GVPT Methods Workshop

How to create a reproducible journal article or report with R and Git

Harriet Goers

28/02/2022

Contents

This document steps you through sourcing, cleaning, analysis, and writing up a reproducible journal article or report using R and Git. By the end of it, you will be able to source your data, clean it, run your calculations, and include updated analysis in your formatted journal article or report with one click. I hope that this write-up, along with the workshop presentation, can serve as useful references for you going forward.

There are a couple of packages that I rely heavily on in my day-to-day research. These include: the `tidyverse` suite; `rio`; `here`; `googlesheets4`, `lubridate`; and `countrycode`. I will step you through each of these (and some more) now.

STEP 1: Github and RStudio

Git and Github are magic. They solve *most* of your version control and collaboration problems. Want to know how your data has changed since you last sourced it? Git can do that! Want to work with someone else on code or a RMarkdown document like you might with a Google doc? Git can do that! Did you completely blow it and want to back track to a working script? Git can also do that! I will show you how you can use Git and Github to your advantage.

When I start a new project, I first make a new Github repository (repo).

> To download Git, set up a Github account, and link it to your RStudio, I highly recommend following the steps provided in the Installation chapter and Connect Git, Github, RStudio chapter of Happy Git and Github for the useR.
>
> I am not going to lie, this is a difficult process. In fact, this is a real pain. But, you only have to do it once. After you have set it up, everything runs nice a smoothly (usually). I highly recommend following the steps provided in Happy Git. The end result is worth the pain.

1. Go to the `Repositories` tab.

2. Click on the big green `New` button in the top right-hand corner.
3. Give your repo a unique name.
    1. I suggest not including spaces in this name. It makes the subsequent file paths associated with the project more difficult to work with.
4. Give your project a brief description.
    1. This is optional; however, it can be useful for people who want to use your repo for replication. You can give a brief description of the question your research answers and a link to the associated journal article.
5. Choose whether the repo will be public or private.
    1. I usually start private. When my research is finalized, I switch to public.
    2. You can add collaborators to private repos.
6. Add a README file.
    1. This serves a similar function to the homepage for your repo. In it, I include instructions for replicating the analysis, the abstract or description of my research, and any useful links.
7. Click the big, green `Create repository` button.

You now have a new Github repo, ready for your scripts. You now need to link this to your RStudio.

1. Click the big, green `Code` button in the top right-hand corner.
2. Copy the URL provided.
3. Head over to RStudio.
4. Click the drop-down menu in the top right-hand corner of RStudio.
5. Click `New Project…`.
6. Select the third option: `Version Control`.
7. Select `Git`.
8. Paste the URL you copied earlier into the `Repository URL` input.
9. Select where you would like the local files to be stored on your computer from the `Create project as a subdirectory of` input.
10. Click `Create project`.

STEP 2: Set up your project

You are now working in an RStudio project.

> For more on the benefits of RStudio projects, I recommend reading the Workflow: projects chapter of R4DS.

We are now going to create the folders into which we will organize our analysis:

- `source`
    - This is where I keep the scripts I use to read raw data from its original source into my RStudio environment.
    - It is important to keep a copy of the raw data for your record. It is useful to know what has changed, if anything, each time you source raw data. Your `source` scripts will save copies of the raw data in the `data-raw` folder.
- `data-raw`
    - This is where I store a copy of the raw data in its original format.
- `munge`

  - This is where I keep my scripts for cleaning the raw data. I will also run any calculations needed for my analysis.
  - I will link these scripts to their `source` counterparts to ensure that every time I am touching the raw data, I am using the most up-to-date version of that data.
  - It is important to keep a copy of this cleaned data for your record. You may want to share this data as a research product in and of itself. Your `munge` scripts will save copies of the cleaned data in the `data` folder.

- `data`
  - This is where I store a copy of the cleaned data this I will use in my analysis.
  - When we upload this data to Github, you will be able to share its associated URL will other researchers. They will, therefore, always have the most up-to-date version of your data for their analysis. They will also have a full record of any changes made to the data.

- `analysis`
  - This is where I store my RMarkdown document of my article or report. It will generally be linked to my `munge` scripts (which are, in turn, linked to my `source` scripts). More on this later.
  - This is also where I store a script providing my models or simulations.

  You should come up with a folder structure that works for you. I find the one outlined above to be sufficiently disaggregated to keep me sane.

  For example, some people really like numbering their folders and scripts to communicate (to themselves as much as others) the right order in which to do things.

1. Create your folders.
2. Provide a brief outline of each of the folders (similar to the one above) in your `README.md` file.
    1. Remember: this is the homepage of your repo, so it serves as a useful reference point for people who are new to your research and analysis.
    2. It might be useful to create a snippet that does these two steps for you. For more details, see Henry Overos's presentation.

STEP 3: Collect and store your data

This step will obviously be unique to your project. I will step you through my method for sourcing data from five common sources: your local computer; a URL; Github; GoogleSheets; and a package.

Other than sourcing from your local computer, writing a script that sources live data directly allows you to automatically ensure that you are always accessing the most up-to-date data available to you. This should encourage regular reflection on whether your models explaining the relationship between your dependent and independent variables stands the test of time. You can come back to your work in one, five, ten years, knit your RMarkdown document (a step we will come to shortly) and see if it all holds up.

> A quick defense of `.csv`: this is a great default file type for your data. Almost anyone on any computer can open and read a `.csv`. It is small and uncomplicated, which makes it very flexible. If you are not dealing with large quantities of data, you should save your data as a `.csv`.

The general process for this step is as follows:

1. Create a new script in your `source` folder.
2. Source your data.
3. Store your data as objects with meaningful names in your environment.

4. Save a copy of them into your `data-raw` folder.
5. Push to Github.
    1. Go to the `Git` tab in your environment panel.
    2. Tick the `Staged` box next to the files you want to commit.
    3. Click the `Commit` button in the top left-hand corner of the `Git` tab.
    4. Write a meaningful message in the `Commit message` panel in the top right-hand corner of the pop-up.
    5. Click the `Commit` button below this panel.
    6. Close the new pop-up.
    7. Click the `Push` button in the top right-hand corner of the pop-up (above the `Commit message` panel).
    8. Click out of all pop-ups.

Sourcing from your local computer

1. Save a copy of your data in your `data-raw` folder.
2. Read that data into your environment.

```
iris_raw <- rio::import(here::here("data-raw", "iris.csv"))
```

> I rely heavily on these two packages: rio and here. They make for incredibly robust and succinct code. `Rio` is a wrapper package that detects the file type of the file you are reading in and calls the relevant function. `Here` adjusts all file paths to include your top-level directory. This eliminates the need for others working on their own computers from manually adjusting the file paths included in your scripts. For more information on each, read the package documentation provided above.

Sourcing from a URL

1. Navigate to the data you want to collect from the web.
2. Generally speaking, the website will provide a button to download the data. If you hover over this button, you will be able to see the underlying URL pop up in the bottom left-hand corner of your screen. If it has a file type at the end (for example, `.zip`, `.csv`, `.xlsx`) you can right-click on that button and copy the URL. On a Mac, the option is `Copy Link Address`.
3. This is your file path. You can use `rio::import` to read it into your environment.
4. You can then use `rio::export` to save a copy of that raw data in your `data-raw` folder.

```
# Downloading the UCDP Armed Conflict Dataset from https://ucdp.uu.se/downloads/
ucdp_acd_raw <- rio::import("https://ucdp.uu.se/downloads/ucdpprio/ucdp-prio-acd-211-RData.zip")

rio::export(ucdp_acd_raw, here::here("data-raw", "ucdp_acd_raw.csv"))
```

> Many organizations offering open source data provide APIs for accessing this data. For example, UCDP provides this API for you to use. Generally speaking, if you are frequently sourcing large quantities of data from an organization that provides an API, you should use that API rather than directly downloading the entire file as we have done above. R provides many straightforward packages for interacting with APIs, including `httr` and `jsonlite`. You should check them out if this is something you do regularly.

Sourcing from Github

Github is a great collaborative tool for researchers. You can access and share datasets via public repos.

1. Find the data stored on Github.
    1. For example, here is a link to a dataset of general country-year data I commonly use as controls in my analysis: https://github.com/hgoers/SRDP_controls_database/blob/main/data/country_level.csv.
2. Click on the `Raw` button in the banner above the displayed dataset.
3. Copy the URL.
4. Again, this is your file path. You can use `rio::import` to read it into your environment.
5. You can then use `rio::export` to save a copy of that raw data in your `data-raw` folder.

```
country_controls_raw <- rio::import("https://raw.githubusercontent.com/hgoers/SRDP_controls_database/ma

rio::export(country_controls_raw, here::here("data-raw", "country_controls_raw.csv"))
```

Sourcing from GoogleSheets

If you are working with others to collect original data, you might be using GoogleSheets to collaboratively build that dataset. We can use the `googlesheets4` package to access that data directly.

> This requires some set-up, because you have to give RStudio permission to access your GoogleSheets. This is very simple. To do it, simply follow these steps provided in the `googlesheets4` package documentation.

1. Open the GoogleSheet in your browser.
2. Copy the entire URL, or unique sheet ID.
    1. More details can be found here.
3. Read in the data using `googlesheets4::read_sheet()`.
4. Store a copy in your `data-raw` folder or provide a link to the GoogleSheet, which provides its own version control.

```
gapminder_mini_raw <- googlesheets4::read_sheet("1k94ZVVl6sdj0AXfK9MQOuQ4rOhd1PULqpAu2_kr9MAU")
```

Sourcing from a package

Many people have done the hard work of sourcing common datasets and turned these into packages for us to more easily access. If you are trying to access a very popular dataset, someone has probably made a package to access it. A great example of this is the `wbstats` package, which provides easy access to the World Bank data API.

1. Grab the unique ID for the variable you want to collect. For example, the unique ID for GDP is NY.GDP.MKTP.CD.
    1. You can find this in the variable description, or as the last part of the URL: https://data.worldbank.org/indicator/NY.GDP.MKTP.CD.
2. Supply this to the `wbstats::wb_data()` function.
3. Save a copy to your `data-raw` folder.

```
gdp_raw <- wbstats::wb_data("NY.GDP.MKTP.CD", return_wide = FALSE)

rio::export(gdp_raw, here::here("data-raw", "gdp_raw.csv"))
```

STEP 4: Clean your data

Again, this step is going to be very unique to your data needs. Here, I will outline some useful tips for connecting these steps to ensure that you are working with the most up-to-date data. I will also offer some useful packages and functions I draw on regularly at this step of my analysis.

Sourcing your previous steps

You can directly integrate your `source` script into the first part of your `munge` script. When you source a script, it runs and pulls all the contained objects into your environment, ready for you to use.

```
source(here::here("source", "src_data.R")) # add your source file(s)
```

> A word of caution is in order here. This is the ideal. In this ideal world, datasets don't change in shape, websites remain nicely formatted, and URLs don't change. However, this is the real and rather messy world. If you suspect (or know) that your data source changes every year, you will probably want to import your raw data, stored in the `data-raw` folder, directly, rather than source your `source` script. This adds steps to your process, but it saves you overwriting the data collected from a pain-staking web-scrape that you rely on.

Check the structure of your data

You should make sure that your data is in the correct format at this stage of the process. For example, check that data that should be numeric is numeric and that dummy variables are factors. You can use the `dplyr::glimpse` function to examine the structure of your data.

```
messy_df <- tibble(year = c("2012", "2013", "2020"),
                   date = c("2012-01-12", "2013-05-21", "2020-09-23"),
                   democracy = c(1, 0, 0),
                   conflict_type = c(3, 2, 1))

glimpse(messy_df)
```

```
## Rows: 3
## Columns: 4
## $ year          <chr> "2012", "2013", "2020"
## $ date          <chr> "2012-01-12", "2013-05-21", "2020-09-23"
## $ democracy     <dbl> 1, 0, 0
## $ conflict_type <dbl> 3, 2, 1
```

```
clean_df <- messy_df %>%
  mutate(year = as.numeric(year),
         date = lubridate::ymd(date),
         democracy = factor(democracy),
         conflict_type = factor(conflict_type, labels = c("Interstate", "Interstate, dyadic", "Intrasta

glimpse(clean_df)
```

```
## Rows: 3
## Columns: 4
## $ year          <dbl> 2012, 2013, 2020
## $ date          <date> 2012-01-12, 2013-05-21, 2020-09-23
## $ democracy     <fct> 1, 0, 0
## $ conflict_type <fct> "Intrastate", "Interstate, dyadic", "Interstate"
```

Get summary statistics for your data

I like to use `skimr::skim` to view summary statistics for each of my variables. I will show it in action in the next step.

Create a scope dataframe

When I am collecting data from multiple different sources, I like to create a scope dataframe that includes all of the observations I need to have in my analysis. For example, if I want to analyse trends across all countries globally from 1960 to 2022, I create a dataframe that includes a row (observation) for every country and every year in that range. I then `dplyr::left_join` any data I collect to this scope dataframe. If I am missing data for a country-year, it comes up as an `NA`.

```
scope_df <- wbstats::wb_countries() %>% # get all current countries
  mutate(year = "1960, 2022") %>% # set the start and end dates
  separate_rows(year, sep = ", ", convert = TRUE) %>% # create a new observation for the start and end
  group_by(country) %>%
  expand(year = full_seq(year, 1)) %>% # fill in the years between the start and end year
ungroup()

head(scope_df)
```

```
## # A tibble: 6 x 2
##   country      year
##   <chr>       <dbl>
## 1 Afghanistan  1960
## 2 Afghanistan  1961
## 3 Afghanistan  1962
## 4 Afghanistan  1963
## 5 Afghanistan  1964
## 6 Afghanistan  1965
```

```
gdp_clean <- gdp_raw %>%
  select(country, year = date, gdp = value) %>%
  mutate(country = countrycode::countrycode(country, "country.name", "country.name"))

full_df <- scope_df %>%
  left_join(gdp_clean)

skimr::skim(full_df)
```

Table 1: Data summary

Name	full_df
Number of rows	18837
Number of columns	3
Column type frequency:	
character	1
numeric	2
Group variables	None

Variable type: character

skim_variable	n_missing	complete_rate	min	max	empty	n_unique	whitespace
country	0	1	4	80	0	299	0

Variable type: numeric

skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100	hist
year	0	1.00	1991	1.818000e+01	1960	1975	1991	2007	2.022000e+03	
gdp	10146	0.46	1958688966	1.002532e+12	824448	154123209	2001779555	13965467312	2.014332e+13	

Ensuring consistent country names

I work with a lot of country-year data. When you pull data for countries from different sources, you often have to grapple with different country names. For example, the United States can also be listed as US, USA, United States of America, etc. You need to make sure country names are consistent across your dataset to ensure you are capturing all data for each country. I use the `countrycode` package to do this.

```
sad_country_names <- tibble(year = 2022,
                            old_country_names = c("United States", "USA", "Russia", "Russian Federation

sad_country_names %>%
  mutate(new_country_names = countrycode::countrycode(old_country_names,
                                                      origin = "country.name",
                                                      destination = "country.name"),
         iso3c = countrycode::countrycode(old_country_names,
                                          origin = "country.name",
                                          destination = "iso3c"))
```

```
## # A tibble: 6 x 4
##    year old_country_names  new_country_names iso3c
##   <dbl> <chr>              <chr>             <chr>
## 1  2022 United States      United States     USA
## 2  2022 USA                United States     USA
## 3  2022 Russia             Russia            RUS
## 4  2022 Russian Federation Russia            RUS
## 5  2022 Myanmar            Myanmar (Burma)   MMR
## 6  2022 Burma              Myanmar (Burma)   MMR
```

STEP 5: Write your journal article in RMarkdown

Next, you need to write up your interesting findings. If you use RMarkdown, you can fully integrate these steps into that write-up. Every graph, table, and in-text number you include can be calculated using code and is, therefore, robust to changes in the data or your calculations in your sourcing or munging steps. They are also, therefore, fully traceable. This makes your analysis transparent and easy to replicate.

I will provide some general tips here. For a good introduction to RMarkdown, please see the RMarkdown chapter in R4DS.

Source your whole process, from sourcing to munging

As we did in our `munge` scripts for our `source` scripts, you can source your `munge` scripts in your RMarkdown document. When you run this line, you will run the `munge` script. This will first run the `source` script and then clean and save your data, ready for analysis.

A word of warning to those doing complex analysis. You should avoid this step if you are collecting large amounts of data or running computationally-intensive models or simulations. Predictably, if you ask your RMarkdown to source, clean, and calculate your data and analysis every time you knit it, this will take a long time. You will also have difficultly tracing any errors if they pop up. You are best to access the clean data directly by importing it from your `data` folder. I would also put all model-building or simulations in a separate script stored in this `analysis` folder that stores the output objects in your `data` folder.

Front load your packages, data, sourcing, and functions

Start your RMarkdown document with a series of code chunks that:

1. Load any relevant packages,
2. Read in data, or source your munge script,
3. Create any custom functions you will use in the body of the text.

This front-loads the processes that are most prone to error. For example, if someone downloads your repo and tries to compile your RMarkdown, but they don't have a required package, R will throw them an error early. They can then install that package and knit your RMarkdown. Happy days.

This recommendation follows from Emily Riederer's RMarkdown Driven Development. I highly recommend reading this blog post if you want more information.

Let Zotero and RMarkdown do your referencing for you

Zotero (a really amazing reference manager) and RMarkdown have brilliant integration. I would highly recommend using them in your writing. This is a good guide for setting them up.

Create your own ggplot theme

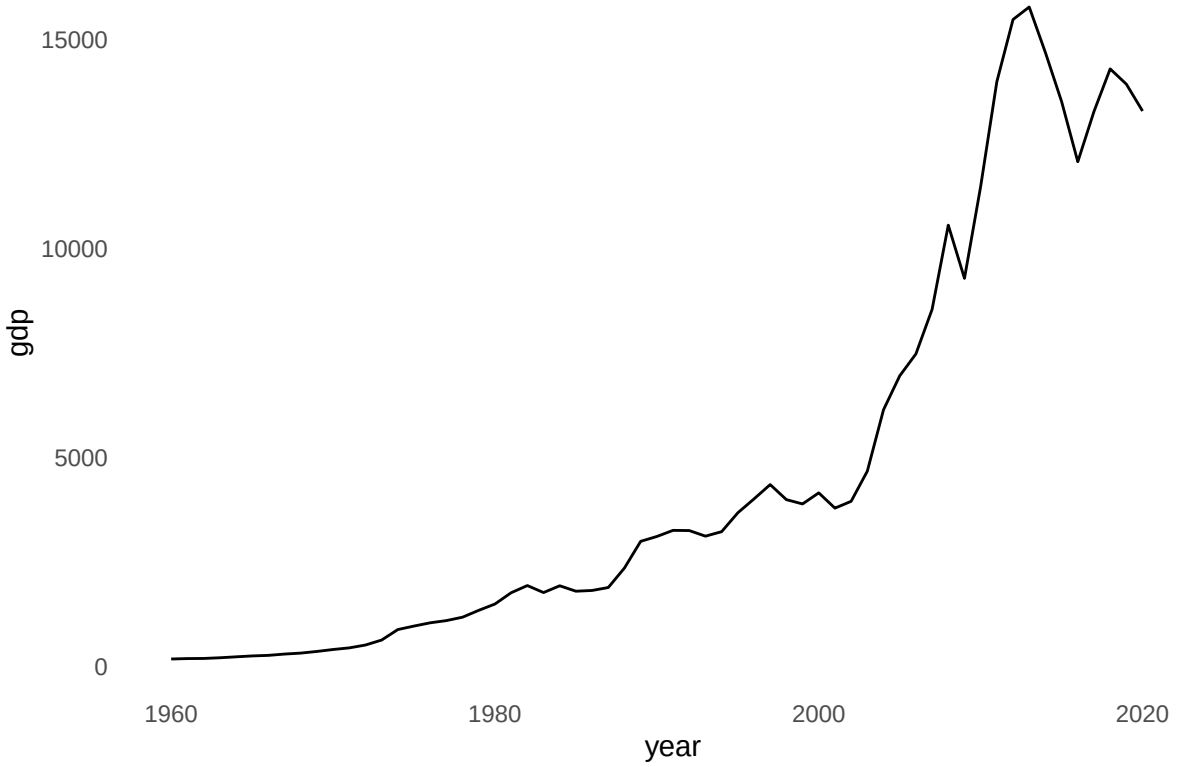
This is incredibly easy. Let me show you:

```
theme_custom <- function() {

  theme_minimal() +
  theme(plot.title.position = "plot",
        panel.grid = element_blank())

}

full_df %>%
  filter(country == "Australia") %>%
  mutate(gdp = gdp / 100000000) %>%
  ggplot(aes(x = year, y = gdp)) +
  geom_line() +
  theme_custom() +
  labs(title = "Australia's GDP (current USD), 1960-2020")
```



For more information on `ggplot` options, see here.

Use other people's templates and code (if they're okay with it!)

There are a couple of political science academics who are championing this approach to reproducible research. One such academic is Steven V. Miller. I highly recommend scrolling through his blog and having a sticky beak at his Github. He provides very useful resources, including templates for formatting your RMarkdown as a journal article.

Don't stop at RMarkdown

There are plenty of ways you can communicate your research using R. For example, you can write your thesis using thesisdown, your next book using bookdown, and your next website using blogdown. Yihui Xie is a great person to follow for all things RMarkdown/Bookdown/Latex integration.

Create a package

Finally, by far the best practice is to turn this repo and project into a package. A package allows users to very easily: access your data; see your analysis; use your functions for sourcing and munging your data. The more easily accessible your work is, the bigger the impact it will have.

Package development also requires you to be more deliberate about your documentation and take a far more function-forward approach to your coding. This has immense long-run benefits for you and anyone trying to replicate your work.

There are a couple more steps required to move from what you have created today to a package. Don't let that stop you. That gap is smaller than you think. For example, the folder structure for which I advocate resembles that required for packages. If you are ready to create your own package, I would highly recommend Hadley Wickham and Jenny Bryan's R Packages.

STEP 6: Publish

By now, you have a soup-to-nuts project. That's amazing! You should now share it so others can build on your work. I recommend pushing everything to Github and then converting your repo from private to public.

1. Head over to your repo.
2. Go to the `Settings` tab.
3. Scroll to the bottom of the `General` tab, all the way to the `Danger Zone`.
4. Click `Change visibility` and follow the prompts.

Now, shout your successes from the roof-top. You didn't even need to wait three months for a snarky reviewer's comments.

STEP 7: Revisit your work

1. Let a bit of time pass. Maybe go and work on something else or go on a well-deserved holiday.
    1. Perhaps wait until a new tranch of data gets released. For example, a few of the datasets we looked at today are updated annually.
2. Navigate to your RMarkdown file.
3. Click `Knit`.
4. Skip through to your results table (or better yet, your results graph because visualizations > tables).
5. Do your results hold up?
    1. If yes, yay! Time for another well-deserved holiday.
    2. If not, perhaps it is interesting to see why not.
        1. Was this new year an unusual one (perhaps there was a global pandemic that put things a bit off-kilter)?
        2. Was your theory only relevant to previous years?
        3. Is there something you could add to the model that would better explain this, and other, variation?