Computational Reproducibility

Technology has greatly improved how scientists work. The internet has made it easy to share information – including data, materials, and code – and new software and online platforms have emerged to facilitate the workflow of scientists. One important goal of a scientific workflow is to make sure that the final work you publish is computationally reproducible. **Computational reproducibility** means that when you use the **same data** as in the published article, you can reproduce the **same results**. In other words, if the authors of the published article send you their data and code, you should be able to get the exact same numbers as they report in their article. Current research on the computational reproducibility of scientific articles suggests it is often not possible to run the original code on the data to reproduce results. Sometimes the code will simply not run on the data, or not all analyses are part of the code.

However, computational reproducibility is important, both for other scholars to be able to verify your results, and to build on your results. We could consider computational reproducibility a minimum standard of your own workflow. However, meeting this standard requires training. When I was a PhD, we often had a problem known as 'data rot'. When I submitted an article for publication, and received the reviews after several months, I could not always easily reproduce my own analyses. For example, I might not have stored how I dealt with outliers, and could not exactly reproduce the original results. Sometimes, 'data rot' had eaten away at either my data or my analysis code, and it no longer worked.

Obviously, there is no such thing as 'data rot'. The problem was I did not use a reproducible workflow. In this assignment, we will learn what a computationally reproducible workflow looks like, and how you can share computationally reproducible results with your published paper. The goal in this assignment is for someone else (or for yourself, one year from now) to be able to take your data, run your code, and get exactly the same results as you reported in your work.

Although there are multiple ways to achieve a fully reproducible workflow, in this assignment I aim to introduce you to what I believe might be one emerging standard for a fully reproducible workflow. You learn to work with a version control system (such as GitHub, which integrates nicely with the Open Science Framework) as you are programming in R, which stores previous versions of files. You will then learn how to write a completely reproducible data analysis script (including figures), that you can save as an HTML file or a PDF file, using RMarkdown. Finally, we will take a look at Code Ocean, a novel online platform that allows you to share computationally reproducible

code online, making it extremely easy for others to run (small variations of) your code. You will not learn how to become an experienced programmer in this assignment, but you will see what a fully reproducible workflow would look like, and get some initial experience with tools you will most likely want to explore more in the future.

Getting software and code to work on your system might be a challenge, and regrettably, I can't offer ICT support. Differences between Windows, Linux, and Apple operating systems means you might need to search the internet for solutions to problems you run into – this is very normal, and even experienced programmers do this all the time. If you get stuck, you can check what you did against what the assignment should look like by visiting the public versions of part of this assignment:

GitHub repository: https://github.com/Lakens/reproducibility_assignment

OSF project: https://osf.io/jky8s/

Code Ocean container: https://codeocean.com/capsule/2529779/tree/v1

Step 1: Setting up a GitHub repository

If you haven't created a GitHub account before, do so now. Go to https://github.com/ and create an account. Git is a version control system for tracking changes in computer files and coordinating work on those files among multiple people. Version control allows you to track changes to files and revert back to previous versions if needed. GitHub and GitLab are web-based hosting services that make it easier to use version control with Git. We will be using GitHub because it is what I am most familiar with, and it integrates with slightly more tools, but feel free to use GitLab instead.

If you have an account, you can create a new repository. A **repository** is a collection of folders and files that make up your project. In the top-right of the GitHub page, click the + symbol, and select 'New repository' from the dropdown menu.
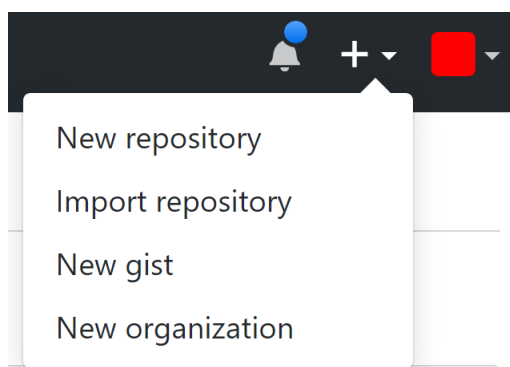
The first thing to do is name your repository. When it comes to naming folders and files, it is important to follow **best practices for file naming**:

- Keep names short, but clear. data_analysis_project is easier to understand for others than dat_an_prjct

- Do not use spaces. Options include:
  - Underscore: this_is_a_file.R (this is my personal favorite)
  - Camelcase: ThisIsAFile.R
  - Dashes: this-is-a-file.R
  - No spaces: thisisafile.R
- If you want to number multiple sequential files, do not use 1_start, 2_end, but use leading zeroes whenever you might number more than 10 files, so for example 01, 02, etc., or 001, 002, etc.
- Do not use special characters such as $#&*{}: in file names.
- If you want to use date information, use the YYYYMMDD format.

Let's name our repository: reproducibility_assignment

Owner		Repository name
🟥 Lakens ▾	/	reproducibility_assignment ✔

You can add a short description (e.g., 'This is an assignment to practice an open and reproducible data analysis workflow'). If you are an academic or student, you can get an academic account, which gives some extra options, such as keeping repositories private: https://education.github.com/pack

Click the checkbox before 'Initialize this repository with a README'. A readme file is a useful way to provide a more detailed description of your project, that will be visible when people visit your GitHub project page.

You are also asked whether you want to add a **license**. Adding a license is a way to easily communicate to others how they can use the data, code, and materials that you will share in your GitHub repository. Note that not making a choice about a license is also a choice: if you do not add a license your work is under exclusive copyright by default, which means others can't easily re-use it. You can learn more about licenses, but for now, a simple choice is the MIT license, which puts only very limited restrictions on reuse, but more restrictive licenses also exist. You can select the choice of license (such as the MIT license) from the dropdown menu. It lets people do anything they want with your code as long as they provide attribution back to you and don't hold you liable. There are also creative commons licenses that you can use when you are sharing something else than software, such as research materials (for example, this educational material is shared under a CC-BY-NC-SA 4.0 license).

We are now ready to create the repository. Click **Create repository**

It might feel unintuitive, but it is important to remember that you are not expected to directly interact with your new GitHub repository through the GitHub website. The repository page will give you information about the contents of the repository, and the history of the files in the repository, but it is not particularly easy to add files or download files directly through the website. The idea is that you use other software to interact with your GitHub repository.

Step 2: Cloning your GitHub repository into RStudio

R Studio can communicate with GitHub. To allow R Studio to work together with GitHub, you first need to set up the system. A detailed explanation for different operating systems is provided here. First, download Git: https://git-scm.com/downloads for your operating system, and install it (you can accept all defaults during the installation process). If you haven't done so already, download and install R: https://cran.r-project.org/, and download and install the free version of R Studio (scroll down for the installers): https://www.rstudio.com/products/rstudio/download/.

In R Studio, go to Tools > Global Options, and select the Git/SVN menu option.
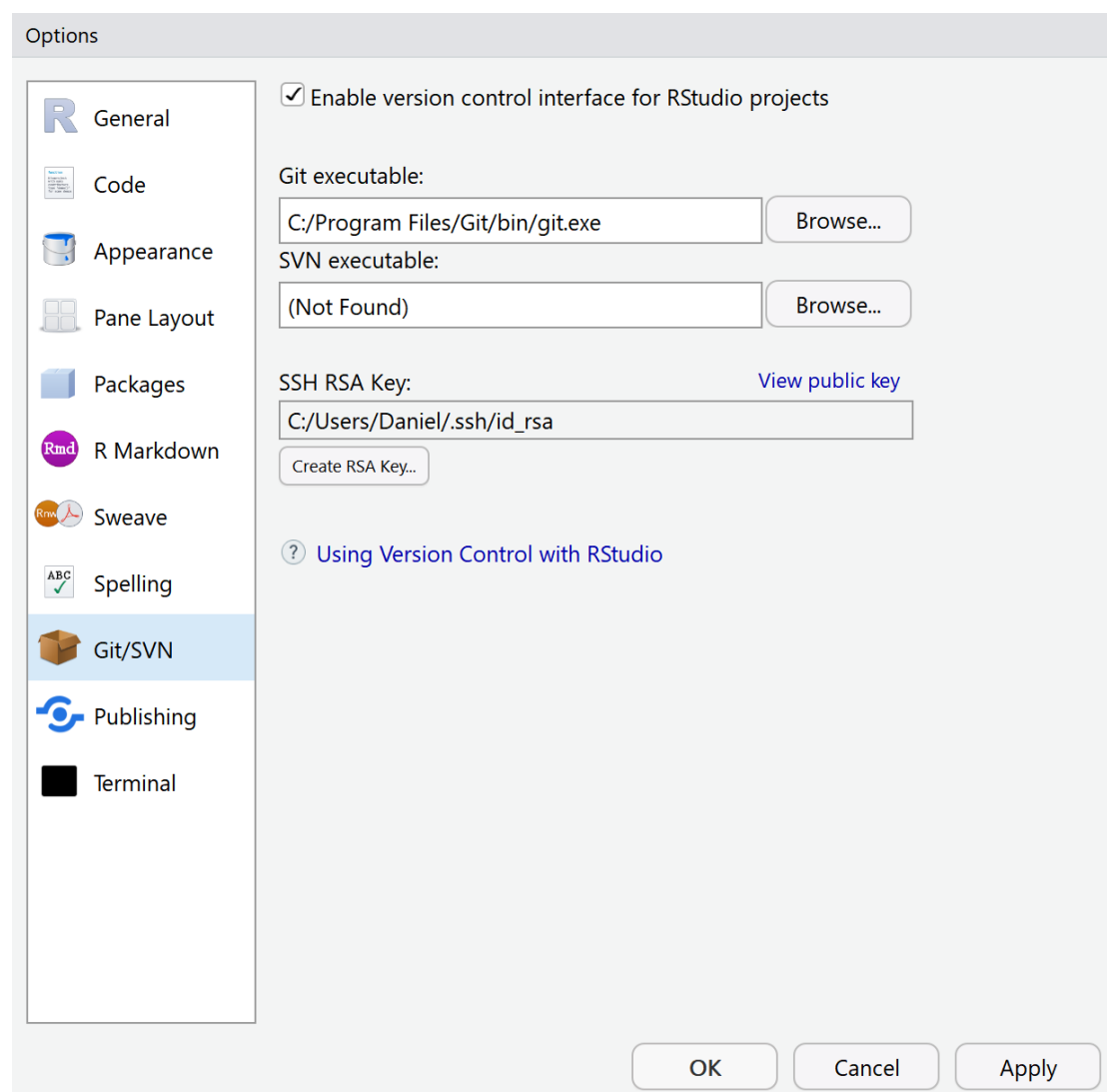
Check if the Git executable ("git.exe") has been found automatically. If not, you will need to click the 'Browse…' button and find it manually. It will always be in the location where you installed Git.

Click the 'Create RSA Key…' button. A window will appear:
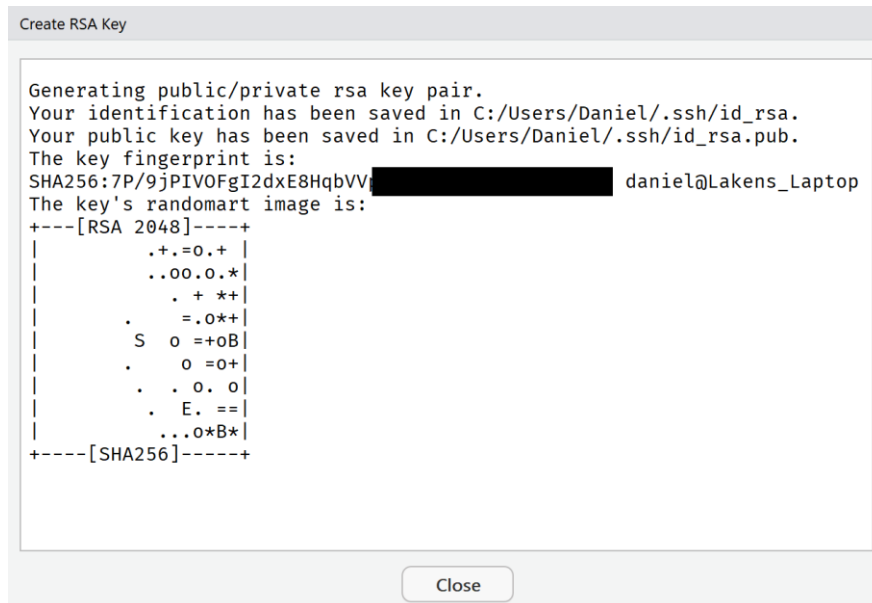
You can close the window. Still under the RStudio options, click the blue hyperlink 'View public key'. A window will appear, telling you that you can use CTRL+C to copy the key. Do so.

Go to GitHub, and got to settings and then select the option SSH and GPG keys:
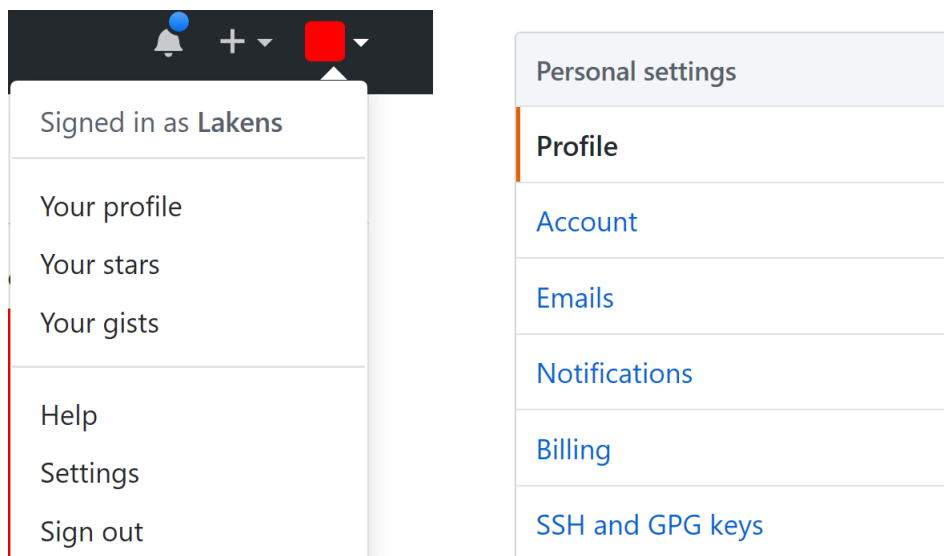
Click 'New SSH key'

Enter a name (e.g., RStudio) and paste the key in the correct window. Click 'Add SSH Key'. This will allow you to send code to from R Studio to your GitHub repositories without having to enter your GitHub login name and password every time. In other words, R Studio is now connected to your GitHub account and repository. You are now ready to create a **version controlled project** in R Studio.
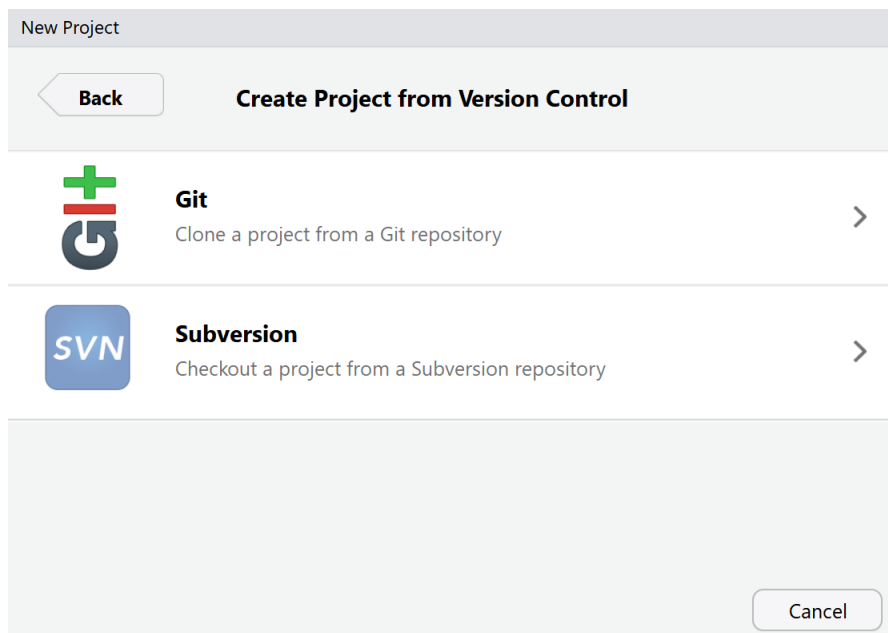
**Restart RStudio**. In RStudio, go to File>New Project:
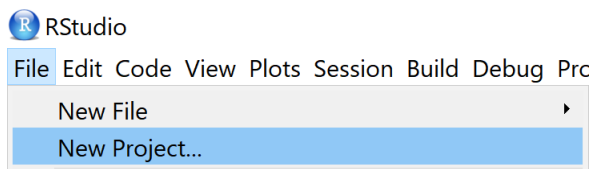
You get three choices. Choose the 'Version Control' option:
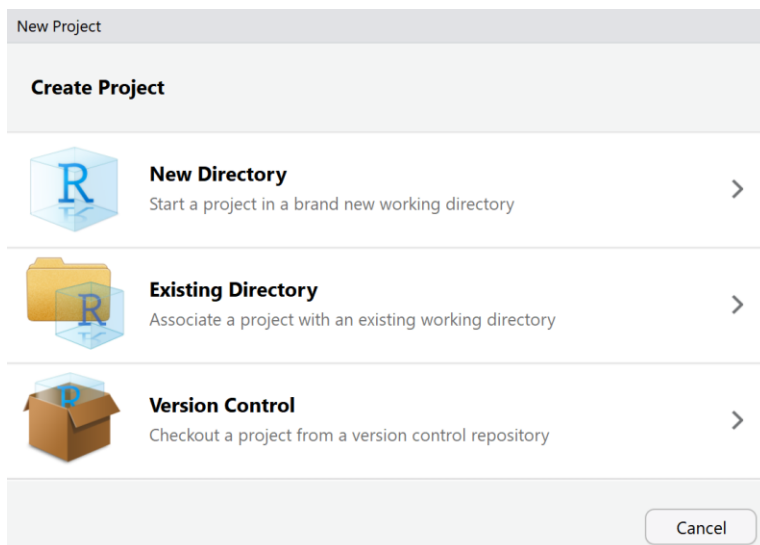

Choose the 'Git' option:

We will be cloning the online GitHub repository we created. Cloning is a term used in Git that means creating a local copy of all files in your repository on your computer. You can copy-paste the URL from your GitHub repository (e.g., https://github.com/Lakens/reproducibility_assignment). If you copy-paste this URL in the top field, it will automatically create a Project directory name that is similar to the name you gave your project on GitHub. You can select a folder on your computer by clicking the 'Browse' button to indicate where you want to save the local copy of your repository.
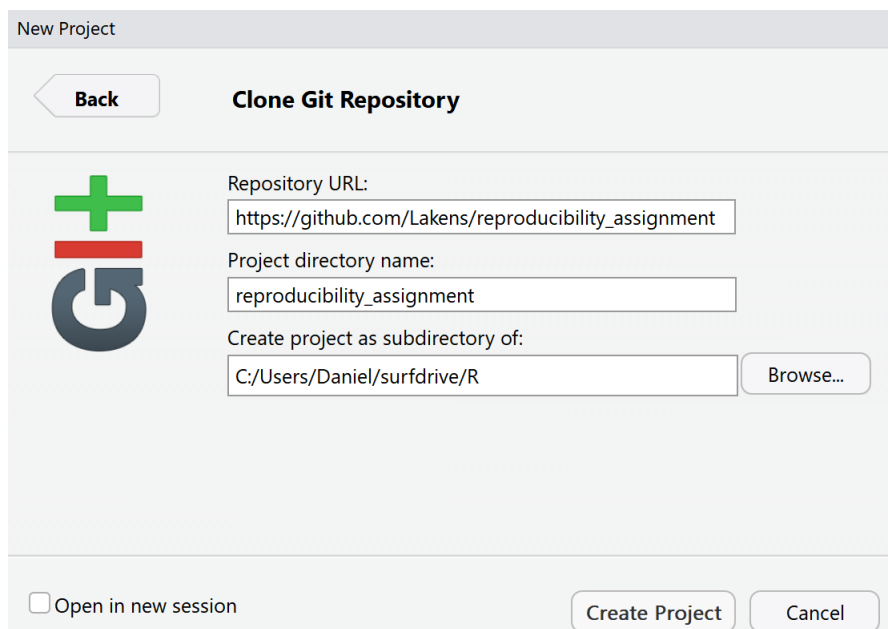

Click 'Create Project'. R will quickly download the files from your repository, and open the new project. You will see that the project creation was successful because the 'Files' tab

in the RStudio interface shows we have downloaded some files from our GitHub repository (the README.md and LICENSE files). RStudio also created a .Rproj file and a .gitignore file. The **project file** is used to store information about the project, and that is required to use GitHub.
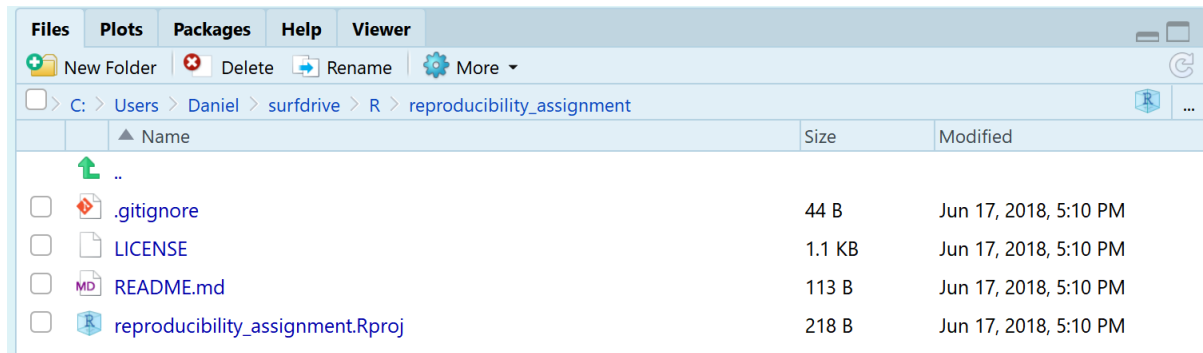
We can also see this is a version control project in the top right of the interface, where there is now a 'Git' tab. If we click it, we see:
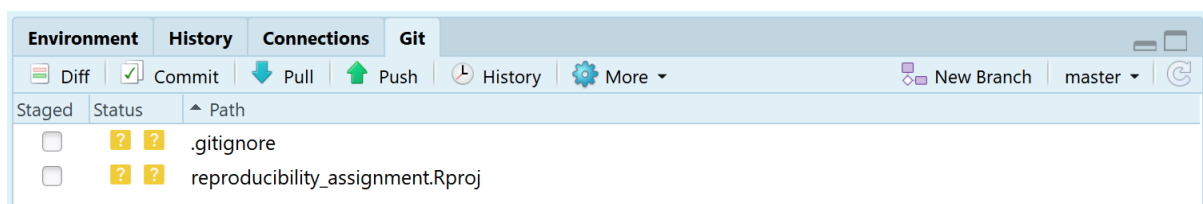
We see a range of buttons, such as the Diff, Commit, Pull, and Push buttons. These will be used to interact with GitHub. Many computer programmers interact with GitHub through the command line, such as:

```
$ git commit -m "This is a git commit message"
```

Learning to use git through the command line is not needed for most people who just want basic version control. Here, I will exclusively focus on version control and git through the menu options in RStudio. It is now time to create a file for which we want to control the versions we make of it.

Step 3: Creating an R Markdown file

R Markdown files provide a way to save and execute code, while at the same time allowing you to create reports of your data analysis (and even full scientific articles that you can submit for publication!). A complete introduction to R Markdown is available here. The main strength of R Markdown documents is that they allow you to create a fully reproducible document. This means that you do not just have some analysis code in an R script, but a manuscript that combines text and code and that you can **compile** to create a PDF or html version of the manuscript. HTML or PDF files have the advantage

that people can read them with regular software. The R Markdown file contains code that performs the analyses **each time the document is compiled**. Instead of copy-pasting values from your analysis software into a word document, you combine code and text in the RMarkdown file to create a manuscript where every number or figure can be traced back to the exact code that generated it. That has the advantage that everyone can use your RMarkdown file and generate the same document (e.g., your manuscript) as you.

You can still make errors in the analysis if you use R Markdown files. The important difference is that you will be making programming errors that are stored in the R Markdown document. Compared to a typo when copying numbers from your analysis to a word document, errors in your analysis in your RMarkdown file will lead to the same document. Because the document is reproducible, all errors are reproducible as well. It is impossible to prevent all errors, but it is possible to make them reproducible. This will make it easier to identify and correct errors. I understand you might worry about others seeing your errors if you allow them to see exactly what you have done. But **we all make mistakes**, and it is important for science to be able to identify and correct these mistakes. An important aspect of moving to a more reproducible workflow, and sharing all files underlying your manuscript publicly, is that we will have to learn to **accept that we all make errors**, and appreciate people who correct them.

Let's start by creating a new R Markdown document in R Studio by clicking New File > R Markdown…
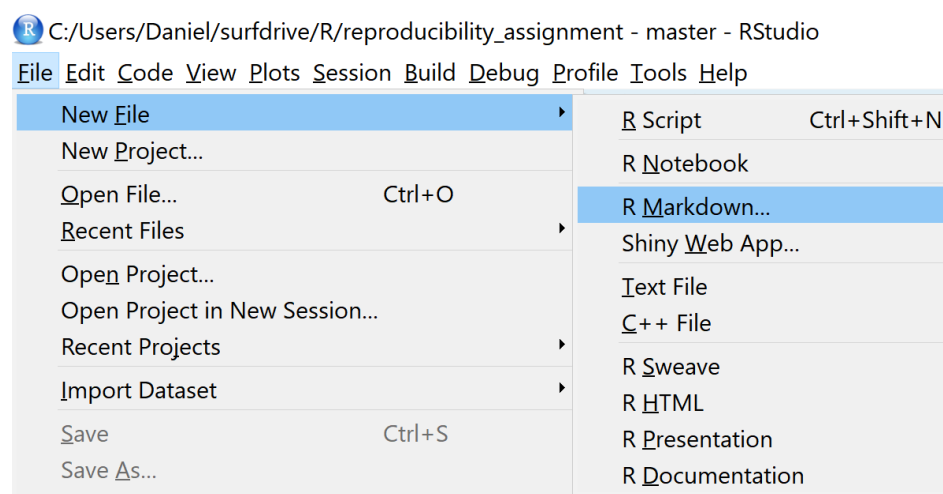
This gives you a new window where you can specify the title of your RMarkdown document and an author name. Enter the title 'Main Analysis', and feel free to change the Author subfield to anything you prefer. RMarkdown files can be compiled (also referred to as 'knitted') into an HTML file, a PDF document, or a word document. To generate PDF files you need to install MiKTex which we won't do for this assignment (a good tutorial how to install MiKTeX is available here). So leave the default output format to HTML and click OK.
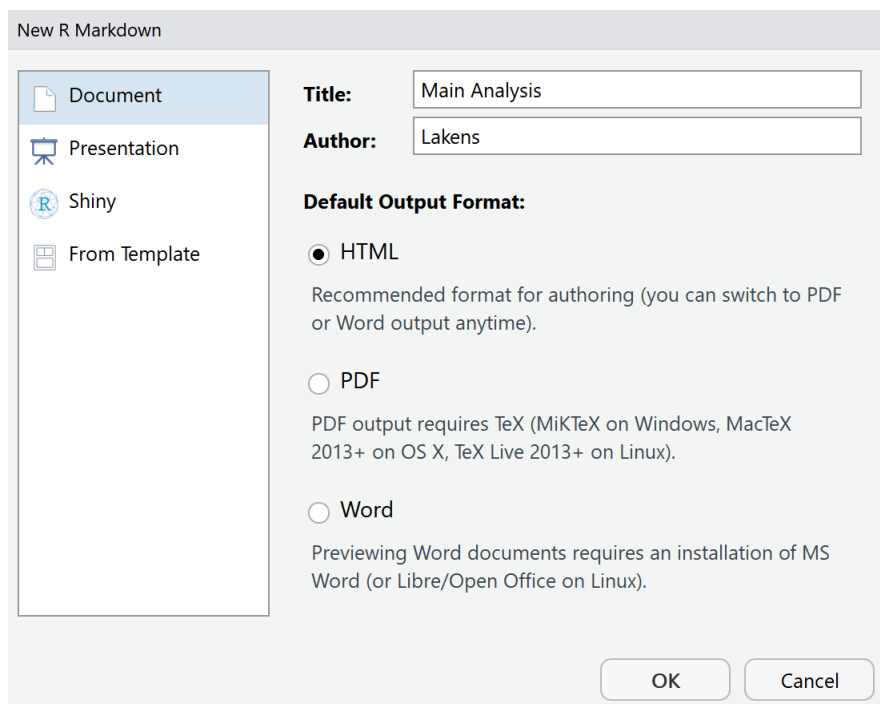
Let's start by saving the new file: Click the save button, and save the file under the name 'main_analysis.Rmd'. Because we are working in an R Studio project, the file will automatically be saved in the same folder as all other files in this project. If you look at the files tab in the bottom right pane, you will see the new file appear. Now let's take a look at the R Markdown file.

The R Markdown file by default includes several sections to get you started. First, there is a header section. In the header section, there is code that determines how the final document is rendered. This section is sensitive, in the sense that it needs to be programmed exactly right – including spaces and tabs – so it is not recommended to change it too much without looking up detailed documentation on how to change this section. If you want the technical details: A R Markdown file is fed to knitr software, which creates a normal markdown file, which then uses pandoc software to generate the specific document you requested. All of this happens automatically.
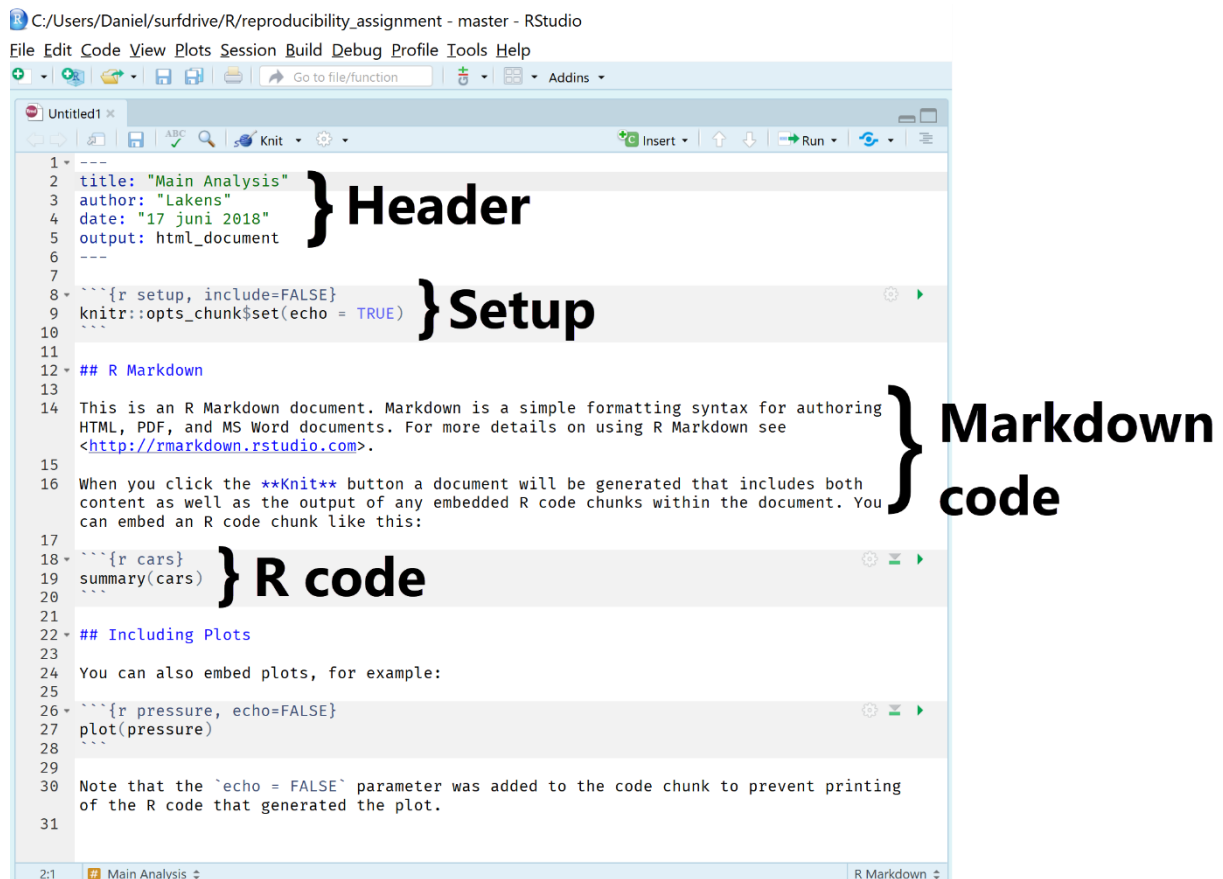
The header is followed by a **set-up** section where you can define general options for the entire R Markdown file. Then, we see the two main sections: **Markdown code**, which is a markup language in plain text formatting syntax that can be easily converted into HTML or other formats. Then, we see **R code** that is used to analyze data or create figures. To see the final result of this code, hit the Knit button in the toolbar at the top of the pane.

Either a new window will appear that allows you to view the HTML file that was created, or your document will appear in the 'viewer' tab in RStudio. You see the formatted HTML document that combined both text and the output of R code.

Close the window – we are now ready to analyze our data.

Step 4: Reproducible Data Analysis in R Studio

Delete all text from ## R Markdown on down – only keep the header and set-up sections of the default document.

First, we need to analyze some data. We will download this data directly from an existing GitHub repository I created. Students in an introduction to psychology course performed a simple Stroop experiment. During the Stroop, participants named the colors in a congruent trial (e.g., the word 'red' written in a red font) and incongruent trial (e.g., the word 'red' written in a green font). The time they took to name all words was

12

recorded in seconds (e.g., 21.3 seconds) for both the congruent and incongruent trial. There are four columns in the dataset:

- Participant Number
- Response Time for Congruent Stimuli
- Response Time for Incongruent Stimuli
- Year of Data Collection

Click the button '+C Insert' to insert code – a dropdown menu will be visible. Select R.
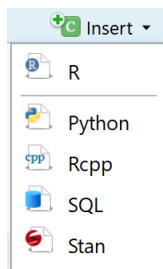
In the R Markdown file, you'll see a new section of R code that starts with ```{r} and ends with ```. You can also just create these sections by manually typing in these two lines.

Copy-paste the code below – make sure to get all the text – and paste it between the start line (```) and the end line (```) of the R code chunk. Note that there is not hard return between 'str' and 'oop.txt' – when copy-pasting this, you probably need to correct this.

```
stroop_data <-
read.table("https://raw.githubusercontent.com/Lakens/Stroop/master/s
troop.txt", sep = "\t", header = TRUE)

write.table(stroop_data, file = "stroop.csv", quote=F, row.names=F)
```

After copy-pasting the text, the code section should look like this (again, be aware of any difficulties when copy-pasting text from a PDF file into RStudio):

```{r}
stroop_data <- read.table("https://raw.githubusercontent.com/Lakens/Stroop/master/stroo
p.txt", sep = "\t", header = TRUE)

write.table(stroop_data, file = "stroop.csv", quote=F, row.names=F)
```

This code creates a data.frame called 'stroop_data" that contains data, and then saves this data in a .csv file called 'stroop.csv'. Click the Knit button: 🖌 Knit to look at the document. You should see something like:
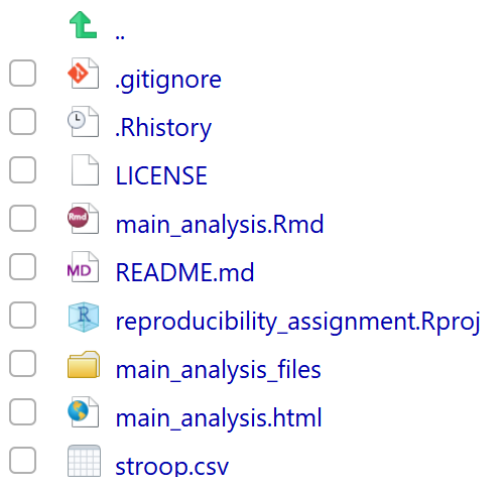
Main Analysis

Lakens

17 juni 2018

```
stroop_data <- read.table("https://raw.githubusercontent.com/Lakens/Stroop/master/stroop.txt", s
ep = "\t", header = TRUE)

write.table(stroop_data, file = "stroop.csv", quote=F, row.names=F)
```

This might not look very impressive – but the real action is in the file pane in the bottom right part of the screen. Close the window showing the HTML output and look at the file pane. You should now see a bunch of files:

⬆ ..
☐ 🔶 .gitignore
☐ 🕘 .Rhistory
☐ 📄 LICENSE
☐ 📕 main_analysis.Rmd
☐ 📝 README.md
☐ Ⓡ reproducibility_assignment.Rproj
☐ 📁 main_analysis_files
☐ 🌐 main_analysis.html
☐ 🗒 stroop.csv

One file is stroop.csv – this is our data file of the Stroop data that we downloaded from the internet and saved to our project folder, using R code.

There is really no need to keep downloading the file from the internet when we can also just load it from the local folder. So let's change the code. We won't completely delete this code – we will just **comment it out** by placing a # in front of it. This way, we can still remember where we downloaded the code from, but we won't use the code.

Because it is always important to **provide comments in the code you write**, add the explanation:

`#run only once to download the data`

above the line where we downloaded the code. Then, select the lines of code in the chunk, and click (on windows) CTRL+SHIFT+C (or click 'Code' in the toolbar and then 'comment/uncomment lines'). This should add # in front of all lines, making it comments instead of code that is executed every time. You should end up with:

```r
# #run only once to download the data
# stroop_data <- read.table("https://raw.githubusercontent.com/Lakens/Stroop/master/stroop.txt", sep = "\t", header = TRUE)
#
# write.table(stroop_data, file = "stroop.csv", quote=F, row.names=F)
```

Now we need to add a line of code that we will run, and with which we will load the stroop.csv dataset from the local folder. Underneath the last commented out line of code, but within the R code block, add:

```r
stroop_data <- read.csv("stroop.csv", sep = " ", header = TRUE)
```

```r
# #run only once to download the data
# stroop_data <- read.table("https://raw.githubusercontent.com/Lakens/Stroop/master/stroop.txt", sep = "\t", header = TRUE)
#
# write.table(stroop_data, file = "stroop.csv", quote=F, row.names=F)

stroop_data <- read.csv("stroop.csv", sep = " ", header = TRUE)
```

Click save, or press CTRL+S, to save the file. Knit the file. We see:

Main Analysis

Lakens

17 juni 2018

```r
# #run only once to download the data
# stroop_data <- read.table("https://raw.githubusercontent.com/Lakens/Stroop/master/stroop.txt",
 sep = "\t", header = TRUE)
#
# write.table(stroop_data, file = "stroop.csv", quote=F, row.names=F)

stroop_data <- read.csv("stroop.csv", sep = " ", header = TRUE)
```

Close the HTML file. We've done quite a lot of work. It would be a shame if this work was lost. So this seems to be the perfect time to save a version of our R Markdown file, not just locally, but also on GitHub.

15

Step 5: Committing and Pushing to GitHub

It is time to store our changes in the cloud, on GitHub. This process takes two steps. First, we record the changes to the repository (aka the code and files we created), which is called '**commit**'. This does not require an internet connection, because we are just recording the changes locally. However, then we want to make sure these recorded changes are also stored on GitHub, which requires you to **push** the files to GitHub.

If we look at the Git tab in the top right pane in R Studio, we see the Commit button, the Push button, and we see a bunch of files. The status of these files is indicated by two question marks in yellow. These question marks indicate these files are not yet tracked by GitHub. Let's change this.
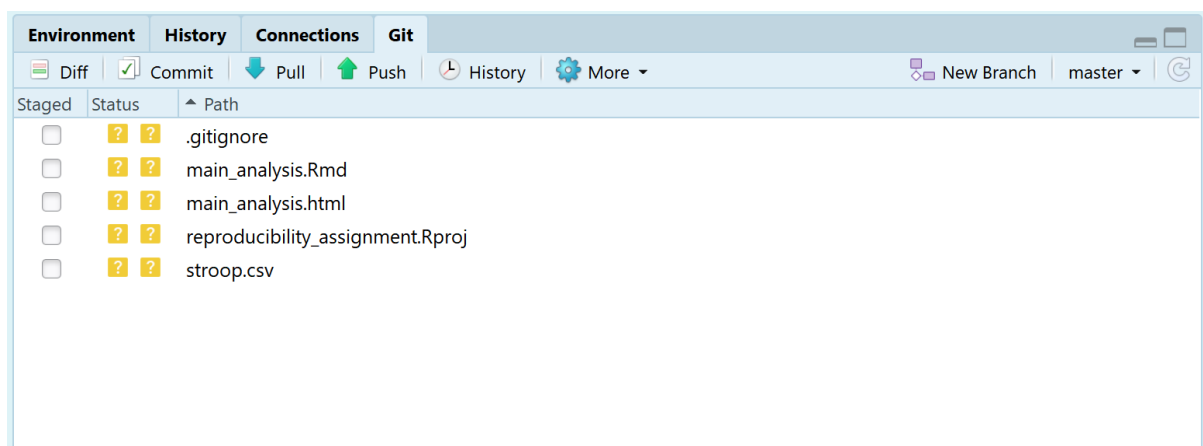

Click the commit button. A menu opens. You can choose to 'stage' the changes that have been made. Staging basically mean selecting which of the files you want to record, or **commit**. You can do this in several ways, such as double clicking each file, or selecting all files and clicking 'Enter'. When staging all files, the yellow question marks change to a green 'A' symbol. Every commit should be accompanied by a **commit message** where you describe which changes you have made – after all, we are recording our changes. You can type in anything you want – it is common to write something like 'initial commit' the first time you commit changes. The menu should look like the screenshot below:
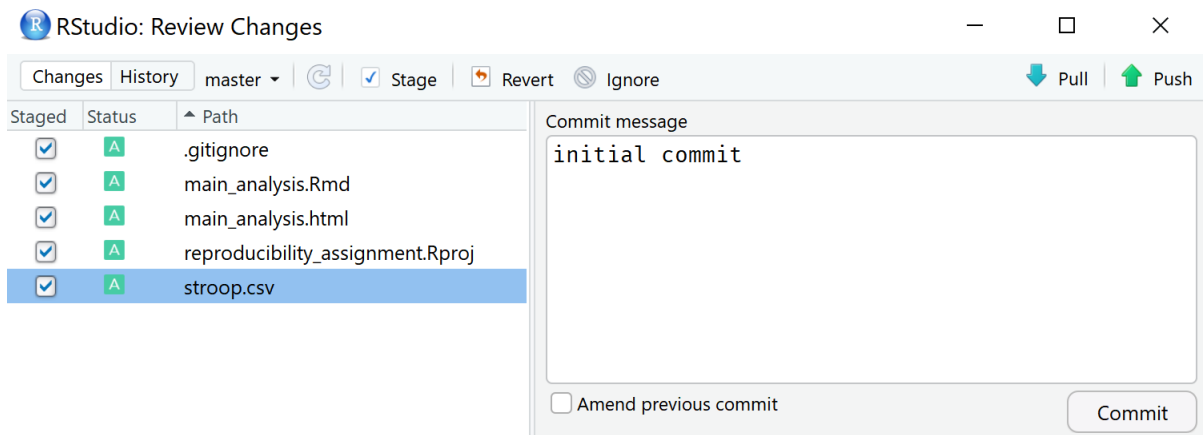
Now we are ready to **commit** these changes. Click the 'Commit' button. A new window opens that shows all changes that have been committed. We see that 5 files have changed. You can close this window and close the previous menu.

R Studio now reminds you that there is a difference between the local copy of your repository, and the remote version of the repository on GitHub. In the Git tab you see a reminder: "Your branch is ahead of 'origin/master' by 1 commit.".
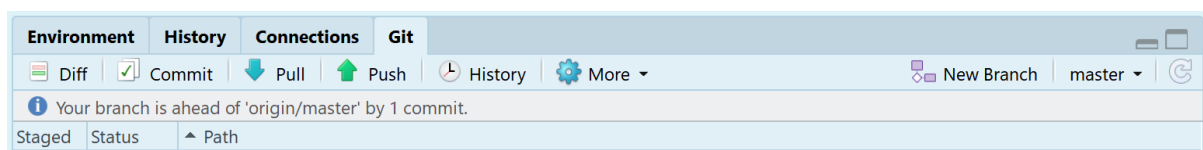
This means the files we have updated and recorded on our computer with a commit are not yet synchronized with the **remote repository** on GitHub. We can solve that by 'pushing' (aka synchronizing) the changes to the remote repository. Simply click the **push** button: . Another pop-up window appears:

```
Git Push                                                    Close

>>> C:/Program Files/Git/bin/git.exe push origin refs/heads/master
To https://github.com/Lakens/reproducibility_assignment
   9a2eaad..4dbbace  master -> master
```

This window informs us there were no errors, and we successfully pushed the changes to the remote version of the repository. You can close this window.

You can check that you successfully pushed all files to GitHub by visiting the GitHub page for your repository in the browser. You should see something like:

Congratulations on your first GitHub push! If you want to read a more extensive introduction to Git, you can read this tutorial paper by Matt Vuorre and James P. Curley.

Step 6: Reproducible Data Analysis

So far, we have only read in data. The goal of an R Markdown file is to create a manuscript that contains a fully **reproducible data analysis**. In this assignment, I cannot teach you how to analyze data in R (but I can highly recommend learning it – there are plenty of excellent online resources). Instead of programming from scratch,

visit [this raw text version of the R Markdown file](#) that will analyze the Stroop data. In the website, select all text (CTRL+A), copy it (CTRL+C). Then go to your main_analysis.Rmd file in R Studio. Select all text (CTRL+A) and hit delete. That's right – delete everything. You don't need to worry about losing anything – you have a **version controlled file** in your GitHub repository, which means you can always go back to a previous version! In the (now empty) main_analysis.Rmd file, press CTRL+V and paste all text. The file should look like the screenshot below.

This R Markdown file does a number of things, which we will explain in detail below. For example, it will automatically install libraries it needs, load the data, and create a report in HTML. You can hit the Knit button, and the HTML document should load. You should see output as in the second screenshot below.

```
1  ---
2  title: "Main Analysis"
3  author: "Lakens"
4  date: "20 juli 2018"
5  output: html_document
6  ---
7
8  ```{r global_options, echo=FALSE, warning=FALSE, message=FALSE,
    include=FALSE}
9  #This code will check if ggplot2 and reshape2 are installed.
10 #If not, this code will install these packages.
11 if(!require(ggplot2)){install.packages('ggplot2')}
12 library(ggplot2)
13 if(!require(reshape2)){install.packages('reshape2')}
14 library(reshape2)
15 knitr::opts_chunk$set(echo=FALSE, warning=FALSE, message=FALSE, include=TRUE)
16 ```
17
18 # Introduction
19
20 Here, we analyze a simple dataset of a Stroop experiment. Students in an
   introduction to psychology course completed an online Stroop task
   (http://faculty.washington.edu/chudler/java/ready.html) and named the colors
   in congruent trials (e.g., the word 'red' written in a red font) and in
   incongruent trials (e.g., the word 'red' written in a green font). The time
   they took to name all words was self-reported in seconds (e.g., 21.3 seconds)
   for both the congruent and incongruent blocks. In this analysis, we are
   interested in examining whether there is a Stroop effect.
```
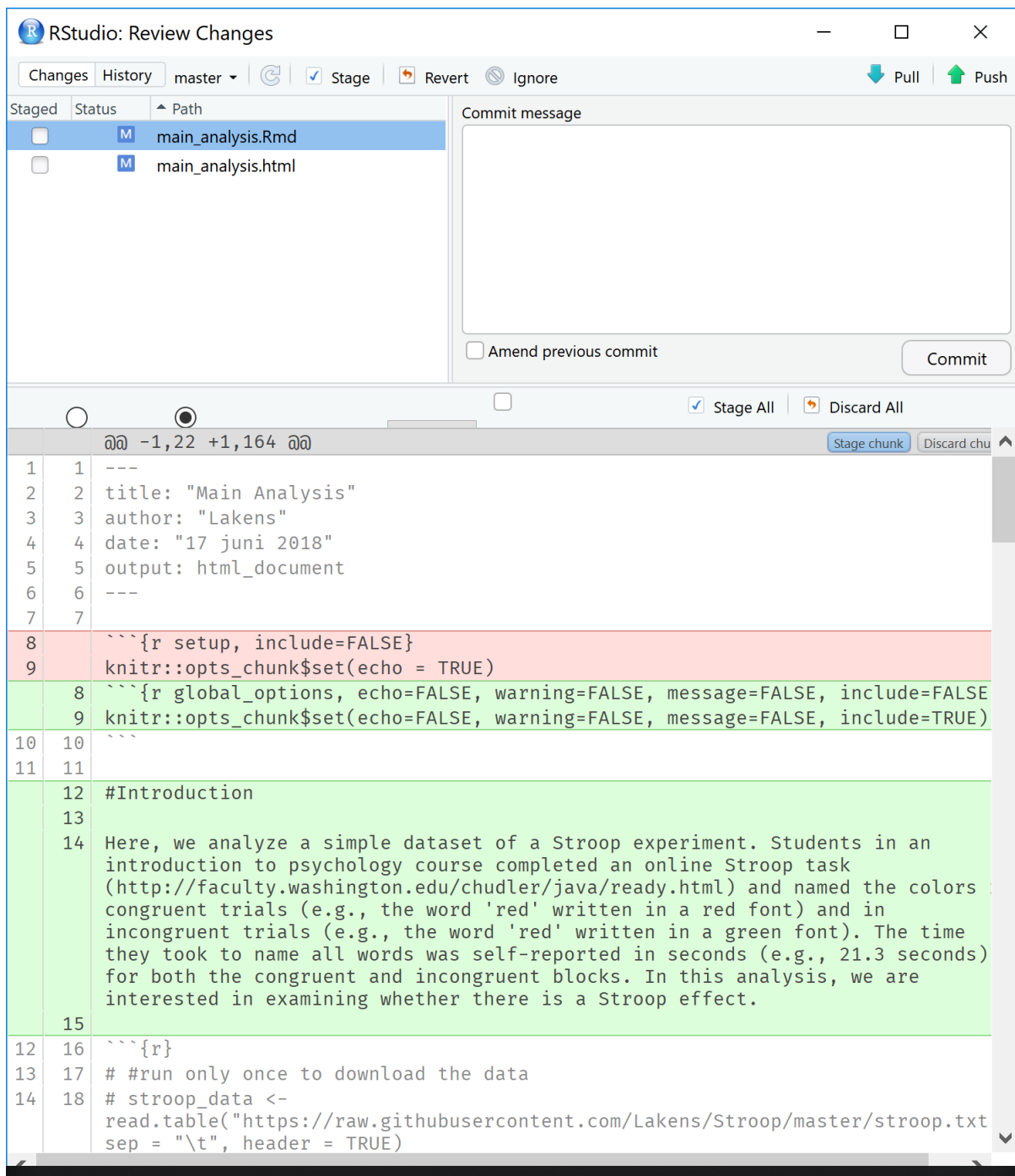
Results

The mean reaction time (in seconds) of participants in the Congruent condition ($M = 15.1$, $SD = 4.1$) was lower than the mean of participants in the Incongruent condition ($M = 23$, $SD = 4.78$, $r = 0.37$). A dependent t-test indicated that based on our preregistered alpha level of 0.01 we could reject the null-hypothesis, $t(130) = 18.04$, $p < 0.001$. As we can expect from the Stroop effect, the standardized effect size is very large, Hedges' $g_{av} = 1.76$. The congruency effect is very clear when we plot the data from the two groups.
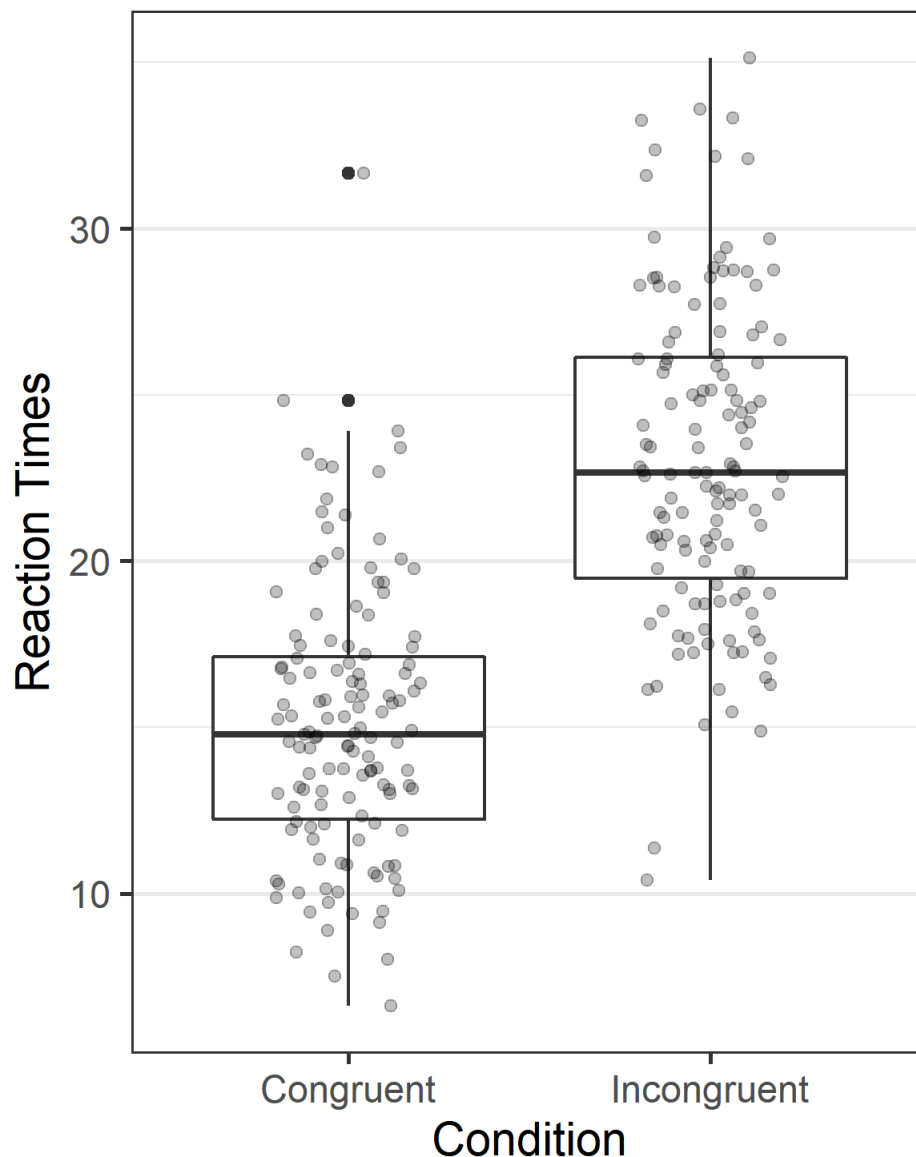
It is important to note that none of the numbers that are in this text are static, or copy-pasted. They are all calculated at the moment that the document is created, directly from the raw data. The same is true for the figures, which are created from the raw data the moment the manuscript is compiled. If you have access to the .Rmd (RMarkdown) file, you can perfectly **reproduce** the reported data analysis.

Since we have made substantial changes, this is the perfect moment to **commit** and **push** the changes to GitHub! Go to the Git tab in the top right pane. Click 'Commit'. The window below will open. If the main_analysis.Rmd file is selected, you will see red and green chunks of text. These tell you what was old (red) and what is new (green).

RStudio: Review Changes — □ ✕

Changes | History | master ▾ | ⟳ | ✓ Stage | ↩ Revert | ⊘ Ignore | ⬇ Pull | ⬆ Push

Staged	Status	▲ Path
☐	M	main_analysis.Rmd
☐	M	main_analysis.html

Commit message

☐ Amend previous commit   Commit

☐   ✓ Stage All   ↩ Discard All

```
@@ -1,22 +1,164 @@                                      Stage chunk | Discard chu

 1   1  ---
 2   2  title: "Main Analysis"
 3   3  author: "Lakens"
 4   4  date: "17 juni 2018"
 5   5  output: html_document
 6   6  ---
 7   7
 8      ```{r setup, include=FALSE}
 9      knitr::opts_chunk$set(echo = TRUE)
     8  ```{r global_options, echo=FALSE, warning=FALSE, message=FALSE, include=FALSE
     9  knitr::opts_chunk$set(echo=FALSE, warning=FALSE, message=FALSE, include=TRUE)
10  10  ```
11  11
    12  #Introduction
    13
    14  Here, we analyze a simple dataset of a Stroop experiment. Students in an
        introduction to psychology course completed an online Stroop task
        (http://faculty.washington.edu/chudler/java/ready.html) and named the colors
        congruent trials (e.g., the word 'red' written in a red font) and in
        incongruent trials (e.g., the word 'red' written in a green font). The time
        they took to name all words was self-reported in seconds (e.g., 21.3 seconds)
        for both the congruent and incongruent blocks. In this analysis, we are
        interested in examining whether there is a Stroop effect.
    15
12  16  ```{r}
13  17  # #run only once to download the data
14  18  # stroop_data <-
        read.table("https://raw.githubusercontent.com/Lakens/Stroop/master/stroop.txt
        sep = "\t", header = TRUE)
```

Select all files that have changed, and 'stage' them (for example by pressing enter). The checkboxes in front of the files, under the 'Staged' column, should be checked.

Staged	Status	▲ Path
☑	M	main_analysis.Rmd
☑	M	main_analysis.html

Type in a commit message, such as 'update mean analysis' in the 'commit message' field. Press the 'Commit' button. Close the window that pops up to inform you about the result of the commit. Then click 'push'. Close the window that informs you about the push command, and close the commit window. You can always visit the GitHub

repository online and look at the full history of your document to see all changes that have been made.

Let's take a look at some sections of our new R Markdown document. First the header:

```
```{r global_options, echo=FALSE, warning=FALSE, message=FALSE,
include=FALSE}
knitr::opts_chunk$set(echo=FALSE, warning=FALSE, message=FALSE,
include=TRUE)
```
```

This sets general options for the code chunks in the R Markdown file. The echo, warning, and message = FALSE hide the code chunks, warning messages, and other messages, where the 'include=true' will make all figures appear in the text. You can set some of these variables to TRUE, and hit Knit to see what they change. Sometimes you might want to share the HTML file with all code visible, for example when sharing with collaborators.

If you scroll down, you can see the introduction text, the code that generates the first figure, and the code that performs the analyses. These variables are used in the Result section. Let's look at this section:

```
#Results

The mean reaction time (in seconds) of participants in the Congruent
condition (*M* = `r round(mean(stroop_data$Congruent), digits = 2)`, *SD*
= `r round(sd(stroop_data$Congruent), digits = 2)`)  was lower than the
mean of participants in the Incongruent condition (*M* = `r
round(mean(stroop_data$Incongruent), digits = 2)`, *SD* = `r
round(sd(stroop_data$Incongruent), digits = 2)`, *r* = `r
round(cor(stroop_data$Congruent, stroop_data$Incongruent), digits = 2)`).
An independent *t*-test indicated we could reject the null-hypothesis,
based on an alpha of 0.05, *t*(`r round(ttest_result$parameter,
digits=2)`) = `r round(ttest_result$statistic, digits=2)`, *p* `r
ifelse(ttest_result$p.value > 0.001," = ", " < ")` `r
ifelse(ttest_result$p.value > 0.001, formatC(round(ttest_result$p.value,
digits=3), digits=3, format="f"), "0.001")`. As we can expect from the
Stroop effect, the standardized effect size is very large, Hedges' *g~av~*
= `r round(d_unb, digits=2)`, 95% CI [`r round(ci_l_d_av, digits=2)`;`r
round(ci_u_d_av, digits=2)`].
```

This section shows how you can **mix text and R code**. The start of this code is normal text. The *M* is still normal text (the * * make the M italicized, just as further down the ~av~ indicates these letters should be subscript), but then you see R code. In R Markdown you can embed R code within `r      `. Any R code within the two apostrophes will be executed. In this case, the mean of the Congruent reaction times is calculated, and rounded to 2 digits. You can see this number in the text.

Learning to program takes time. You can see some things are quite tricky to program. For example, the code:

```
`r ifelse(ttest_result$p.value > 0.001," = ", " < ")` `r
ifelse(ttest_result$p.value > 0.001,
formatC(round(ttest_result$p.value, digits=3), digits=3,
format="f"), "0.001")`
```

is a lot of code to make sure the exact *p*-value is reported, unless this p-value is smaller than 0.001, in which case 'p < 0.001' is printed (the papaja package below makes reporting statistics a lot easier!). The first time you need to program something like this takes a lot of time – but remember you can re-use code in the future, and you can steal a lot of code from others! You can complete a full introduction to Markdown here.

Extra: APA formatted manuscripts in papaja

If you want to write a reproducible manuscript in **APA style** (common in for example psychology) you might want to try out the R package papaja created by Frederik Aust. Install the papaja package. **Restart RStudio**.Then, create a new R Markdown document, but instead of selecting the document option, select the 'From Template' option, and select the template APA article (6th edition) provided by the papaja package.

You will see a template with a lot of fields for you to fill in, such as the title, author names and affiliation, the author note, the abstract, etc. Papaja takes care that all this information ends up in a nice lay-out – exactly following the APA rules. This means that if you have installed MiKTeX (to be able to convert to PDF), you can knit the document to a pdf, and submit an APA formatted document that is completely reproducible. For a tutorial covering all options in papaja, including how to add citations: https://crsh.github.io/papaja_man/index.html

```
● Untitled1 ×
○ ○ | ○ | ▤ | ABC ✓ Q | ✎ Knit ▾ ○ ▾                                                    ○ Insert ▾ | ⇧ ⇩ | → Run ▾ | ✦ ▾ | ☰
  1 ▾ ---
  2   title            : "The title"
  3   shorttitle       : "Title"
  4
  5   author:
  6     - name         : "First Author"
  7       affiliation  : "1"
  8       corresponding : yes     # Define only one corresponding author
  9       address      : "Postal address"
 10       email        : "my@email.com"
 11     - name         : "Ernst-August Doelle"
 12       affiliation  : "1,2"
 13
 14   affiliation:
 15     - id           : "1"
 16       institution  : "Wilhelm-Wundt-University"
 17     - id           : "2"
 18       institution  : "Konstanz Business School"
 19
 20   author_note: >
 21     Complete departmental affiliations for each author (note the indentation, if you start a new paragraph).
 22
 23     Enter author note here.
 24
 25   abstract: >
 26     Enter abstract here (note the indentation, if you start a new paragraph).
 27
 28   keywords         : "keywords"
 29   wordcount        : "X"
 30
 1:1    ▤ Title ▾                                                                                         R Markdown ▾
```

Step 7: Organizing Your Data and Code

It is important to always **organize your data files and analysis files**. This helps others to quickly find the files you are looking for. In general, I recommend the **TIER protocol**: https://www.projecttier.org/tier-protocol/. If share a datafile that is slightly larger than the one in this example, make sure you add a codebook so others understand what all variables mean.

When you are organizing your code, **take great care to make sure that any personally identifying information in your data is stored safely**. Open science is great, but you are responsible to share data responsibly. This means that you need to **ask participants permission to share their data** in the informed consent form (a useful resource is the Research Data Management Support page of the University of Utrecht). Whenever you collect personal data, make sure you handle this data responsibly. Information specialists at your university library should be able to help.

Step 8: Archiving Your Data and Code

Although we have uploaded our data and code to GitHub, when you publish your article and want to **share your data and code**, it is important to remember that GitHub is not a data repository that guarantees long term data storage. GitHub is currently owned by Microsoft, and companies can choose to do with their free service whatever they want. This makes it less suitable to link to GitHub in scientific articles, because articles be around decades from now. For scientific publications, you will want to link to a stable long-term data repository. For a **list of data repositories**, click HERE. We will use the

Open Science Framework (OSF) in this assignment as a stable data storage, because it is very easy to just integrate our GitHub repository within an OSF project.

Log in to the OSF at https://osf.io/ (create an account if you haven't already done so). Click 'Create new project'. Give your project a name (for example 'Stroop Reproducible Analysis Assignment').

It is again important to add a **license** to your work, also on your OSF project. After having created a project, you can click 'Add a license':

You can again choose an MIT license. You will have to fill in a year, and the copyright holders – in this case, that is you, so fill in your name (it will appear in the license text).

Then click Save.

Although we could upload all our files to the OSF, we can also simply link our GitHub project to the OSF. In the menu bar of the OSF project, click on 'Add-ons'. In the list, scroll to GitHub:

Follow the step-by-step guide to connect to GitHub provided by the OSF:
https://help.osf.io/hc/en-us/articles/360019929813-Connect-GitHub-to-a-Project

Select your repository that contains the reproducibility assignment and click 'Save'.

Click the title of the OSF project page to go back to the main project page. You will now see in the 'Files' pane that the GitHub repository is linked:

This is a good moment to click the 'Make Public' button in the top right of your project. After making the project public, people will be able to find it on the OSF. If you don't want to make your project public just yet, but you do want to give others access to your files, you can create a **'View-only' link** on the OSF. Go to 'Contributors' and click the +Add button next to View-only links. For a step-by-step guide, see [this tutorial](#).


Create a link to share this project so those who have the link can view—but not edit—the project.

You can use a view-only link to share access to your files only with reviewers. You can create an anonymized view-only link to hide your contributor names in the project - this is particularly useful in **blinded peer review**. Giving access to the files during peer review greatly helps reviewers – it will be a lot easier for them to answer any questions they might have about your materials, data, or code. Be aware it means that people you don't know will have access to your files. So far, I don't know of any negative experiences with this process, but it is important to be aware that others have access to your files before they are published.

The OSF page now just links to the files on the GitHub page. It does not independently store them. This means we do not yet have a **long term stable data storage solution**.

To create a snapshot of all files in the GitHub repository that will be stored for a long time, you have to create a **Registration** of your project. We will not create a Registration of your project in this assignment. Creating a registration starts several formal procedures: data in linked repositories (such as GitHub) are stored by the OSF, and the project appears in the list of registrations. You should only register when you want to create a stable copy of your work. Below you see an example of the files in an OSF project that has been registered. You see that the GitHub repository that was linked to

the project has been turned into an Archive of GitHub – this creates a stable version of the project, as it was at the moment you registered.

A good moment to create a stable version of your project is when your manuscript is accepted for publication. You can create a Registration and use the Digital Object Identifier (DOI) to link to the code, data, and materials in the paper (you can add this link to the DOI to the manuscript as you check the proofs of your article before it is published). Note that it is recommended to link to the materials using the **DOI**. The DOI is a persistent link (meaning it will keep working) where a website address might change. A registration does not automatically get a DOI. After creating the Registration, you need to click the 'Create DOI' link to create a persistent object identifier.

Contributors: Daniel Lakens
Date registered: 2016-11-10 04:52 PM
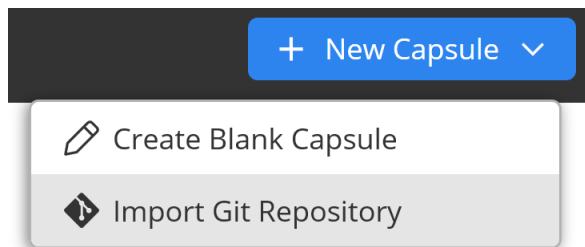Date created: 2016-07-14 09:04 AM
Create DOI
Category: 📦 Project

If you are ready to create a Registration, follow the instructions on the OSF: https://help.osf.io/hc/en-us/articles/360019930893-Register-Your-Project. As an example of a Registration that was made to store all work related to one of my scientific publications, see https://doi.org//10.17605/OSF.IO/9Z6WB (this link is itself an example of how to link to an OSF project using a DOI).

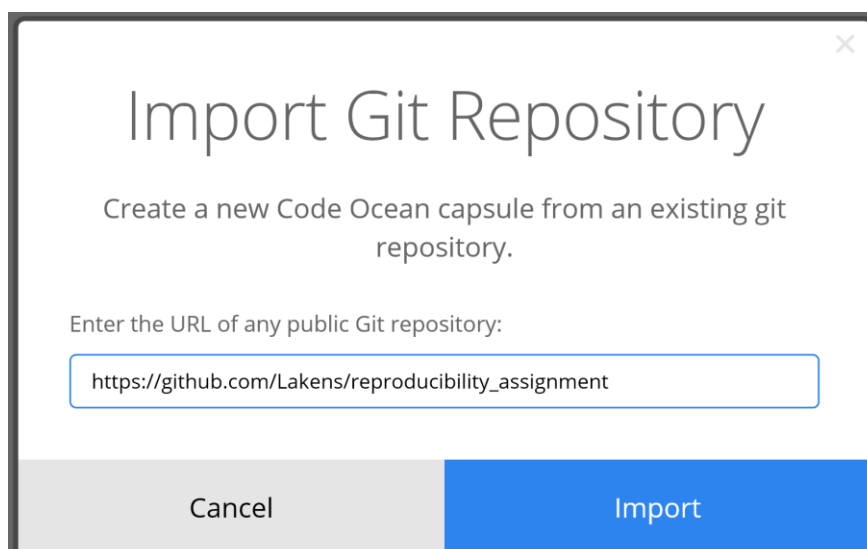## EXTRA: Sharing Reproducible Code on Code Ocean

If you have used the workflow above to create a reproducible manuscript, you might want to make it easy for other people to explore your data and code. People can simply clone your GitHub repository – but this still requires them to install the software you used, and even if they have R installed, it requires them to install the packages you used to analyze your data. This can potentially lead to reproducibility problems. Packages in R

update and change over time, and code that works on one machine might not run well on another computer. Furthermore, even when shared perfectly, downloading all files and getting the code up and running takes time. Several solutions exist, such as Packrat, which is a dependency management system for R, or Docker, which can create a container that works as a virtual machine that includes a computing environment including all the libraries, code and data that you need to reproduce an analysis.
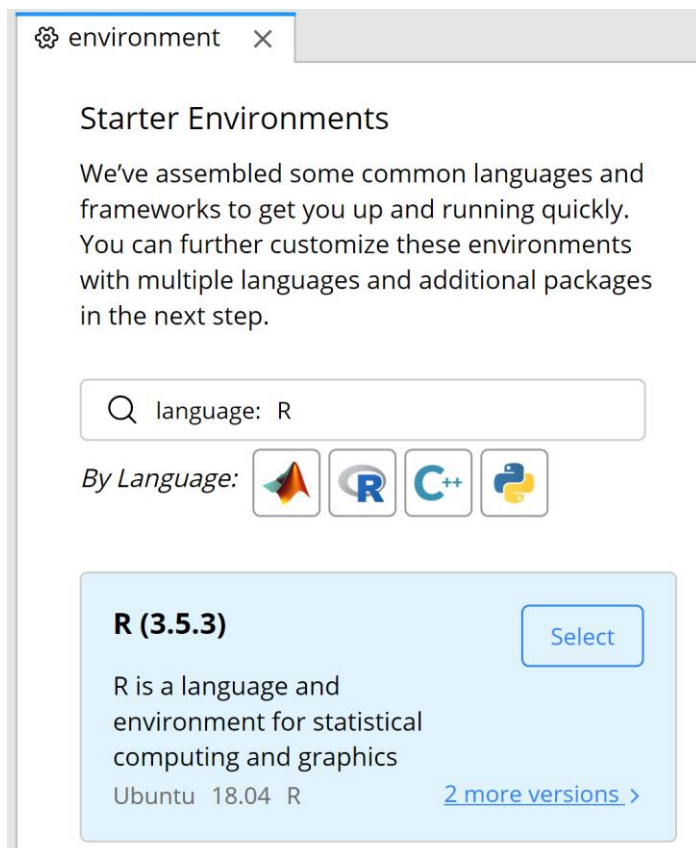
Here, I'll focus on a software solution that is designed to be easier to use than Docker, but provides many of the same benefits, called Code Ocean. Code Ocean is a cloud-based computational reproducibility platform. You can create a computing capsule that runs online and contains all packages your code needs to run. Although Code Ocean does not (yet) guarantee long term storage of data and code, it is an interesting way to make your reproducible code available to fellow researchers, and makes it very easy for researchers (or reviewers) to make small changes to your code, and examine the results. Create a (free) account on CodeOcean. Go to the dashboard. Click the 'New Capsule' button and choose the option 'Import Git Repository.

Enter the web address of your GitHub repository and click import.

Click on 'Environment' in the left pane. In the middle pane, click the R icon to select the programming language. At the time of writing, the default language is 3.5.3, but you can click on '2 more versions' and select R 3.6.

We need to set up the Code Ocean environment to contain the packages we need. Click +Add behind apt-get, and type pandoc and click the return key two times (no need to specify a version of pandoc). Click +Add behind R (CRAN) and type in ggplot2, and click return twice (no need to select a specific version). Click +Add again, and type reshape2, click return twice. Click add once more, and type rmarkdown and click return twice.

Package Managers	Packages
apt-get ⚙	pandoc 1.19.2.4~dfsg-1build4 + Add
R (CRAN) ⚙	ggplot2 3.2.0 reshape2 1.4.3 rmarkdown 1.14 + Add

In the left pane, drag and drop the main_analysis.Rmd file into the 'code' folder, and the 'stroop.csv' file into the 'data' folder. Select the code folder, and click the + button in the bar on the top of the left pane to add a file. Name the file 'run.sh'. This file will tell Code Ocean what to run. Select the file. The middle pane will be empty. Add the following code:

```bash
#!/bin/bash


Rscript -e "rmarkdown::render(input = 'main_analysis.Rmd', \
  output_dir = '../results', clean = TRUE)"
```

We need to make one final change. Select the main_analysis.Rmd. Scroll down to line 28 where the data is read in. Change:

```r
stroop_data <- read.csv("stroop.csv", sep = " ", header = TRUE)
```

to

```r
stroop_data <- read.csv("../data/stroop.csv", sep = " ", header = TRUE)
```

We are no all done! Click the 'Reproducible Run' button in the right pane. You will get output:

Click on the 'main_analysis.html'. You will see the script has generated our reproducible results. Commit the changes. You could (in principle – for this assignment we won't) make this completely reproducible container openly available and share it with your published paper.

There is now a completely reproducible analysis file online. Anyone can not just reproduce your data analysis – they can go into the main_analysis.Rmd file, and change anything they want in your code, and run it again. For example, let's say you dislike the black straight line in the first scatterplot, and you want it to be red. It is easy to change 'black' to 'red' in line 42, and re-run the analysis, and you will get a figure with a red line. Although that might in itself not be very exciting, the ability to easily re-analyze data might be useful in more realistic scenarios. For example, imagine you are reviewing a paper where the researchers do not plot the data. Without having to install any software, you can just type in `hist(stroop_data$Congruent)` after the data has been read in (e.g., on line 30), run the code again, and you will see a histogram for the reaction times in the Congruent condition. Give it a try.

Conclusion

In this assignment, we used a number of platforms and software solutions, such as GitHub, the Open Science Framework, R Studio, R, and R Markdown. Following through the example here is not the same as being able to use these tools in your research. Learning to use these tools will take time. There will be many frustrations when the code or software doesn't work as you want, or when you have gotten your local and remote GitHub repositories so much out of sync you just need to delete everything on your local computer and re-download all files from GitHub (`git reset --hard [HEAD]` is your friend). There are a lot of resources available online to find answers, or to ask for help. In my experience, it is some work, but it is doable, even if you have very limited knowledge of programming. You can get a basic reproducible workflow up and running by simply using all the steps described here, and then learn new skills as you need them. Learning these skills is appreciated both within and outside of academia (which is useful for PhD students) and will quickly save you time (e.g., when recreating figures for a revision, or when analyzing very similar datasets in the future). A reproducible workflow also improves the quality of your scientific work, and makes it easier for other scientists to re-use your work in the future.