Reports and manuscripts with Quarto

What is `quarto`?

- *Open-source document format and computational notebook system*
- Integrates text, code, and outout
- Can create multiple different types of products (documents, slides, websites, books)

Why not R Markdown?

Only because quarto is newer and more featured!

- Anything you already know how to do in R Markdown you can do in quarto, and more![1]
- All of these slides, website, etc. are all made in quarto.
- If you know and love R Markdown, by all means keep using it!

1. Slight caveat...

Quarto workflow

1. Create a Quarto document
2. Write code
3. Write text
4. Repeat 2-3 in whatever order you want
5. Render

How does it work?

- You text in markdown and code in R
- `knitr` processes the code chunks, executes the R code, and inserts the code outputs (e.g., plots, tables) back into the markdown document
- `pandoc` transforms the markdown document into various output formats

Text and code...

```
# My header

Some text

Some *italic text*

Some **bold text**

- Eggs
- Milk

```{r}
x <- 3
x
```
```

... becomes ...

My title

Some text

Some *italic text*

Some **bold text**

- Eggs
- Milk

```
1  x <- 3
2  x
```

```
[1] 3
```

If you prefer, you can use the visual editor

My title

Some text

Some *italic text*

Some **bold text**

```{r}
x <- 3
x
```

R chunks

Everything within the chunks has to be valid R[1]

````
```{r}
x <- 3
```
````

````
```{r}
x + 4
```
````

```
[1] 7
```

Chunks run in order, continuously, like a single script

1. You can also use other languages, like Python!

YAML

At the top of your Quarto document, a header written in *yaml* describes options for the document

```yaml
---
title: "My document"
author: Louisa Smith
format: html
---
```

There are a *ton* of possible options, but importantly, this determines the document output

Output

https://quarto.org/docs/output-formats/all-formats.html

Exercises

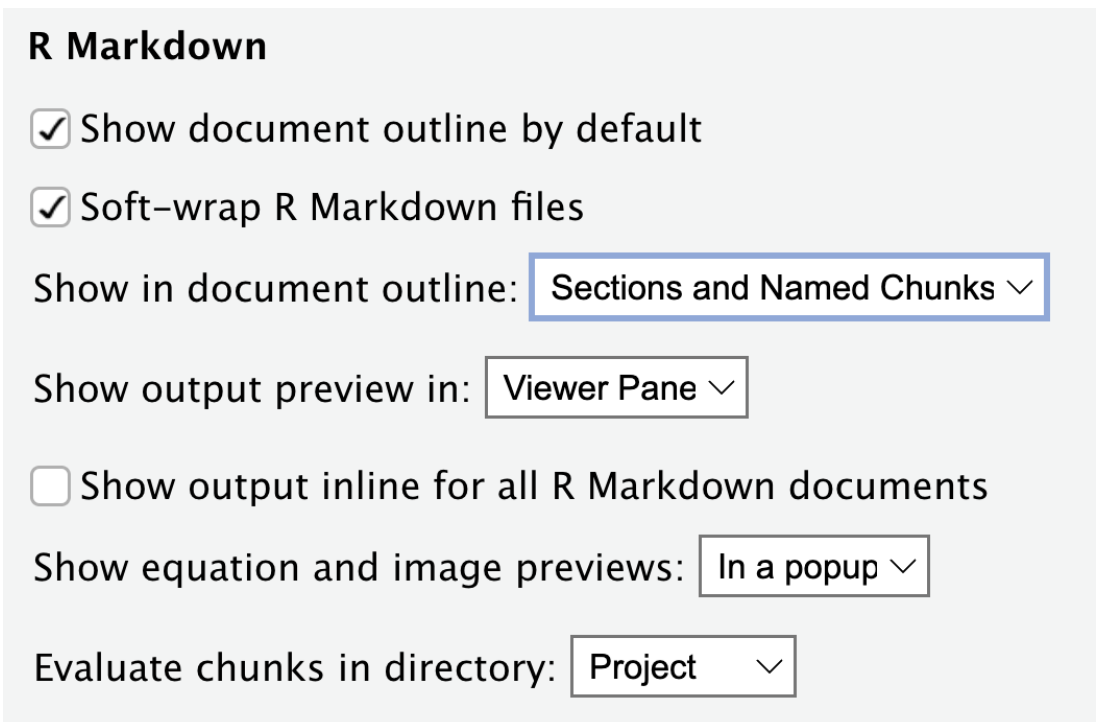
We're going to focus on html output

- It's easy to transition to Word (`format: docx`) but it's not as good for constant re-rendering
- You need a LaTeX installation for pdf
  - I recommend `{tinytex}`

Exercises

You can choose whether you want to have chunk output show up within the document (vs. just the console) when you are running Quarto (and RMarkdown) documents interactively

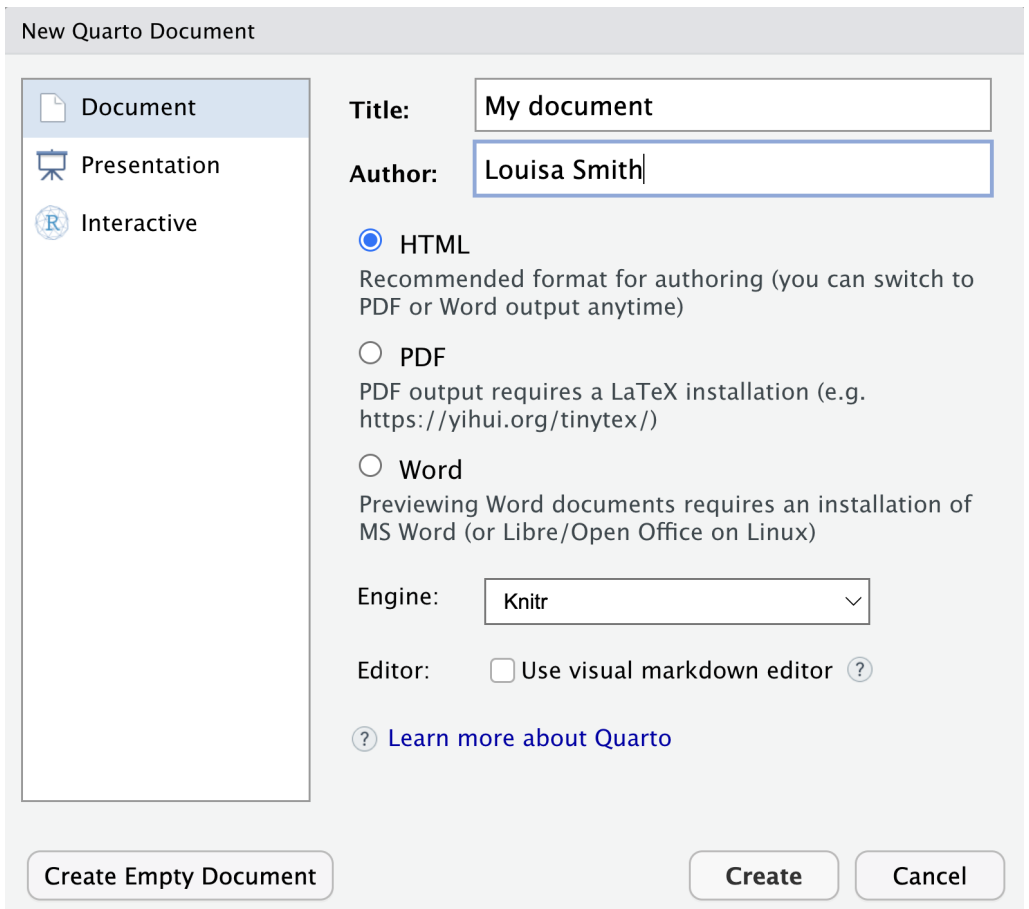
Tools > Global options



Exercises

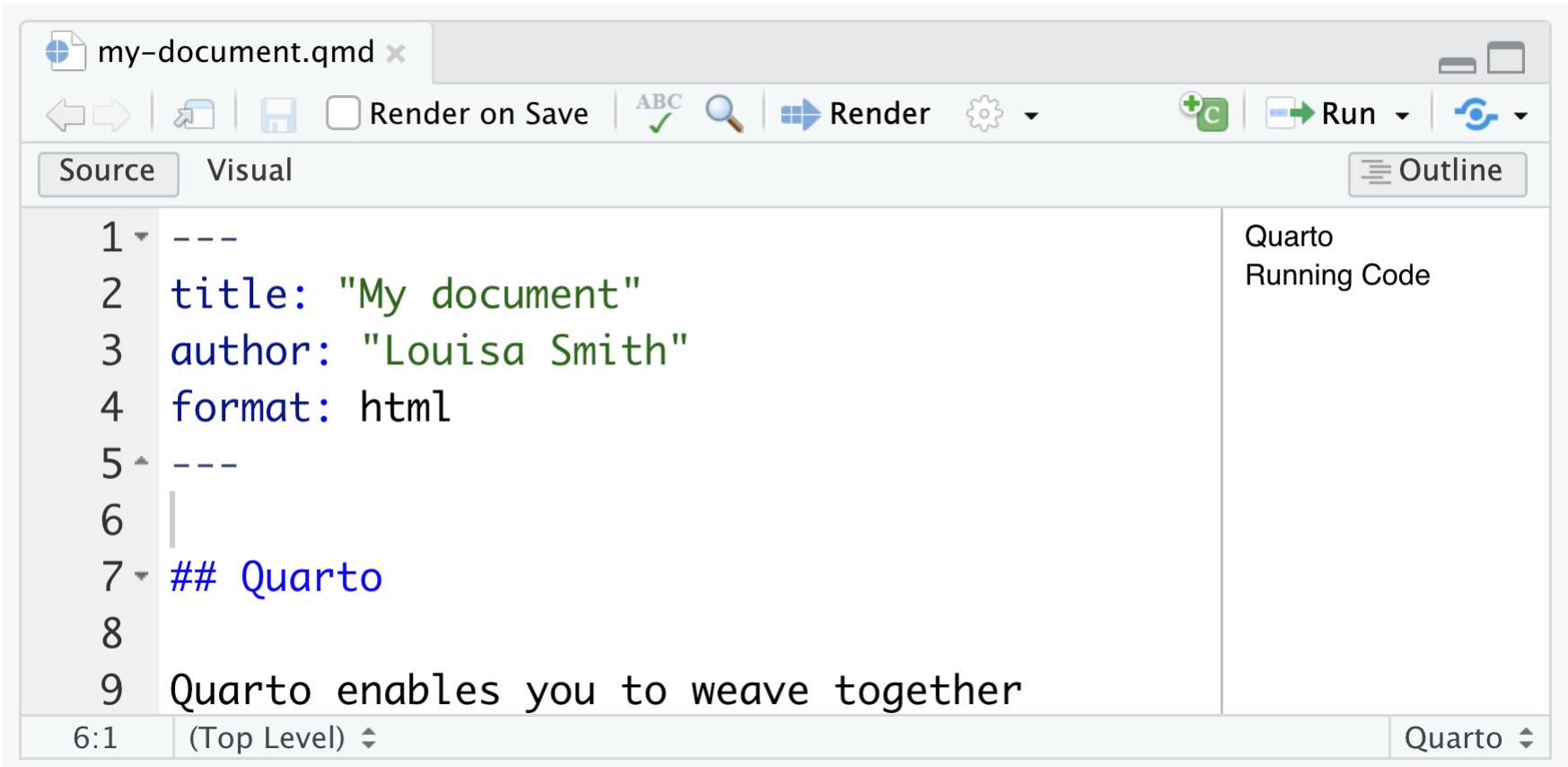
Open up your epi590r-2023-in-class R project!

File > New File > Quarto Document



Exercises

- Try toggling between Source and Visual views
- Toggle on and off the Outline
- Click Render and look at the output



Quarto options

Chunk options

In your Quarto document, you had a chunk:

````
```{r}
#| echo: false
2 * 2
```
````

`#| echo: false` tells `knitr` not to show the code within that chunk

In RMarkdown, you would have written this `{r, echo = FALSE}`. You can still do that with Quarto, but it's generally easier to read, particularly for long options (like

Chunk options

Some of the ones I find myself using most often:

- `#| eval: false`: Don't evaluate this chunk! Really helpful if you're trying to isolate an error, or have a chunk that takes a long time
- `#| error: true`: Render this *even if* the chunk causes an error
- `#| cache: true`: Store the results of this chunk so that it doesn't need to re-run every time, as long as there are no changes
- `#| warning: false`: Don't print warnings
- `#| message: false`: Don't print messages

Document options

You can tell the *entire* document not to evaluate or print code (so just include the text!) at the top:

```yaml
---
title: "My document"
author: Louisa Smith
format: html
execute:
  eval: false
  echo: false
---
```

Careful! YAML is *really* picky about spacing.

Document options

There are lots of different options for the document

- For example, you can choose a theme:

```yaml
---
format:
  html:
    theme: yeti
---
```

- Remember the pickiness: when you have a format option, `html:` moves to a new line and the options are indented 2 spaces

Exercises

Download the quarto document with some `{gtsummary}` tables from yesterday

- There's an error in the code! Try to render it. Play around with `eval:` and `error:` chunk and document options to help you a) find the error and b) render the document despite the error. Then fix the error.
- I don't like the output from the first chunk, where the passages are loaded. Make it so that we don't see this chunk's code or output.
- Play around with themes!

Quarto tables, figures, and stats

Chunks can produce figures and tables

```
```{r}
#| label: tbl-one
#| tbl-cap: "This is a great table"
knitr::kable(mtcars)
```
```

Table 1: This is a great table

	mpg	cyl	disp	hp	drat	wt	qse
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.0
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.6

	mpg	cyl	disp	hp	drat	wt	qse
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.4
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.0
Valiant	18.1	6	225.0	105	2.76	3.460	20.2
Duster 360	14.3	8	360.0	245	3.21	3.570	15.8
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.0
Merc 230	22.8	4	140.8	95	3.92	3.150	22.9
Merc 280	19.2	6	167.6	123	3.92	3.440	18.3
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.9
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.4

	mpg	cyl	disp	hp	drat	wt	qse
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.6
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.0
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.9
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.8
Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.4
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.4
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.5

	mpg	cyl	disp	hp	drat	wt	qse
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.9
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.0
Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.8
AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.3
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.4
Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.0

	mpg	cyl	disp	hp	drat	wt	qse
Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.9
Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.7
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.9
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.5
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.5
Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.6
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.6

Chunks can produce figures or tables

```
1 ```{r}
2 #| label: fig-hist
3 #| fig-cap: "This is a histogram"
4 hist(rnorm(100))
5 ```
```

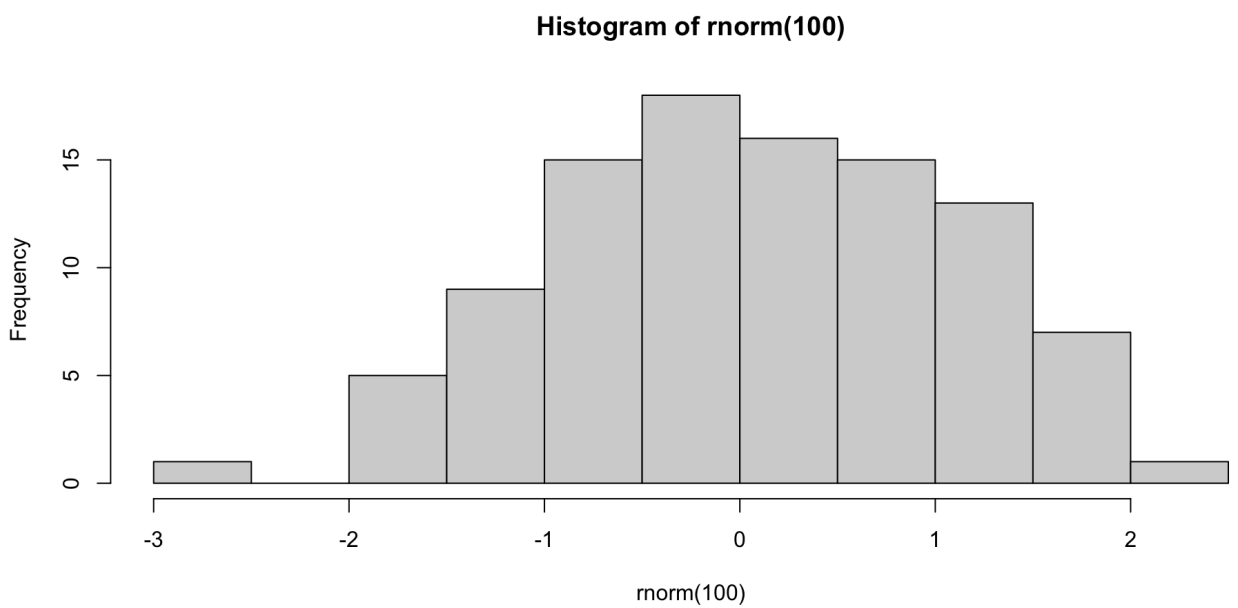


Figure 1: This is a histogram

Cross-referencing

You can then refer to those with `@tbl-one` and `@fig-hist` and the Table and Figure ordering will be correct (and linked)

```
@fig-hist contains a histogram and @tbl-one a table.
```

gets printed as:

Figure 1 contains a histogram and Table 1 a table.

There are currently some bugs with cross-referencing in Word docs which will be

Inline R

Along with just regular text, you can also run R code *within* the text:

```
There were `r 3 + 4` participants
```

becomes:

There were 7 participants

Inline stats

I often create a list of stats that I want to report in a manuscript:

```
stats <- list(n = nrow(data),
              mean_age = mean(data$age))
```

I can then print these numbers in the text with:

There were `` `r stats$n` `` participants with a mean age of `` `r stats$mean_age` ``.

which turns into:

There were 1123 participants with a mean age of 43.5.

Inline stats from {gtsummary}

We saw very, very briefly yesterday:

```
1  inline_text(income_table, variable = "age_bir")
```

```
[1] "595 (95% CI 538, 652; p<0.001)"
```

We pulled a statistic from our univariate table

If we're making a table, we probably want to report numbers from it

````
```{r}
#| label: tbl-descr
#| tbl-cap: "Descriptive statistics"
#| output-location: slide
table1 <- tbl_summary(
  nlsy,
  by = sex_cat,
  include = c(sex_cat, race_eth_cat, region_cat,
              eyesight_cat, glasses, age_bir)) |>
  add_overall(last = TRUE)
table1
```
````

If we're making a table, we probably want to report numbers from it

Table 2:
Descriptive statistics

Characteristic	**Male**, N = 6,403[1]	**Female**, N = 6,283[1]	**Overall**, N = 12,686[1]
race_eth_cat			
Hispanic	1,000 (16%)	1,002 (16%)	2,002 (16%)
Black	1,613 (25%)	1,561 (25%)	3,174 (25%)
Non-Black, Non-Hispanic	3,790 (59%)	3,720 (59%)	7,510 (59%)
region_cat			
Northeast	1,296 (21%)	1,254 (20%)	2,550 (20%)
North Central	1,488 (24%)	1,446 (23%)	2,934 (24%)
South	2,251 (36%)	2,317 (38%)	4,568 (37%)
West	1,253 (20%)	1,142 (19%)	2,395 (19%)
Unknown	115	124	239
eyesight_cat			
Excellent	1,582 (38%)	1,334 (31%)	2,916 (35%)
Very good	1,470 (35%)	1,500 (35%)	2,970 (35%)

[1] n (%); Median (IQR)

Characteristic	**Male**, N = 6,403[1]	**Female**, N = 6,283[1]	**Overall**, N = 12,686[1]
Good	792 (19%)	1,002 (23%)	1,794 (21%)
Fair	267 (6.4%)	365 (8.5%)	632 (7.5%)
Poor	47 (1.1%)	85 (2.0%)	132 (1.6%)
Unknown	2,245	1,997	4,242
glasses	1,566 (38%)	2,328 (54%)	3,894 (46%)
Unknown	2,241	1,995	4,236
age_bir	25 (21, 29)	22 (19, 27)	23 (20, 28)
Unknown	3,652	3,091	6,743

[1] n (%); Median (IQR)

I want to report some stats!

How about the median (IQR) age of the male participants at the birth of their first child?

```
1  inline_text(table1, variable = "age_bir", column = "Male")
```

```
[1] "25 (21, 29)"
```

Or the frequency and percentage of women from the South?

```
1  inline_text(table1, variable = "region_cat", level = "South", colu
```

```
[1] "2,317 (38%)"
```

And the overall stats on people who wear glasses?

```
1  inline_text(table1, variable = "glasses", column = "stat_0",
2              pattern = "{n}/{N} ({p}%)")
```

Better yet...

We can integrate these into the text of our manuscript:

```
A greater proportion of female (`r inline_text(table1,
variable = "glasses", column = "Female")`) than male

(`r inline_text(table1, variable = "glasses", column =
"Male")`) participants wore glasses.
```

Which becomes:

A greater proportion of female (2,328 (54%)) than male (1,566 (38%)) participants wore glasses.

Readability

Because this can be hard to read, I'd suggest storing those stats in a chunk before the text:

````
```{r}
glasses_f <- inline_text(table1, variable = "glasses",
                         column = "Female")
glasses_m <- inline_text(table1, variable = "glasses",
                         column = "Male")
```
A greater proportion of female (`r glasses_f`) than male (`r glass
````

Exercises

Return to the quarto document with the tables.

- Choose a table to label and caption, and then write a sentence that *cross-references* it (e.g., Table 1 shows the descriptive statistics)
- From that table, choose at least two statistics to pull out of the table and include in the text using `inline_text()`.
- Add another statistic to the text that you calculate yourself using the `nlsy` data, e.g., the mean number of hours of sleep on weekends.

Package management for R

What is `{renv}`?

`{renv}` is an R package for managing project dependencies and creating reproducible environments

Benefits of using {renv}

1. **Isolation:** Creates project-specific environments separate from the global R library.
2. **Reproducibility:** Ensures consistent package versions for code reproducibility.
3. **Collaboration:** Facilitates sharing and collaborating on projects with others.

Getting Started with {renv}

1. Install {renv}

```
install.packages("renv")
```

2. Initialize a project

```
renv::init()
```

3. Install packages

```
install.packages("other_package")
# only an option when using renv!
install.packages("github_user/github_package")
```

4. Track dependencies via a lockfile

```
renv::snapshot()
```

Behind the scenes

- Your project `.Rprofile` is updated to include:

```
source("renv/activate.R")
```

- This is run every time R starts, and does some management of the library paths to make sure when you call `install.packges("package")` or `library(package)` it does to the right place (`renv/library/R-{version}/{computer-specifics}`)
- A `renv.lock` file (really just a text file) is created to store the names and versions of the packages.

renv.lock


```
{
  "R": {
    "Version": "4.3.0",
    "Repositories": [
      {
        "Name": "CRAN",
        "URL": "https://cran.rstudio.com"
      }
    ]
  },
  "Packages": {
    "R6": {
      "Package": "R6",
      "Version": "2.5.1",
      "Source": "Repository",
      "Repository": "CRAN",
      "Requirements": [
        "R"
      ],
      "Hash": "470851b6d5d0ac559e9d01bb352b4021"
    },
    base64enc": {
      "Package": "base64enc",
      "Version": "0.1-3",
      "Source": "Repository",
      "Repository": "CRAN",
      "Requirements": [
```

```
      "R"
    ],
    "Hash": "543776ae6848fde2f48ff3816d0628bc"
  },
```

Using `{renv}` later

Restore an environment

```
renv::restore()
```

Install new packages

```
install.packages("other_package")
```

Update the lockfile

```
renv::snapshot()
```

Collaboration with `{renv}`

- Share the project's `renv.lock` file with collaborators to ensure consistent environments
- When they run `renv::restore()`, the correct versions of the packages will be installed on their computer

```
renv::restore()
```

Other helpful functions

Remove packages that are no longer used:

```
renv::clean()
```

Check the status of the project library with respect to the lockfile:

```
renv::status()
```

This will tell you to `renv::snapshot()` to add packages you've installed but haven't snapshotted, or `renv::restore()` if you're missing packages you need but which aren't installed

Conclusion

`{renv}` benefits:

- Isolation, reproducibility, and collaboration

Getting started with `{renv}`

1. Initialize a project using `renv::init()`
2. Install packages and store with `renv::snapshot()`
3. Restore later or elsewhere with `renv::restore()`

Exercises

3. Install a new R package of your choice. (Not sure what to choose? Try one of these fun packages. For example, I did `install.packages("hadley/emo")`.)
4. Create an R script and save it in your R project. Include some code that requires the package. For example:

```
1  emo::ji("banana")
```

4. Run `renv::status()` to make sure your project picked up the package as a dependency.
5. Run `renv::snapshot()` to record that package in your lockfile.
6. Open your lockfile and look for your new package. For example, mine now has:

```
"emo": {
      "Package": "emo",
      "Version": "0.0.0.9000",
      "Source": "git",
      "RemoteType": "git",
      "RemoteUrl": "https://github.com/hadley/emo",
      "RemoteHost": "api.github.com",
      "RemoteUsername": "hadley",
      "RemoteRepo": "emo",
      "RemoteRef": "master",
      "RemoteSha": "3f03b11491ce3d6fc5601e210927eff73bf8e350",
      "Requirements": [
        "R",
        "assertthat",
        "crayon",
        "glue",
        "lubridate",
        "magrittr",
```

Functions

Functions in R

I've been denoting functions with parentheses: `func()`

We've seen functions such as:

- `mean()`
- `tbl_summary()`
- `init()`
- `create_github_token`

Functions take **arguments** and return **values**

Looking inside a function

If you want to see the code within a function, you can just type its name without the parentheses:

```
1  usethis::create_github_token
```

```
function (scopes = c("repo", "user", "gist", "workflow"), description = "DESCRIBE THE TOKEN'S USE
CASE",
    host = NULL)
{
    scopes <- glue_collapse(scopes, ",")
    host <- get_hosturl(host %||% default_api_url())
    url <- glue("{host}/settings/tokens/new?scopes={scopes}&description={description}")
    withr::defer(view_url(url))
    hint <- code_hint_with_host("gitcreds::gitcreds_set", host)
    ui_todo("\n    Call {ui_code(hint)} to register this token in the \\\n    local Git
credential store\n    It is also a great idea to store this token in any password-management \\\n
software that you use")
    invisible()
}
<bytecode: 0x10d9131a0>
<environment: namespace:usethis>
```

Structure of a function

```
1  func <- function()
```

You can name your function like you do any other object

- Just avoid names of existing functions

Structure of a function

```
1 func <- function(arg1,
2                  arg2 = defaul
3 }
```

What objects/values do you need to make your function work?

- You can give them default values to use if the user doesn't specify others

Structure of a function

```
1 func <- function(arg1,
2                  arg2 = defaul
3
4 }
```

Everything else goes within curly braces

- Code in here will essentially look like any other R code, using any inputs to your functions

Structure of a function

Structure of a function

```
1 func <- function(arg1,
2                  arg2 = defaul
3   new_val <- # do something wi
4   return(new_val)
5 }
```

Return something new that the code has produced

- The `return()` statement is actually optional. If you don't put it, it will return the last object in the code. When you're starting out, it's safer to always explicitly write out what you want to return.

Example: a new function for the mean

Let's say we are not satisfied with the `mean()` function and want to write our own.

Here's the general structure we'll start with.

```
1 new_mean <- function() {
2
3 }
```

New mean: arguments

We'll want to take the mean of a vector of numbers.

It will help to make an example of such a vector to think about what the input might look like, and to test the function. We'll call it `x`:

```
1  x <- c(1, 3, 5, 7, 9)
```

We can add `x` as an argument to our function:

```
1  new_mean <- function(x) {
2
3  }
```

New mean: function body

Let's think about how we calculate a mean in math, and then translate it into code:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

So we need to sum the elements of `x` together, and then divide by the number of elements.

We can use the functions `sum()` and `length()` to help us.

We'll write the code with our test vector first, before inserting it into the function:

```
1  n <- length(x)
2  sum(x) / n
```

```
[1] 5
```

New mean: function body

Our code seems to be doing what we want, so let's insert it. To be explicit, I've stored the answer (within the function) as `mean_val`, then returned that value.

```
1 new_mean <- function(x) {
2     n <- length(x)
3     mean_val <- sum(x) / n
4     return(mean_val)
5 }
```

Testing a function

Let's plug in the vector that we created to test it:

```
1  new_mean(x = x)
```

```
[1] 5
```

And then try another one we create on the spot:

```
1  new_mean(x = c(100, 200, 300))
```

```
[1] 200
```

Adding another argument

Let's say we plan to be using our `new_mean()` function to calculate proportions (i.e., the mean of a binary variable). Sometimes we'll want to report them as multiplier by multiplying the proportion by 100.

Let's name our new function `prop()`. We'll use the same structure as we did with `new_mean()`.

```
1 prop <- function(x) {
2    n <- length(x)
3    mean_val <- sum(x) / n
```

Testing the code

Now we'll want to test on a vector of 1's and 0's.

```
x <- c(0, 1, 1)
```

To calculate the proportion and turn it into a percentage, we'll just multiply the mean by 100.

```
multiplier <- 100
multiplier * sum(x) / length(x)
```

```
[1] 66.66667
```

Testing the code

We want to give users the option to choose between a proportion and a percentage. So we'll add an argument `multiplier`. When we want to just return the proportion, we can just set `multiplier` to be 1.

```
1 multiplier <- 1
2 multiplier * sum(x) / length(x)
```

```
[1] 0.6666667
```

```
1 multiplier <- 100
2 multiplier * sum(x) / length(x)
```

Adding another argument

If we add `multiplier` as an argument, we can refer to it in the function body.

```
1 prop <- function(x, multiplier) {
2     n <- length(x)
3     mean_val <- multiplier * sum(x) / 
4     return(mean_val)
5 }
```

Adding another argument

Now we can test:

```
1 prop(x = c(1, 0, 1, 0), multiplier =
```

```
[1] 0.5
```

```
1 prop(x = c(1, 0, 1, 0), multiplier =
```

```
[1] 50
```

Making a default argument

Since we don't want users to have to specify `multiplier = 1` every time they just want a proportion, we can set it as a **default**.

```
1 prop <- function(x, multiplier = 1) {
2   n <- length(x)
3   mean_val <- multiplier * sum(x) / n
4   return(mean_val)
5 }
```

Now we only need to specify that argument if we want a percentage.

```
1 prop(x = c(0, 1, 1, 1))
```

```
[1] 0.75
```

```
1 prop(x = c(0, 1, 1, 1), multiplier = 100)
```

```
[1] 75
```

Caveats

- This is obviously not the best way to write this function!
- For example, it will still work if `x = c(123, 593, -192)`…. but it certainly won't give you a proportion or a percentage!
- We could also put `multiplier =` *any number*, and we'll just be multiplying the answer by that number – this is essentially meaningless.
- We also haven't done any checking to see whether the user is even entering numbers! We could put in better error messages so users don't just get an R default error message if they do something wrong.

```
1 prop(x = c("blah", "blah", "blah"))
```

Exercises

Create some functions!

{targets}

What is {targets}?

"a Make-like pipeline tool for statistics and data science in R"

- manage a sequence of computational steps
- only update what needs updating
- ensure that the results at the end of the pipeline are still valid

Script-based workflow

01-data.R

```r
library(tidyverse)
data <- read_csv("data.csv", col_types = cols()) %>%
    filter(!is.na(Ozone))
write_rds(data, "data.rds")
```

02-model.R

```r
library(tidyverse)
data <- read_rds("data.rds")
model <- lm(Ozone ~ Temp, data) %>%
    coefficients()
write_rds(model, "model.rds")
```

03-plot.R

```r
library(tidyverse)
model <- read_rds("model.rds")
data <- read_rds("data.rds")
ggplot(data) +
    geom_point(aes(x = Temp, y = Ozone)) +
    geom_abline(intercept = model[1], slope = model[2])
ggsave("plot.png", plot)
```

Problems with script-based workflow

- **Reproducibility**: if you change something in one script, you have to remember to re-run the scripts that depend on it
- **Efficiency**: that means you'll usually rerun all the scripts even if they don't depend on the change
- **Scalability**: if you have a lot of scripts, it's hard to keep track of which ones depend on which
- **File management**: you have to keep track of which files are inputs and which are outputs and where they're saved

Based on example in https://books.ropensci.org/targets

{targets} workflow

R/functions.R

```r
get_data <- function(file) {
  read_csv(file, col_types = cols()) %>%
    filter(!is.na(Ozone))
}

fit_model <- function(data) {
  lm(Ozone ~ Temp, data) %>%
    coefficients()
}

plot_model <- function(model, data) {
  ggplot(data) +
    geom_point(aes(x = Temp, y = Ozone)) +
    geom_abline(intercept = model[1], slope = model[2])
```

{targets} workflow

_targets.R

```
1  library(targets)
2
3  tar_source()
4  tar_option_set(packages = c("tidyverse"))
5
6  list(
7    tar_target(file, "data.csv", format = "file"),
8    tar_target(data, get_data(file)),
9    tar_target(model, fit_model(data)),
10   tar_target(plot, plot_model(model, data))
11 )
```

Run `tar_make()` to run pipeline

`use_targets()` will generate a `_targets.R` script for you to fill in.

{targets} workflow

Targets are “hidden” away where you don’t need to manage them

```
├── _targets.R
├── data.csv
├── R/
│   ├── functions.R
├── _targets/
│   ├── objects
│          ├── data
│          ├── model
│          ├── plot
```

You can of course have multiple files in `R/`; `tar_source()` will source them all

My typical workflow with `{targets}`

1. Read in some data and do some cleaning until it's in the form I want to work with.
2. Wrap that in a function and save the file in `R/`.
3. Run `use_targets()` and edit `_targets.R` accordingly, so that I list the data file as a target and `clean_data` as the output of the cleaning function.
4. Run `tar_make()`.
5. Run `tar_load(clean_data)` so that I can work on the next step of my workflow.
6. Add the next function and corresponding target when I've solidified that step.

I usually include `library(targets)` in my project `.Rprofile` so that I can always call

`_targets.R` tips and tricks

```
list(
  tar_target(
    data_file,
    "data/raw_data.csv",
    format = "file"
  ),
  tar_target(
    raw_data,
    read.csv(data_file)
  ),
  tar_target(
    clean_data,
    clean_data_function(raw_data)
  )
```

I like to pair my functions/targets by name so that the workflow is clear to me

`_targets.R` tips and tricks

```
preparation <- list(
  ...,
  tar_target(
    clean_data,
    clean_data_function(raw_data)
  )
)
modeling <- list(
  tar_target(
    linear_model,
    linear_model_function(clean_data)
  ),
  ...
)
```

By grouping the targets into lists, I can easily comment out chunks of the pipeline

_targets.R tips and tricks

```r
## prepare ----
prepare <- list(
  ### cleanData.csv ----
  tar_target(
    cleanData.csv,
    file.path(path_to_data,
              "cleanData.csv"),
    format = "file"
  ),
  ### newdat ----
  tar_target(
    newdat,
    read_csv(cleanData.csv,
             guess_max = 20000)
```

In big projects, I comment my `_targets.R` file so that I can use the RStudio outline

Key `{targets}` functions

- `use_targets()` gets you started with a `_targets.R` script to fill in
- `tar_make()` runs the pipeline and saves the results in `_targets/objects/`
- `tar_make_future()` runs the pipeline in parallel[1]
- `tar_load()` loads the results of a target into the global environment (e.g., `tar_load(clean_data)`)
- `tar_read()` reads the results of a target into the global environment (e.g., `dat <- tar_read(clean_data)`)
- `tar_visnetwork()` creates a network diagram of the pipeline
- `tar_outdated()` checks which targets need to be updated
- `tar_prune()` deletes targets that are no longer in `_targets.R`
- `tar_destroy()` deletes the `.targets/` directory if you need to burn everything down and start again

1. Note: `{targets}` is moving to a new distributed computing strategy using `{crew}`

Advanced `{targets}`

“target factories”

repo status Active `tarchetypes` makes it easy to add certain kinds of common tasks to reproducible pipelines. Most of its functions create families of targets for parameterized R Markdown, simulation studies, and other general-purpose scenarios.

repo status Active `stantargets` is a workflow framework for Bayesian data analysis with `cmdstanr`. With concise, easy-to-use syntax, it defines versatile families of targets tailored to Bayesian statistics, from a single MCMC run with postprocessing to large simulation studies.

repo status Active Like `stantargets`, `jagstargets` is a workflow framework for Bayesian data analysis, with support for both single MCMC runs and large-scale simulation studies. It invokes JAGS through the `R2jags` package, which has nice features such as the ability to parallelize chains across local R processes.

`{tarchetypes}`: reports

Render documents that depend on targets loaded with `tar_load()` or `tar_read()`.

- `tar_render()` renders an R Markdown document
- `tar_quarto()` renders a Quarto document (or project)

It can't detect dependencies like `tar_load(ends_with("plot"))`

What does `report.qmd` look like?

````
---
title: "My report"
---
```{r}
library(targets)
tar_load(results)
tar_load(plots)
```
There were `r results$n` observations with a mean age of `r result
```{r}
library(ggplot2)
plots$age_plot
```
````

Because `report.qmd` depends on `results` and `plots`, it will only be re-rendered if either of those targets change.

{tarchetypes}: branching

Using data from the National Longitudinal Survey of Youth,

`_targets.R`

```
library(targets)
library(tarchetypes)
tar_source()

targets_setup <- list(
  tar_target(
    csv,
    "data/nlsy.csv",
    format = "file"
  ),
  tar_target(
    dat,
    readr::read_csv(csv,
      show_col_types = FALSE)
  )
)
```

`R/functions.R`

```
model_function <- function(outcome_var,
                           sex_val, dat) {

  lm(as.formula(paste(outcome_var,
      " ~ age_bir + income + factor(region)")) ,
     data = dat,
     subset = sex == sex_val)
}

coef_function <- function(model) {
  coef(model)[["age_bir"]]
}
```

we want to investigate the relationship between age at first birth and hours of sleep on weekdays and weekends among moms and dads separately

Option 1

Create (and name) a separate target for each combination of sleep variable (`"sleep_wkdy"`, `"sleep_wknd"`) and sex (male: `1`, female: `2`):

```
targets_1 <- list(
  tar_target(
    model_1,
    model_function(outcome_var = "sleep_wkdy", sex_val = 1, dat =
  ),
  tar_target(
    coef_1,
    coef_function(model_1)
  )
)
```

... and so on...

Option 2

Use `tarchetypes::tar_map()` to map over the combinations for you (static branching):

```
targets_2 <- tar_map(
  values = tidyr::crossing(
    outcome = c("sleep_wkdy", "sleep_wknd"),
    sex = 1:2
  ),
  tar_target(
    model_2,
    model_function(outcome_var = outcome, sex_val = sex, dat = dat
  ),
  tar_target(
    coef_2,
    coef_function(model_2)
  )
)
```

Option 2, cont.

Use `tarchetypes::tar_combine()` to combine the results of a call to `tar_map()`:

```
combined <- tar_combine(
  combined_coefs_2,
  targets_2[["coef_2"]],
  command = vctrs::vec_c(!!!.x),
)
tar_read(combined_coefs_2)
```

`command = vctrs::vec_c(!!!.x)` is the default, but you can supply your own function to combine the results

Option 3

Use the `pattern =` argument of `tar_target()` (dynamic branching):

```
targets_3 <- list(
  tar_target(
    outcome_target,
    c("sleep_wkdy", "sleep_wknd")
  ),
  tar_target(
    sex_target,
    1:2
  ),
  tar_target(
    model_3,
    model_function(outcome_var = outcome_target, sex_val = sex_tar
    pattern = cross(outcome_target, sex_target)
  )
```

Branching

Dynamic	Static
Pipeline creates new targets at runtime.	All targets defined in advance.
Cryptic target names.	Friendly target names.
Scales to hundreds of branches.	Does not scale as easily for tar_visnetwork() etc.
No metaprogramming required.	Familiarity with metaprogramming is helpful.

From https://books.ropensci.org/targets/dynamic.html#branching

Branching

- The book also has an example of using metaprogramming to map over different functions
  - i.e. fit multiple models with the same arguments
- Static and dynamic branching can be combined
  - e.g. `tar_map(values = ..., tar_target(..., pattern = map(...)))`
- Branching can lead to slowdowns in the pipeline (see book for suggestions)

{tarchetypes}: repetition

`tar_rep()` repeats a target multiple times with the same arguments

```
targets_4 <- list(
  tar_rep(
    bootstrap_coefs,
    dat |>
      dplyr::slice_sample(prop = 1, replace = TRUE) |>
      model_function(outcome_var = "sleep_wkdy", sex_val = 1, dat
      coef_function(),
    batches = 10,
    reps = 10
  )
)
```

The pipeline gets split into `batches` x `reps` chunks, each with its own random seed

{tarchetypes}: mapping over iterations

```
sensitivity_scenarios <- tibble::tibble(
  error = c("small", "medium", "large"),
  mean = c(1, 2, 3),
  sd = c(0.5, 0.75, 1)
)
```

`tar_map_rep()` repeats a target multiple times with different arguments

```
targets_5 <- tar_map_rep(
  sensitivity_analysis,
  dat |>
    dplyr::mutate(sleep_wkdy = sleep_wkdy + rnorm(nrow(dat), mean,
    model_function(outcome_var = "sleep_wkdy", sex_val = 1, dat =
    coef_function() |>
    data.frame(coef = _),
  values = sensitivity_scenarios,
```

```
 9    batches = 10,
10    reps = 10
11  )
```

{tarchetypes}: mapping over iterations

```
tar_read(sensitivity_analysis) |> head()
```

Ideal for sensitivity analyses that require multiple iterations of the same pipeline with different parameters

```
tar_read(sensitivity_analysis) |>
  dplyr::group_by(error) |>
  dplyr::summarize(q25 = quantile(coef, .25),
                   median = median(coef),
                   q75 = quantile(coef, .75))
```

Summary

- `{targets}` is a great tool for managing complex workflows
- `{tarchetypes}` makes it even more powerful
- The user manual is a great resource for learning more

Exercises

We'll clone a repo with `{targets}` already set up and add some additional steps to the analysis.

Reports and manuscripts with Quarto

What is `quarto`?

- *Open-source document format and computational notebook system*
- Integrates text, code, and outout
- Can create multiple different types of products (documents, slides, websites, books)

Why not R Markdown?

Only because quarto is newer and more featured!

- Anything you already know how to do in R Markdown you can do in quarto, and more![1]
- All of these slides, website, etc. are all made in quarto.
- If you know and love R Markdown, by all means keep using it!

1. Slight caveat...

Quarto workflow

1. Create a Quarto document
2. Write code
3. Write text
4. Repeat 2-3 in whatever order you want
5. Render

How does it work?

- You text in markdown and code in R
- `knitr` processes the code chunks, executes the R code, and inserts the code outputs (e.g., plots, tables) back into the markdown document
- `pandoc` transforms the markdown document into various output formats

Text and code...

````
# My title

Some text

Some *italic text*

Some **bold text**


```{r}
x <- 3
x
```
````

... becomes ...

My title

Some text

Some *italic text*

Some **bold text**

```
1  x <- 3
2  x
```

```
[1] 3
```

If you prefer, you can use the visual editor

My title

Some text

Some *italic text*

Some **bold text**

```{r}
x <- 3
x
```

R chunks

Everything within the chunks has to be valid R[1]

````
```{r}
x <- 3
```
````

````
```{r}
x + 4
```
````

```
[1] 7
```

Chunks run in order, continuously, like a single script

1. You can also use other languages, like Python!

YAML

At the top of your Quarto document, a header written in *yaml* describes options for the document

```yaml
---
title: "My document"
author: Louisa Smith
format: html
---
```

There are a *ton* of possible options, but importantly, this determines the document output

Output

https://quarto.org/docs/output-formats/all-formats.html

Exercises

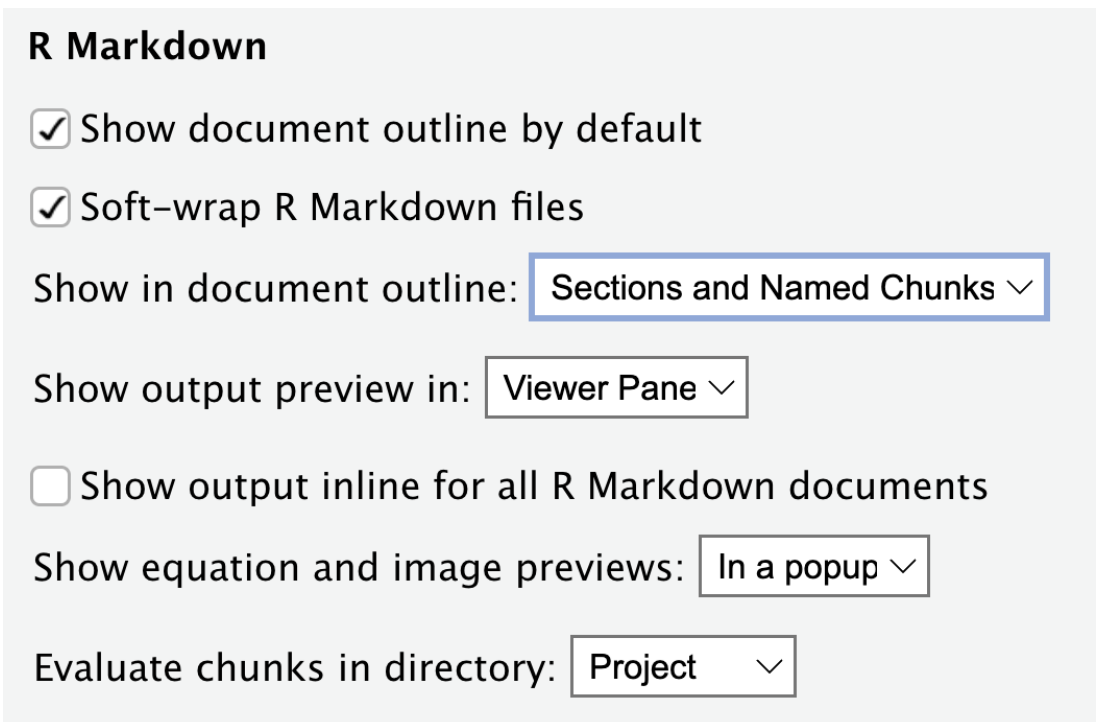
We're going to focus on html output

- It's easy to transition to Word (`format: docx`) but it's not as good for constant re-rendering
- You need a LaTeX installation for pdf
  - I recommend `{tinytex}`

Exercises

You can choose whether you want to have chunk output show up within the document (vs. just the console) when you are running Quarto (and RMarkdown) documents interactively

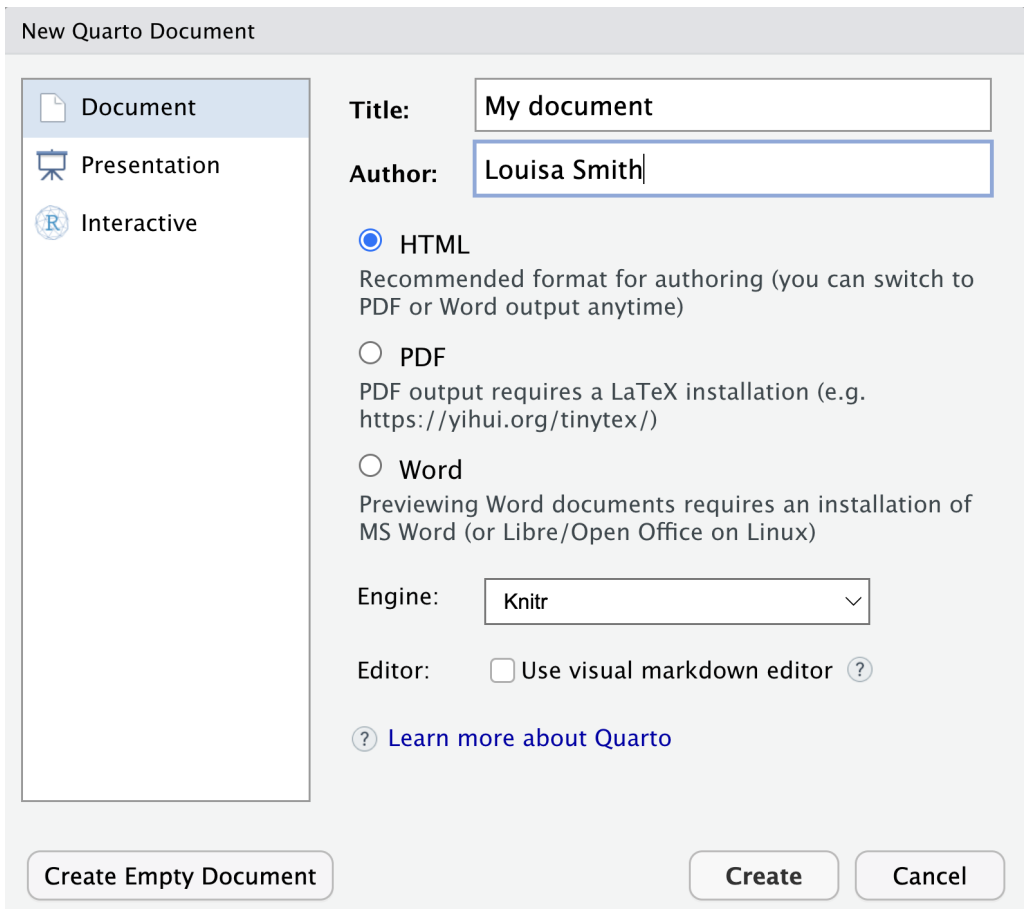
Tools > Global options



Exercises

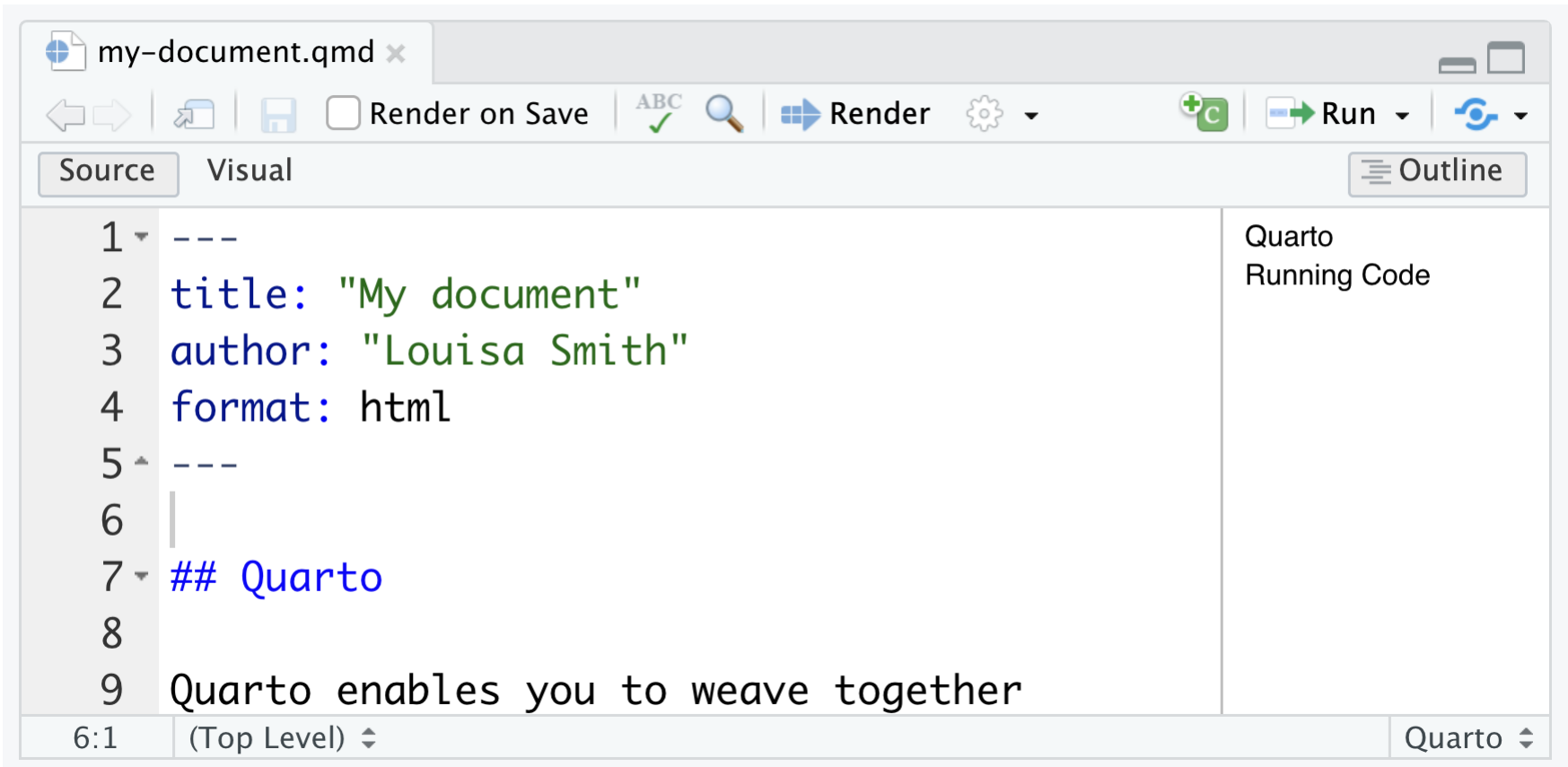
Open up your epi590r-2023-in-class R project!

File > New File > Quarto Document



Exercises

- Try toggling between Source and Visual views
- Toggle on and off the Outline
- Click Render and look at the output



Quarto options

Chunk options

In your Quarto document, you had a chunk:

````
```{r}
#| echo: false
2 * 2
```
````

`#| echo: false` tells `knitr` not to show the code within that chunk

In RMarkdown, you would have written this `{r, echo = FALSE}`. You can still do that with Quarto, but it's generally easier to read, particularly for long options (like

Chunk options

Some of the ones I find myself using most often:

- `#| eval: false`: Don't evaluate this chunk! Really helpful if you're trying to isolate an error, or have a chunk that takes a long time
- `#| error: true`: Render this *even if* the chunk causes an error
- `#| cache: true`: Store the results of this chunk so that it doesn't need to re-run every time, as long as there are no changes
- `#| warning: false`: Don't print warnings
- `#| message: false`: Don't print messages

Document options

You can tell the *entire* document not to evaluate or print code (so just include the text!) at the top:

```yaml
---
title: "My document"
author: Louisa Smith
format: html
execute:
  eval: false
  echo: false
---
```

Careful! YAML is *really* picky about spacing.

Document options

There are lots of different options for the document

- For example, you can choose a theme:

```yaml
---
format:
  html:
    theme: yeti
---
```

- Remember the pickiness: when you have a format option, `html:` moves to a new line and the options are indented 2 spaces

Exercises

Download the quarto document with some `{gtsummary}` tables from yesterday

- There's an error in the code! Try to render it. Play around with `eval:` and `error:` chunk and document options to help you a) find the error and b) render the document despite the error. Then fix the error.
- I don't like the output from the first chunk, where the passages are loaded. Make it so that we don't see this chunk's code or output.
- Play around with themes!

Quarto tables, figures, and stats

Chunks can produce figures and tables

```
1 ```{r}
2 #| label: tbl-one
3 #| tbl-cap: "This is a great table"
4 knitr::kable(mtcars)
5 ```
```

Table 1: This is a great table

	mpg	cyl	disp	hp	drat	wt	qse
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.0
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.6
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.4

	mpg	cyl	disp	hp	drat	wt	qse
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.0
Valiant	18.1	6	225.0	105	2.76	3.460	20.2
Duster 360	14.3	8	360.0	245	3.21	3.570	15.8
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.0
Merc 230	22.8	4	140.8	95	3.92	3.150	22.9
Merc 280	19.2	6	167.6	123	3.92	3.440	18.3
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.9
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.4
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.6
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.0

	mpg	cyl	disp	hp	drat	wt	qse
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82
Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90

	mpg	cyl	disp	hp	drat	wt	qse
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.0
Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.8
AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.3
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.4
Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.0
Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.9
Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.7

	mpg	cyl	disp	hp	drat	wt	qsec
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50
Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60

Chunks can produce figures or tables

```r
```{r}
#| label: fig-hist
#| fig-cap: "This is a histogram"
hist(rnorm(100))
```
```

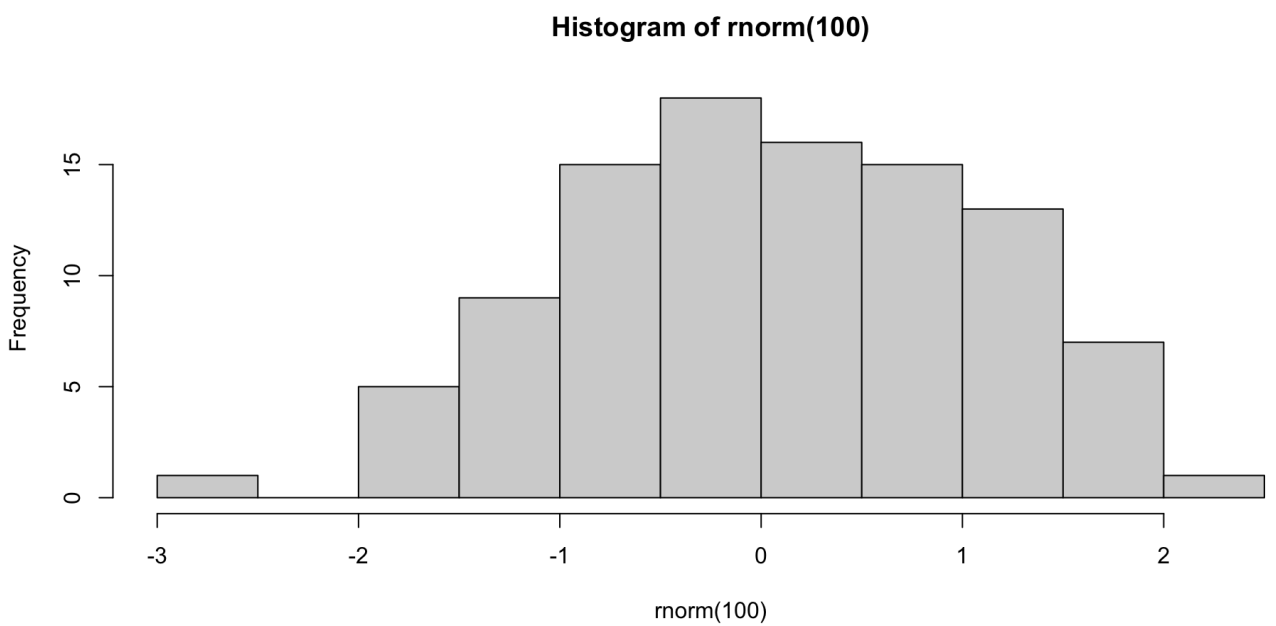


Figure 1: This is a histogram

Cross-referencing

You can then refer to those with `@tbl-one` and `@fig-hist` and the Table and Figure ordering will be correct (and linked)

```
@fig-hist contains a histogram and @tbl-one a table.
```

gets printed as:

Figure 1 contains a histogram and Table 1 a table.

There are currently some bugs with cross-referencing in Word docs which will be

Inline R

Along with just regular text, you can also run R code *within* the text:

```
There were `r 3 + 4` participants
```

becomes:

There were 7 participants

Inline stats

I often create a list of stats that I want to report in a manuscript:

```r
stats <- list(n = nrow(data),
              mean_age = mean(data$age))
```

I can then print these numbers in the text with:

There were `` `r stats$n` `` participants with a mean age of `` `r stats$mean_age` ``.

which turns into:

There were 1123 participants with a mean age of 43.5.

Inline stats from {gtsummary}

We saw very, very briefly yesterday:

```
1  inline_text(income_table, variable = "age_bir")
```

```
[1] "595 (95% CI 538, 652; p<0.001)"
```

We pulled a statistic from our univariate table

If we're making a table, we probably want to report numbers from it

````
```{r}
#| label: tbl-descr
#| tbl-cap: "Descriptive statistics"
#| output-location: slide
table1 <- tbl_summary(
  nlsy,
  by = sex_cat,
  include = c(sex_cat, race_eth_cat, region_cat,
              eyesight_cat, glasses, age_bir)) |>
  add_overall(last = TRUE)
table1
```
````

If we're making a table, we probably want to report numbers from it

Table 2:
Descriptive statistics

Characteristic	**Male**, N = 6,403[1]	**Female**, N = 6,283[1]	**Overall**, N = 12,686[1]
race_eth_cat			
Hispanic	1,000 (16%)	1,002 (16%)	2,002 (16%)
Black	1,613 (25%)	1,561 (25%)	3,174 (25%)
Non-Black, Non-Hispanic	3,790 (59%)	3,720 (59%)	7,510 (59%)
region_cat			
Northeast	1,296 (21%)	1,254 (20%)	2,550 (20%)
North Central	1,488 (24%)	1,446 (23%)	2,934 (24%)
South	2,251 (36%)	2,317 (38%)	4,568 (37%)
West	1,253 (20%)	1,142 (19%)	2,395 (19%)
Unknown	115	124	239
eyesight_cat			
Excellent	1,582 (38%)	1,334 (31%)	2,916 (35%)
Very good	1,470 (35%)	1,500 (35%)	2,970 (35%)

[1] n (%); Median (IQR)

Characteristic	**Male**, N = 6,403[1]	**Female**, N = 6,283[1]	**Overall**, N = 12,686[1]
Good	792 (19%)	1,002 (23%)	1,794 (21%)
Fair	267 (6.4%)	365 (8.5%)	632 (7.5%)
Poor	47 (1.1%)	85 (2.0%)	132 (1.6%)
Unknown	2,245	1,997	4,242
glasses	1,566 (38%)	2,328 (54%)	3,894 (46%)
Unknown	2,241	1,995	4,236
age_bir	25 (21, 29)	22 (19, 27)	23 (20, 28)
Unknown	3,652	3,091	6,743

[1] n (%); Median (IQR)

I want to report some stats!

How about the median (IQR) age of the male participants at the birth of their first child?

```
inline_text(table1, variable = "age_bir", column = "Male")
```

```
[1] "25 (21, 29)"
```

Or the frequency and percentage of women from the South?

```
inline_text(table1, variable = "region_cat", level = "South", colu
```

```
[1] "2,317 (38%)"
```

And the overall stats on people who wear glasses?

```
inline_text(table1, variable = "glasses", column = "stat_0",
            pattern = "{n}/{N} ({p}%)")
```

```
[1] "3,894/8,450 (46%)"
```

Better yet...

We can integrate these into the text of our manuscript:

```
A greater proportion of female (`r inline_text(table1,
variable = "glasses", column = "Female")`) than male

(`r inline_text(table1, variable = "glasses", column =
"Male")`) participants wore glasses.
```

Which becomes:

A greater proportion of female (2,328 (54%)) than male (1,566 (38%)) participants wore glasses.

Readability

Because this can be hard to read, I'd suggest storing those stats in a chunk before the text:

````
```{r}
glasses_f <- inline_text(table1, variable = "glasses",
                         column = "Female")
glasses_m <- inline_text(table1, variable = "glasses",
                         column = "Male")
```
A greater proportion of female (`r glasses_f`) than male (`r glass
````

Exercises

Return to the quarto document with the tables.

- Choose a table to label and caption, and then write a sentence that *cross-references* it (e.g., Table 1 shows the descriptive statistics)
- From that table, choose at least two statistics to pull out of the table and include in the text using `inline_text()`.
- Add another statistic to the text that you calculate yourself using the `nlsy` data, e.g., the mean number of hours of sleep on weekends.

Package management for R

What is `{renv}`?

`{renv}` is an R package for managing project dependencies and creating reproducible environments

Benefits of using {renv}

1. **Isolation:** Creates project-specific environments separate from the global R library.
2. **Reproducibility:** Ensures consistent package versions for code reproducibility.
3. **Collaboration:** Facilitates sharing and collaborating on projects with others.

Getting Started with `{renv}`

1. Install `{renv}`

```
install.packages("renv")
```

2. Initialize a project

```
renv::init()
```

3. Install packages

```
install.packages("other_package")
# only an option when using renv!
install.packages("github_user/github_package")
```

4. Track dependencies via a lockfile

```
renv::snapshot()
```

Behind the scenes

- Your project `.Rprofile` is updated to include:

```
source("renv/activate.R")
```

- This is run every time R starts, and does some management of the library paths to make sure when you call `install.packges("package")` or `library(package)` it does to the right place (`renv/library/R-{version}/{computer-specifics}`)
- A `renv.lock` file (really just a text file) is created to store the names and versions of the packages.

renv.lock


```
{
  "R": {
    "Version": "4.3.0",
    "Repositories": [
      {
        "Name": "CRAN",
        "URL": "https://cran.rstudio.com"
      }
    ]
  },
  "Packages": {
    "R6": {
      "Package": "R6",
      "Version": "2.5.1",
      "Source": "Repository",
      "Repository": "CRAN",
      "Requirements": [
        "R"
      ],
      "Hash": "470851b6d5d0ac559e9d01bb352b4021"
    },
    base64enc": {
      "Package": "base64enc",
      "Version": "0.1-3",
      "Source": "Repository",
      "Repository": "CRAN",
      "Requirements": [
```

```
      "R"
    ],
    "Hash": "543776ae6848fde2f48ff3816d0628bc"
  },
```

Using {renv} later

Restore an environment

```
renv::restore()
```

Install new packages

```
install.packages("other_package")
```

Update the lockfile

```
renv::snapshot()
```

Collaboration with `{renv}`

- Share the project's `renv.lock` file with collaborators to ensure consistent environments
- When they run `renv::restore()`, the correct versions of the packages will be installed on their computer

```
renv::restore()
```

Other helpful functions

Remove packages that are no longer used:

```
renv::clean()
```

Check the status of the project library with respect to the lockfile:

```
renv::status()
```

This will tell you to `renv::snapshot()` to add packages you've installed but haven't snapshotted, or `renv::restore()` if you're missing packages you need but which aren't installed

Conclusion

`{renv}` benefits:

- Isolation, reproducibility, and collaboration

Getting started with `{renv}`

1. Initialize a project using `renv::init()`
2. Install packages and store with `renv::snapshot()`
3. Restore later or elsewhere with `renv::restore()`

Exercises

3. Install a new R package of your choice. (Not sure what to choose? Try one of these fun packages. For example, I did `install.packages("hadley/emo")`.)
4. Create an R script and save it in your R project. Include some code that requires the package. For example:

```
1  emo::ji("banana")
```

4. Run `renv::status()` to make sure your project picked up the package as a dependency.
5. Run `renv::snapshot()` to record that package in your lockfile.
6. Open your lockfile and look for your new package. For example, mine now has:

```
"emo": {
      "Package": "emo",
      "Version": "0.0.0.9000",
      "Source": "git",
      "RemoteType": "git",
      "RemoteUrl": "https://github.com/hadley/emo",
      "RemoteHost": "api.github.com",
      "RemoteUsername": "hadley",
      "RemoteRepo": "emo",
      "RemoteRef": "master",
      "RemoteSha": "3f03b11491ce3d6fc5601e210927eff73bf8e350",
      "Requirements": [
        "R",
        "assertthat",
        "crayon",
        "glue",
        "lubridate",
        "magrittr",
```

Functions

Functions in R

I've been denoting functions with parentheses: `func()`

We've seen functions such as:

- `mean()`
- `tbl_summary()`
- `init()`
- `create_github_token`

Functions take **arguments** and return **values**

Looking inside a function

If you want to see the code within a function, you can just type its name without the parentheses:

```
1  usethis::create_github_token
```

```
function (scopes = c("repo", "user", "gist", "workflow"), description = "DESCRIBE THE TOKEN'S USE
CASE",
    host = NULL)
{
    scopes <- glue_collapse(scopes, ",")
    host <- get_hosturl(host %||% default_api_url())
    url <- glue("{host}/settings/tokens/new?scopes={scopes}&description={description}")
    withr::defer(view_url(url))
    hint <- code_hint_with_host("gitcreds::gitcreds_set", host)
    ui_todo("\n    Call {ui_code(hint)} to register this token in the \\\n    local Git
credential store\n    It is also a great idea to store this token in any password-management \\\n
software that you use")
    invisible()
}
<bytecode: 0x10d9131a0>
<environment: namespace:usethis>
```

Structure of a function

```
1  func <- function()
```

You can name your function like you do any other object

- Just avoid names of existing functions

Structure of a function

```
1 func <- function(arg1,
2                  arg2 = defaul
3 }
```

What objects/values do you need to make your function work?

- You can give them default values to use if the user doesn't specify others

Structure of a function

```
1  func <- function(arg1,
2                   arg2 = defaul
3
4  }
```

Everything else goes within curly braces

- Code in here will essentially look like any other R code, using any inputs to your functions

Structure of a function

Structure of a function

```
1 func <- function(arg1,
2                  arg2 = defaul
3   new_val <- # do something wi
4   return(new_val)
5 }
```

Return something new that the code has produced

- The `return()` statement is actually optional. If you don't put it, it will return the last object in the code. When you're starting out, it's safer to always explicitly write out what you want to return.

Example: a new function for the mean

Let's say we are not satisfied with the `mean()` function and want to write our own.

Here's the general structure we'll start with.

```
1 new_mean <- function() {
2
3 }
```

New mean: arguments

We'll want to take the mean of a vector of numbers.

It will help to make an example of such a vector to think about what the input might look like, and to test the function. We'll call it `x`:

```
1 x <- c(1, 3, 5, 7, 9)
```

We can add `x` as an argument to our function:

```
1 new_mean <- function(x) {
2
3 }
```

New mean: function body

Let's think about how we calculate a mean in math, and then translate it into code:

So we need to sum the elements of `x` together, and then divide by the number of elements.

We can use the functions `sum()` and `length()` to help us.

We'll write the code with our test vector first, before inserting it into the function:

```
1  n <- length(x)
2  sum(x) / n
```

```
[1] 5
```

New mean: function body

Our code seems to be doing what we want, so let's insert it. To be explicit, I've stored the answer (within the function) as `mean_val`, then returned that value.

```
1 new_mean <- function(x) {
2   n <- length(x)
3   mean_val <- sum(x) / n
4   return(mean_val)
5 }
```

Testing a function

Let's plug in the vector that we created to test it:

```
new_mean(x = x)
```

```
[1] 5
```

And then try another one we create on the spot:

```
new_mean(x = c(100, 200, 300))
```

```
[1] 200
```

Adding another argument

Let's say we plan to be using our `new_mean()` function to calculate proportions (i.e., the mean of a binary variable). Sometimes we'll want to report them as multiplier by multiplying the proportion by 100.

Let's name our new function `prop()`. We'll use the same structure as we did with `new_mean()`.

```
1 prop <- function(x) {
2     n <- length(x)
3     mean_val <- sum(x) / n
```

Testing the code

Now we'll want to test on a vector of 1's and 0's.

```
1  x <- c(0, 1, 1)
```

To calculate the proportion and turn it into a percentage, we'll just multiply the mean by 100.

```
1  multiplier <- 100
2  multiplier * sum(x) / length(x)
```

```
[1] 66.66667
```

Testing the code

We want to give users the option to choose between a proportion and a percentage. So we'll add an argument `multiplier`. When we want to just return the proportion, we can just set `multiplier` to be 1.

```
multiplier <- 1
multiplier * sum(x) / length(x)
```

```
[1] 0.6666667
```

```
multiplier <- 100
multiplier * sum(x) / length(x)
```

```
[1] 66.66667
```

Adding another argument

If we add `multiplier` as an argument, we can refer to it in the function body.

```
1 prop <- function(x, multiplier) {
2    n <- length(x)
3    mean_val <- multiplier * sum(x) / 
4    return(mean_val)
5 }
```

Adding another argument

Now we can test:

```
1 prop(x = c(1, 0, 1, 0), multiplier =
```

```
[1] 0.5
```

```
1 prop(x = c(1, 0, 1, 0), multiplier =
```

```
[1] 50
```

Making a default argument

Since we don't want users to have to specify `multiplier = 1` every time they just want a proportion, we can set it as a **default**.

```
1 prop <- function(x, multiplier = 1) {
2   n <- length(x)
3   mean_val <- multiplier * sum(x) / n
4   return(mean_val)
5 }
```

Now we only need to specify that argument if we want a percentage.

```
1 prop(x = c(0, 1, 1, 1))
```

```
[1] 0.75
```

```
1 prop(x = c(0, 1, 1, 1), multiplier = 100)
```

Caveats

- This is obviously not the best way to write this function!
- For example, it will still work if `x = c(123, 593, -192)`…. but it certainly won't give you a proportion or a percentage!
- We could also put `multiplier =` *any number*, and we'll just be multiplying the answer by that number – this is essentially meaningless.
- We also haven't done any checking to see whether the user is even entering numbers! We could put in better error messages so users don't just get an R default error message if they do something wrong.

```
1 prop(x = c("blah", "blah", "blah"))
```

Exercises

Create some functions!

{targets}

What is {targets}?

"a Make-like pipeline tool for statistics and data science in R"

- manage a sequence of computational steps
- only update what needs updating
- ensure that the results at the end of the pipeline are still valid

Script-based workflow

01-data.R

```r
library(tidyverse)
data <- read_csv("data.csv", col_types = cols()) %>%
    filter(!is.na(Ozone))
write_rds(data, "data.rds")
```

02-model.R

```r
library(tidyverse)
data <- read_rds("data.rds")
model <- lm(Ozone ~ Temp, data) %>%
    coefficients()
write_rds(model, "model.rds")
```

03-plot.R

```r
library(tidyverse)
model <- read_rds("model.rds")
data <- read_rds("data.rds")
ggplot(data) +
    geom_point(aes(x = Temp, y = Ozone)) +
    geom_abline(intercept = model[1], slope = model[2])
ggsave("plot.png", plot)
```

Problems with script-based workflow

- **Reproducibility**: if you change something in one script, you have to remember to re-run the scripts that depend on it
- **Efficiency**: that means you'll usually rerun all the scripts even if they don't depend on the change
- **Scalability**: if you have a lot of scripts, it's hard to keep track of which ones depend on which
- **File management**: you have to keep track of which files are inputs and which are outputs and where they're saved

Based on example in https://books.ropensci.org/targets

{targets} workflow

R/functions.R

```r
get_data <- function(file) {
  read_csv(file, col_types = cols()) %>%
    filter(!is.na(Ozone))
}

fit_model <- function(data) {
  lm(Ozone ~ Temp, data) %>%
    coefficients()
}

plot_model <- function(model, data) {
  ggplot(data) +
    geom_point(aes(x = Temp, y = Ozone)) +
    geom_abline(intercept = model[1], slope = model[2])
```

{targets} workflow

_targets.R

```
1  library(targets)
2
3  tar_source()
4  tar_option_set(packages = c("tidyverse"))
5
6  list(
7    tar_target(file, "data.csv", format = "file"),
8    tar_target(data, get_data(file)),
9    tar_target(model, fit_model(data)),
10   tar_target(plot, plot_model(model, data))
11 )
```

Run `tar_make()` to run pipeline

`use_targets()` will generate a `_targets.R` script for you to fill in.

{targets} workflow

Targets are “hidden” away where you don’t need to manage them

```
├── _targets.R
├── data.csv
├── R/
│   ├── functions.R
├── _targets/
│   ├── objects
│          ├── data
│          ├── model
│          ├── plot
```

You can of course have multiple files in `R/`; `tar_source()` will source them all

My typical workflow with `{targets}`

1. Read in some data and do some cleaning until it's in the form I want to work with.
2. Wrap that in a function and save the file in `R/`.
3. Run `use_targets()` and edit `_targets.R` accordingly, so that I list the data file as a target and `clean_data` as the output of the cleaning function.
4. Run `tar_make()`.
5. Run `tar_load(clean_data)` so that I can work on the next step of my workflow.
6. Add the next function and corresponding target when I've solidified that step.

I usually include `library(targets)` in my project `.Rprofile` so that I can always call

`_targets.R` tips and tricks

```
list(
  tar_target(
    data_file,
    "data/raw_data.csv",
    format = "file"
  ),
  tar_target(
    raw_data,
    read.csv(data_file)
  ),
  tar_target(
    clean_data,
    clean_data_function(raw_data)
  )
```

I like to pair my functions/targets by name so that the workflow is clear to me

`_targets.R` tips and tricks

```
preparation <- list(
  ...,
  tar_target(
    clean_data,
    clean_data_function(raw_data)
  )
)
modeling <- list(
  tar_target(
    linear_model,
    linear_model_function(clean_data)
  ),
  ...
)
```

By grouping the targets into lists, I can easily comment out chunks of the pipeline

_targets.R tips and tricks

```r
## prepare ----
prepare <- list(
  ### cleanData.csv ----
  tar_target(
    cleanData.csv,
    file.path(path_to_data,
              "cleanData.csv"),
    format = "file"
  ),
  ### newdat ----
  tar_target(
    newdat,
    read_csv(cleanData.csv,
             guess_max = 20000)
```

In big projects, I comment my `_targets.R` file so that I can use the RStudio outline

Key `{targets}` functions

- `use_targets()` gets you started with a `_targets.R` script to fill in
- `tar_make()` runs the pipeline and saves the results in `_targets/objects/`
- `tar_make_future()` runs the pipeline in parallel[1]
- `tar_load()` loads the results of a target into the global environment (e.g., `tar_load(clean_data)`)
- `tar_read()` reads the results of a target into the global environment (e.g., `dat <- tar_read(clean_data)`)
- `tar_visnetwork()` creates a network diagram of the pipeline
- `tar_outdated()` checks which targets need to be updated
- `tar_prune()` deletes targets that are no longer in `_targets.R`
- `tar_destroy()` deletes the `.targets/` directory if you need to burn everything down and start again

1. Note: `{targets}` is moving to a new distributed computing strategy using `{crew}`

Advanced `{targets}`

“target factories”

repo status Active `tarchetypes` makes it easy to add certain kinds of common tasks to reproducible pipelines. Most of its functions create families of targets for parameterized R Markdown, simulation studies, and other general-purpose scenarios.

repo status Active `stantargets` is a workflow framework for Bayesian data analysis with `cmdstanr`. With concise, easy-to-use syntax, it defines versatile families of targets tailored to Bayesian statistics, from a single MCMC run with postprocessing to large simulation studies.

repo status Active Like `stantargets`, `jagstargets` is a workflow framework for Bayesian data analysis, with support for both single MCMC runs and large-scale simulation studies. It invokes JAGS through the `R2jags` package, which has nice features such as the ability to parallelize chains across local R processes.

`{tarchetypes}`: reports

Render documents that depend on targets loaded with `tar_load()` or `tar_read()`.

- `tar_render()` renders an R Markdown document
- `tar_quarto()` renders a Quarto document (or project)

It can't detect dependencies like `tar_load(ends_with("plot"))`

What does `report.qmd` look like?

````
---
title: "My report"
---
```{r}
library(targets)
tar_load(results)
tar_load(plots)
```
There were `r results$n` observations with a mean age of `r result
```{r}
library(ggplot2)
plots$age_plot
```
````

Because `report.qmd` depends on `results` and `plots`, it will only be re-rendered if either of those targets change.

{tarchetypes}: branching

Using data from the National Longitudinal Survey of Youth,

`_targets.R`

```
library(targets)
library(tarchetypes)
tar_source()

targets_setup <- list(
  tar_target(
    csv,
    "data/nlsy.csv",
    format = "file"
  ),
  tar_target(
    dat,
    readr::read_csv(csv,
      show_col_types = FALSE)
  )
)
```

`R/functions.R`

```
model_function <- function(outcome_var,
                           sex_val, dat) {

  lm(as.formula(paste(outcome_var,
      " ~ age_bir + income + factor(region)")) ,
     data = dat,
     subset = sex == sex_val)
}

coef_function <- function(model) {
  coef(model)[["age_bir"]]
}
```

we want to investigate the relationship between age at first birth and hours of sleep on weekdays and weekends among moms and dads separately

Option 1

Create (and name) a separate target for each combination of sleep variable (`"sleep_wkdy"`, `"sleep_wknd"`) and sex (male: `1`, female: `2`):

```
targets_1 <- list(
  tar_target(
    model_1,
    model_function(outcome_var = "sleep_wkdy", sex_val = 1, dat =
  ),
  tar_target(
    coef_1,
    coef_function(model_1)
  )
)
```

... and so on...

Option 2

Use `tarchetypes::tar_map()` to map over the combinations for you (static branching):

```
targets_2 <- tar_map(
  values = tidyr::crossing(
    outcome = c("sleep_wkdy", "sleep_wknd"),
    sex = 1:2
  ),
  tar_target(
    model_2,
    model_function(outcome_var = outcome, sex_val = sex, dat = dat
  ),
  tar_target(
    coef_2,
    coef_function(model_2)
  )
)
```

Option 2, cont.

Use `tarchetypes::tar_combine()` to combine the results of a call to `tar_map()`:

```
combined <- tar_combine(
  combined_coefs_2,
  targets_2[["coef_2"]],
  command = vctrs::vec_c(!!!.x),
)
tar_read(combined_coefs_2)
```

`command = vctrs::vec_c(!!!.x)` is the default, but you can supply your own function to combine the results

Option 3

Use the `pattern =` argument of `tar_target()` (dynamic branching):

```
targets_3 <- list(
  tar_target(
    outcome_target,
    c("sleep_wkdy", "sleep_wknd")
  ),
  tar_target(
    sex_target,
    1:2
  ),
  tar_target(
    model_3,
    model_function(outcome_var = outcome_target, sex_val = sex_tar
    pattern = cross(outcome_target, sex_target)
  )
```

Branching

Dynamic	Static
Pipeline creates new targets at runtime.	All targets defined in advance.
Cryptic target names.	Friendly target names.
Scales to hundreds of branches.	Does not scale as easily for tar_visnetwork() etc.
No metaprogramming required.	Familiarity with metaprogramming is helpful.

From https://books.ropensci.org/targets/dynamic.html#branching

Branching

- The book also has an example of using metaprogramming to map over different functions
  - i.e. fit multiple models with the same arguments
- Static and dynamic branching can be combined
  - e.g. `tar_map(values = ..., tar_target(..., pattern = map(...)))`
- Branching can lead to slowdowns in the pipeline (see book for suggestions)

{tarchetypes}: repetition

`tar_rep()` repeats a target multiple times with the same arguments

```
targets_4 <- list(
  tar_rep(
    bootstrap_coefs,
    dat |>
      dplyr::slice_sample(prop = 1, replace = TRUE) |>
      model_function(outcome_var = "sleep_wkdy", sex_val = 1, dat
      coef_function(),
    batches = 10,
    reps = 10
  )
)
```

The pipeline gets split into `batches` x `reps` chunks, each with its own random seed

{tarchetypes}: mapping over iterations

```
sensitivity_scenarios <- tibble::tibble(
  error = c("small", "medium", "large"),
  mean = c(1, 2, 3),
  sd = c(0.5, 0.75, 1)
)
```

`tar_map_rep()` repeats a target multiple times with different arguments

```
targets_5 <- tar_map_rep(
  sensitivity_analysis,
  dat |>
    dplyr::mutate(sleep_wkdy = sleep_wkdy + rnorm(nrow(dat), mean,
    model_function(outcome_var = "sleep_wkdy", sex_val = 1, dat =
    coef_function() |>
    data.frame(coef = _),
  values = sensitivity_scenarios,
```

```
 9    batches = 10,
10    reps = 10
11  )
```

{tarchetypes}: mapping over iterations

```
tar_read(sensitivity_analysis) |> head()
```

Ideal for sensitivity analyses that require multiple iterations of the same pipeline with different parameters

```
tar_read(sensitivity_analysis) |>
  dplyr::group_by(error) |>
  dplyr::summarize(q25 = quantile(coef, .25),
                   median = median(coef),
                   q75 = quantile(coef, .75))
```

Summary

- `{targets}` is a great tool for managing complex workflows
- `{tarchetypes}` makes it even more powerful
- The user manual is a great resource for learning more

Exercises

We'll clone a repo with `{targets}` already set up and add some additional steps to the analysis.