Introduction to git and knitr - with Style

Jared Bennett

02 September, 2019

git Basics

Learning Objectives

- Create a GitHub repository
- Create a local Git repository
- Practice adding, and committing changes to your (local) Git repo
- Practice pushing commited changes to a remote repo

0) Useful Links

- A nice tutorial is available on the Berkeley SCF github repo
- Show git branch in command prompt
- Setup git password manager
  - Simple Answer
  - Official Documentation

1) Create a New GitHub Repository

There are two ways to start a repository:

- create the repository on Github using your browser and then use `git clone`
- use `git init` on your machine and then linking it to a remote server e.g. github.

We're going to cover creating one online:

- Open your browser and Sign in to your github account.
- Locate the `+` button (next to your avatar).
- Select the `New repository` option.
- Choose a name for your repository: e.g. `demo-repo`.
- In the **Description** field add a brief description: e.g. "this is a demo repo"
- add a .gitignore for R files.
- Click the green button **Create repository**.

2) Adding a README file

Initially, your repo is located on GitHub. To set it up locally, you must clone the repository from GitHub.

```
git clone https://github.com/berkeley-scf/tutorial-git-basics
```

It is customary to add a `README.md` file at the top level. This file must contain (at least) a description of what the repository is about. The following command will create a `README.md` file with some minimalist content:

```
echo "# Demo Repo" >> README.md
```

So far there you have a "new" file in your local repo, but this change has not been recorded by Git. You can confirm this by checking the status of the repo:

```
git status
```

Notice that Git knows that `README.md` is untracked. So let's add the changes to Git's database:

```
git add README.md
```

Check the status of the repo again:

```
git status
```

Now Git is tracking the file `README.md`. Next thing consists of **committing** the changes

```
git commit -m "Create README"
```

3) Pushing changes to a remote repo

Now that you have linked your local repo with your remote repo, you can start pushing (i.e. uploading) commits to GitHub. As part of the basic workflow with git and github, you want to constantly check the status of your repo

```
git status
```

Now let's push your recent commit to the remote branch (`origin`) from the local branch (`master`):

```
git push origin master
```

Go to your Github repository and refresh the browser. If everything went fine, you should be able to see the contents of your customized `README.md` file.

knitr and R Markdown Files

Learning Objectives:

- Differentiate between `.R` and `.Rmd` files
- To understand dynamic documents
- To gain familiarity with R Markdown `.Rmd` files
- To gain familiarity with code chunks

0) Useful Links

- Dynamic docs
- Knitr in a knutshell
- R Markdown cheatsheet
- Complete Guide

1) Opening and knitting an `Rmd` file

In the menu bar of RStudio, click on **File**, then **New File**, and choose **R Markdown**. Select the default option (Document), and click **Ok**. RStudio will open a new `.Rmd` file in the source pane. And you should be able to see a file with some default content.

Locate the button **Knit HTML**, the one with an icon of a ball of yarn and two needles. Click the button (knit to HTML) so you can see how `Rmd` files are rendered and displayed as HTML documents. Alternatively, you can use a keyboard shortcut: in Mac `Command+Shift+K`, in Windows `Ctrl+Shift+K`

2) What is an `Rmd` file?

Rmd files are a special type of file, referred to as a *dynamic document*. This is the fancy term we use to describe a document that allows us to combine narrative (text) with R code in one single file.

Rmd files are plain text files. This means that you can open an Rmd file with any text editor (not just RStudio) and being able to see and edit its contents.

The main idea behind dynamic documents is simple yet very powerful: instead of working with two separate files, one that contains the R code, and another one that contains the narrative, you use an `.Rmd` file to include both the commands and the narrative.

One of the main advantages of this paradigm, is that you avoid having to copy results from your computations and paste them into a report file. In fact, there are more complex ways to work with dynamic documents and source files. But the core idea is the same: combine narrative and code in a way that you let the computer do the manual, repetitive, and time consuming job.

Rmd files is just one type of dynamic document that you will find in RStudio. In fact, RStudio provides other file formats that can be used as dynamic documents: e.g. `.Rnw`, `.Rpres`, `.Rhtml`, etc.

3) Anatomy of an `Rmd` file

The structure of an `.Rmd` file can be divided in two parts: 1) a **YAML header**, and 2) the **body** of the document. In addition to this structure, you should know that `.Rmd` files use three types of syntaxes: YAML, Markdown, and R.

The *YAML header* consists of the first few lines at the top of the file. This header is established by a set of three dashes `---` as delimiters (one starting set, and one ending set). This part of the file requires you to use YAML syntax (Yet Another Markup Language.) Within the delimiter sets of dashes, you specify settings (or metadata) that will apply to the entire document. Some of the common options are things like:

- `title`
- `author`
- `date`
- `output`

The *body* of the document is everything below the YAML header. It consists of a mix of narrative and R code. All the text that is narrative is written in a markup syntax called **Markdown** (although you can also use LaTeX math notation). In turn, all the text that is code is written in R syntax inside *blocks of code*.

There are two types of blocks of code: 1) **code chunks**, and 2) **inline code**. Code chunks are lines of text separated from any lines of narrative text. Inline code is code inserted within a line of narrative text .

4) How does an Rmd file work?

Rmd files are plain text files. All that matters is the syntax of its content. The content is basically divided in the header, and the body.

- The header uses YAML syntax.
- The narrative in the body uses Markdown syntax.
- The code and commands use R syntax.

The process to generate a nice rendered document from an Rmd file is known as **knitting**. When you *knit* an Rmd file, various R packages and programs run behind the scenes. But the process can be broken down in three main phases: 1) Parsing, 2) Execution, and 3) Rendering.

1) Parsing: the content of the file is parsed (examined line by line) and each component is identified as yaml header, or as markdown text, or as R code.

Each component receives a special treatment and formatting.

The most interesting part is in the pieces of text that are R code. Those are separated and executed if necessary. The commands may be included in the final document. Also, the output may be included in the final document. Sometimes, nothing is executed nor included.

Depending on the specified output format (e.g. HTML, pdf, word), all the components are assembled, and one single document is generated.

5) Yet Another Syntax to Learn

R markdown (`Rmd`) files use markdown as the main syntax to write content.Markdown is a very lightweight type of markup language, and it is relatively easy to learn.

One of the most common sources of confusion when learning about R and Rmd files has to do with the hash symbol `#`. As you know, `#` is the character used by R to indicate comments. The issue is that the `#` character has a different meaning in markdown syntax. Hashes in markdown are used to define levels of headings.

In an Rmd file, a hash `#` that is inside a code chunk will be treated as an R comment. A hash outside a code chunk, will be treated as markdown syntax, making its associated text a given type of heading.

6) Code chunks

There are dozens of options available to control the executation of the code, the formatting and display of both the commands and the output, the display of images, graphs, and tables, and other fancy things. Here's a list of the basic options you should become familiar with:

- `eval`: whether the code should be evaluated
  - `TRUE`
  - `FALSE`
- `echo`: whether the code should be displayed
  - `TRUE`
  - `FALSE`
  - numbers indicating lines in a chunk
- `error`: whether to stop execution if there is an error
  - `TRUE`
  - `FALSE`
- `results`: how to display the output
  - `markup`
  - `asis`
  - `hold`
  - `hide`
- `comment`: character used to indicate output lines
  - the default is a double hash `##`
  - `""` empty character (to have a cleaner display)

Additionally, you can include inline code within your work. If you're describing results, you don't want to hard-code a number or the amount of repititions you ran. Instead, include variables or short functions as **`r 2 + 2`**, which is rendered as 4.

7) LaTeX

Rmarkdown files render LaTeX through an external generator. This means that you can write any math equations or LaTeX syntax within a specific chunk, and install the required LaTeX libraries outside of R, and it will be rendered properly.

Inline code chunks are setoff with single dollar signs, ie `$\beta$` is rendered as β. This is great for small equations, Greek letters, and references to variables.

LaTeX chunks can also be significantly more complicated. Independent chunks are setoff with double dollar signs, ie `$$ Complex LaTeX Thing $$`, such as the following equation:

$$D(\theta_l, T_x) = \begin{cases} \theta^{'}_{l[0]} = \theta_l & i = 0 \\ \theta^{'}_{l[i+1]} = \theta^{'}_{l[i]} * F(\overline{L_{[t-i-T_x]}}) & i \leq T_l \end{cases}$$

Submitting problem sets

Learning Objectives:

- Ensure repository is setup correctly
- Discuss Code Styles
- Turning in homework

0) Useful Links

- Homework Submission
- Hadley Wickham Style Guide
- Google's R Style Guide
- Weird One with Links

You don't need to follow the exact style of any of those - use your own judgment and figure out what style you like and be consistent in using that style. But you should do the following:

- use white space to make it easier to read your code
- have your code lines be no more than 80 characters
- give your objects and functions meaningful (and not overly long) names
- comment your code
- indent your code as needed so one can see what lines of code go together in a block

You should NOT include periods in names of objects (this contradicts Google's style guide). The reason is that periods are used to mean something specific in R's S3 object oriented programming syntax (e.g., `predict.lm`) and that periods are used in other languages specifically for object-oriented syntax. So I'd suggest either `calculate_mle` or `calculateMLE`, not `calculate.mle`.