

# Introduction to Data Analysis in R

## Module 4: Dynamic documents and project management

Andrew Proctor

[andrew.proctor@phdstudent.hhs.se](mailto:andrew.proctor@phdstudent.hhs.se)

February 11, 2019



- 1 Intro
- 2 RMarkdown documents
- 3 RStudio Projects
- 4 Version Control



```
library(tidyverse)
```

```
## -- Attaching packages -----
```

```
## v ggplot2 3.1.0      v purrr  0.3.0  
## v tibble  2.0.1      v dplyr  0.7.8  
## v tidyr   0.8.2      v stringr 1.4.0  
## v readr   1.3.1      v forcats 0.3.0
```

```
## -- Conflicts -----
```

```
## x dplyr::filter() masks stats::filter()  
## x dplyr::lag()     masks stats::lag()
```

```
library(rio)  
library(rprojroot)  
library(here)
```



# Intro



# Goals for Today

- Learn how to create dynamic R documents in a variety of formats using knitr and RMarkdown
- Learn how to how manage the project structure using RStudio projects.
- Learn how to perform local and online online version control using Git and GitHub.



# RMarkdown documents



# Reproducible R Reports

- So far, we have been working purely with basic “R Script” files, which are very similar to Stata do-files.
- But thanks largely to the [knitr package](#), you can easily create reports that interweaves text and R code in a neatly structured manner.
- Output can be structured as PDF documents, HTML webpages, Word documents, or various presentation formats including Beamer (LaTeX) presentations.
- The course website, lecture slides, and exercise instructions have all been generated in R.



# Getting started

- Reports of different file formats are generated using the [knitr package](#).
- Before installing knitr, make sure sure that you have a Latex distribution installed.
- Then install the knitr package and initialize it in the usual manner.

```
# Run only once (ie in the console)  
  # install.packages("knitr")  
  
# Initialize library  
library("knitr")
```





# Knitr and RMarkdown

Knitr allows for the creation of documents structured using two different typesetting languages:

- LaTeX with the .RNW file
- Markdown (specifically *RMarkdown*), which was originally created as a simple language for structuring HTML markup.

For this course, we will focus on the **RMarkdown** format, which has become the dominant method for “knitting” document because of it’s lightweight and flexibility.

- More information about how to generate R reports using the Latex format can be found at  
<https://rpubs.com/YaRrr/SweaveIntro>.



# Creating an RMarkdown document

- After installing knitr, to create an RMarkdown document, go to **File—New File—R Markdown**.
- A popup shows up to ask enter the document Title and Author, as well as what type of document you want to create.

New R Markdown

Document  
Presentation  
Shiny  
From Template

Title:

Author:

Default Output Format:

☒ HTML  
Recommended format for authoring (you can switch to PDF or Word output anytime).

☐ PDF  
PDF output requires TeX (MiKTeX on Windows, MacTeX 2013+ on OS X, TeX Live 2013+ on Linux).

☐ Word  
Previewing Word documents requires an installation of MS Word (or Libre/Open Office on Linux).

OK Cancel



# Writing and Code in RMarkdown

In RMarkdown, expository writing and code “chunks” are differentiated in writing code in specific code chunks.

```
```{r}
# Here is an example of a code chunk
2 +2
```
```

When you create the document, that statement gets evaluated as:

```
# Here is an example of a code chunk
2 +2
```

```
## [1] 4
```



# Inline Chunks

You can also include inline code by using initializing with a backtick and the letter r (no space between), writing the code, then closing the chunk with another backtick.

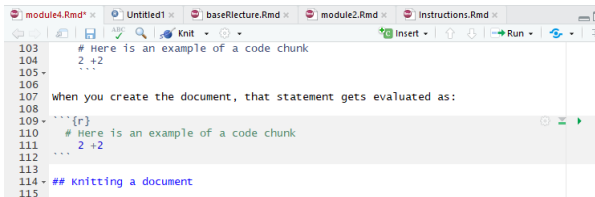
- For example: ``r 2+2``



# Knitting a document

To generate a document in the desired output format from a RMarkdown document, you need to **“Knit”** the document, which appears as a clickable icon on the menu atop the script pane.

You do not need to Knit a document after every change, however. You can just as easily run the code chunks. There are specific specific buttons to run either the current chunk or all of the chunks above a given chunk.



The screenshot shows the RStudio interface with a RMarkdown document open. The script pane on the left contains the following code:

```
103 # Here is an example of a code chunk
104 2 + 2
105 ...
106
107 when you create the document, that statement gets evaluated as:
108
109 ```{r}
110 # Here is an example of a code chunk
111 2 + 2
112 ...
113
114 ## Knitting a document
115
```

The code chunk starting at line 109 is highlighted in grey. The toolbar at the top of the script pane includes a 'Knit' button (a document icon with a double arrow) and a 'Run' button (a green play icon). The 'Run' button is currently selected.



# Writing outside of code chunks

Anything not written inside of these bacted sections is interpret as normal writing.

RMarkdown makes styling your writing particularly easy. Some common formatting options include:

- **Headers:** Headers are defined using hashes (#)
  - A single # indicates a top level heading (and bigger font), while each additional hash indicates a smaller heading size
  - So while # is the largest heading size, #### is a small heading
- **Bold:** To bold text, wrap it in two asterisks: **\*\*Bold Statement\*\***
- **Italics:** To italicize text, wrap in a single asterisk: *\*Italics Statement\**



# Lists and Latex Input

- **Latex input:** Most LaTeX commands (except for generally longer multi-line structures) can be included in RMarkdown documents just as you'd write them in Tex document.
- **Lists/Bullet Points:** Like the bullet points here, you will often want to structure output using lists.
  - To create a bulleted list, start each bulleted line with a dash (-).
  - Make sure to leave an empty line between the start of the list and any other text.
  - To make an indent “sub-list”, start the sub-list with a plus sign (+) and use tab to indent the line twice for each of the sub-list items.



# Ordered Lists

- Ordered lists use the same indent rules as unordered lists, but with no dashes or plus signs.
  - You can also generally uses automatic numbering by repeating the initial letter or number (e.g.)

```
---  
title: "Example"  
output: pdf_document  
---  
  
1. Here is  
1. A list with  
1. Automatic numbering  
  
  a. I could write stuff here  
  
    i. More stuff here  
    i. And here  
    i. And here  
  
  a. Blah blah  
  a. Data science
```

1. Here is
2. A list with
3. Automatic numbering
  - a. I could write stuff here
    - i. More stuff here
    - ii. And here
    - iii. And here
  - b. Blah blah
  - c. Data science





# Seperating Lines in RMarkdown

- Something you might wonder is how to obey the RStudio 80-character margins while allowing your text to wrap normally in the generated documents.
- The answer lies in how new lines are treated in RMarkdown documents.
  - If the line ends with one space or less, a new line in RMarkdown will not be treated as a new line in the documents generated.



## Code chunk options

There are several output options you can specify for how R code and the code output are expressed in reports. These options are expressed as options in the `{r}` declaration at the top of the chunk.

- **echo=FALSE**: do not show the R code itself (but potentially the code output depending on other chunk options).
- **include=FALSE**: do not show the R code or the output in the document.
- **eval=FALSE**: do not actually execute the code inside the chunk, only display it.
- **results=hide**: run the code but do not show the results output.
- **warning=FALSE** / **message=FALSE**: do not show warnings or messages associated with the R code.



# Output options

When a RMarkdown document is generated in a given output format, there are several common things you can do to customize the appearance of the document.

To add options to your document, indent the name of the output type to an indented new line and a colon to it. Then indent under the output type and add the desired options.

```
---  
title: "Example"  
author: "Andrew Proctor"  
output: html_document  
---
```

```
---  
title: "Example"  
author: "Andrew Proctor"  
output:  
  html_document:  
    theme: flatly  
    df_print: kable  
    highlight: haddock  
    toc: yes  
---
```



# Common output options

Here are a few common options:

- **table of contents:** to include a table of contents in your document, use the **toc: yes** option.
- To change the way data frame output is printed, use the **df\_print:** option. Good options are **kable** or **tibble**.
- To add code highlighting for R chunks, use the **highlight:** option.
  - **Options include:** default, tango, pygments, kate, monochrome, espresso, zenburn, haddock, and textmate.
- You can also specify output themes for html documents and beamer presentations. For html documents, possible themes are listed [here](#) while beamer themes are typically supplied by .sty files in your project folder.



# Working Directories in RMarkdown

In RMarkdown documents, the working directory is automatically set to the folder in which the RMarkdown document is saved.

- From there, you can use *relative file paths*. If data etc is in the root of the project folder, then just refer to the file name directly.
- If data is in a subfolder, eg *data*, use a relative path like:

```
import(./data/mydata.rds)
```



# R Notebooks

Aside from the standard RMarkdown documents that we've covered so far, another format worth mentioning is the **R Notebook** format.

- R Notebooks essentially adapt html RMarkdown documents to be even more similar to something like Jupyter Notebooks.
- With R Notebooks, you can ***Preview*** documents without knitting them over again.
- The document also generally has the Notebook-style code-output-code layout.



# RStudio Projects



# Projects Intro

In addition to using RMarkdown documents to make your scripts and reports more compelling, another process upgrade is using **RStudio Projects**.

Projects are useful because they:

- Define a project root folder
- Save a RStudio environment that is unique to each project
- Allow for easy version control





# Working folder benefits of a Project

- A project root folder is not only preferable to the need to use **setwd()**, but also to the default working directory used in RMarkdown documents outside of R Projects. Why?
- Because for substantial research projects, you likely will have a lot of files that you split into different subfolders, one of which is probably something like **code**.
  - In this case, you'd need to use somewhat convoluted relative file paths to indicate that the paths should be from the parent folder of **code**.



# Using RStudio Projects

To create a RStudio Project, go to **File – New Project**. From there, you can choose whether to create a new directory, use an existing directory, or use a Version Control repository.

- In practice, I'd suggest you use either a **New Directory** or **Version Control** depending on whether or not you want to sync your scripts to GitHub.
  - We'll go over version control shortly.

Once you have created a Project, you can either open it from the File Menu or by opening the .RProj file in the project directory root.



# Project workflow structure

## Sample Folder Structure:

- code/
  - data\_prep/
  - analysis/
- data/
  - raw\_data/
  - derived\_data/
- docs/
  - report/
  - presentation/
- images/
- results/
  - tables/
  - figures/



# Some workflow management packages:

The following packages can help with setting up workflow, with predefined structures and integration of other useful tools, such as caching, versioning, and logging.

- [rrtools](#)
- [workflowr](#)
- [ProjectTemplate](#)
- [represtoools](#)



# Reproducibility and the R environment

- A concern with any type of analysis project is that over time, the analysis environment can change – making it harder to reproduce results.
  - The most common concern is that packages may change or become obsolete
  - But also the program itself (R) can change, the OS can change, etc. All potentially leading to the inability to reproduce results.



# Managing the R environment

- A solution to evolving package ecosystems built-in to R Projects is [packrat](#).
  - **packrat** can create a package library specific to the individual project.
- A more robust reproducibility solution is with [Docker](#), which creates “containers” in which not only packages are fixed, but also the software (and even the virtual machine).



# Version Control



# What is version control?

Version control is a means to track changes made to files

Version control allows you to:

- See a history of every change made to files -Annotate changes  
-Allow you to revert files to previous versions





# Local Version Control with Git

The most popular software for managing version control is Git.

- There's a good chance you've at least seen GitHub before, which is an online remote location for storing version control repositories.
- The Git software, however, is equally adept at managing version control on your local computer.

Once Git is installed (and recognized in RStudio), you can use Projects to perform local version control.



# Local Version Control with Git ctd

In **File – New Project – New Directory**, once you have Git installed there is a checkbox that can be selected to enable to “Create a Git repository”.

- A repository is a location that Git uses to track the changes of a file or folder, here the project folder. The Git repository is stored as a folder named “.git” in the project root.
- Creating a Project in this manner with a Git repository will enable version control on your local computer.



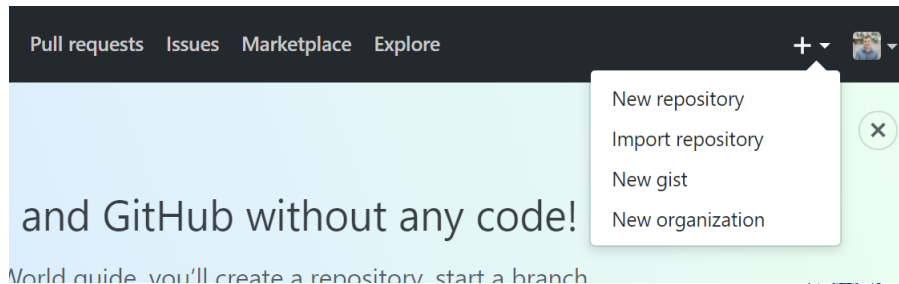
# Remote version control with GitHub

- In addition to local version control, you can also back up the history of file changes to online repositories using GitHub.
- GitHub connects to the local Git repository on your computer, “pushing” and “pulling” changes between the local and remote repositories.
- This also allows for easy collaboration on coding projects, as multiple person can sync to files by connecting to the remote repository.



# Using GitHub for remote version control

With a GitHub account, you can create a new online repository by clicking the “+” icon in the top right of a GitHub page, and then clicking “New Repository”.



# Setting up a new repository

From there, you need to:

- Supply GitHub with a repository name (think folder name)
- Choose whether or not the repository should be public or private (ie whether or not you want other people to be able to visit your GitHub page and view the repository).
  - If you have a GitHub education account, then Private repositories are free. Otherwise, you'd need a paid GitHub subscription.
- Click on the checkbox to enable “Initialize this repository with a README”.
  - Each repository is required to have a readme file, which you may want to comment but is not strictly necessary. Commenting uses Markdown, which is essentially the same as RMarkdown!



# Using a Remote Repository with GitHub

Once you've created an online repository, **Projects** once again allows you to easily connect RStudio with the repository.

- To setup a project for use with GitHub, create a **New Project** and select **Version Control** instead of **New Directory**.
- From there, simply choose “Git” and then copy the url of the repository from GitHub into RStudio.



# Tracking changes with Git

Once you have a Project setup with version control, the first key component of tracking changes is “Committing” them to the repository

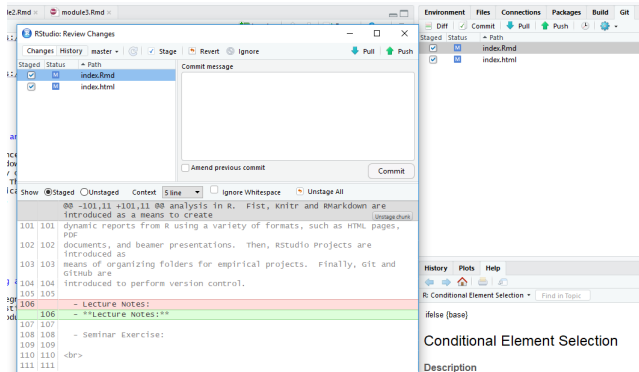
- A “commit” is an update that saves revisions of files into the Git repository.

You can commit changes by going to the “Git” tab in the upper right-hand side of the RStudio IDE.

- In the Git tab, any files that have changed since the last commit are listed. From there, click on the files you’d like to commit and click on the commit button.
- A Commit box appears which shows you the changes since the last revision and asks for a commit message, where you should very briefly describe the changes.



# Tracking changes with Git





# Syncing changes with a remote repository

- If you are just tracking changes with a local repository, commit is sufficient to manage version control.
- But if you are using version control with an remote (ie online) repository, you will two other steps to make sure changes are sync between the local repository and online.
  - To send changes made locally to the online repository, after comitting changes click on “Push.”
  - To sync changes from the online repository to local files, click on “Pull”.



# Viewing previous commits

- To view previous versions of the files (along with annotations supplied with the commit message), click on the clock icon in the Git pane.
- From there, you can see not only a “difference” view of the file changes, but you can also open the document exactly how it was written in a previous commit.
- From there, if you wanted to revert changes, you *could* explicitly revert the file with Git, or simply copy over the file with code from the previous commit — my preferred method of reverting changes.

