

A scenic view of the Elbe river in Magdeburg, Germany. In the foreground, several boats are on the water, including a small tugboat, a barge, and two larger white ferries. A concrete dam or lock structure is visible in the middle ground. In the background, the city skyline is visible, featuring a large cathedral with two tall spires and other buildings. The sky is clear and blue.

09 - Good practices for programming with R

Data Science with R · Summer 2021

Uli Niemann · Knowledge Management & Discovery Lab

<https://brain.cs.uni-magdeburg.de/kmd/DataSciR/>

Outline

- [Miscellaneous tips for reproducible programming](#)
- [Reproducible and efficient project pipelines with drake](#)
- [Recommendations on naming files & folders](#)
- [Package library management with renv](#)
- [Debugging](#)

Recommended reading

📖 Jennifer Bryan, and Jim Hester. [What They Forgot to Teach You About R](#). 2019. Last accessed 10.05.2021.



Photo by [Guillaume Jaillet](#)

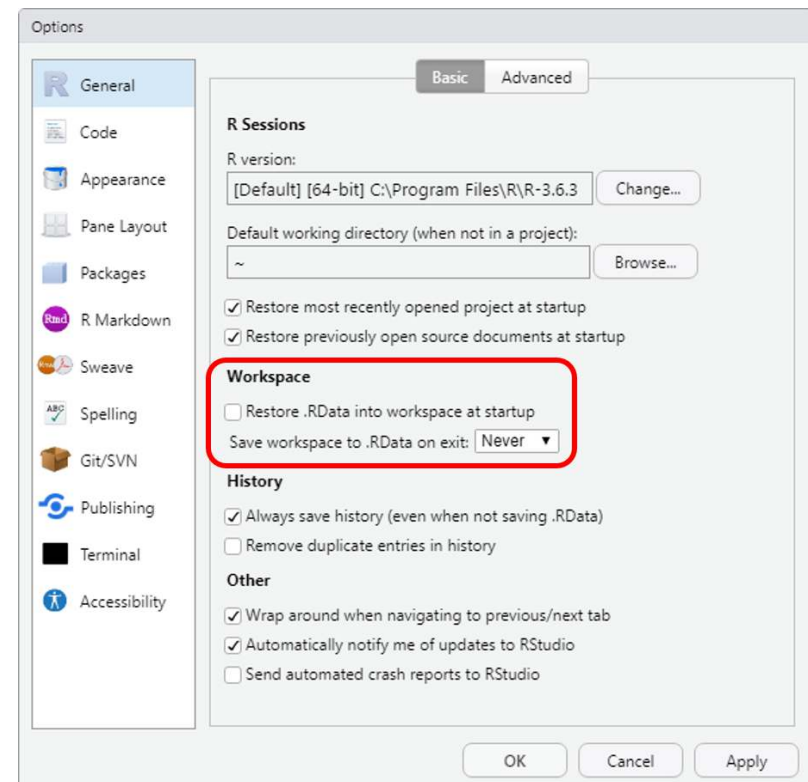
Miscellaneous tips for reproducible programming



Photo by [John Schnobrich](#)

✓ Save code, not the workspace

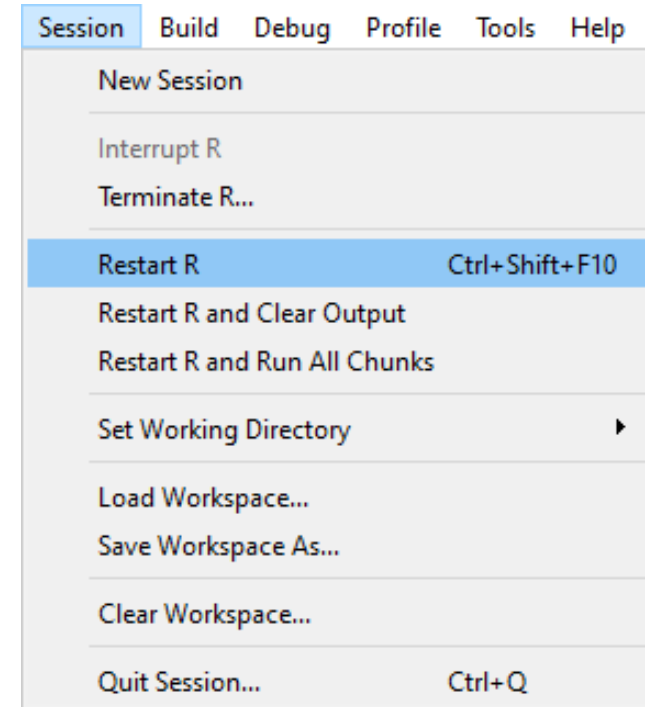
→ Save your commands in an R script (.R file) or R Markdown document (.Rmd) instead of saving the results of your interactive analysis as workspace (.Rdata)



Tip: disable automatic saving of workspace in RStudio

✓ Restart R often

- Restart R in RStudio: *Session* → *Restart R* (keyboard shortcut **Ctrl+⇧Shift+F10**)
- Restart R from the shell: **Ctrl+D** or `q()` to quit current session, then restart R
- Restart development where you left off, i.e., re-run all code above cursor position:
 - in an R script: **Ctrl+Alt+B**
 - in an R Markdown file: **Ctrl+Alt+P**



Tip: restart R every now and then

✓ **Avoid using** `rm(list = ls())`

Unlike restarting R, the command `rm(list = ls())` **does not** create a new R session.

- ¶ It only removes all user-created objects from the environment.
- ¶ Previously loaded packages are still in the search path.
- ¶ Modifications to global options are still there.

`rm(list = ls())` makes a script **vulnerable to hidden dependencies** on commands that were ran in the R session before executing `rm(list = ls())`.

☹️ *"What about time-consuming scripts that take hours or maybe even days? It is impracticable to re-run these scripts every time we need the results."*

Split the computationally expensive part of your code (e.g. fetching and pre-processing large amounts of raw data from a database) into a new script. At the end of this script, you save the output as `.rds` file (R data single).

```
saveRDS(data_prep, "results/data_preprocessed.rds")
```

✓ **Organize code as self-contained "projects":**

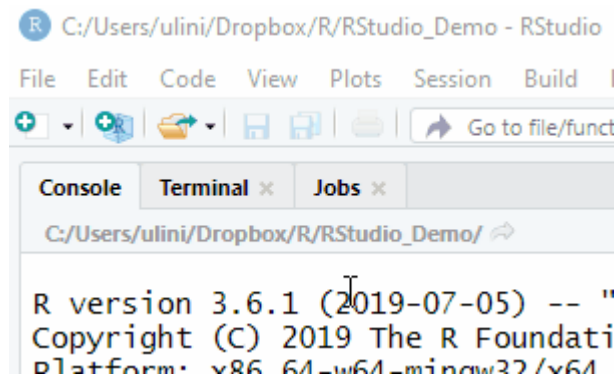
- Organize each logical project into a separate directory.
- Organize all files related to the project into subfolders, e.g. `data/`, `code/`, `figures/`, `reports/`, etc.
- The top-level folder of the project must be clearly recognizable (e.g. by containing a `.Rproj` file)
- Create paths **relative to the top-level folder**.
- Launch the R session from the project's top-level folder.

RStudio projects

Create a new RStudio project:

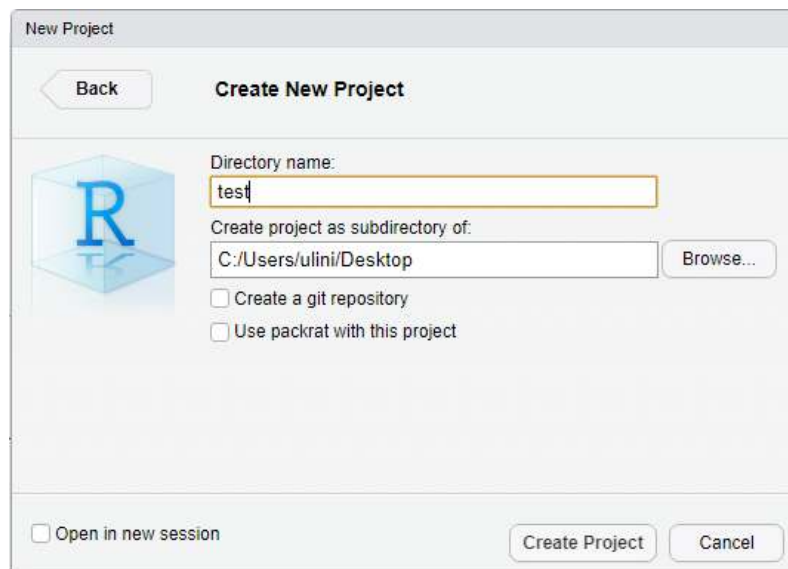
File → New Project → New Directory → Empty Project

The working directory corresponds to the top-level folder of the project (where the `.Rproj` file is). All project files should be organized in subfolders of the project's top-level folder. In the code, the files should be referenced with relative paths.



 Further reading:

- Best practices: [RStudio Projects](#)
- Chapter [Where does your analysis live?](#) in [R for Data Science](#)
- [Project-oriented workflow](#)



A basic folder structure could look like so:

```
my_project
+-- data
|   +-- processed
|   \-- raw
+-- my_project.Rproj
+-- output
+-- R
+-- README.md
\-- run_analyses.R
```

✓ Use `here::here()`¹ instead of `setwd()` to set your working directory.

☹ Specifying absolute file paths are bad practice:

```
library(ggplot2)
setwd("C:/Users/uli/verbose_funicular/foofy/data")
df <- read.delim("raw_foofy_data.csv")
p <- ggplot(df, aes(x, y)) + geom_point()
ggsave("../figs/foofy_scatterplot.png")
```

😊 Use file paths relative to project root folder.

```
library(ggplot2)
library(here)
df <- read.delim(here("data", "raw_foofy_data.csv"))
p <- ggplot(df, aes(x, y)) + geom_point()
ggsave(here("figs", "foofy_scatterplot.png"))
```

→ not self-contained, not portable

- file paths will not work for anyone besides the author
- high chance that objects from one project will leak into another project

¹`here::here()` is a robust version of `file.path()`, because `here()` creates paths that (1) work across different operating systems and (2) always start at the top-level folder of the project.

Example from: Jennifer Bryan. [Project-oriented workflow](#). 2017. Last accessed 10.05.2021.

Exemplary content in `R` folder of your project:

File	Description
01_fetch-data.R	Fetch raw data from a database with millions of rows
02_clean-data.R	Pre-process and tidies data
03-eda.R	Exploratory data analysis, including plotting the data
04-model.R	Train a prediction model (takes some hours ⌚)
05-report.R	Create an R Markdown project report
run-analysis.R	Successively runs all scripts above

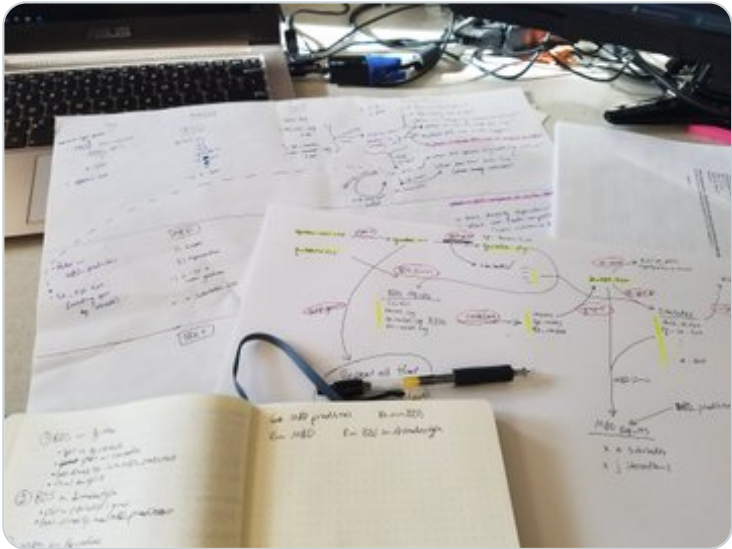
😬 "If we want to make a small change in one of these scripts, do we have to rerun all scripts from scratch again?"



Brianna McHorse, PhD 🏳️‍🌈
@fossilosophy



Replying to @ma_salmon @benmarwick and @wmlandau
Save me from myself and having to remember all this when files change



9:23 PM · Feb 21, 2018



URL

**targets: reproducible and efficient
project pipelines**



The targets package

The targets package is a Make-like pipeline toolkit for Statistics and data science in R. With targets, you can maintain a reproducible workflow without repeating yourself. targets skips costly runtime for tasks that are already up to date, runs the necessary computation with implicit parallel computing, and abstracts files as R objects. A fully up-to-date targets pipeline is tangible evidence that the output aligns with the code and data, which substantiates trust in the results.

— <https://docs.ropensci.org/targets/index.html>



- With targets, we can set up a **pipeline of successively executed commands**.
- The **return value** of each command is stored as a **target**.
- The targets package automatically identifies **dependencies** between **targets**.
- The targets package ensures that only targets **affected** by a change in one of its inputs (= **outdated targets**) need to be recomputed.

Every targets workflow is set up in a file called `_targets.R`:

```
library(targets)
library(tarchetypes)
options(tidyverse.quiet = TRUE)

create_plot <- function(data) {
  ggplot(data, aes(x = Petal.Width, fill = Species)) +
    geom_histogram()
}

tar_option_set(packages = c("dplyr", "forcats", "ggplot2", "readr", "tidyr"))

list(
  tar_target(raw_data, readxl::read_excel("targets_files/raw_iris.xlsx")),
  tar_target(my_data, raw_data %>% mutate(Species = fct_inorder(Species))),
  tar_target(hist, create_plot(my_data)),
  tar_target(fit, lm(Sepal.Width ~ Petal.Width + Species, my_data)),
  tar_render(report, "targets_files/report.Rmd")
)
```

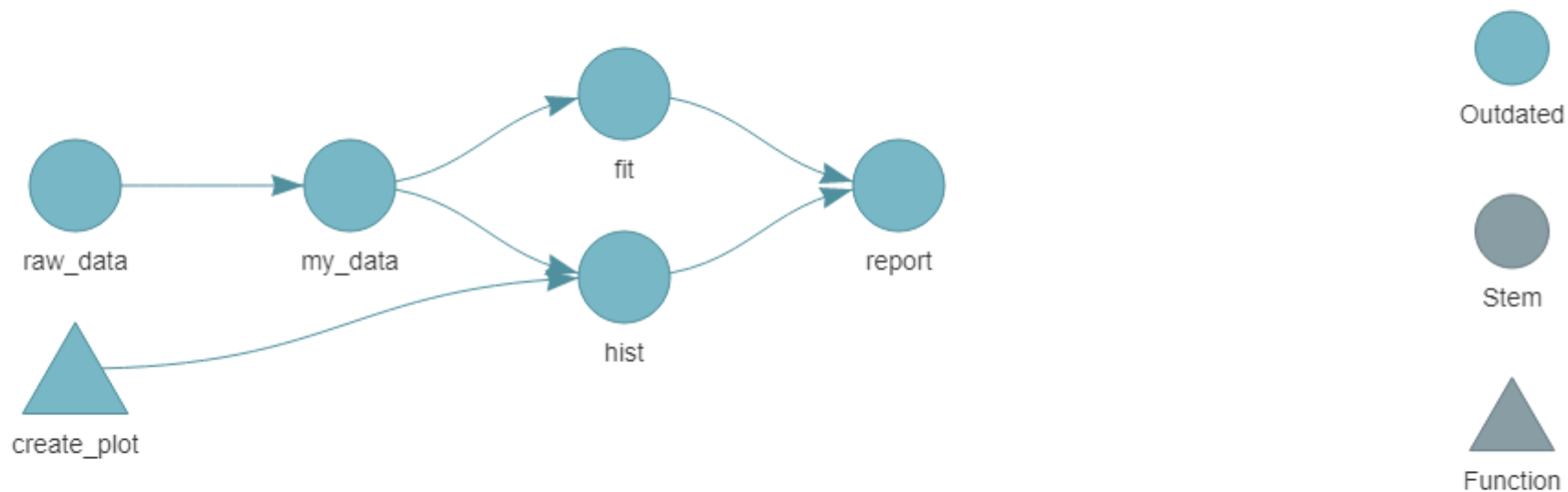
Show a data frame with information about the targets:

```
tar_manifest() %>% select(name, command)
```

```
## # A tibble: 5 x 2
##   name      command
##   <chr>     <chr>
## 1 raw_data  "readxl::read_excel(\"targets_files/raw_iris.xlsx\")"
## 2 my_data   "raw_data %>% mutate(Species = fct_inorder(Species))"
## 3 fit       "lm(Sepal.Width ~ Petal.Width + Species, my_data)"
## 4 hist      "create_plot(my_data)"
## 5 report    "tarchetypes::tar_render_run(path = \"targets_files/report.Rmd\",  \\n      arg~
```

Show the dependencies between the targets:

```
tar_visnetwork()
```



report.Rmd

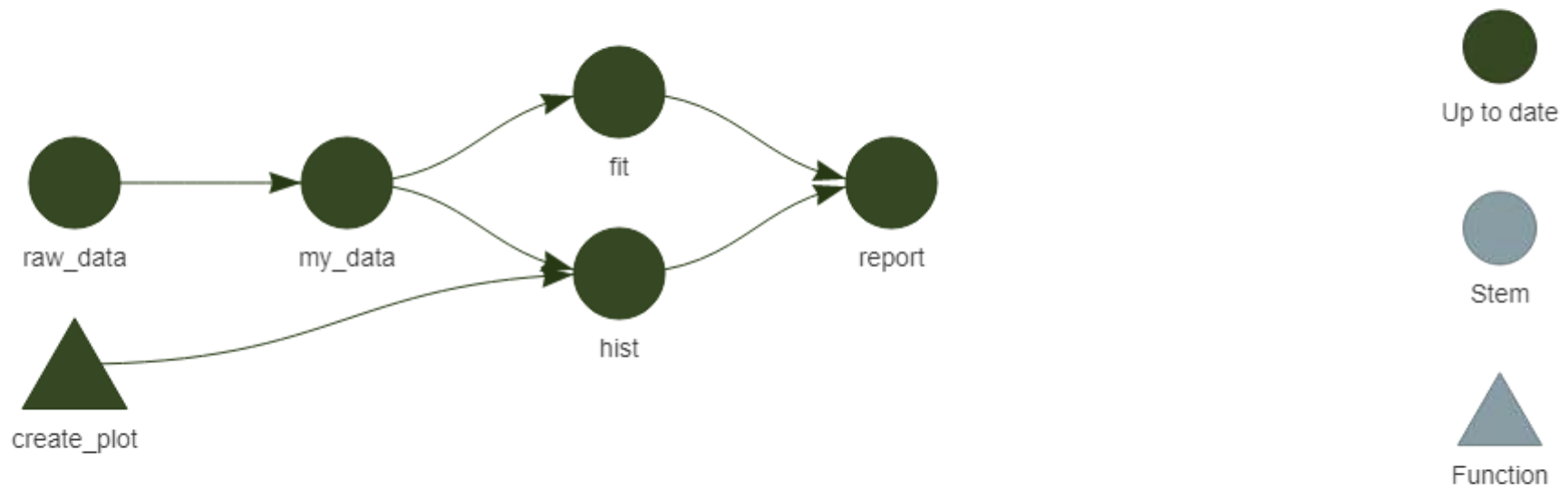
```
---  
title: "Example R Markdown targets file"  
output: github_document  
---  
  
Run `tar_make()` to generate the output `report.md` and its dependencies.  
Because we use `tar_load()` and `tar_read()` below, `targets` knows `report.md` depends on targets  
`fit`, and `hist`.  
  
```${r content}```  
tar_load("fit")
print(fit)
tar_read(hist)
```${r content}```  
  
More:  
  
- Walkthrough: [this chapter of the user manual]  
(https://books.ropensci.org/targets/walkthrough.html#walkthrough)
```


Now we can run the pipeline defined in `_targets.R`. The function creates the targets in the correct order and stores the return values in `_targets/objects/`.

```
tar_make()
```

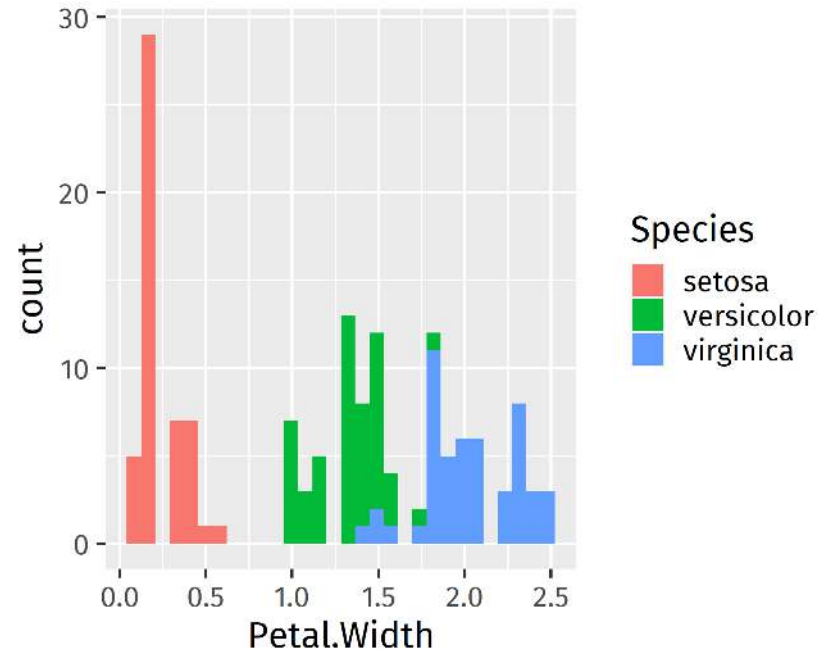
```
## * start target raw_data
## * built target raw_data
## * start target my_data
## * built target my_data
## * start target fit
## * built target fit
## * start target hist
## * built target hist
## * start target report
## * built target report
## * end pipeline
```

```
tar_visnetwork()
```



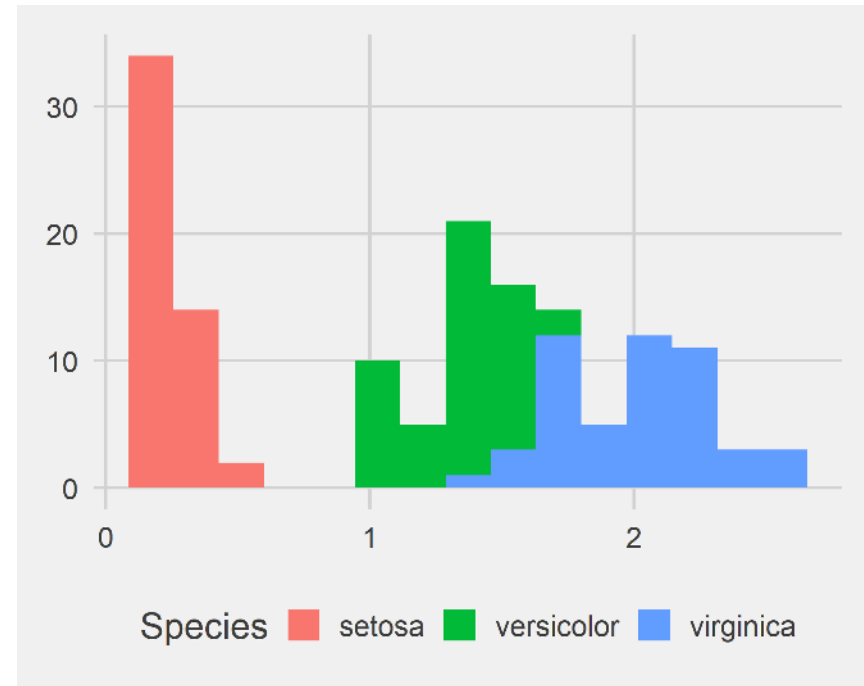
Return a built target with `tar_read()`:

```
tar_read(hist)
```



What happens if we would like to make some changes to the histogram? The number of bins should be set to 15 and a different theme should be applied.

We want the histogram to look like this:



We change the plotting function accordingly:

```
library(ggthemes)
create_plot <- function(data) {
  ggplot(data, aes(x = Petal.Width, fill = Species)) +
    geom_histogram(bins = 15) +
    theme_fivethirtyeight(base_size = 18)
}
```

`_targets.R` now looks like this:

```
library(targets)
library(tarchetypes)
options(tidyverse.quiet = TRUE)

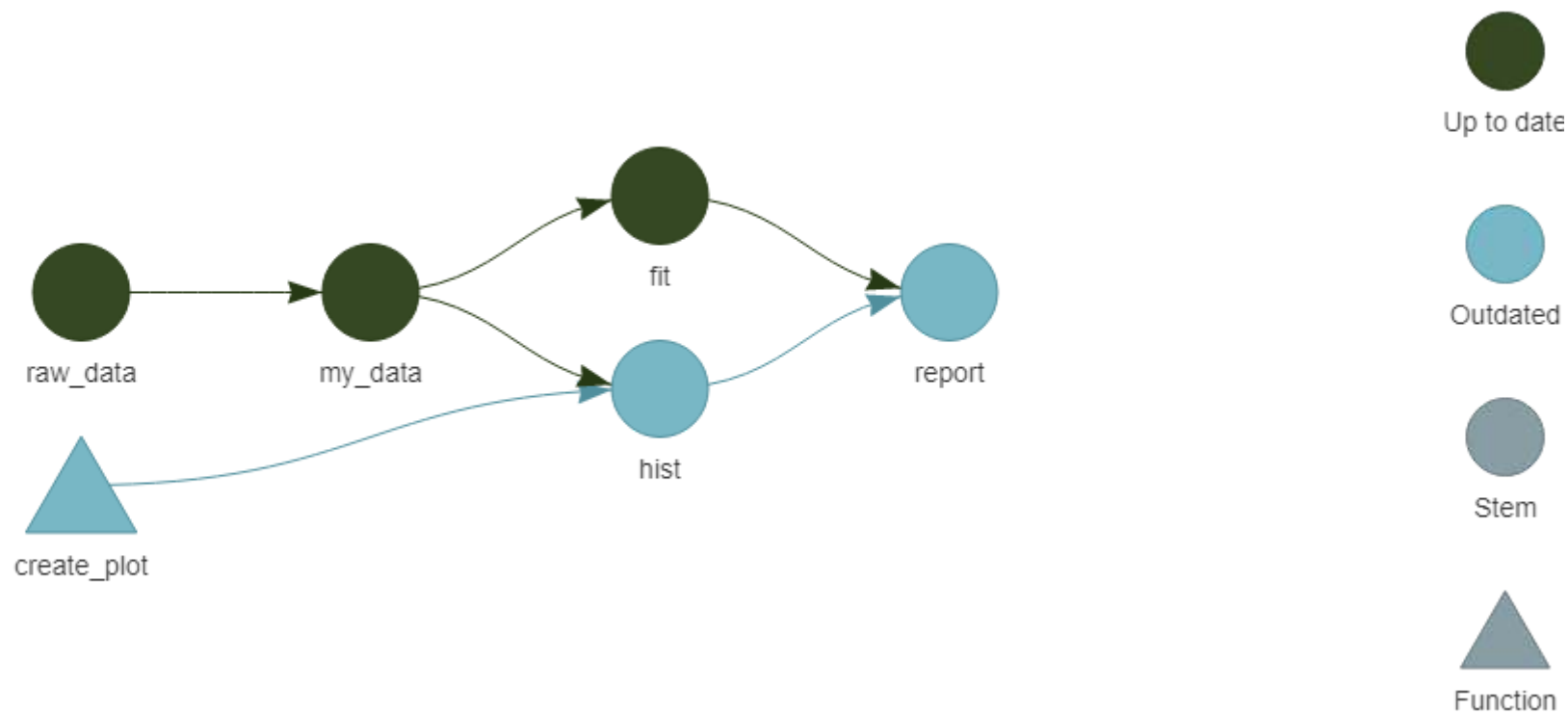
create_plot <- function(data) {
  ggplot(data, aes(x = Petal.Width, fill = Species)) +
    geom_histogram(bins = 15) +
    theme_fivethirtyeight(base_size = 18)
}

tar_option_set(packages = c("dplyr", "forcats", "ggplot2", "ggthemes", "readr", "tidyr"))

list(
  tar_target(raw_data, readxl::read_excel("targets_files/raw_iris.xlsx")),
  tar_target(my_data, raw_data %>% mutate(Species = fct_inorder(Species))),
  tar_target(hist, create_plot(my_data)),
  tar_target(fit, lm(Sepal.Width ~ Petal.Width + Species, my_data)),
  tar_render(report, "targets_files/report.Rmd")
)
```



```
tar_visnetwork()
```



Note that the targets `hist` and `report` (and only these targets) have become outdated because they depend on the modified function `create_plot()`.

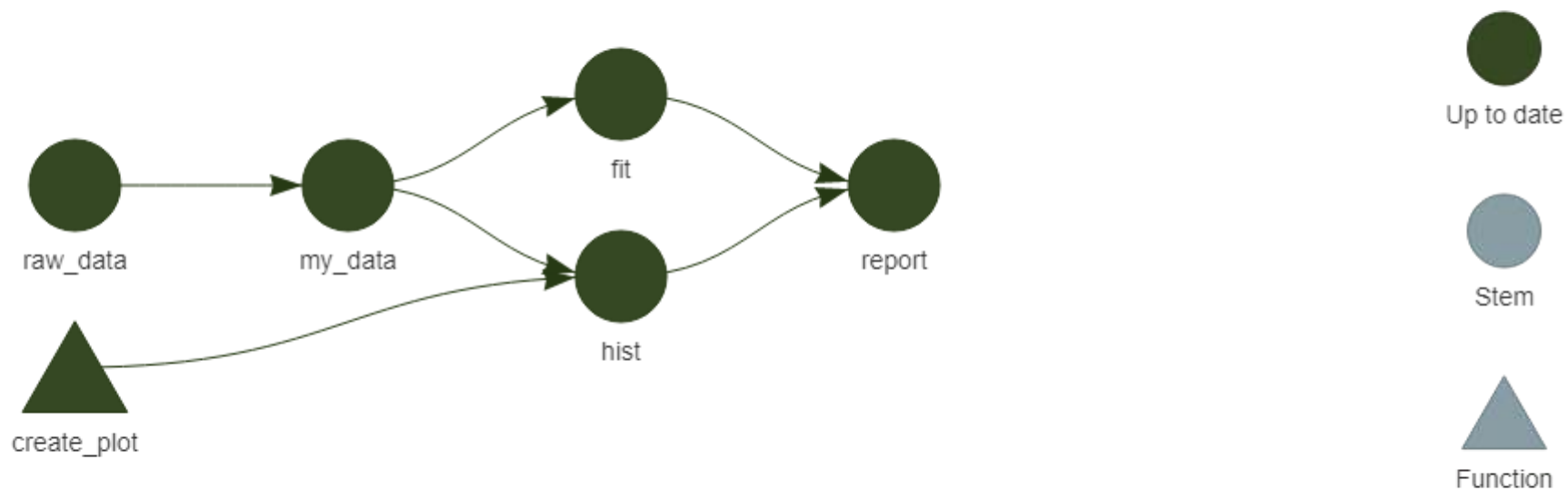
We rerun the pipeline Only `hist` and `report` are recomputed. The values of the other targets are pulled from the cache.

```
tar_make()
```

```
## v skip target raw_data
## v skip target my_data
## v skip target fit
## * start target hist
## * built target hist
## * start target report
## * built target report
## * end pipeline
```

In the dependency graph, all targets are up-to-date again:

```
tar_visnetwork()
```



Recommendations on naming files & folders

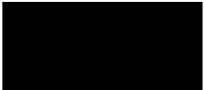


Photo by [Jeff Ackley](#)



[How to name files](#)

by [Jennifer \(Jenny\) Bryan](#)



[PDF version](#)

Package library management with renv



Photo by [ch Claudio Schwarz](#) | [@purzlbaum](#)

Package library management

- R packages are stored in one or more directories on your computer (`.libPaths()`)
 - **user library**: packages that *you* install are (usually) contained in `.libPaths()[1]`
 - **system library**: default packages are (usually) contained in `.libPaths()[2]`
- RStudio Packages tab provides a convenient interface for installing and updating packages
- Use `install.packages(<pkgname>)` to install or update packages via the console
 - installs **binary** version from CRAN (no additional tools required)
 - binary: single, OS-specific file (.zip for Windows, .tgz for macOS)

Sometimes, you need the latest development version hosted on GitHub, BitBucket, etc.

- Use `remotes::install_github(<username>/<pkgname>)` to install or packages via the console
 - installs **source** version
 - source: OS-independent directory with a specific structure
 - example: `remotes::install_github("rstudio/gt")`
 - additional tools for compilation necessary, e.g. Rtools for Windows

Package library management

- many packages are continuously under development even after CRAN release
- package maintainers generally pay attention to backwards compatibility, but there is no guarantee that there will not be changes that introduce existing code to fail

Scenarios where we need another way of package management:

- ¶ *We want to ship the final project code to our customers. The code does not only have to work now, but also in 3 months, 6 months, 1 year from now on.*
- ¶ *We do not want to keep track of code-breaking package updates.*
- ¶ *For other projects, we must be able to update packages, in order to make use of new useful features.*

Package library management with renv

`renv`: **dependency manager** that helps to set up and maintain **project-local R libraries**

- **isolated**: each project has its own library of packages
 - → updating packages globally or in another package will not affect current project
- **portable**: version numbers of all "active" packages are tracked in a "lock file"
 - → facilitates collaboration
- **reproducible**: enables saving a "snapshot", i.e., a state of the project library that you know is working
 - → safety net in case a package update causes problems
- **disk-space-efficient**: packages are installed into a global cache; different projects that use the same version of a package will pull a "shared" package installation from the global cache
 - → also reduces installation time when a package has already been installed by another project that is managed with renv



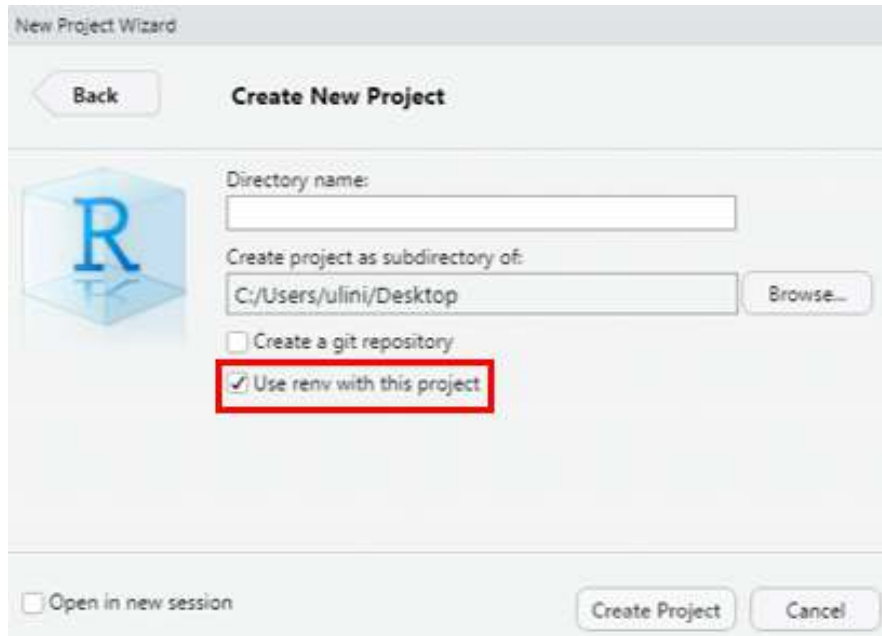
📖 Kevin Ushey. Introduction to renv. <https://rstudio.github.io/renv/articles/renv.html>. Last accessed 10.05.2021.

📖 RStudio. Upgrading Packages: How to Safely Upgrade Packages. <https://environments.rstudio.com/upgrades.html>. Last accessed 10.05.2021.

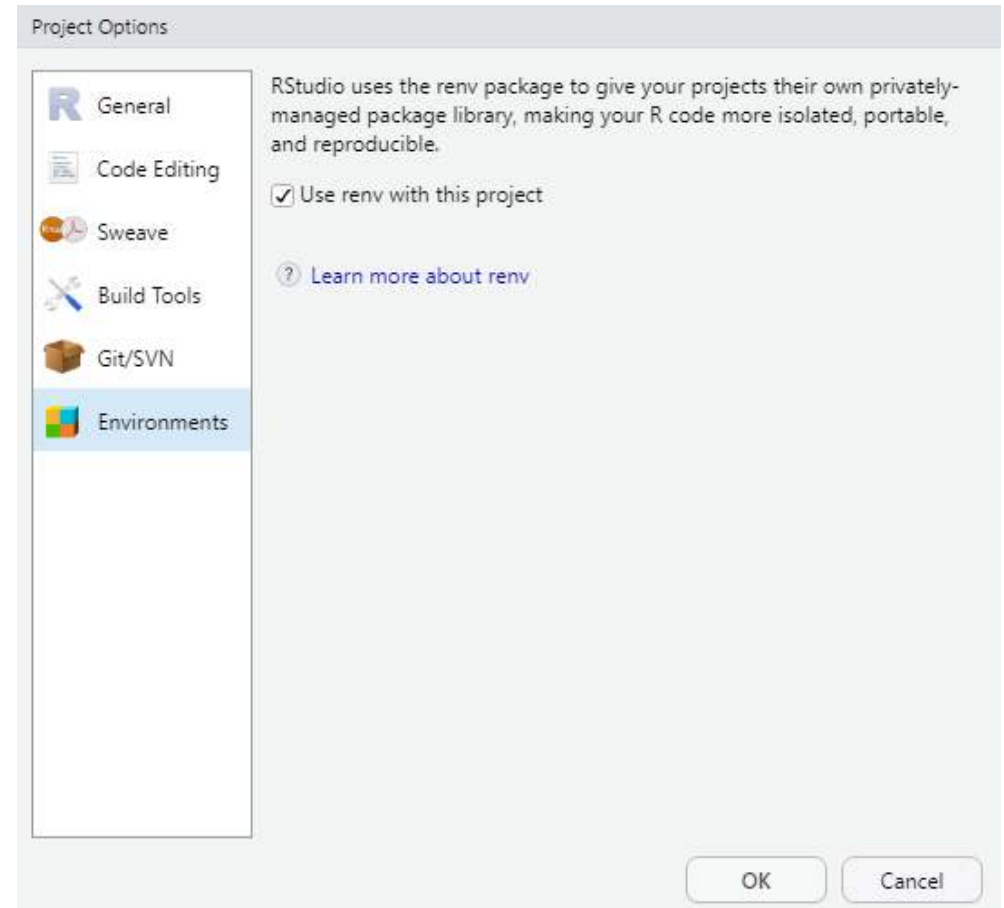


📺 Kevin Ushey. [renv: Project Environments for R. Keynote at RSTUDIO::CONF 2020.](#) Last accessed 10.05.2021.

Activate project-local package library in RStudio

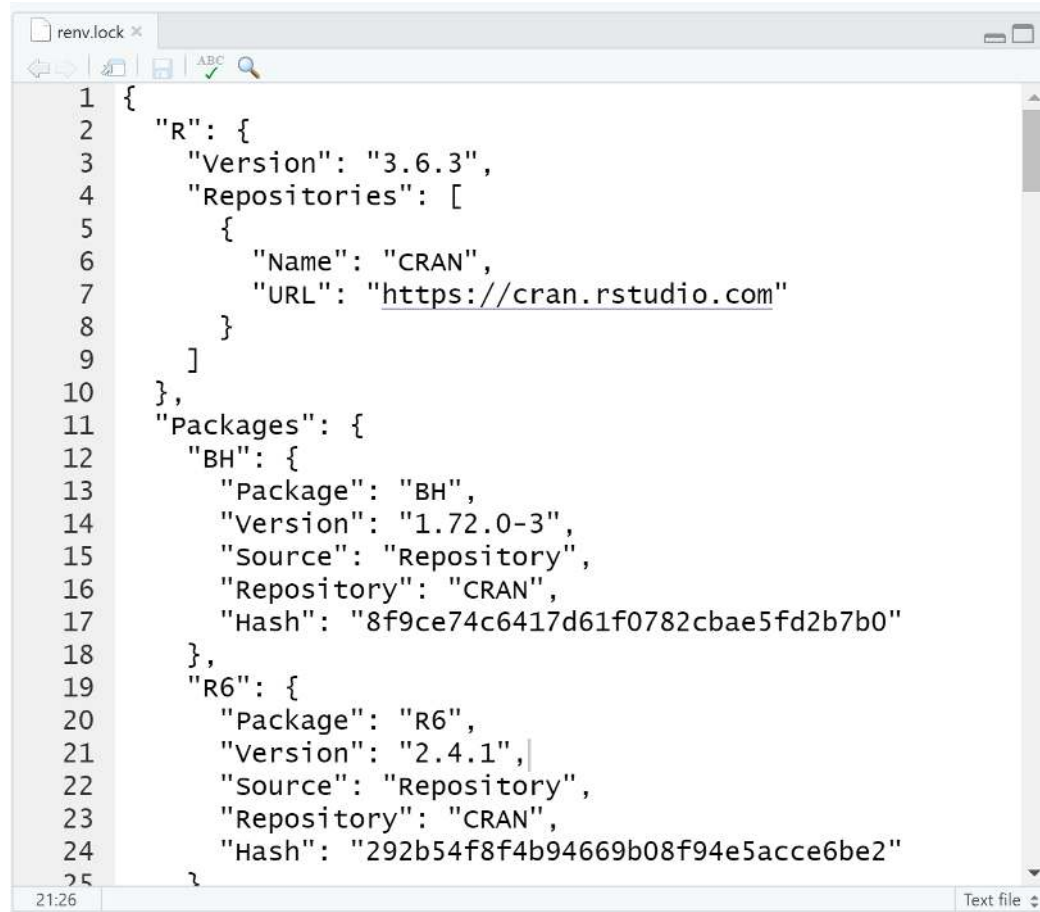


Create a new RStudio project with renv



Use renv in your existing RStudio project

Example of renv.lock file



```
1 {
2   "R": {
3     "Version": "3.6.3",
4     "Repositories": [
5       {
6         "Name": "CRAN",
7         "URL": "https://cran.rstudio.com"
8       }
9     ]
10  },
11  "Packages": {
12    "BH": {
13      "Package": "BH",
14      "Version": "1.72.0-3",
15      "Source": "Repository",
16      "Repository": "CRAN",
17      "Hash": "8f9ce74c6417d61f0782cbae5fd2b7b0"
18    },
19    "R6": {
20      "Package": "R6",
21      "Version": "2.4.1",
22      "Source": "Repository",
23      "Repository": "CRAN",
24      "Hash": "292b54f8f4b94669b08f94e5acce6be2"
25    }
26  }
```

Debugging

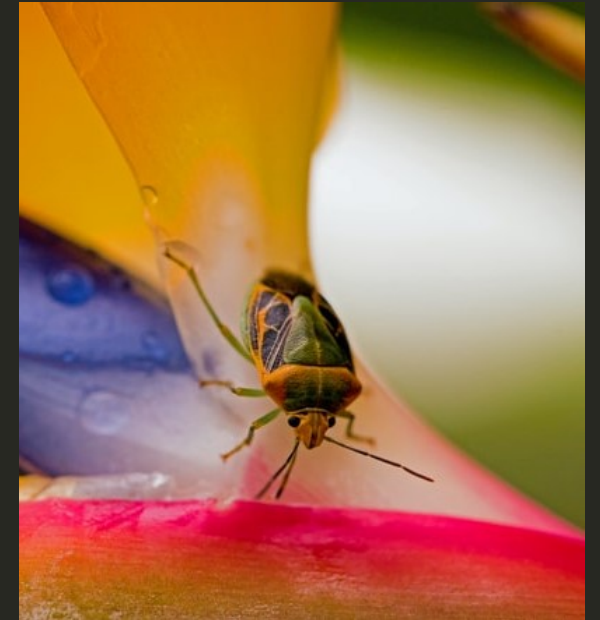


Photo by [David Clode](#)

Error handling

How to find the source of an error in your R code?

Recommended strategy:

1. **Restart R.** → Make sure the error *always* occurs.
2. **Google the error message.** → It is likely that there is already an existing solution on Stack Overflow, RStudio Community, Twitter or other fora.
3. **Debug.**
4. **Ask for help.** → Prepare a reproducible example and ask the internet.

Debugging

```
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) {
  if (!is.numeric(d)) {
    stop("`d` must be numeric", call. = FALSE)
  }
  d + 10
}
```

🤔 What is the result of this code when we run `f("10")`?

```
f("10")
```

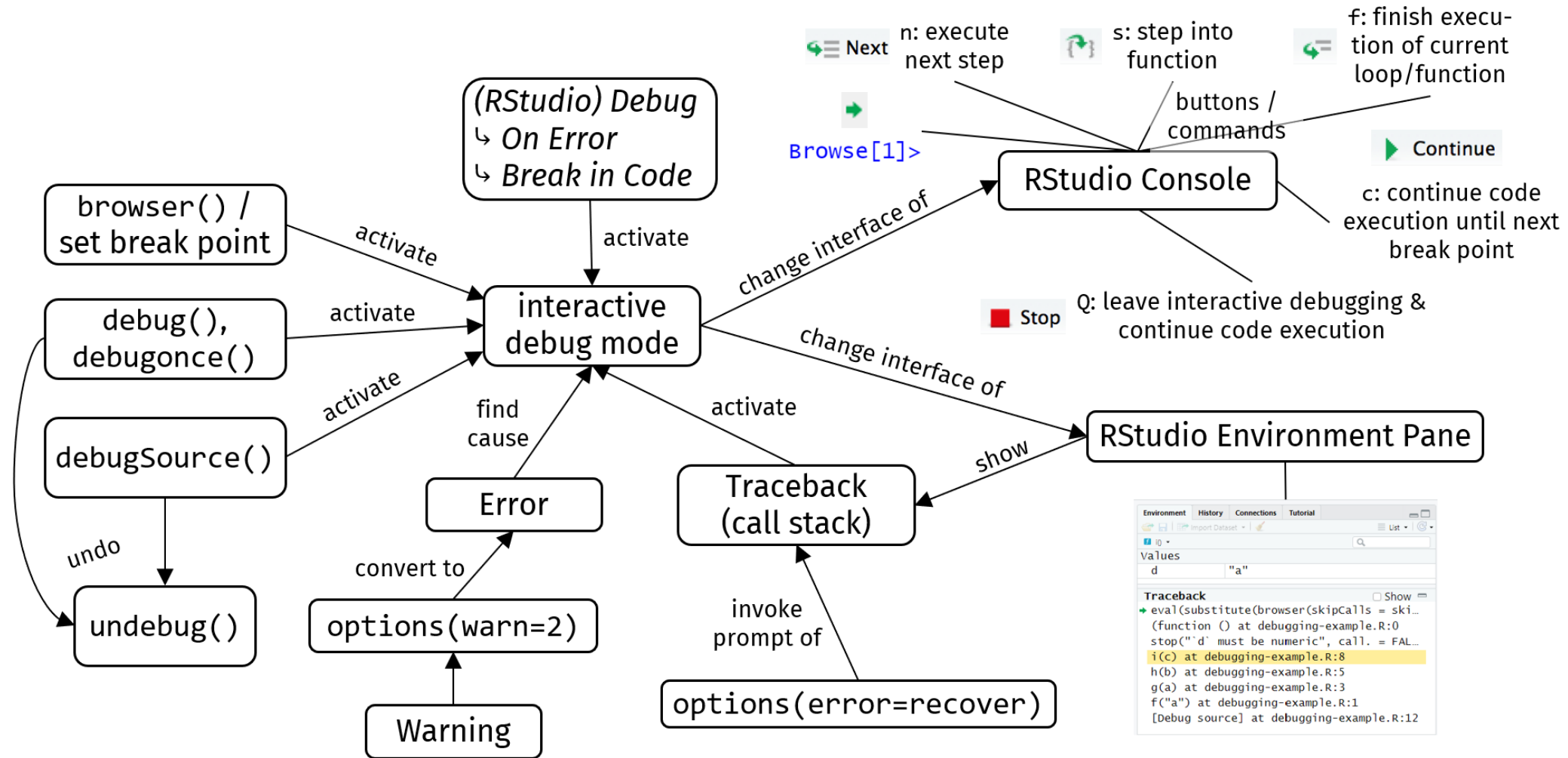
```
## Error: `d` must be numeric
```

Useful resources:

- 📖 Example from: Hadley Wickham. Advanced R. [Chapter Debugging](#). Second Edition. Chapman & Hall/CRC, 2019.
- 📖 Jonathan McPherson. [Debugging with RStudio](#). Last accessed 10.05.2021.
- 📺 Jenny Bryan. [Object of type 'closure' is not subsettable](#). Keynote at RSTUDIO::CONF 2020. Last accessed 10.05.2021.
- 📖 Garrett Grolmund. Hands-On Programming with R. [Appendix E: Debugging R Code](#). O'Reilly, 2014.

Interactive debug mode

1. set break point (click on gray space left to line number or **↑Shift+F9**)
2. console debugging commands (`n, s, f, c, Q`)
3. `browser()`
4. `debug()`, `debugOnce()`
5. `undebug()`
6. RStudio: *Debug* → *On Error* → *Break in Code*
7. `debugSource()`
8. `options(error = recover)`
9. `options(warn = 2)`




Session info

```
## setting value
## version R version 4.0.5 (2021-03-31)
## os Windows 10 x64
## system x86_64, mingw32
## ui RTerm
## language (EN)
## collate English_United States.1252
## ctype English_United States.1252
## tz Europe/Berlin
## date 2021-05-10
```

package	version	date	source
dplyr	1.0.5	2021-03-05	CRAN (R 4.0.4)
forcats	0.5.1	2021-01-27	CRAN (R 4.0.3)
ggplot2	3.3.3	2020-12-30	CRAN (R 4.0.3)
ggthemes	4.2.4	2021-01-20	CRAN (R 4.0.3)
kableExtra	1.3.4	2021-02-20	CRAN (R 4.0.3)
knitr	1.31	2021-01-27	CRAN (R 4.0.3)
purrr	0.3.4	2020-04-17	CRAN (R 4.0.2)

package	version	date	source
readr	1.4.0	2020-10-05	CRAN (R 4.0.3)
stringr	1.4.0	2019-02-10	CRAN (R 4.0.2)
tarchetypes	0.2.0.9000	2021-05-10	Github (ropensci/tarchetypes)
targets	0.4.2	2021-04-30	CRAN (R 4.0.5)
tibble	3.1.1	2021-04-18	CRAN (R 4.0.5)
tidyr	1.1.3	2021-03-03	CRAN (R 4.0.4)
tidyverse	1.3.0	2019-11-21	CRAN (R 4.0.2)

A photograph of a modern residential courtyard. In the center, a large, leafy tree stands on a grassy area. To the left, a paved path leads towards the background. In the middle ground, there are several metal benches. To the right, a multi-story white building with many windows is visible. The sky is blue with some clouds. The overall scene is bright and sunny.

Thank you! Questions?