Lab 000

Data cleaning and workflow [1/N]

Edward Rubin

07 January 2022

Admin

Admin

Basic **workflow** (best) practices (*i.e.*, *Projects*)

- RStudio and projects
- Naming conventions
- Pipes (`%>%`)
- Data cleaning with `dplyr`

Reminders

Reminder: Readings for next week

- ISL Ch1–Ch2
- Prediction Policy Problems by Kleinberg *et al.* (2015)

Improving your workflow

Improving your workflow

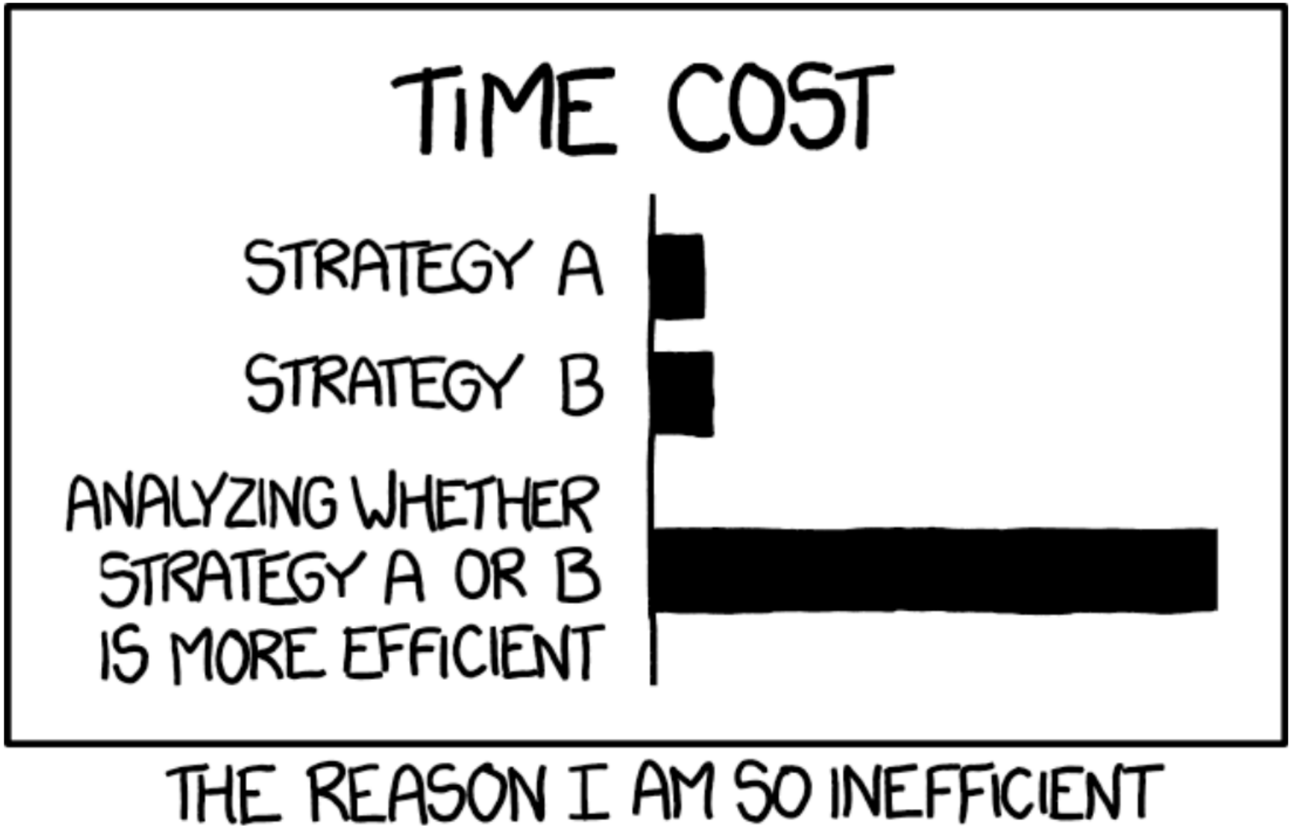
Data cleaning, manipulation, and analysis can be grueling, but optimizing your workflow can speed things along and make them less painful.†

A few dimensions that can help

- Understand how to interact with RStudio
- Use `R` projects
- Follow reasonable naming conventions
- `dplyr` and pipes
- Write your own functions (future lab)
- Use loops and parallelization (future lab)
- Hire an intern/assistant to do your work for you

† Notice that I said *less* painful.

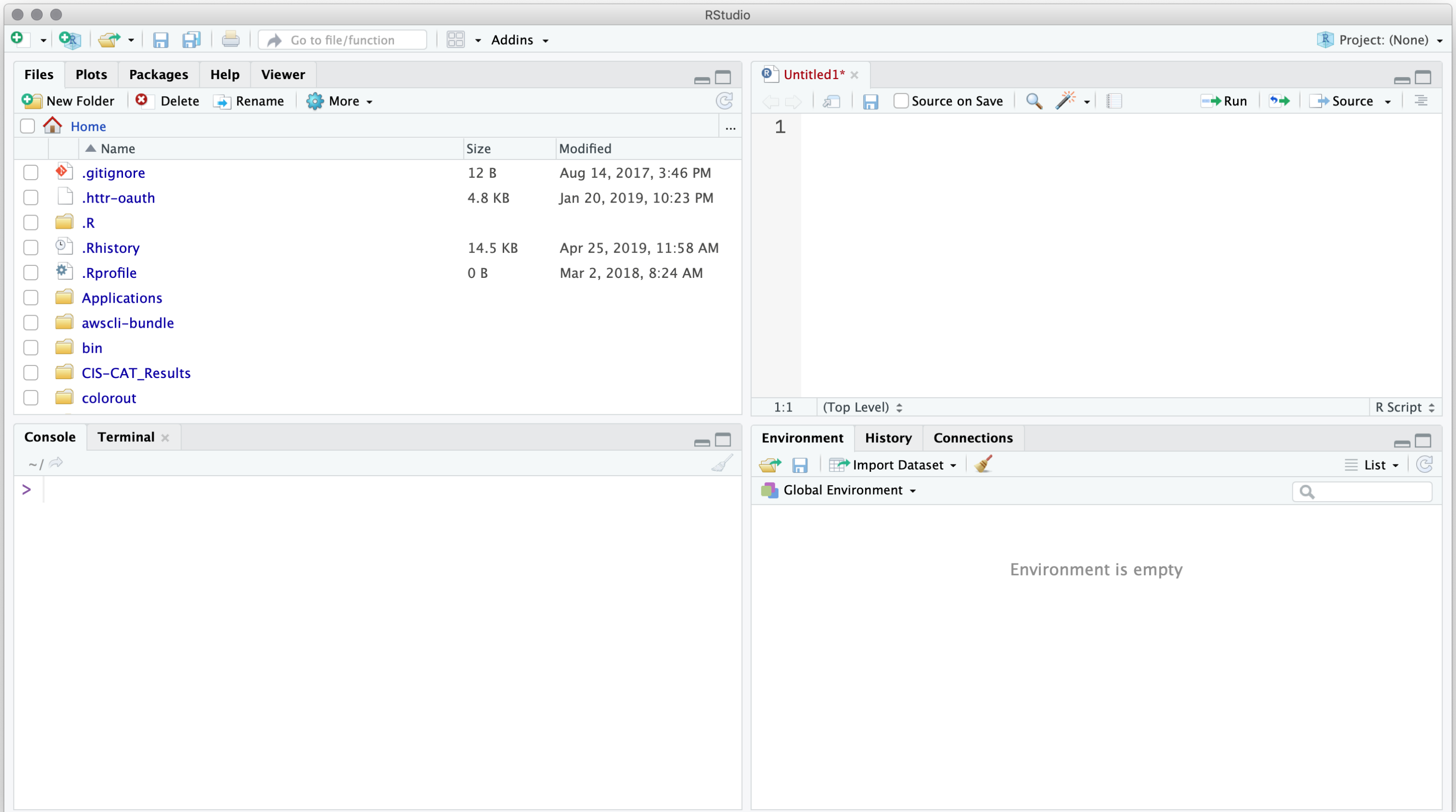
Efficiency



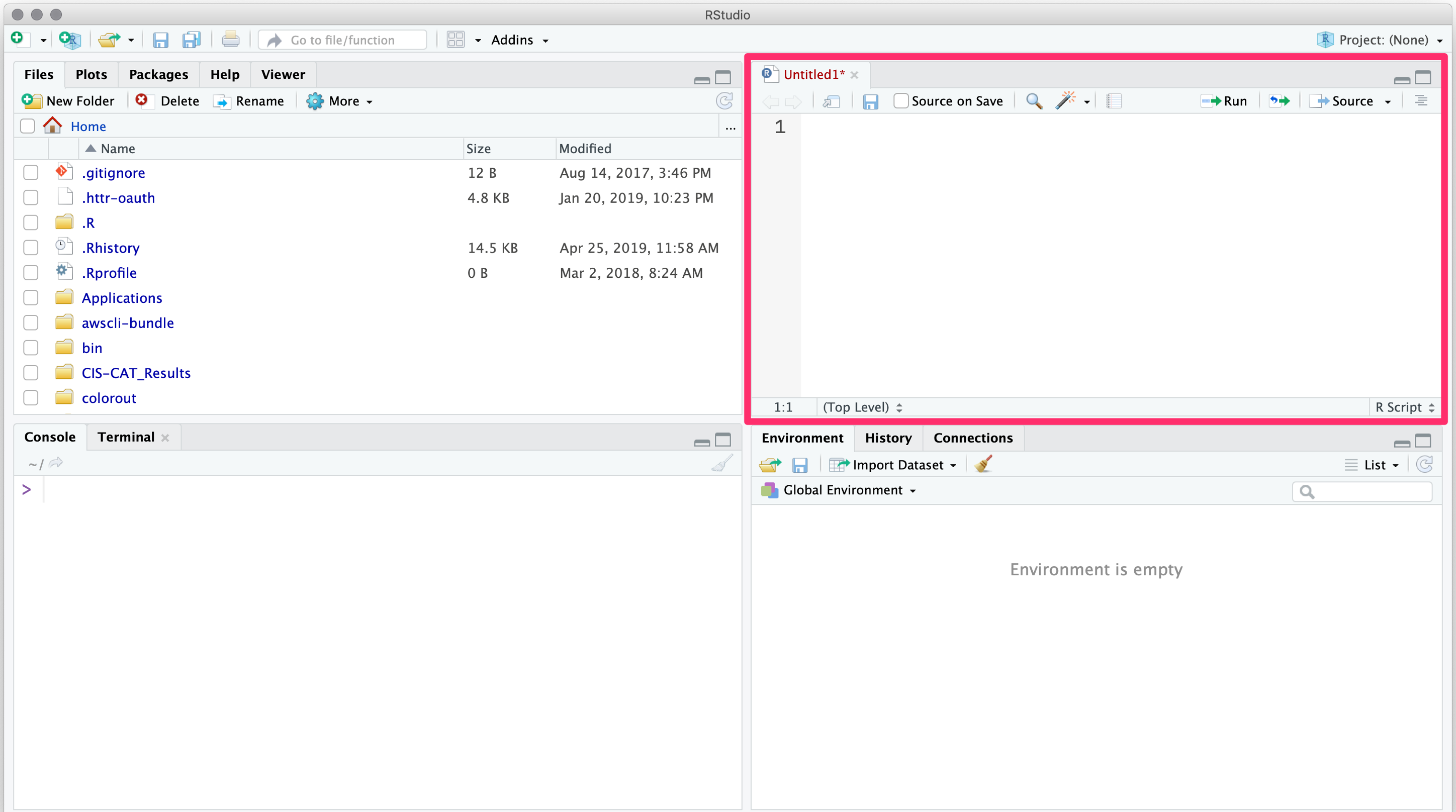
Source: xkcd

RStudio

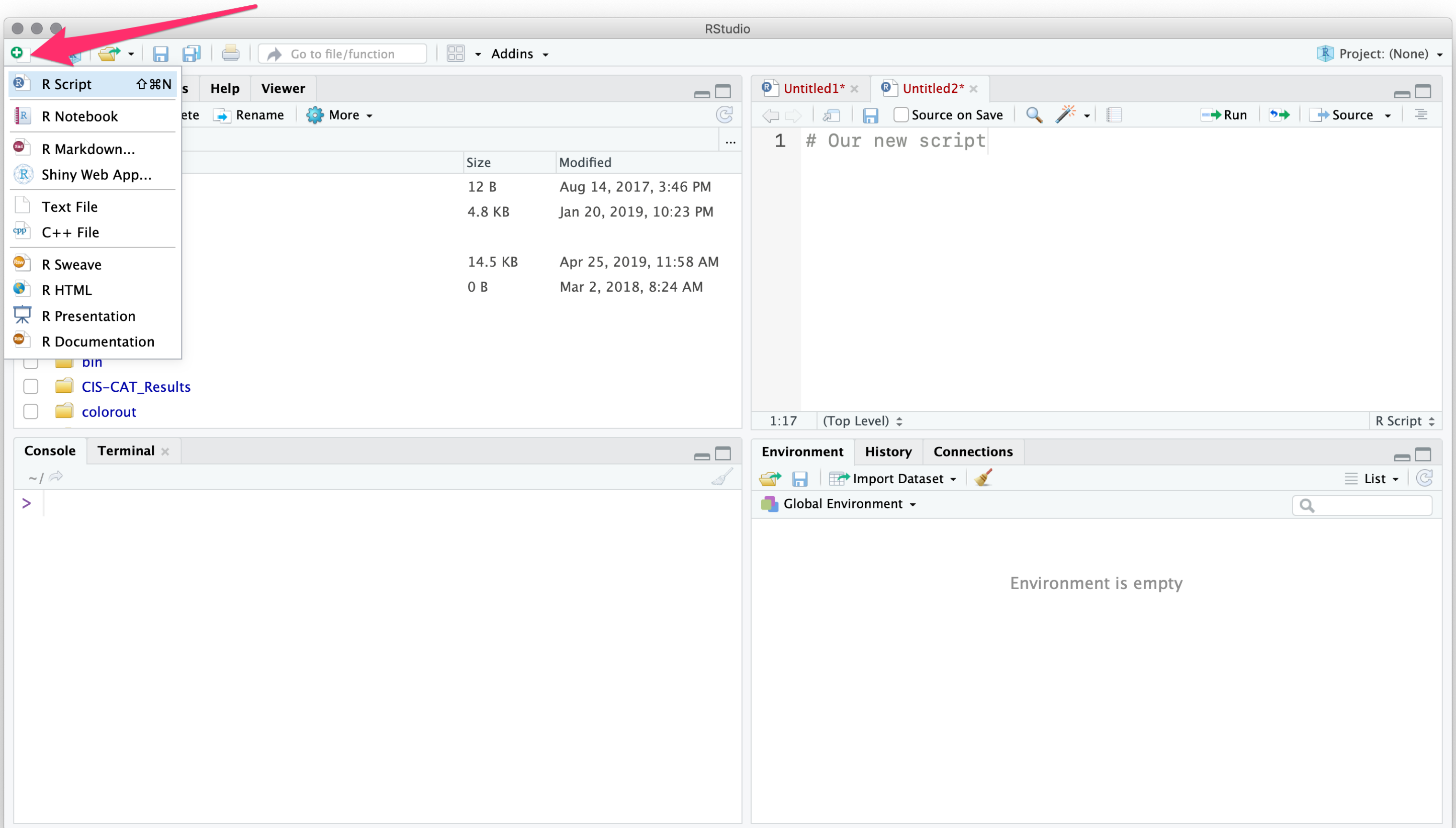
Let's recap some of the major features in `RStudio`…



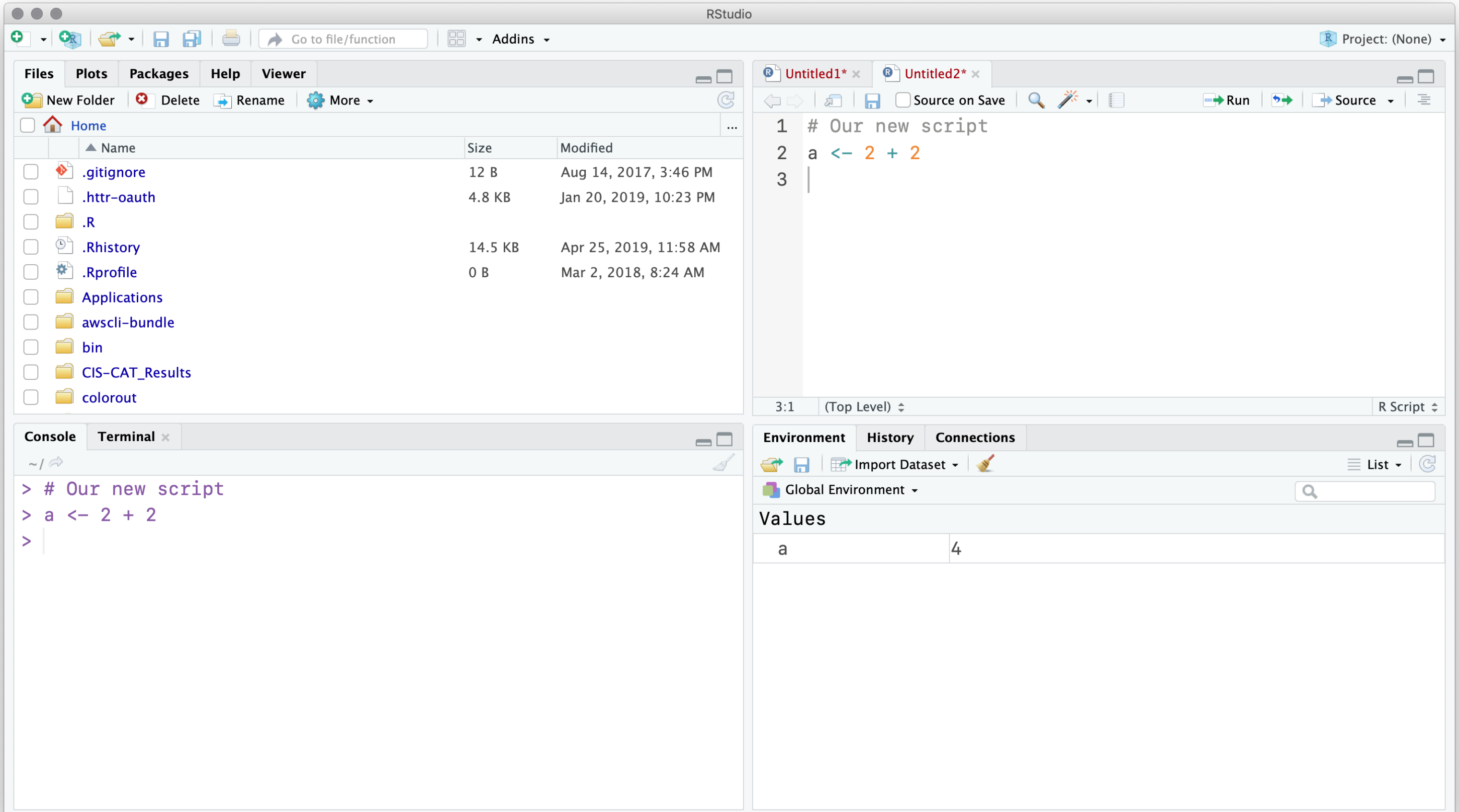
First, you write your `R` scripts (source code) in the **Source** pane.



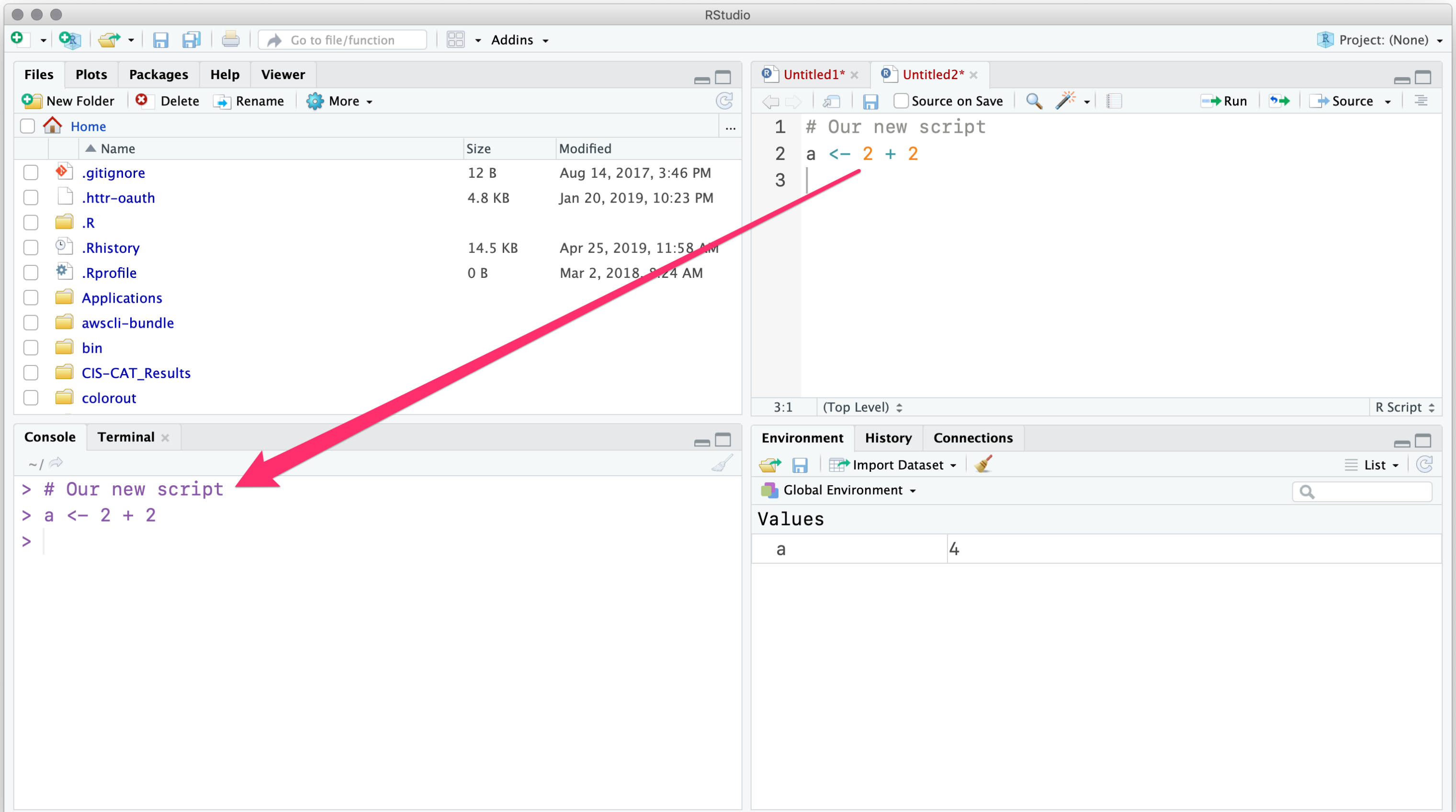
You can use the menubar or ⇧+⌘+N to create new R scripts.



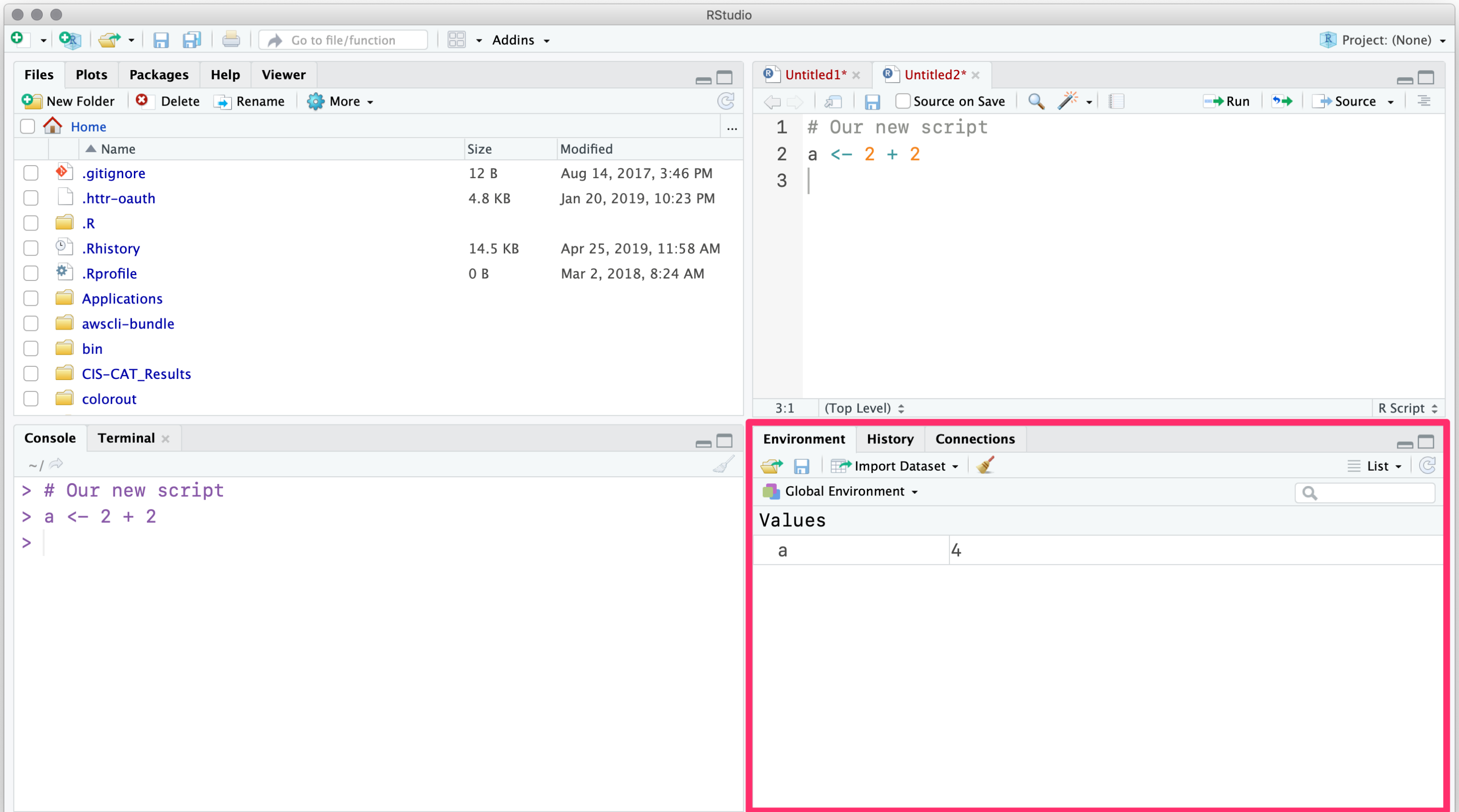
To execute commands from your `R` script, use `⌘+Enter`.



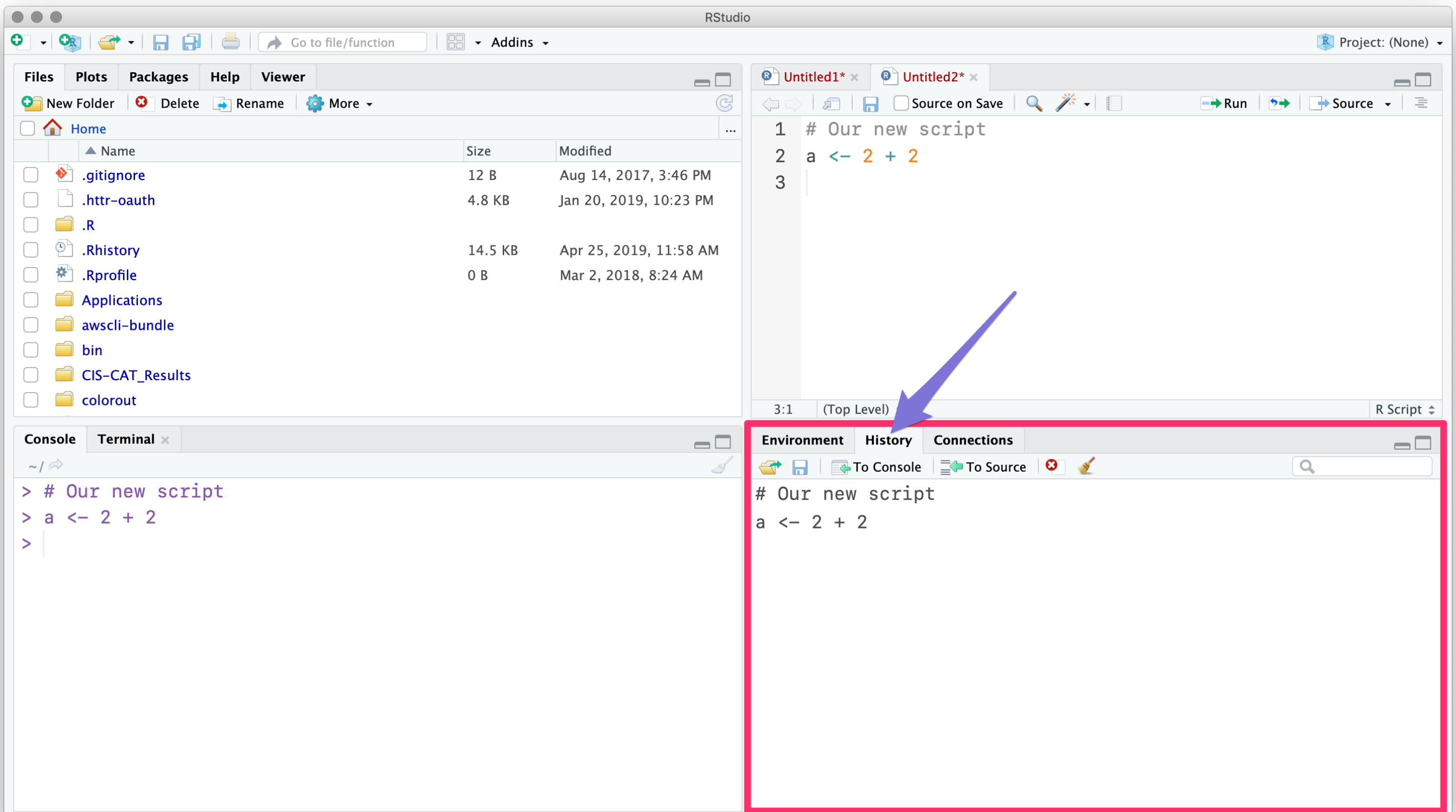
`RStudio` will execute the command in the terminal.



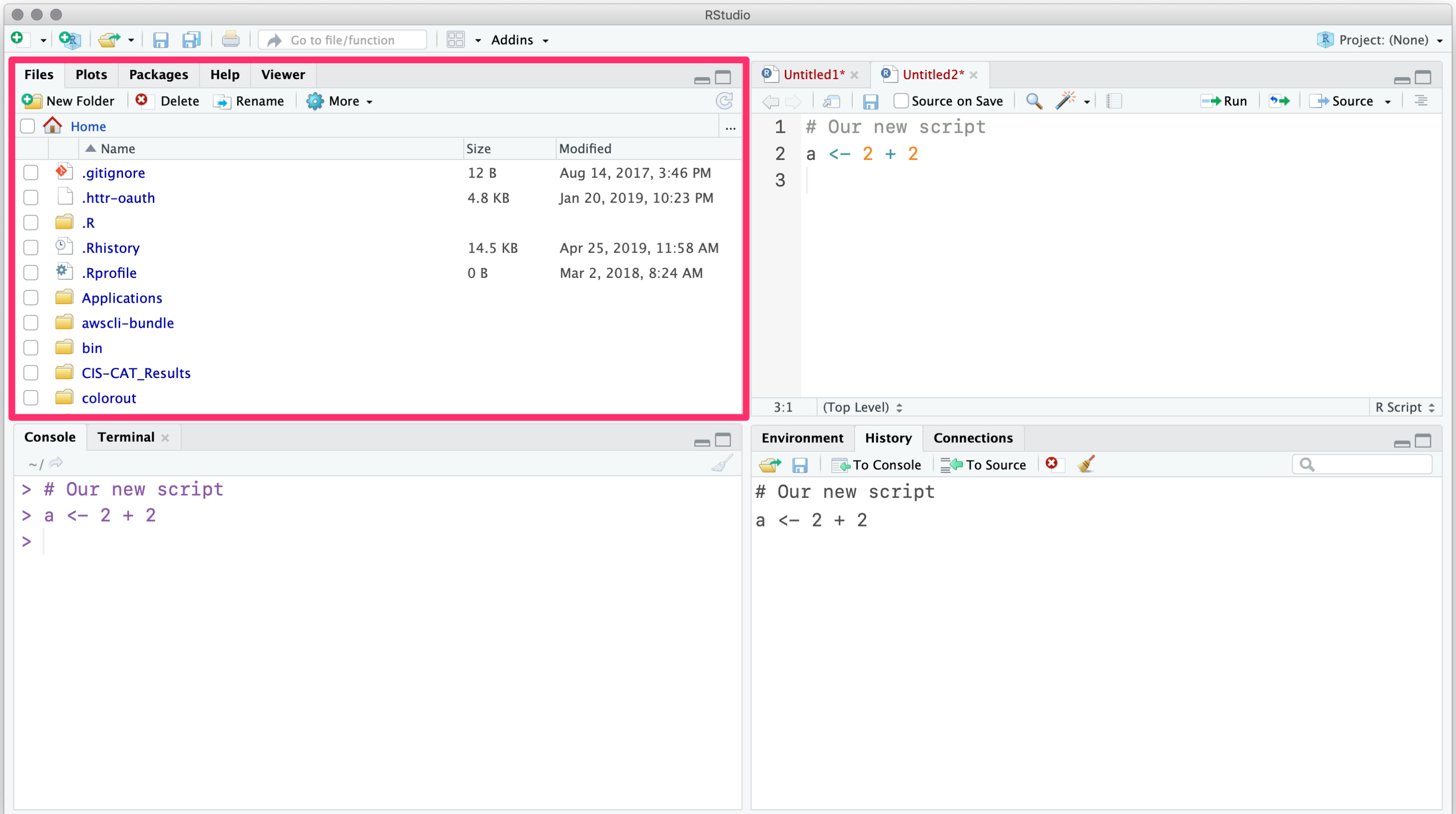
You can see our new object in the **Environment** pane.



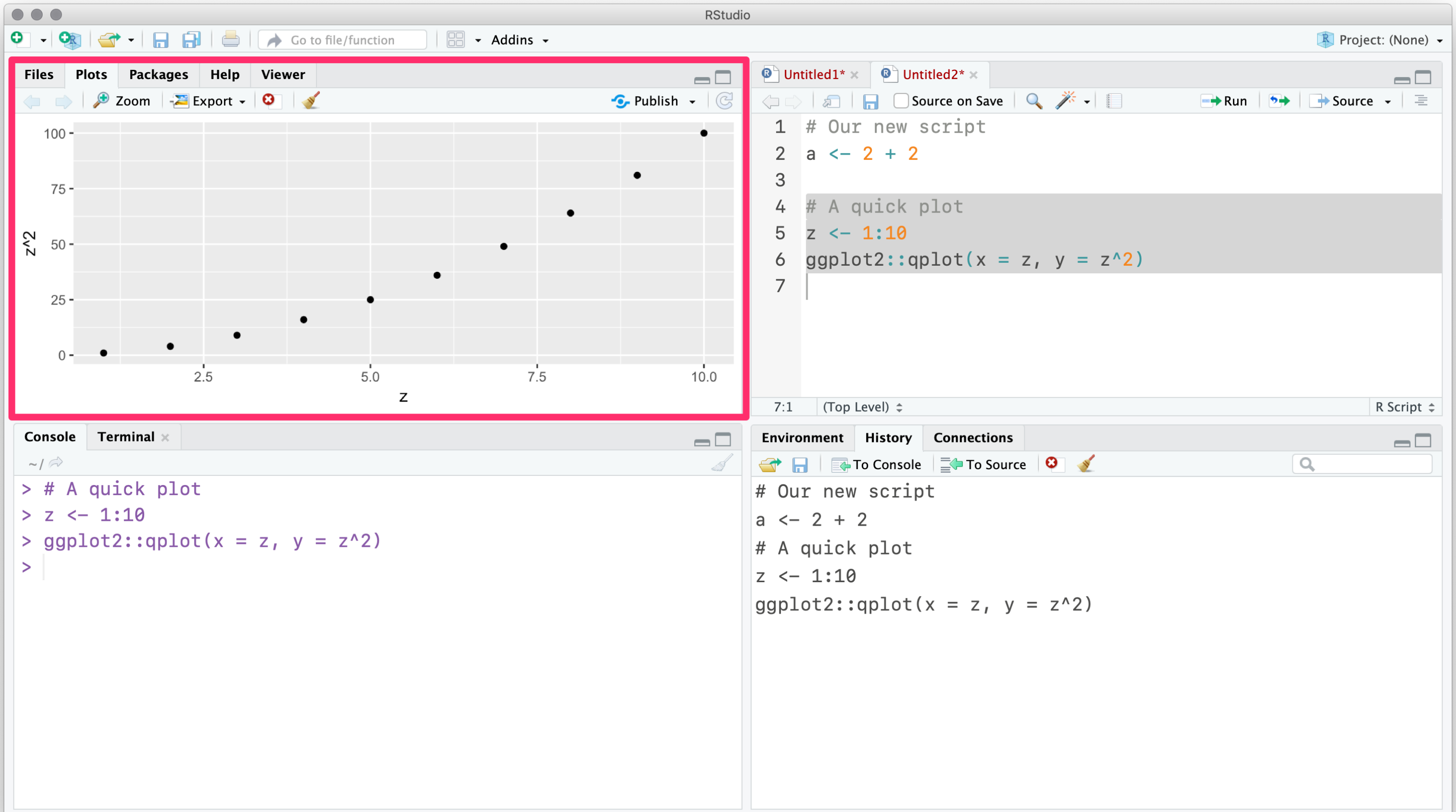
The **History** tab (next to **Environment**) records your old commands.



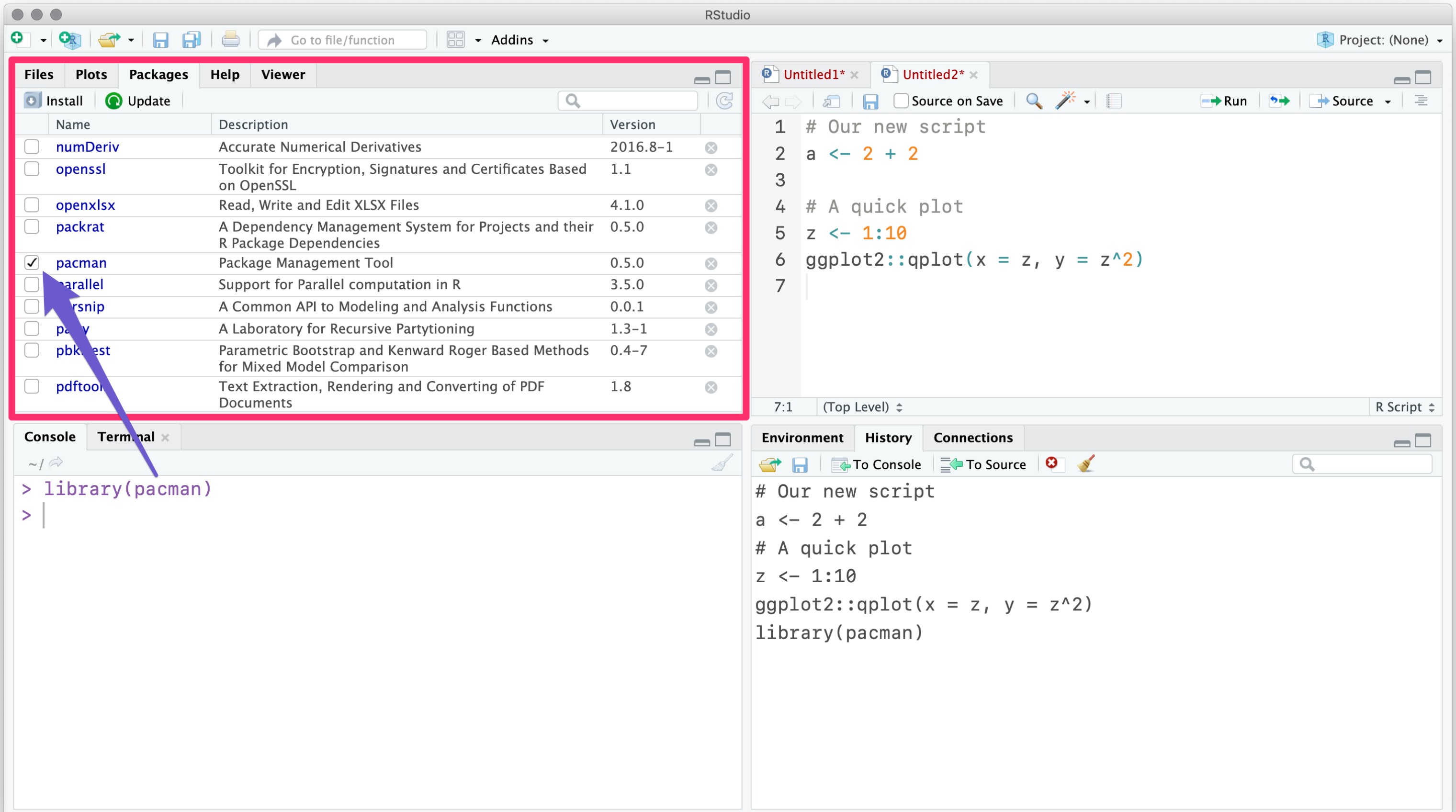
The **Files** pane is file explorer.



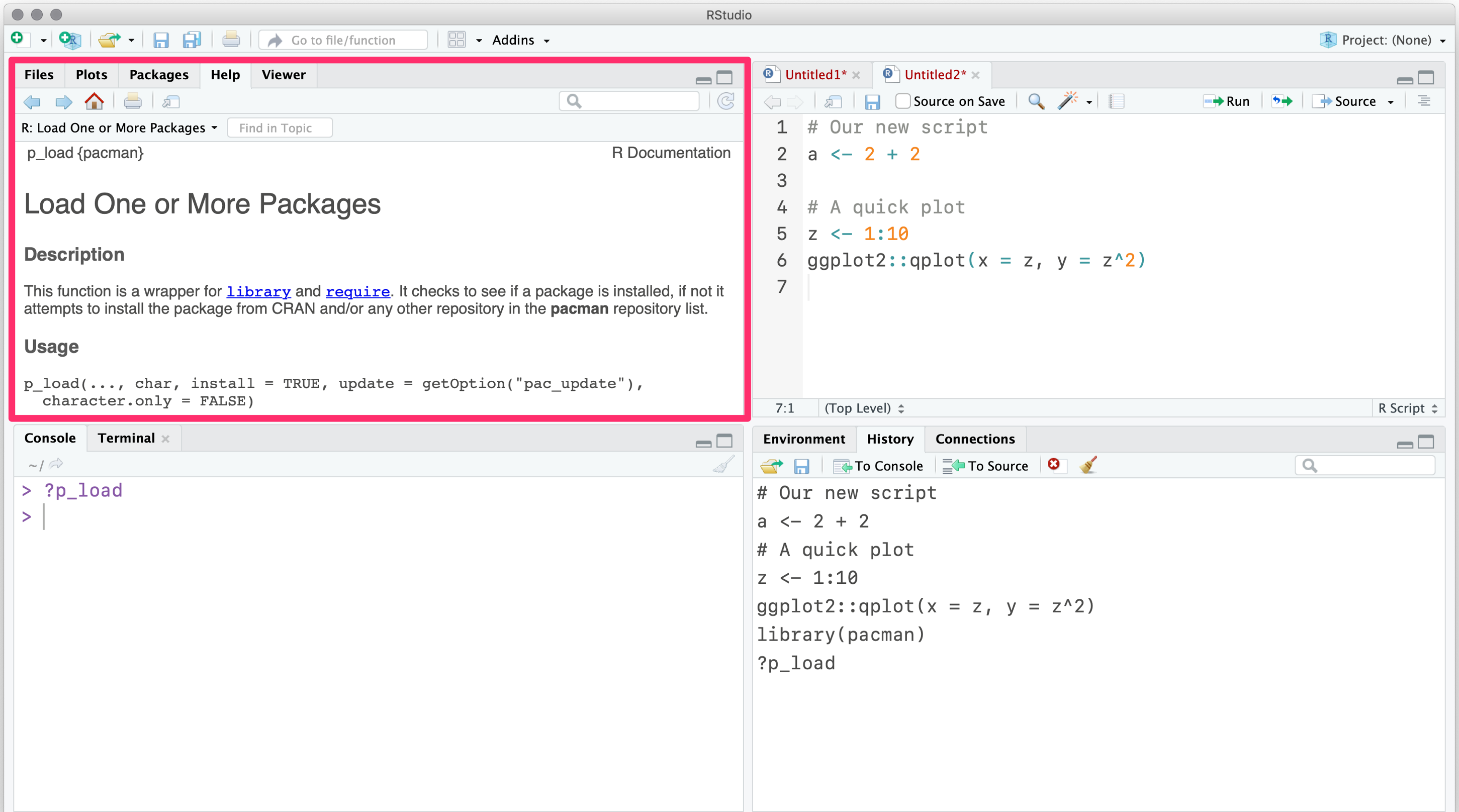
The **Plots** pane/tab shows… plots.



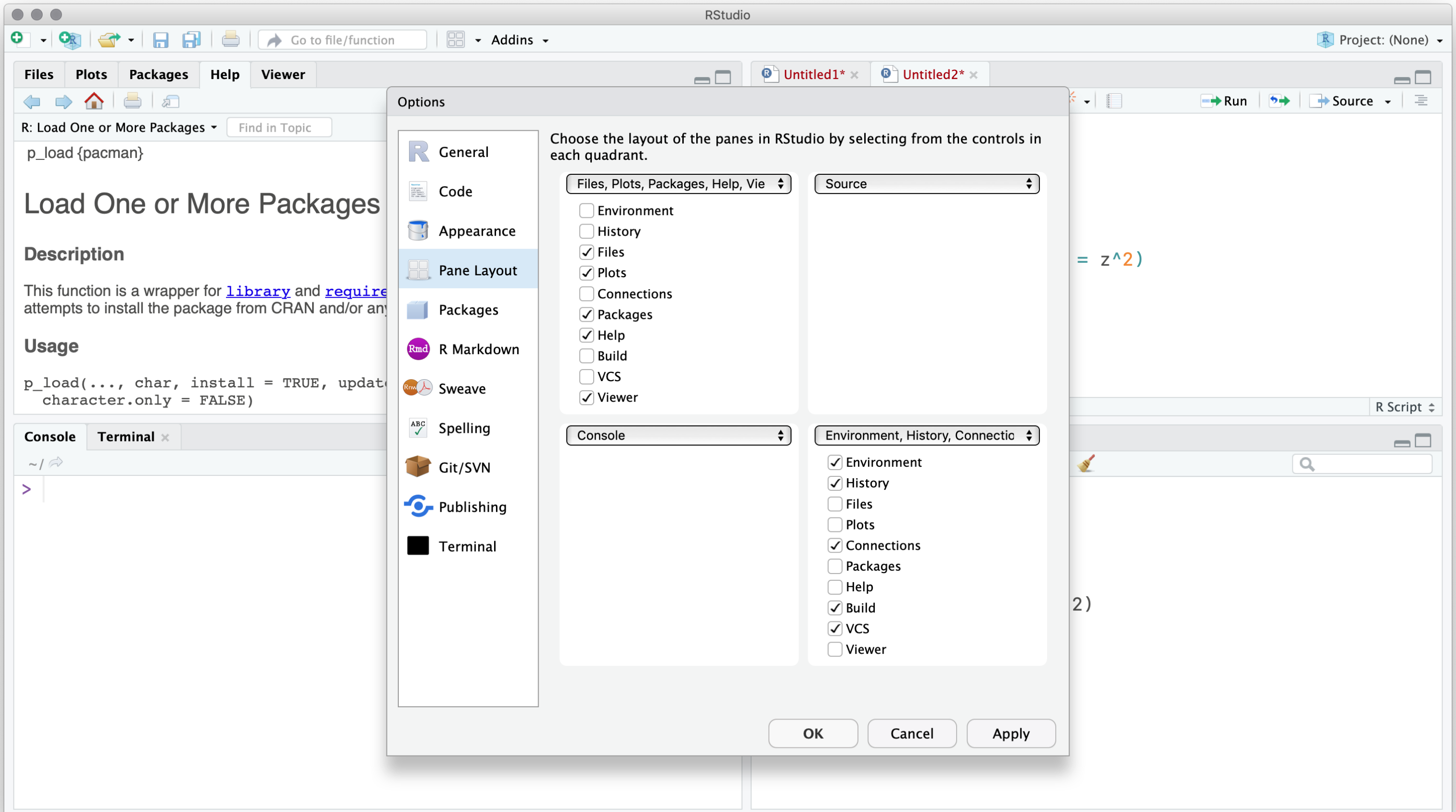
Packages shows installed packages and whether they are **loaded**.



The **Help** tab shows help documentation (also accessible via `?`).



Finally, you can customize the actual layout and many other items.



`R` and `RStudio`

Related best practices

1. Write code in `R` scripts. Troubleshoot in `RStudio`. Then run the scripts.
2. Comment your code. (`# This is a comment`)
3. Name objects/variables/files with intelligible, standardized names.
   - **BAD** `ALLCARS`, `Vl123a8`, `a.fun`, `cens.12931`, `cens.12933`
   - **GOOD** `unique_cars`, `health_df`, `sim_fun`, `is_female`, `age`
4. Write code that is readable (see comments comment above).
5. Use projects in `RStudio` (next). And organize your projects.

Projects

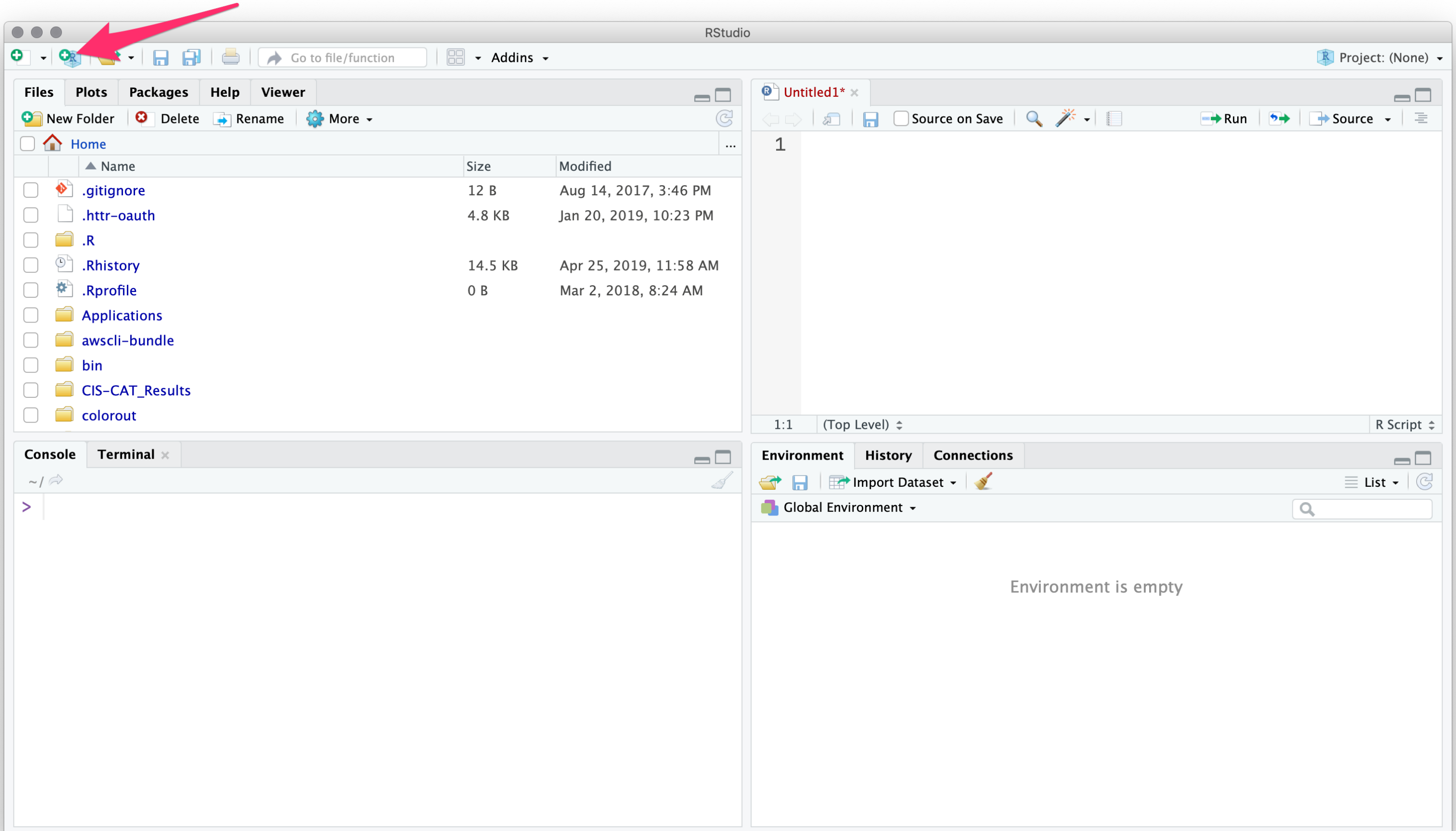
Projects

Projects in `R` offer several benefits

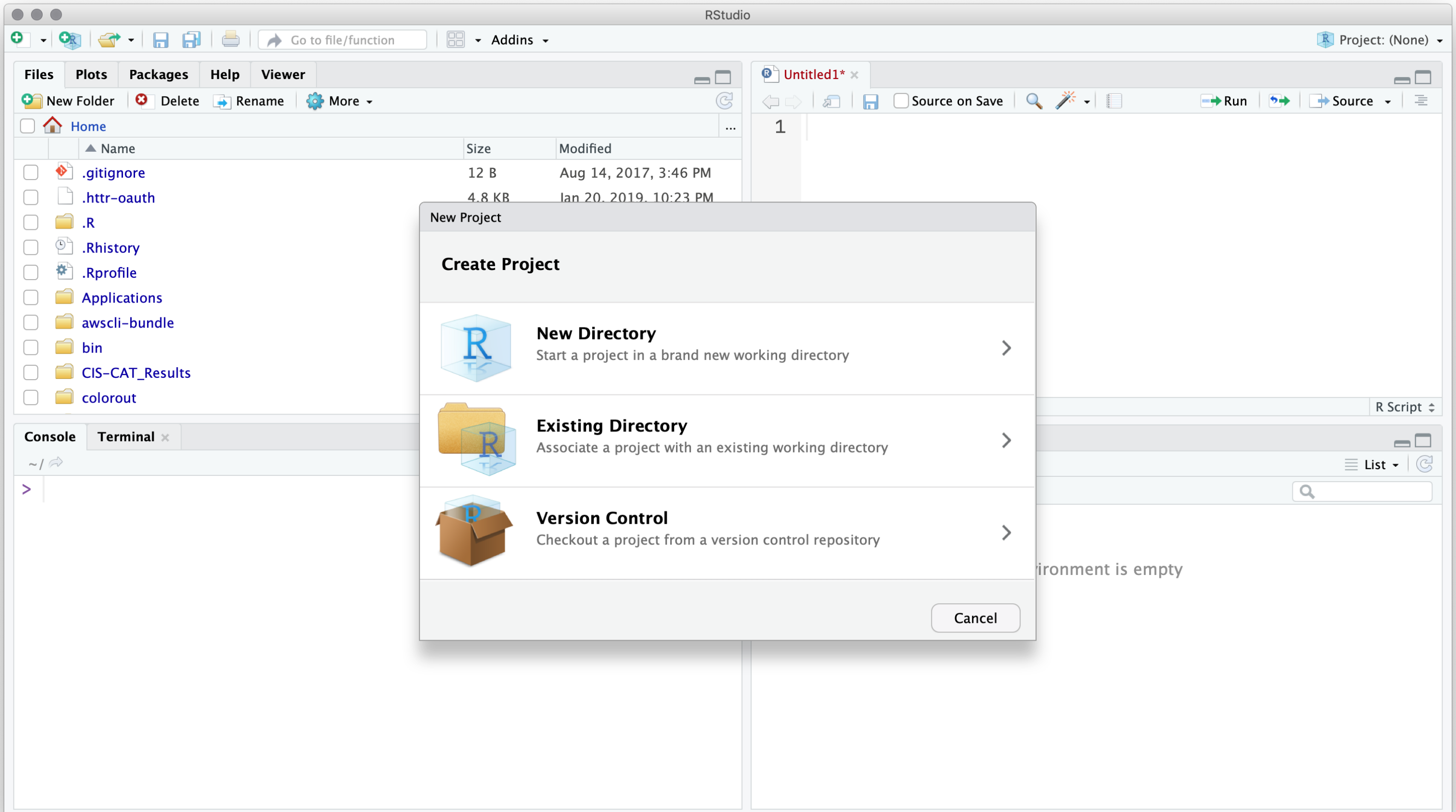
1. Act as an **anchor** for working with files.
2. Make your work (projects) easily **reproducible**.†
3. Help you **quickly jump back** into your work.

† In this class, we're assuming reproducibility is good/desirable.

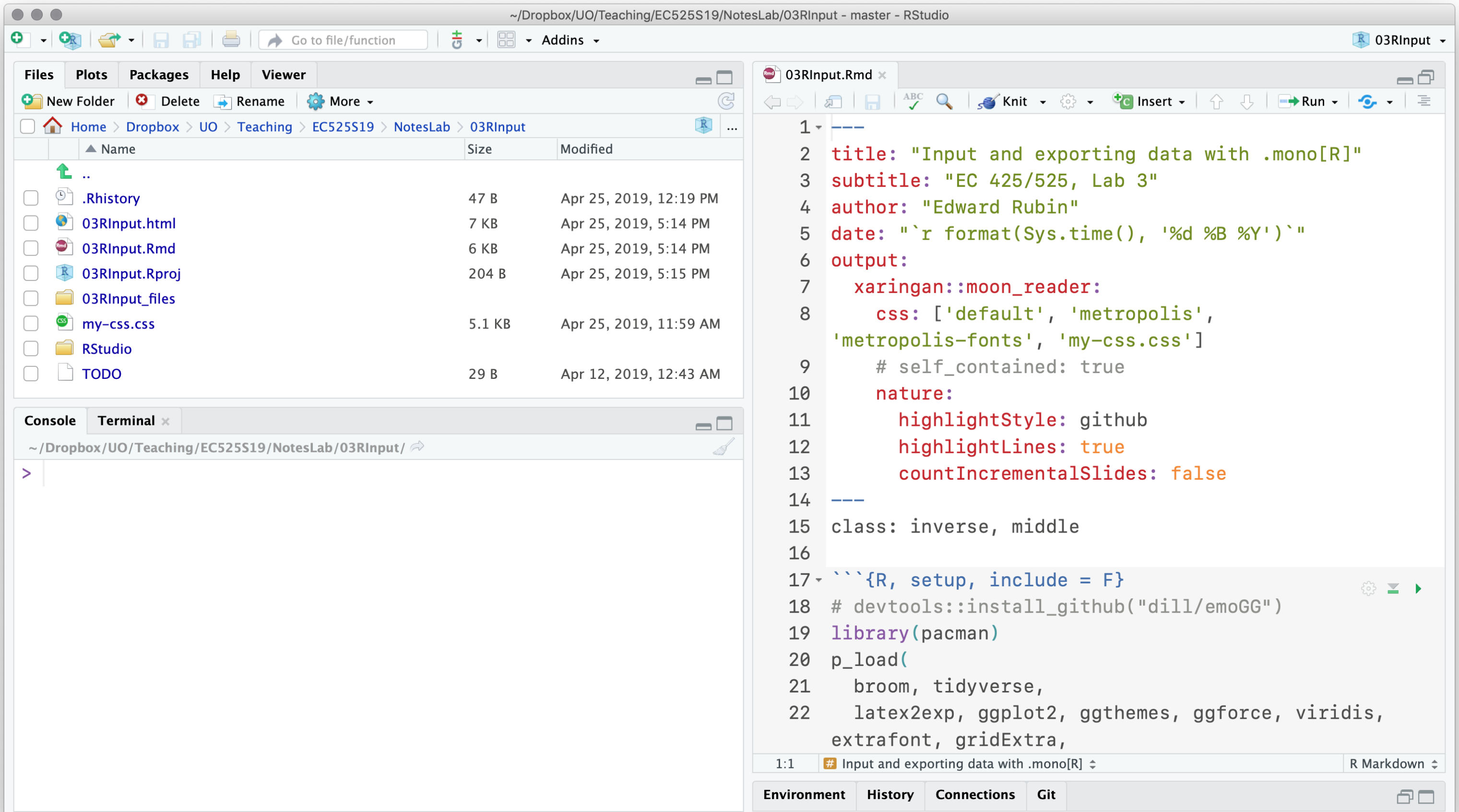
To start a new project, hit the **project icon**.



You'll then choose the folder/directory where your project lives.



`RStudio` will 'load' your previous setup (pane setup, scripts, *etc.*).



Projects

Without a project, you will need to define long file paths that you'll need to keep updating as folder names/locations change.

```
dir_class ← "/Users/edwardarubin/Dropbox/UO/Teaching/EC525S19/"
dir_labs ← paste0(dir_class, "NotesLab/")
dir_lab03 ← paste0(dir_labs, "03RInput/")
sample_df ← read.csv(paste0(dir_lab03, "sample.csv"))
```

With a project, R automatically references the project's folder.

```
sample_df ← read.csv("sample.csv")
```

Double-plus bonus The `here` package extends projects' reproducibility.

Pipes and dplyr

Pipes and dplyr

Introduction

1. Pipes (`%>%`) make your life easier.[†]

2. `dplyr` is your data-work friend.

† Check out `magrittr` for more pipe options, *e.g.*, `%<>%`.

Pipes and dplyr

What is a pipe?

Pipes are a simplifying programming tool; make your code easier to read

Take the **output** of a function as the **input/argument** of another function

In `dplyr`, the expression for a pipe is `%>%`

`R`'s pipe specifically plugs the returned object to the left of the pipe into the first argument of the function on the right fo the pipe, *e.g.*,

```
rnorm(10) %>% mean()
```

```
#> [1] -0.5162503
```

† `|>` native pipe as of R 4.1.0

Pipes and dplyr

Pipes

Pipes avoid nested functions, prevent excessive writing to your disc, and increase the readability of our `R` scripts

Example Three ways to draw 100 N(0,1) observations and calculate the interquartile range (IQR: difference between the 75th and 25th percentiles).

```
# Save each intermediate step
draw ← rnorm(100)
end_points ← quantile(draw, probs = c(0.25, 0.75))
diff(end_points)
# Lots of nesting
diff(quantile(rnorm(100), probs = c(0.25, 0.75)))
# Piping 💪
rnorm(100) %>% quantile(probs = c(0.25, 0.75)) %>% diff()
```

Think russian dolls

Pipes and dplyr

Pipes

By default, `R` pipes the output from the LHS of the pipe into the **first** argument of the function on the RHS of the pipe.

E.g., `a %>% fun(3)` is equivalent to `fun(arg1 = a, arg2 = 3)`.

If you want to pipe output into a different argument, you use a period (`.`).

- `b %>% fun(arg1 = 3, .)` is equivalent to `fun(arg1 = 3, arg2 = b)`.
- `b %>% fun(3, .)` is also equivalent to `fun(arg1 = 3, arg2 = b)`.
- `b %>% fun(., .)` is equivalent to `fun(arg1 = b, arg2 = b)`.

The `magrittr` package contains even more piping power.[†]

† `magrittr` = Magritte (of *this is not a pipe* fame) plus `R`.

dplyr

Before we begin:

1. Ensure `tidyverse` is installed: `install.packages('tidyverse')`
2. Install `nycflights13` package: `install.packages('nycflights13')`
3. Load package libraries: `library(tidyverse, nycflights13)`
4. Test the `flights` dataset: `(flights)`

year	month	day	dep_time	sched_dep_time	dep_delay	arr_time
2013	1	1	517	515	2	830
2013	1	1	533	529	4	850
2013	1	1	542	540	2	923

dplyr

dplyr

Introduction

It's a package. `dplyr` is not installed by default, so you'll need to install it.†

`dplyr` is part of the `tidyverse` (Hadleyverse), and it follows a grammar-based approach to programming/data work.

- `data` compose the subjects of your stories
- `dplyr` provides the *verbs* (action words) :
  `filter()`, `mutate()`, `select()`, `group_by()`, `summarize()`, `arrange()`

Bonus `dplyr` is pretty fast and able to interact with SQL databases.

† or just `p_load(dplyr)` after loading `pacman`.

dplyr

Manipulating variables: `mutate()`

`dplyr` streamlines adding/manipulating variables in your data frame.

Function `mutate(.data, ... )`

- Required argument `.data`, an existing data frame
- Additional arguments Names and values of the new variables
- Output An updated data frame

Example

```
mutate(.data = our_df, new1 = 7, new2 = x * y)
```

dplyr

`mutate()`

Example Take the data frame

```
my_df <- data.frame(x = 1:3, y = 5:7)
```

`mutate()` allows us to create many new variables with one call.

```
mutate(.data = my_df,
  xy = x * y,
  x2 = x^2,
  xy2 = xy^2,
  is_max = x == max(x)
)
```

x	y	xy	x2	xy2	is_max
1	5	5	1	25	false
2	6	12	4	144	false
3	7	21	9	441	true

Notice `mutate()` returns the original *and* new columns.

`mutate()` *vs.* `transmute()`

As their names imply, `mutate()` and `transmute()` are very similar functions.

- `mutate()` returns the original *and* new columns (variables).
- `transmute()` returns only the new columns (variables).

Note Both functions return a new object as *output*—they do not update the object in `R`'s memory. (This is the case for all functions in `dplyr`.)

`%>%` and `dplyr`

Each `dplyr` function begins with a `.data` argument so that you can easily pipe in data frames (recall: `mutate(.data, ... )`).

The common workflow in `dplyr` will look something like

```
new_df <- old_df %>% mutate(cool stuff here)
```

which takes `old_df`, does some cool stuff with `mutate()`, and then saves the output of `mutate()` as `new_df`.

dplyr

`filter()`

The `filter()` function does what its name implies: it **filters the rows** of your data frame **based upon logical conditions**.

Example

```r
# Create a dataset
some_df <- data.frame(
  x = 1:10,
  y = 11:20
)
```

```r
# Only keep rows where x is 3
some_df %>% filter(x == 3)
```

x	y
3	13

dplyr

`filter()`

The `filter()` function does what its name implies: it **filters the rows** of your data frame **based upon logical conditions**.

Example

```r
# Create a dataset
some_df <- data.frame(
  x = 1:10,
  y = 11:20
)
```

```r
# Only keep rows where x > 7
some_df %>% filter(x > 7)
```

x	y
8	18
9	19
10	20

dplyr

`filter()`

The `filter()` function does what its name implies: it **filters the rows** of your data frame **based upon logical conditions**.

Example

```
# Create a dataset
some_df <- data.frame(
  x = 1:10,
  y = 11:20
)
```

```
# Keep rows where y/x > 3
some_df %>% filter(y/x > 3)
```

x	y
1	11
2	12
3	13
4	14

dplyr

`filter()`

The `filter()` function does what its name implies: it **filters the rows** of your data frame **based upon logical conditions**.

Example

```
# Create a dataset
some_df <- data.frame(
  x = 1:10,
  y = 11:20
)
```

```
# Keep rows where x>8 OR y<12
some_df %>%
  filter(x > 8 | y < 12)
```

x	y
1	11
9	19
10	20

dplyr

`filter()`

The `filter()` function does what its name implies: it **filters the rows** of your data frame **based upon logical conditions**.

Example

```r
# Create a dataset
some_df <- data.frame(
  x = 1:10,
  y = 11:20
)
```

```r
# Keep rows where 16 ≤ y ≤ 18
some_df %>%
  filter(between(y, 16, 18))
```

x	y
6	16
7	17
8	18

`filter()`

The `filter()` function does what its name implies: it **filters the rows** of your data frame **based upon logical conditions**.

Example

```r
# Create a dataset
some_df <- data.frame(
  x = 1:10,
  y = 11:20
)
```

```r
# Keep rows where y > 20
some_df %>% filter(y > 20)
```

x	y
No data available in table	

If you filter your data frame down to nothing, `R` returns a 0-row data frame with the names/number of columns from the original data frame.

dplyr

`select()`

Just as `filter()` grabs row-based subsets of your data frame, `select()` grabs column-based subsets.

You can select columns using their **names**

```
our_df %>% select(var10, var100)
```

you can select columns using their **numbers**

```
our_df %>% select(10, 100)
```

or you can select columns using **helper fuctions**

```
our_df %>% select(starts_with("var10"))
```

`select()` helps you narrow down a dataset to its necessary features.

dplyr

`summarize()`

Hopefully you're starting to see that functions' names in `dplyr` tell you what the function does.

`summarize()`† summarizes variables—you choose the variables and the summaries (*e.g.*, `mean()` or `min()`).

```
the_df %>% summarize(
  mean(x), mean(y), mean(z),
  min(x), max(x),
)
```

would return a 1×5 data frame with the means of `x`, `y`, and `z`; the minimum of `x`; and the maximum of `x`.

† or `summarise()` if you ❤️ 🇬🇧

dplyr

`summarize()` and `group_by()`

While sample-wide summarizes are certainly interesting, `dplyr` has one last gem for us: `group_by()`.

`group_by()` groups your observations by the variable(s) that you name.

Specifically, `group_by()` returns a *grouped data frame* that you can then feed to `summarize()`, `mutate()`, or `transmuate` to perform grouped calculations, *e.g.*, each group's mean.

Example: Grouped summaries

```r
# Create a new data frame
our_df <- data.frame(
  x = 1:6,
  y = c(0, 1),
  grp = rep(c("A", "B"), each = 3)
)
```

```r
# For dataset 'our_df'...
our_df %>%
  # Group by 'grp'
  group_by(grp) %>%
  # Take means of 'x' and 'y'
  summarize(mean(x), mean(y))
```

x	y	grp
1	0	A
2	1	A
3	0	A
4	1	B
5	0	B
6	1	B

grp	mean(x)	mean(y)
A	2.000	0.333
B	5.000	0.667

dplyr

Example: Grouped mutation

```r
# Create a new data frame
our_df ← data.frame(
  x = 1:6,
  y = c(0, 1),
  grp = rep(c("A", "B"), each = 3)
)
```

```r
# Add grp means for x and y
our_df %>%
  group_by(grp) %>%
  mutate(
    x_m = mean(x), y_m = mean(y)
  )
```

x	y	grp
1	0	A
2	1	A
3	0	A
4	1	B
5	0	B
6	1	B

x	y	grp	x_m	y_m
1	0	A	2.000	0.333
2	1	A	2.000	0.333
3	0	A	2.000	0.333
4	1	B	5.000	0.667
5	0	B	5.000	0.667
6	1	B	5.000	0.667

`arrange()`

`arrange()` will sorts the rows of a data frame using the inputted columns.

`R` defaults to starting with the "lowest" (smallest) at the top of the data frame. Use a `-` in front of the variable's name to reverse sort.

```r
# As is
our_df
```

x	y	grp
1	0	A
2	1	A
3	0	A
4	1	B
5	0	B
6	1	B

```r
# Arrang by y, grp, then -x
our_df %>% arrange(y, grp, -x)
```

x	y	grp
3	0	A
1	0	A
5	0	B
2	1	A
6	1	B
4	1	B

The tidyverse

There's more! `dplyr` and `tidyr` offer even more...[†]

- *Viewing data* `glimpse()`, `top_n()`
- *Sampling* `sample_n()`, `sample_frac()`
- *Summaries* `first()`, `last()`, `nth()`, `n_distinct()`
- *Duplicates* `distinct()`
- *Missingness* `na_if()`, `replace_na()`, `drop_na()`, `fill()`

The folks at RStudio have put together some great cheatsheets, *e.g.*,

- `dplyr`
- data import
- data wrangling

† And these are only two of the packages in the `tidyverse`.

Exercises

Some selected exercises from R for Data Science by Hadley Wickham

1. Exercise 5.2.4.1
2. Exercise 5.3.1.2, 5.3.1.3, 5.3.1.4
3. Exercise 5.5.2.4
4. Exercise 5.7.1.3

Table of contents