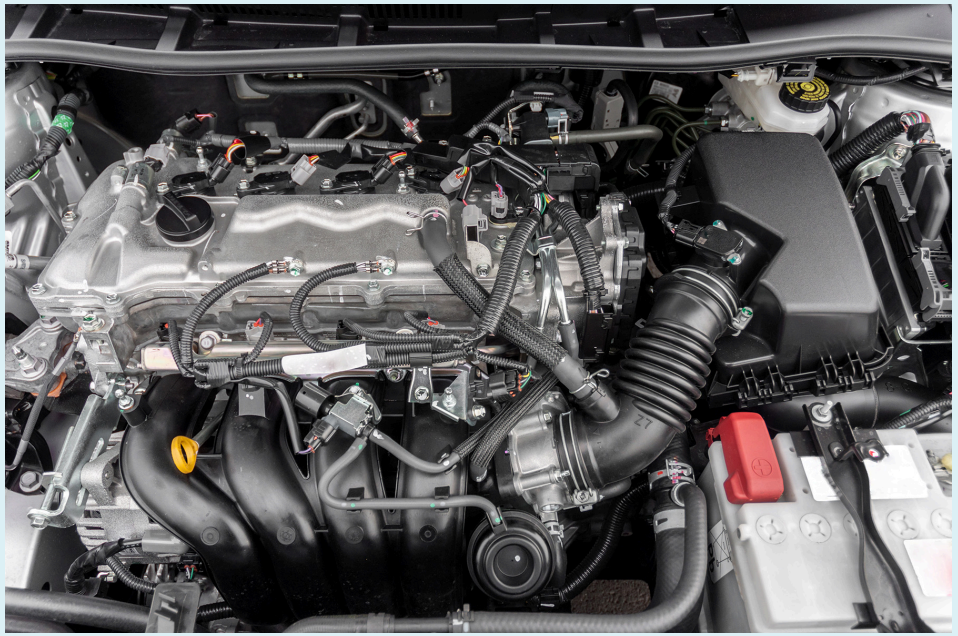
R:

- Windows: Download and install R from CRAN.
- Mac: Download and install R from CRAN.

IDE for R

- Integrated Development Environment(IDE) is a software application that provides comprehensive facilities to computer programmers for software development.
    - R is the "engine" of the car, and IDE is the navigator of the car. And RStudio is the most popular IDE for R.
    - Download and install RStudio from RStudio.
- NOTE: You should *install R first* and *then install RStudio*.

Using IDE(RStudio) v.s Terminal(R)



R

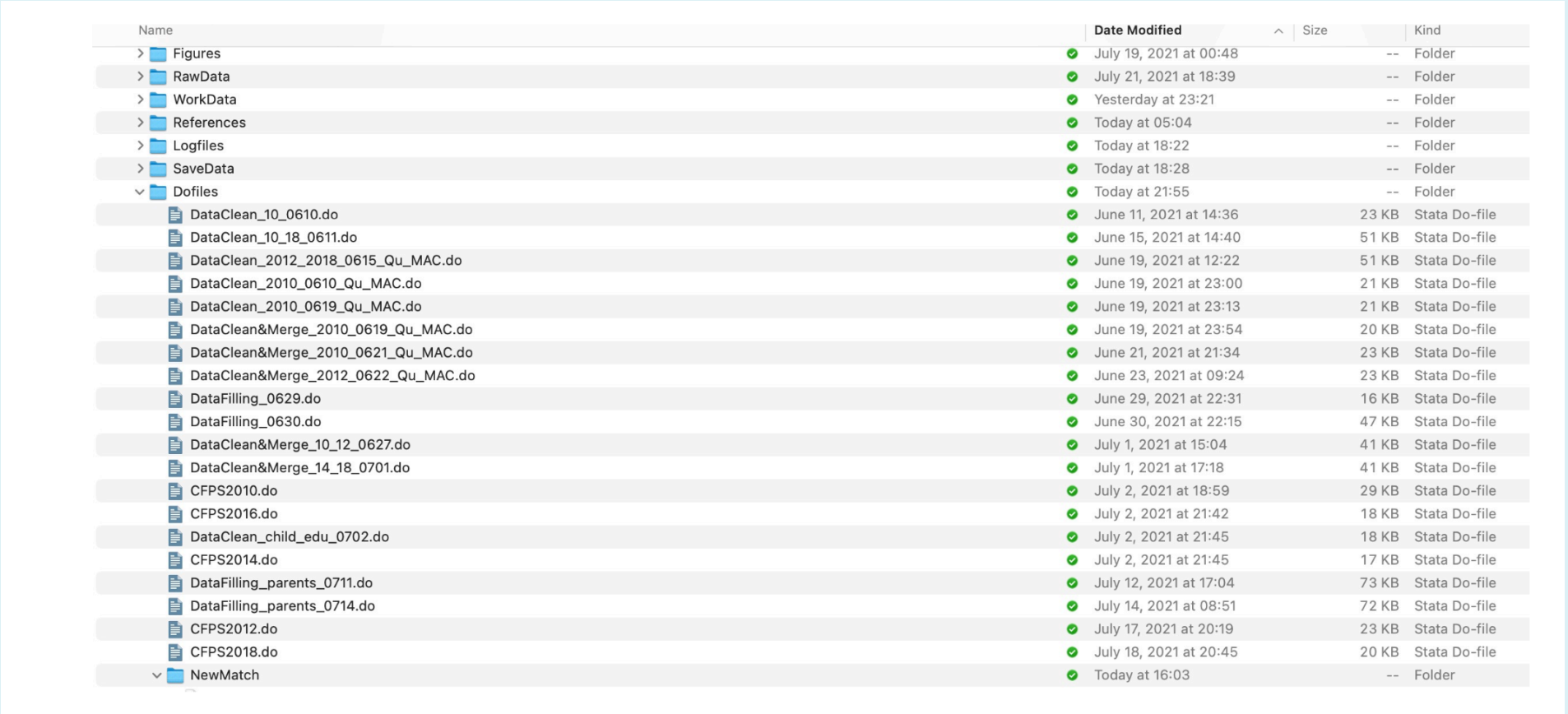
RStudio

Source: ModernDive(2025)

Introduction to Version Control, Git and GitHub

Why do we need Version Control?

- Once upon a time, there was a researcher who worked on a project for a while.



Name	Date Modified	Size	Kind
Figures	July 19, 2021 at 00:48	--	Folder
RawData	July 21, 2021 at 18:39	--	Folder
WorkData	Yesterday at 23:21	--	Folder
References	Today at 05:04	--	Folder
Logfiles	Today at 18:22	--	Folder
SaveData	Today at 18:28	--	Folder
Dofiles	Today at 21:55	--	Folder
DataClean_10_0610.do	June 11, 2021 at 14:36	23 KB	Stata Do-file
DataClean_10_18_0611.do	June 15, 2021 at 14:40	51 KB	Stata Do-file
DataClean_2012_2018_0615_Qu_MAC.do	June 19, 2021 at 12:22	51 KB	Stata Do-file
DataClean_2010_0610_Qu_MAC.do	June 19, 2021 at 23:00	21 KB	Stata Do-file
DataClean_2010_0619_Qu_MAC.do	June 19, 2021 at 23:13	21 KB	Stata Do-file
DataClean&Merge_2010_0619_Qu_MAC.do	June 19, 2021 at 23:54	20 KB	Stata Do-file
DataClean&Merge_2010_0621_Qu_MAC.do	June 21, 2021 at 21:34	23 KB	Stata Do-file
DataClean&Merge_2012_0622_Qu_MAC.do	June 23, 2021 at 09:24	23 KB	Stata Do-file
DataFilling_0629.do	June 29, 2021 at 22:31	16 KB	Stata Do-file
DataFilling_0630.do	June 30, 2021 at 22:15	47 KB	Stata Do-file
DataClean&Merge_10_12_0627.do	July 1, 2021 at 15:04	41 KB	Stata Do-file
DataClean&Merge_14_18_0701.do	July 1, 2021 at 17:18	41 KB	Stata Do-file
CFPS2010.do	July 2, 2021 at 18:59	29 KB	Stata Do-file
CFPS2016.do	July 2, 2021 at 21:42	18 KB	Stata Do-file
DataClean_child_edu_0702.do	July 2, 2021 at 21:45	18 KB	Stata Do-file
CFPS2014.do	July 2, 2021 at 21:45	17 KB	Stata Do-file
DataFilling_parents_0711.do	July 12, 2021 at 17:04	73 KB	Stata Do-file
DataFilling_parents_0714.do	July 14, 2021 at 08:51	72 KB	Stata Do-file
CFPS2012.do	July 17, 2021 at 20:19	23 KB	Stata Do-file
CFPS2018.do	July 18, 2021 at 20:45	20 KB	Stata Do-file
NewMatch	Today at 16:03	--	Folder

My Messy Stata Dofiles

Why do we need Version Control?

Writing	July 17, 2021 at 21:59	--	Folder
农民工工资趋同_0511_Qu——无分组.docx	May 15, 2021 at 13:00	2.4 MB	Microso...(.docx)
农民工工资趋同_0515_Qu——无分组.docx	May 16, 2021 at 12:03	3 MB	Microso...(.docx)
城市职工的结果.docx	May 16, 2021 at 13:41	279 KB	Microso...(.docx)
农民工工资趋同_0515_by_QU.docx	May 16, 2021 at 18:59	1.9 MB	Microso...(.docx)
农民工工资趋同_0516_by_QU.docx	May 18, 2021 at 12:19	1.9 MB	Microso...(.docx)
农民工工资趋同_0518_qu.docx	May 19, 2021 at 11:34	1.3 MB	Microso...(.docx)
农民工工资趋同_20210606_qu02.docx	June 7, 2021 at 02:09	827 KB	Microso...(.docx)
农民工工资趋同_20210606_中国工业经济.docx	June 7, 2021 at 08:19	829 KB	Microso...(.docx)
农民工工资趋同_20210606.docx	June 7, 2021 at 08:27	826 KB	Microso...(.docx)
农民工工资趋同_20210606.pdf	June 7, 2021 at 08:28	2.2 MB	PDF Document
农民工工资趋同_20210607.docx	June 7, 2021 at 08:50	826 KB	Microso...(.docx)
最终版本	July 17, 2021 at 09:51	--	Folder
农民工工资趋同_20210607_final.docx	June 7, 2021 at 08:50	826 KB	Microso...(.docx)

My Messy Drafts

- Different versions of the same file are a mess for the data scientist.
- Replicability and Reproducibility are the most important things for social science research.
  - Many journals require you to provide your code and data for replication.

Why do we need Version Control?

- Version Control is a system that records changes to a file or set of files over time so that you can recall specific versions later.
  - Like a tracking system for your files.
- Git is a distributed version control system that allows you to track changes in your files. And it can be deployed on
  - a local server(PC)
  - or cloud server or web-based platform.
- **Guidebook: Pro Git 2nd Edition, free download and Chinese Edition.**

Common Misceonceptions

> "Github is a data science tool for sharing data"

- It's built more for version controlling plain-text code (that analyzes data) and text (that documents it).

> "Git is only relevant for software developers"

- It also has distinct benefits for the applied researchers' workflow

> Version control is *only* useful for collaborative projects"

- No, in fact putting your solo work under version control at first.
    - then move on to more complicated collaborations.

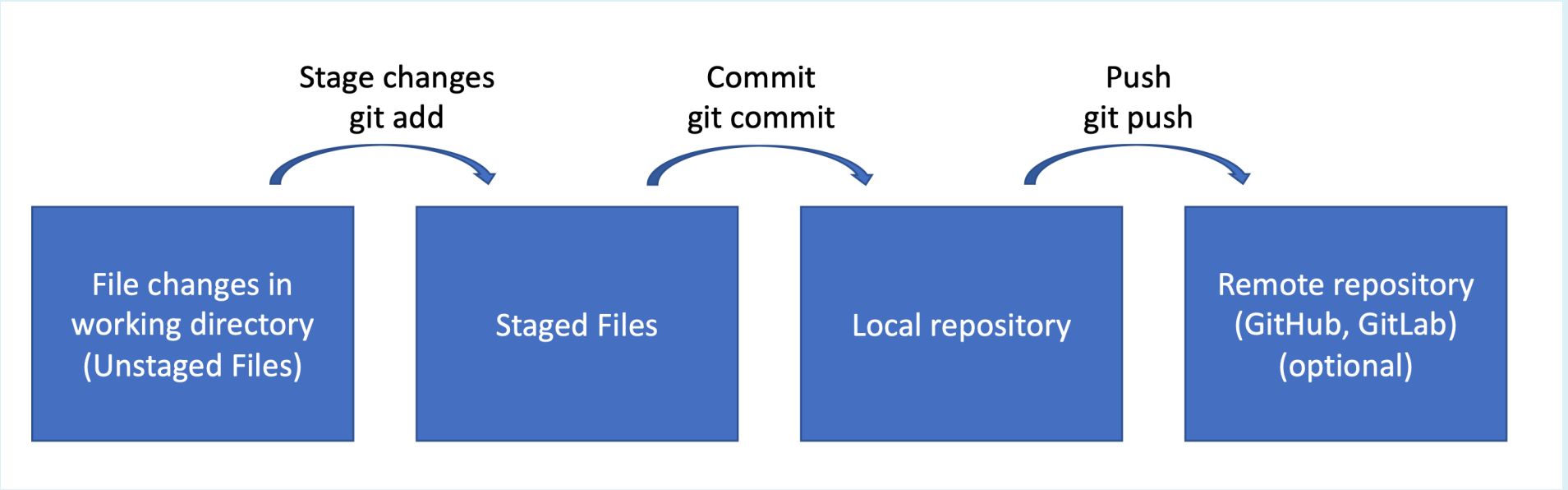
Git: The Basic Workflow

1. *do work in your own working directory (your own PC),like coding, writing, and so on.*

2. *stage files(暂存文件), thus adding snapshots of them to your staging area.*

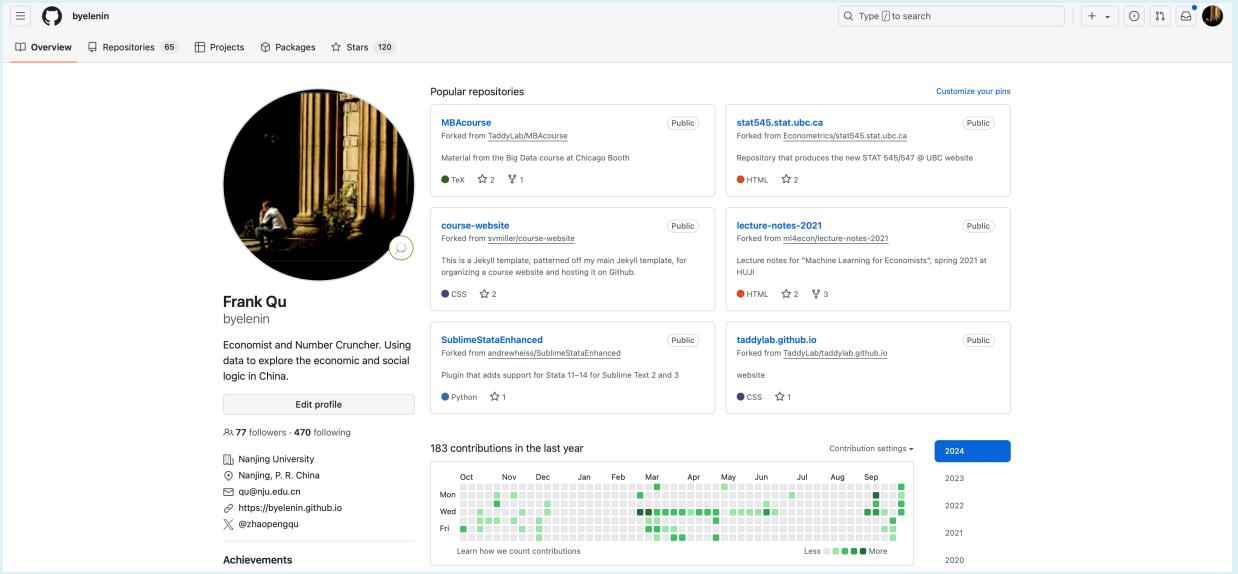
3. *commit(提交), which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.*

4. *if you have a remote repository, push your commit to it.(上传到云端)*

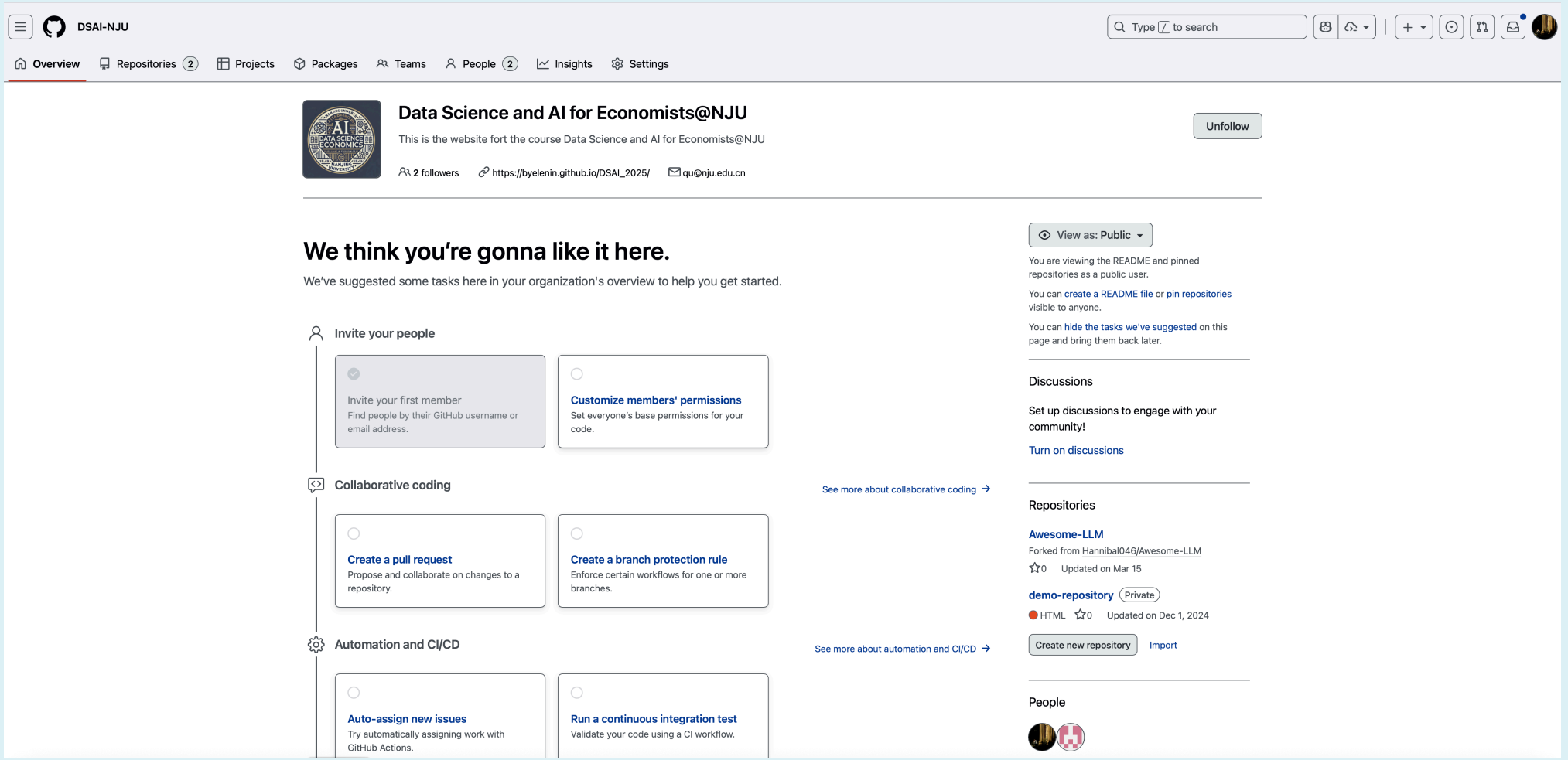


Using GitHub as your remote git server

- You can use Git on your local server, but you can also use GitHub to manage your files.
- GitHub is a web-based platform that provides hosting for software development and version control using Git.
  - Other providers are GitLab, Bitbucket, Gitee(Chinese version of GitHub) etc.
- My github page



Our Course Repository on GitHub



Git and GitHub Management

- Git is originally a command-line tool, and you can use it in the following ways:

1. Command Line Interface (CLI): Shell Command-line Tools

2. GUI: A graphical user interface for Git, such as GitHub Desktop, SourceTree, GitKraken,

3. IDE: Integrated Development Environment, such as RStudio, VS Code...

4. Web-based Platform: A web-based platform for Git management, such as GitHub, GitLab, Bitbucket...

- As a beginner, I recommend you use RStudio/GitHub Web/GitHub Desktop/ for Git management.

Lab: GitHub Practice with RStudio[1]

[1] This section is heavily borrowed from Grant McDermott's course materials (Lecture 2) at the University of Oregon.Although I have made some minor modifications to the original materials, the core content remains the same.

Git Installing and Setup

- Open your terminal for MacOS or Command Prompt for Windows and type the following command:
  - `git --version`.
- Git installing
  - Windows: Download and install Git from here.
  - Mac: Download and install Git from here.

Git Installing and Setup

- Git Setup: Open your terminal for MacOS or Command Prompt for Windows and type the following command:
  - `git config --global user.name "Your Name"`
  - `git config --global user.email "Your email"`
  - `git config --global --list`
- Git Help: Type `git help` or `git help -a` for more information.

GitHub Practice: Register an account

- Register an account
  - Sign up: Create an account on GitHub.
  - Educational Account: Apply for an educational account on GitHub.
  - The Big Benefits: Free Copilot Access and Unlimited Private Repositories.

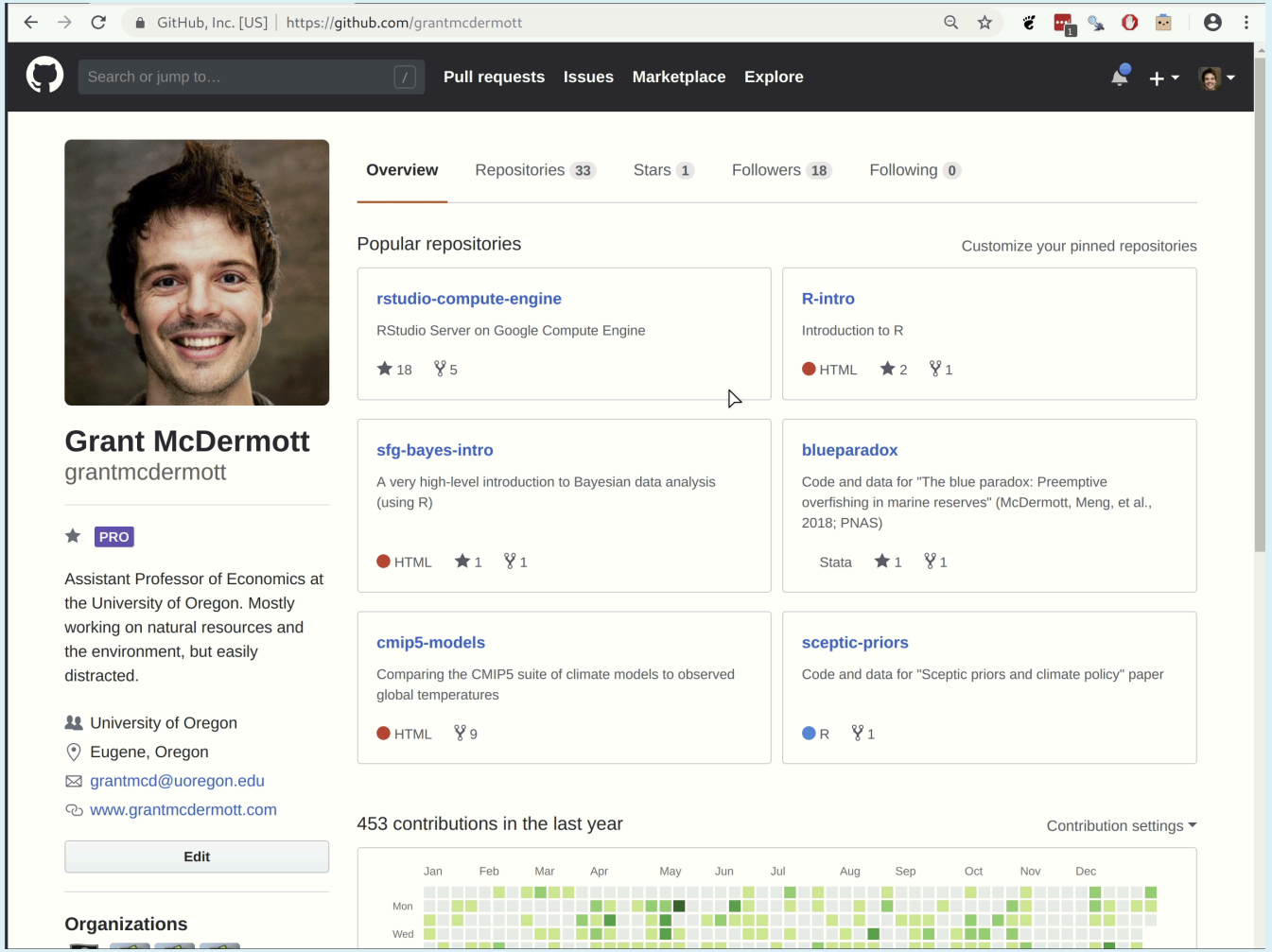
GitHub Practice with RStudio

- Link a GitHub repository (i.e. "repo") to an RStudio Project. Here are the steps we're going to follow:

1. Create the repo on GitHub and initialize with a README.
2. Copy the HTTPS/SSH link (the green "Clone or Download" button).[1]
3. Open up RStudio.
4. Navigate to File -> New Project -> Version Control -> Git.
5. Paste your copied link into the "Repository URL:" box.
6. Choose the project path ("Create project as subdirectory of:") and click Create Project.

[1] It's easiest to start with HTTPS, but SSH is advised for more advanced users.

Link a GitHub repo to an RStudio Project



Make some local changes

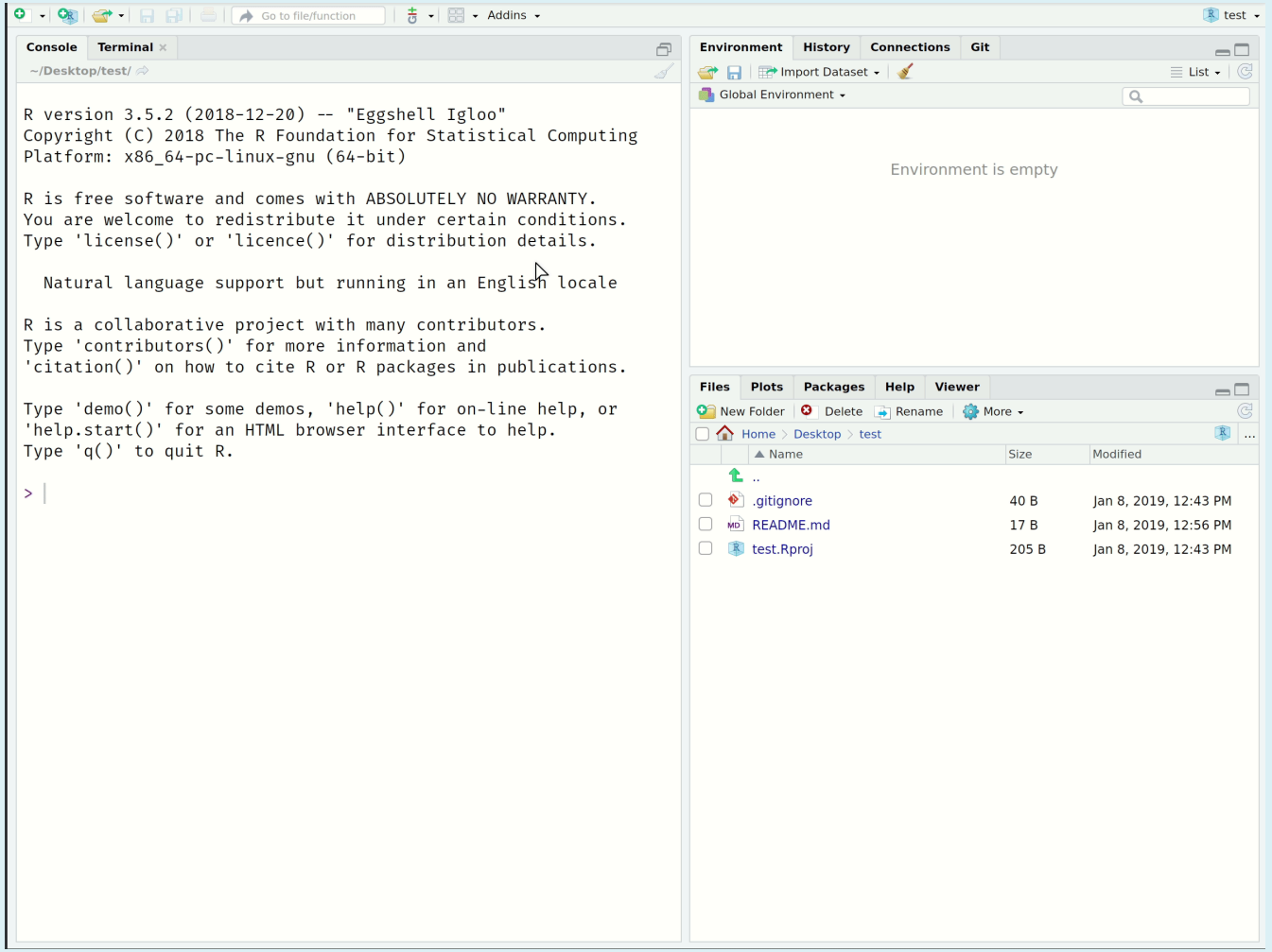
Look at the top-right panel in your RStudio IDE. Do you see the "Git" tab?

- Click on it.
- There should already be some files in there, which we'll ignore for the moment.

Now open up the README file (see the "Files" tab in the bottom-right panel).

- Add some text like "Hello World!" and save the README.
- Do you see any changes in the "Git" panel? Good. (Raise your hand if not.)

Make some local changes



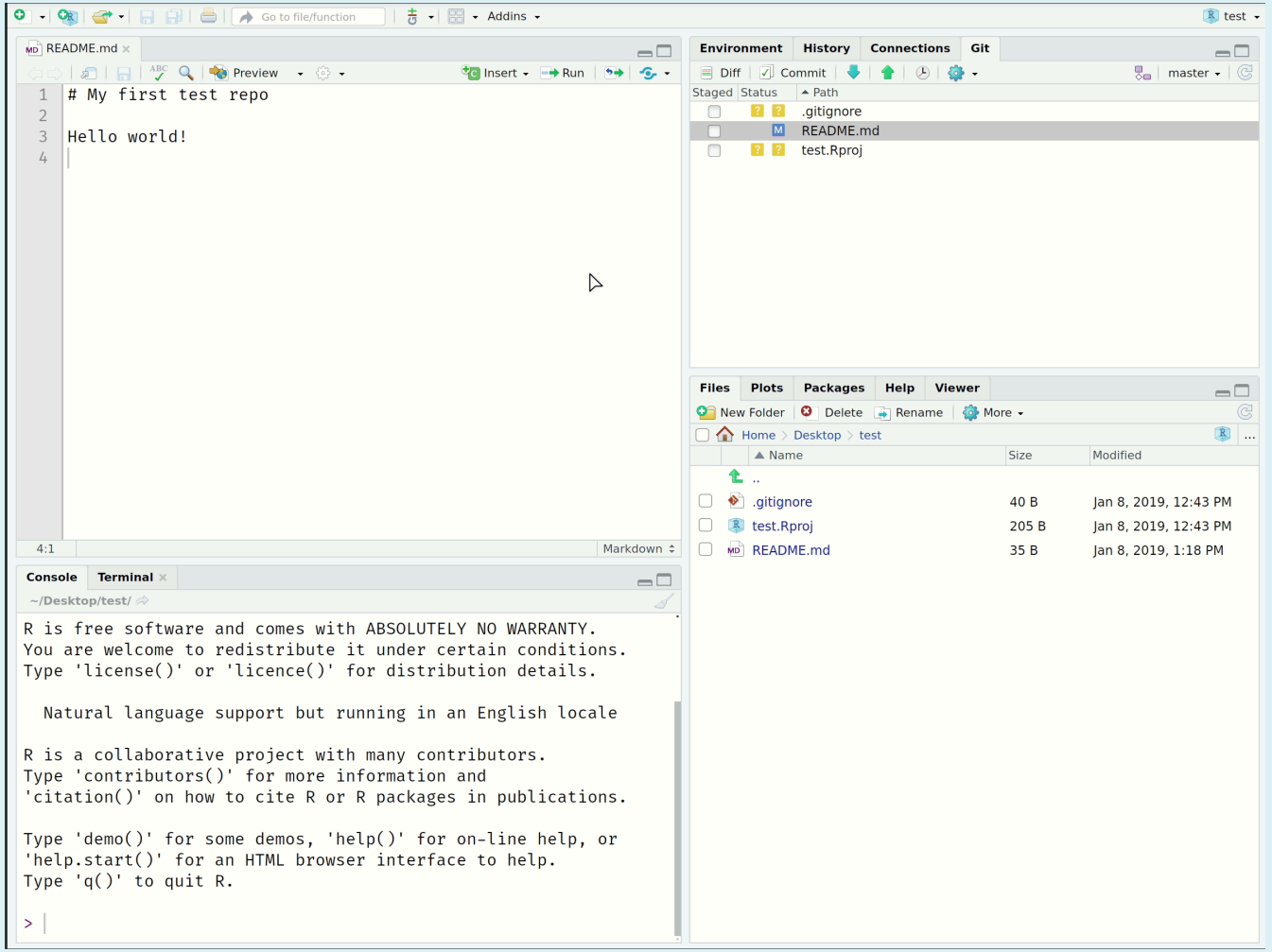
Main Git operations

- Now that you've cloned your first repo and made some local changes, it's time to learn the four main Git operations.

1. Stage (or "add")
   - Tell Git that you want to add changes to the repo history (file edits, additions, deletions, etc.)
2. Commit
   - Tell Git that, yes, you are sure these changes should be part of the repo history.
3. Pull
   - Get any new changes made on the GitHub repo (i.e. the upstream remote), either by your collaborators or you on another machine.
4. Push
   - Push any (committed) local changes to the GitHub repo

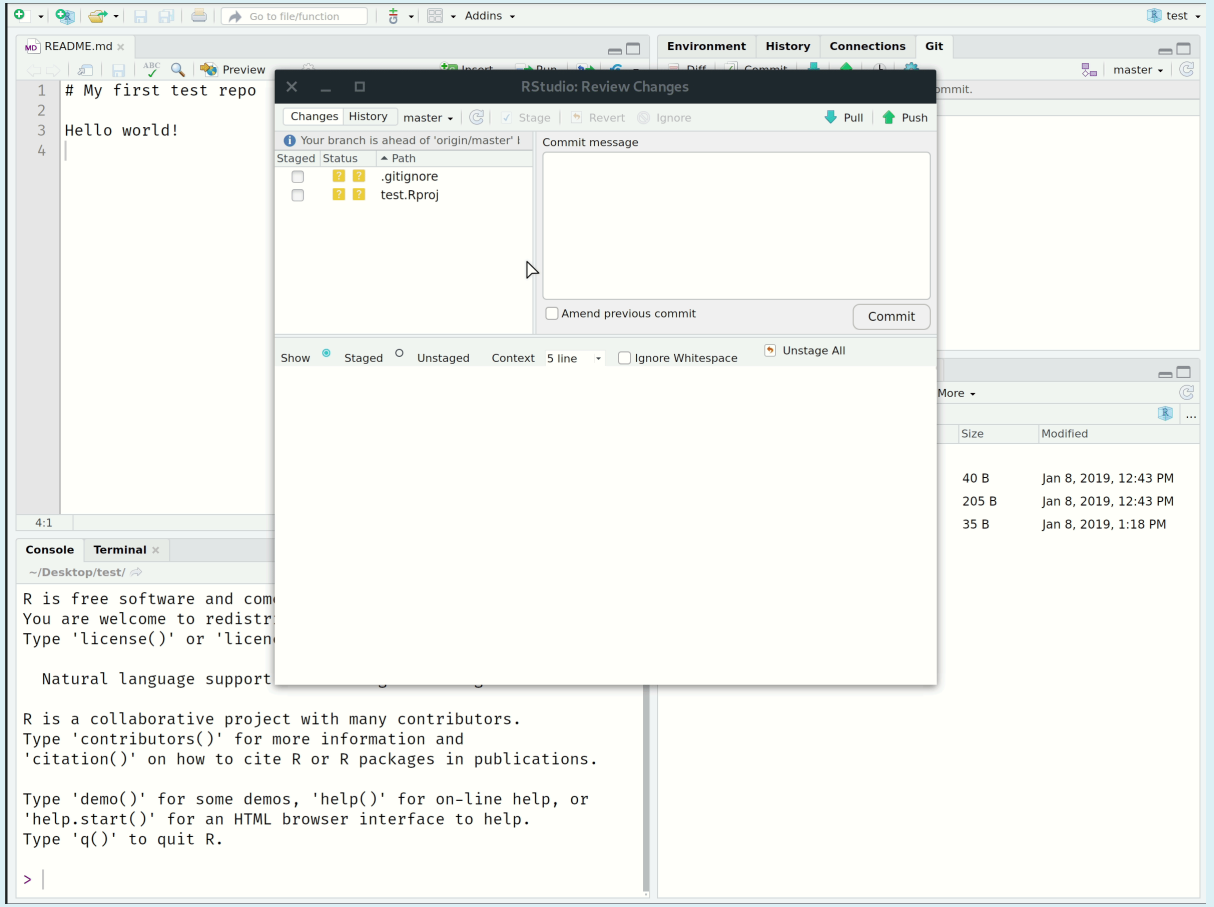
- For the moment, it will be useful to group the first two operations and last two operations together.

Stage and Commit



- Note the helpful commit message to ourselves.

Push and Pull



Recap

Here's a step-by-step summary of what we just did.

- *Made same changes* to a file and saved them locally.
- *Staged* these local changes.
- *Committed* these local changes to our Git history with a helpful message.
- *Pulled* from the GitHub repo just in case anyone else made changes too (not expected here, but good practice).
- *Pushed* our changes to the GitHub repo.

- Always **pull** from the upstream repo *before* you **push** any changes. Seriously, do this even on solo projects; making it a habit will save you headaches down the road.

Why this workflow?

- Creating the repo on GitHub first means that it will always be "upstream" of your (and any other) local copies.
- In effect, this allows GitHub to act as the central node in the distributed VC network.
- Especially valuable when you are collaborating on a project with others — more on this later — but also has advantages when you are working alone.
- If you would like to move an existing project to GitHub, my advice is still to create an empty repo there first, clone it locally, and then copy all your files across.

- RStudio Projects are great.
    - they interact seamlessly with Git(Hub), as we've just seen.
    - They also solve absolute vs. relative path problems, since the .Rproj file acts as an anchor point for all other files in the repo.

Git from the shell

Why bother with the shell?

The GitHub + RStudio Project combo is ideal for new users.

- RStudio's Git integration and built-in GUI cover all the major operations.
- RStudio Projects FTW.

However, I want to go over Git shell commands so that you can internalise the basics.

- The shell is more powerful and flexible. Does some things that the RStudio Git GUI can't.
- Potentially more appropriate for projects that aren't primarily based in R. (Although, no real harm in using RStudio Projects to clone a non-R repo.)

How to open the shell?

Windows:

- PowerShell (recommended)
- Command Prompt (cmd)
- Windows Terminal (modern terminal app)

macOS:

- Terminal app with zsh (default since 2019)
- Terminal app with bash (optional)

Main Git shell commands

Clone a repo.

```
$ git clone REPOSITORY-URL
```

See the commit history (hit spacebar to scroll down or q to exit).

```
$ git log
```

What has changed?

```
$ git status
```

Main Git shell commands (cont.)

Stage ("add") a file or group of files.

```
$ git add NAME-OF-FILE-OR-FOLDER
```

You can use wildcard characters to stage a group of files (e.g. sharing a common prefix). There are a bunch of useful flag options too:

- Stage all files.

  ```
  $ git add -A
  ```

- Stage updated files only (modified or deleted, but not new).

  ```
  $ git add -u
  ```

- Stage new files only (not updated).

  ```
  $ git add .
  ```

Main Git shell commands (cont.)

Commit your changes.

```
$ git commit -m "Helpful message"
```

Pull from the upstream repository (i.e. GitHub).

```
$ git pull
```

Push any local changes that you've committed to the upstream repo (i.e. GitHub).

```
$ git push
```

Merge conflicts

Collaboration time

- Turn to the person next to you. You are now partners temporarily.
- **Student 1**: Invite Student 2 to join you as a collaborator on the "test" GitHub repo that you created earlier. (See the *Settings* tab of your repo.)
- **Student 2**: Clone Student 1's repo to your local machine.Change into a new directory first or give it a different name to avoid conflicts with your own "test" repo.
- **Student 2**: Make some edits to the README (e.g. delete lines of text and add your own). Stage, commit and push these changes.
- **Student 1**: Now make your own changes to the README on your local machine. Stage, commit and then try to push them (*after* pulling from the GitHub repo first).
- Did **Student 1** encounter a `merge conflict` error?
- Good, that's what we were trying to trigger.Let's learn how to fix them.

Merge conflicts

- First, let's confirm what's going on.

```
$ git status
```

- As part of the response, you should see something like:

```
Unmerged paths:
  (use "git add <file>..." to mark resolution)

    * both modified:   README.md
```

- Git is protecting **Student 1** by refusing the merge. It wants to make sure that you don't accidentally overwrite all of your changes by pulling **Student 2's** version of the README.

- In this case, the source of the problem was obvious. Once we start working on bigger projects, however, `git status` can provide a helpful summary to see which files are in conflict.

Merge conflicts (cont.)

- Let's see what's happening here by opening up the README file.
- RStudio is a good choice, although your preferred text editor is fine.
- You should see something like:

```
# README
Some text here.
<<<<<<< HEAD
Text added by Partner 2.
=======
Text added by Partner 1.
>>>>>>> 814e09178910383c128045ce67a58c9c1df3f558.
More text here.
```

Merge conflicts (cont.)

What do these symbols mean?

```
# README
Some text here.
<<<<<<< HEAD
Text added by Partner 2.
=======
Text added by Partner 1.
>>>>>>> 814e09178910383c128045ce67a58c9c1df3f558.
More text here.
```

- `<<<<<<< HEAD` Indicates the start of the merge conflict.
- `=======` Indicates the break point used for comparison.
- `>>>>>>> <long string>` Indicates the end of the lines that had a merge conflict.

Merge conflicts (cont.)

- Fixing these conflicts is a simple matter of (manually) editing the README file.
  - Delete the lines of the text that you don't want.
  - Then, delete the special Git merge conflict symbols.
- Once that's done, you should be able to stage, commit, pull and finally push your changes to the GitHub repo without any errors.

Caveats

- **Student 1** gets to decide what to keep because they fixed the merge conflict.
- The full commit history is preserved, so Student 2 can always recover their changes if desired.
- A more elegant and democratic solution to merge conflicts (and repo changes in general) is provided by Git branches. We'll get there next.

Aside: Line endings and different OSs

Problem

During your collaboration, you may have encountered a situation where Git is highlighting differences on seemingly unchanged sentences.

- If that is the case, check whether your partner is using a different OS to you.

The "culprit" is the fact that Git evaluates an invisible character at the end of every line. This is how Git tracks changes. (More info here and here.)

- For Linux and MacOS, that ending is "LF"
- For Windows, that ending is "CRLF" (of course it is...)

Aside: Line endings and different OSs

Solution

Open up the shell and enter

```
$ git config --global core.autocrlf input
```

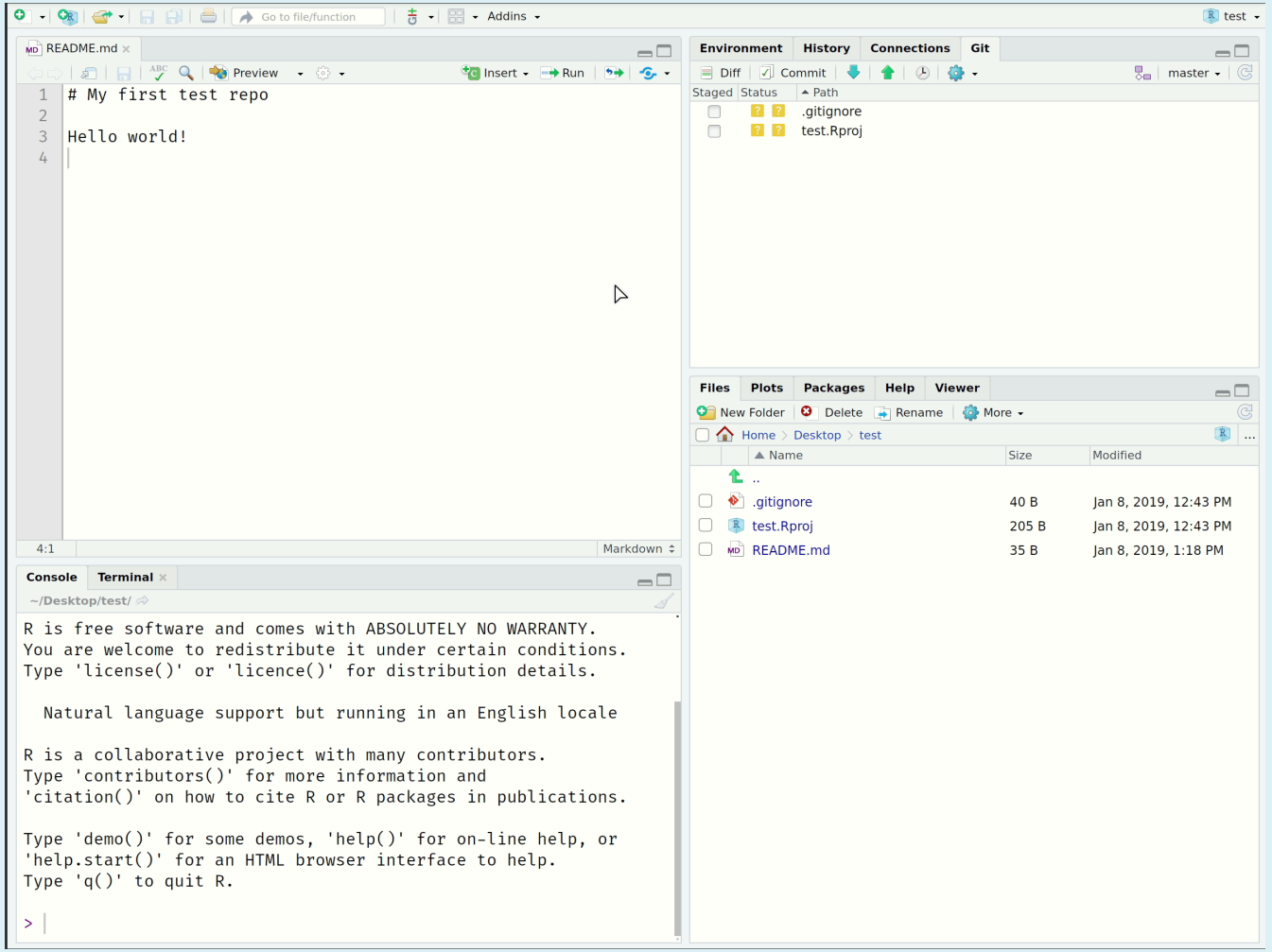
(Windows users: Change `input` to `true`).

Branches and forking

What are branches and why use them?

- **Branches** are one of Git's coolest features.
- Allow you to take a snapshot of your existing repo and try out a whole new idea *without affecting* your main (i.e. "master") branch.
- Only once you (and your collaborators) are 100% satisfied, would you merge it back into the master branch.
    - This is how most new features in modern software and apps are developed.
    - It is also how bugs are caught and fixed.
    - But researchers can easily — and should! — use it to try out new ideas and analysis (e.g. robustness checks, revisions, etc.)
- If you aren't happy, then you can just delete the experimental branch and continue as if nothing happened.

Create a new branch in RStudio



Branch shell commands

- Create a new branch on your local machine and switch to it:

```
$ git checkout -b NAME-OF-YOUR-NEW-BRANCH
```

- Push the new branch to GitHub:

```
$ git push origin NAME-OF-YOUR-NEW-BRANCH
```

- List all branches on your local machine:

```
$ git branch
```

Branch shell commands(cont.)

- Switch back to (e.g.) the master branch:

```
$ git checkout master
```

- Delete a branch

```
$ git branch -d NAME-OF-YOUR-FAILED-BRANCH
$ git push origin :NAME-OF-YOUR-FAILED-BRANCH
```

Merging branches + Pull requests

You have two options:

1. Locally

- Commit your final changes to the new branch (say we call it "new-idea").
- Switch back to the master branch: `$ git checkout master`
- Merge in the new-idea branch changes: `$ git merge new-idea`
- Delete the new-idea branch (optional): `$ git branch -d new-idea`

2. Remotely (i.e. *pull requests* on GitHub)

- Pull requests are a way to notify collaborators—or yourself—that you've finished a feature and want it reviewed or merged.
- You write a summary of all the changes contained in the branch.
- You then assign suggested reviewers of your code — including yourself potentially — who are then able to approve these changes ("Merge pull request") on GitHub.
- Let's practice this now in class...

Your first pull request

You know that "new-idea" branch we just created a few slides back? Switch over to it if you haven't already.

- Remember: `$ git checkout new-idea` (or just click on the branches tab in RStudio)

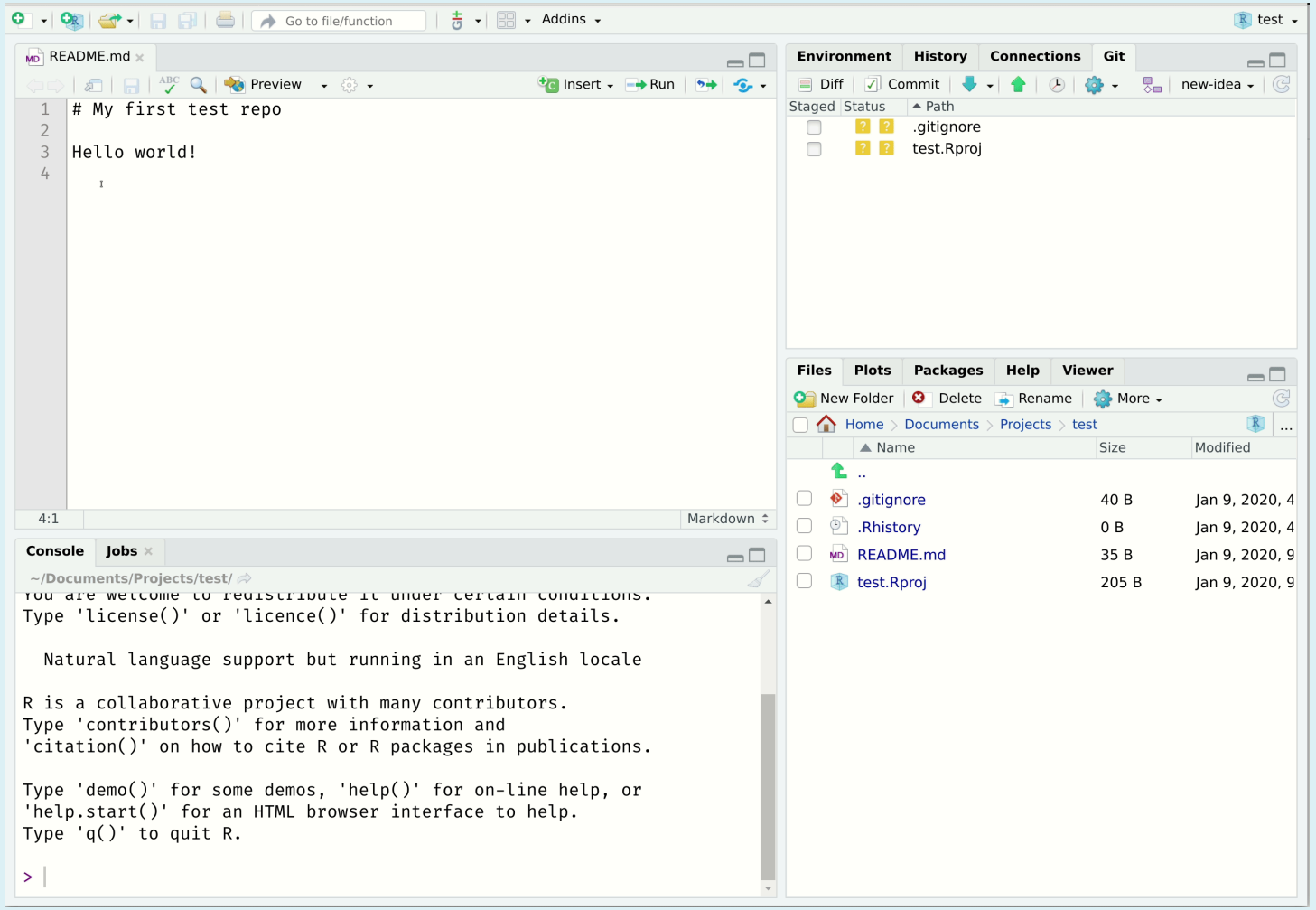
Make some local changes and then commit + push them to GitHub.

- The changes themselves don't really matter. Add text to the README, add some new files, whatever.

After pushing these changes, head over to your repo on GitHub.

- You should see a new green button with "Compare & pull request". Click it.
- Add a meta description of what this PR accomplishes. You can also change the title if you want.
- Click "Create pull request".
- (Here's where you or your collaborators would review all the changes.)
- Once satisfied, click "Merge pull request" and then confirm.

Your first pull request (cont.)



Forks

Git forks lie somewhere between cloning a repo and branching from it.

- In fact, if you fork a repo then you are really creating a copy of it.

Forking a repo on GitHub is very simple; just click the "Fork" button in the top-right corner of said repo.

- This will create an independent copy of the repo under your GitHub account.
- Once you fork a repo, you are free to do anything you want to it. (It's yours.) However, forking — in combination with pull requests — is actually how much of the world's software is developed. For example:
- Outside user *B* forks *A*'s repo. She adds a new feature (or fixes a bug she's identified) and then issues an upstream pull request.
- *A* is notified and can then decide whether to merge *B*'s contribution with the main project.

Forks (cont.)

Creating forks is super easy as we've just seen. However, maintaining them involves some more leg work if you want to stay up to date with the original repo.

- GitHub: "Syncing a fork"
- This isn't going to be an issue for completed projects. E.g. Forking the repo that contains the code and data of a published paper.

OSS contribution

Remember that "OSS contribution" component of the course (i.e. 10% of your final grade)? Well, now is a good time to tell you that forks, branches, and pull requests are effectively what I will be expecting of you.

- Grades aside, I want to encourage you to start thinking about contributing to software projects in general.
- Seriously, it can be something as simple as correcting typos or language. Many great programmers and data scientists are not English first-language speakers. Helping to improve package documentation is a small way to say thanks. (More here.)

Other tips

README

README files are special in GitHub because they act as repo landing pages.

- For a project tied to a research paper, this is where you should be explicit about the goal of the research paper, the software requirements, how to run the analysis, and so forth (e.g. here).
- On the other end of the scale, many GitHub repos are basically standalone README files. Think of these as version-controlled blog posts (e.g. here).

README files can also be added to the *sub-directories* of a repo, where they will act as a landing pages too.

- Particularly useful for bigger projects. Say, where you are using multiple programming languages (e.g. here), or want to add more detail about a dataset (e.g. here).

READMEs should be written in Markdown, which GH automatically renders.

- We'll learn more about Markdown (and its close relation, R Markdown) during the course of our homework assignments.

.gitignore

A .gitignore file tells Git what to — *wait for it* — ignore.

This is especially useful if you want to exclude whole folders or a class of files (e.g. based on size or type).

- Proprietary data files should be ignored from the beginning if you intend to make a repo public at some point.
- Very large individual files (>100 MB) exceed GitHub's maximum allowable size and should be ignored regardless. See here and here.

I typically add compiled datasets to my .gitignore in the early stages of a project.

- Reduces redundant version control history, where the main thing is the code that produces the compiled dataset, not the end CSV in of itself. ("Source is real.")
- Simple to remove from my .gitignore once the project is being finalised (e.g. paper is being submitted).

.gitignore (cont.)

You can create a .gitignore file in multiple ways.

- A .gitignore file was automatically generated if you cloned your repo with an RStudio Project.
- You could also have the option of adding one when you first create a repo on GitHub.
- Or, you can create one with your preferred text editor. (Must be saved as ".gitignore".)

Once the .gitignore file is created, simply add in lines of text corresponding to the files that should be ignored.

- To ignore a single a file: `FILE-I-WANT-TO-IGNORE.csv`
- To ignore a whole folder (and all of its contents, subfolders, etc.): `FOLDER-NAME/**`
- The standard shell commands and special characters apply.
  - E.g. Ignore all CSV files in the repo: `*.csv`
  - E.g. Ignore all files beginning with "test": `test*`
  - E.g. Don't ignore a particular file: `!somefile.txt`

Summary

Recipe (shell commands in grey)

1. Create a repo on GitHub and initialize with a README.
2. Clone the repo to your local machine. Preferably using an RStudio Project, but as you wish. (E.g. Shell command: `$ git clone REPOSITORY-URL`)
3. Stage any changes you make: `$ git add -A`
4. Commit your changes: `$ git commit -m "Helpful message"`
5. Pull from GitHub: `$ git pull`
6. (Fix any merge conflicts.)
7. Push your changes to GitHub: `$ git push`

Repeat steps 3—7 (but especially steps 3 and 4) often.

Homework

- Find a way to access a **stable and reliable internet** connection.
- Apply the **educational account on GitHub** if you haven't already.
- Then send your Github ID to out TA.