R Refresher

Professor Zuduo Zheng, School of Civil Engineering,
University of Queensland

2024-02-23

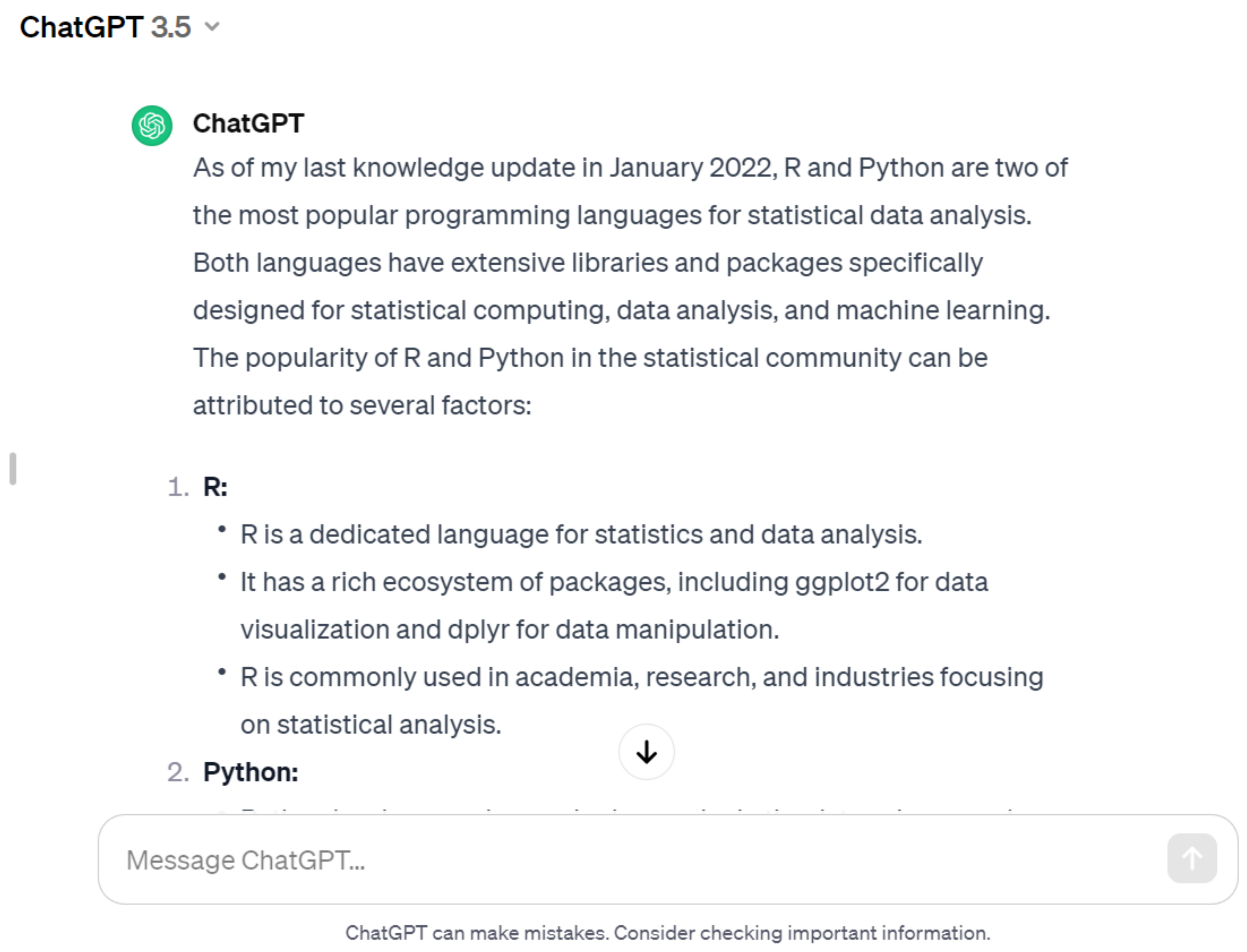
Introduction

- Since you have learned how to use R in CIVL2530, in this tutorial we will only cover some essential information directly related to the data analysis tasks in this course.
- You need four things to run the code in our tutorials: **R**, **RStudio**, a collection of R packages called the **tidyverse**, and a handful of other packages.
- Packages are the fundamental units of reproducible R code. They include reusable functions, documentation that describes how to use them, and sample data.

Why use R

- Free and open-source
- R packages and community
- An end-to-end workbench for data analysis
- Reproducibility and reporting
- User friendliness substantially enhanced by RStudio IDE
- The default language for data analysis. (to many statisticians and data analysts)

ChatGPT's Opinion



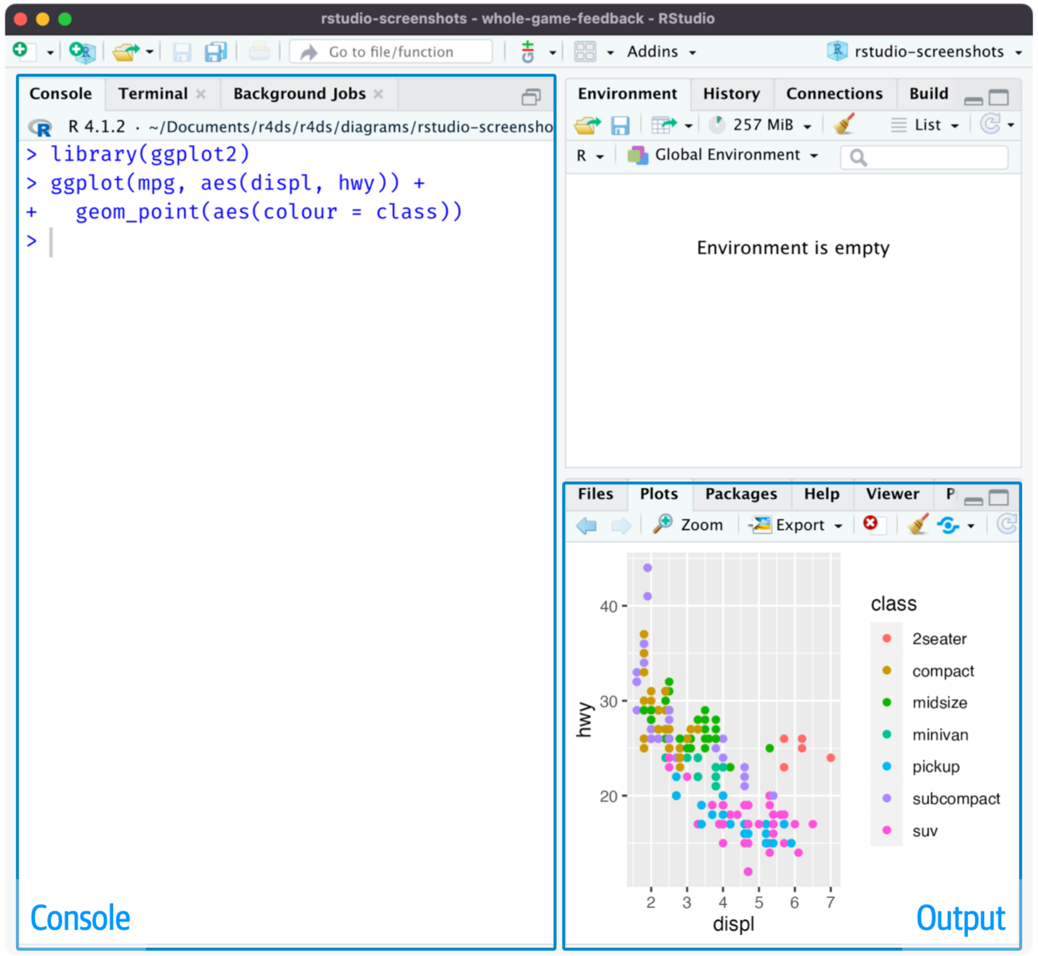
ChatGPT

As of my last knowledge update in January 2022, R and Python are two of the most popular programming languages for statistical data analysis. Both languages have extensive libraries and packages specifically designed for statistical computing, data analysis, and machine learning. The popularity of R and Python in the statistical community can be attributed to several factors:

1. **R:**
   - R is a dedicated language for statistics and data analysis.
   - It has a rich ecosystem of packages, including ggplot2 for data visualization and dplyr for data manipulation.
   - R is commonly used in academia, research, and industries focusing on statistical analysis.
2. **Python:**

Message ChatGPT...
ChatGPT can make mistakes. Consider checking important information.

RStudio IDE



```
> library(ggplot2)
> ggplot(mpg, aes(displ, hwy)) +
+   geom_point(aes(colour = class))
>
```

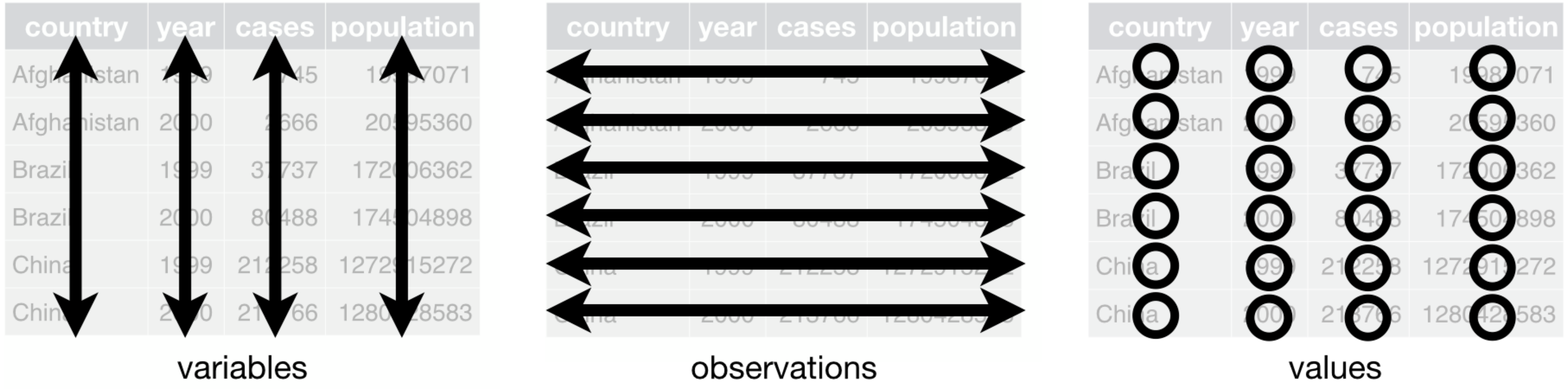
Two key regions: type R code in the console pane on the left, and look for plots in the output pane on the right.

Data frame

- A data frame is also used to store data with two dimensions, which are tightly coupled collections of variables
- used as the fundamental data structure by most of R‘s modeling software
- Probably the most powerful and flexible data structure you need to use most time in R

Data frame

A tidy data frame should look like this:



Tibble

- a special case of the base data.frame class, developed in response to lessons learned over many years of data analysis with data frames. Tibble is the central data structure for the set of packages known as the *tidyverse*, including *dplyr*, *ggplot2*, *tidyr*, and *readr*.
- tibbles have a more consistent behaviour with better usability in many cases. Most importantly, when a tibble object is printed to the console it automatically shows only the first 10 rows and condenses additional columns. By contrast, a data frame fills up the entire console screen with values which can lead to confusion.

Create a data frame

There are many ways to create a data frame. Two common options are:

1. Directly creating a (small) data frame Task:

Create two versions of the data frame below. The 1st version as a dataframe, and the 2nd as a tibble.

speed	volume	density	condition
34	1800	0.7	congested
56	1200	0.5	heavy
89	800	0.1	free
65	1300	0.55	heavy
24	1500	0.78	congested

Create a data frame: an example

```r
traffic <- data.frame(speed     = c(34, 56, 89, 65, 24),
                      volume    = c(1800, 1200, 800, 1300, 1500),
                      density   = c(0.7, 0.5, 0.1, 0.55, 0.78),
                      condition = c("congested", "heavy", "free", "
print(traffic)
```

```
  speed volume density condition
1    34   1800    0.70 congested
2    56   1200    0.50     heavy
3    89    800    0.10      free
4    65   1300    0.55     heavy
5    24   1500    0.78 congested
```

```r
summary(traffic)
```

```
     speed          volume        density        condition        
 Min.   :24.0   Min.   : 800   Min.   :0.100   Length:5          
 1st Qu.:34.0   1st Qu.:1200   1st Qu.:0.500   Class :character  
 Median :56.0   Median :1300   Median :0.550   Mode  :character  
 Mean   :53.6   Mean   :1320   Mean   :0.526                     
 3rd Qu.:65.0   3rd Qu.:1500   3rd Qu.:0.700                     
 Max.   :89.0   Max.   :1800   Max.   :0.780                     
```

```r
#tibble(traffic)
```

Create a tibble: an example

```r
library(tidyverse)
traffic_tib <- tibble(speed     = c(34, 56, 89, 65, 24),
                      volume    = c(1800, 1200, 800, 1300, 1500),
                      density   = c(0.7, 0.5, 0.1, 0.55, 0.78),
                      condition = c("congested", "heavy", "free", "
print(traffic_tib)
```

```
# A tibble: 5 × 4
  speed volume density condition
  <dbl>  <dbl>   <dbl> <chr>
1    34   1800    0.7  congested
2    56   1200    0.5  heavy
3    89    800    0.1  free
4    65   1300    0.55 heavy
5    24   1500    0.78 congested
```

Exercise

- What are the variable types for the data in the table below?
- Change “condition” from character to factor and make the output dataframe as a tibble.

speed	volume	density	condition
34	1800	0.7	congested
56	1200	0.5	heavy
89	800	0.1	free
65	1300	0.55	heavy
24	1500	0.78	congested

Solution

```r
library(tidyverse)
traffic <- mutate(traffic, condition = factor(condition)) |> as.tib
print(traffic)
```

```
# A tibble: 5 × 4
  speed volume density condition
  <dbl>  <dbl>   <dbl> <fct>
1    34   1800    0.7  congested
2    56   1200    0.5  heavy
3    89    800    0.1  free
4    65   1300    0.55 heavy
5    24   1500    0.78 congested
```

```r
glimpse(traffic)
```

```
Rows: 5
Columns: 4
$ speed     <dbl> 34, 56, 89, 65, 24
$ volume    <dbl> 1800, 1200, 800, 1300, 1500
$ density   <dbl> 0.70, 0.50, 0.10, 0.55, 0.78
$ condition <fct> congested, heavy, free, heavy, congested
```

Create a data frame

2. Directly import data into R as data frame or tibble

```r
library(tidyverse)
station <- read_csv("pm_station_1014.csv")
glimpse(station)
```

```
Rows: 400
Columns: 3
$ speed     <dbl> 53, 51, 49, NA, 35, 61, 46, 52, 58, 49, 59, 50, 35, 50, 59,
…
$ volume    <dbl> 5, 5, 7, 4, NA, NA, NA, NA, 6, 4, 2, 4, 7, NA, NA, NA, 7, 4,
…
$ occupancy <dbl> 2, 2, 3, 2, 2, 1, 3, 6, 2, 1, NA, 2, 2, NA, 3, NA, 2, 5, 2,
…
```

Exercise: Tibbles versus Data Frames

Which answers about data frames and tibbles are correct?

1. The printed output to the console is the same for tibbles and data frames;
2. All functions defined for data frames also work on tibbles;
3. Tibbles also show the data type in the console output;
4. To use tibble object the tibbles package needs to be loaded;
5. The table dimensions are not shown in the console output for tibbles.

File management

- List all the objects you have created so far using *ls()*;

```r
#|echo:true
ls()
```

```
[1] "station"     "traffic"     "traffic_tib"
```

- Delete objects using *rm(x)*;

```r
#|echo:true
rm(traffic) # remove traffic
ls() # check
```

```
[1] "station"     "traffic_tib"
```

Question

How to delete everything in your work environment?

Delete everything

```r
#|echo:true
rm(list = ls())
ls()
```

```
character(0)
```

Warning:: Think carefully before using the codes above.
Question: When do you think it is a good idea to delete everything?

Working directory

- Save what you have done using *save.image()*;
- Where did you save it? check your current working directory:

```r
getwd()
```

```
[1] "C:/Users/uqzzhen5/Dropbox/Teaching/UQ/CIVL3530/2024/tutorial_1"
```

Also, you can get your current working directory at the top of the console. It is a good practice to explicitly set the working directory:

```r
setwd("C:/Users/uqzzhen5/Dropbox/Teaching/UQ/CIVL3530/2024/tutorial
```

File management

- For Windows users, unlike you would do in other occasions you can not use a backslash to separate two components of the path; to get a single backslash, you need to type two backslashes.
- To save some typing and improve the readability, a recommended way of providing a path in R is to use forward slashes (the Mac/Linux style), for example, “C:/Users/zhengz2/Dropbox/bookproject/chapter2_start_R”.

File management

- The path used in the example above is an absolute path which points to the same place regardless of your working directory. For Mac/Linux users, an absolute path starts with a slash “/”.
- Using absolute paths should be avoided because it hurts your script’s reproducibility as it would be extremely rare to have the same directory configuration on another computer.
- So, using **relative paths** whenever possible, more specifically, paths relative to your working directory.

File path: example

```r
library(tidyverse)
station <- read_csv("C:/Users/uqzzhen5/Dropbox/Teaching/UQ/CIVL353
```

The asolute path should be avoided. Use this instead:

```r
library(tidyverse)
station <- read_csv("pm_station_1014.csv")
```

R project

The recommended way of managing your files is through project in the IDE (integrated development environment) provided by RStudio, which allows you to keep together all the files associated with a project. If you manage your files in this way, you can easily access all the files you would need to re-run the project: input data, R scripts, figures, and etc.

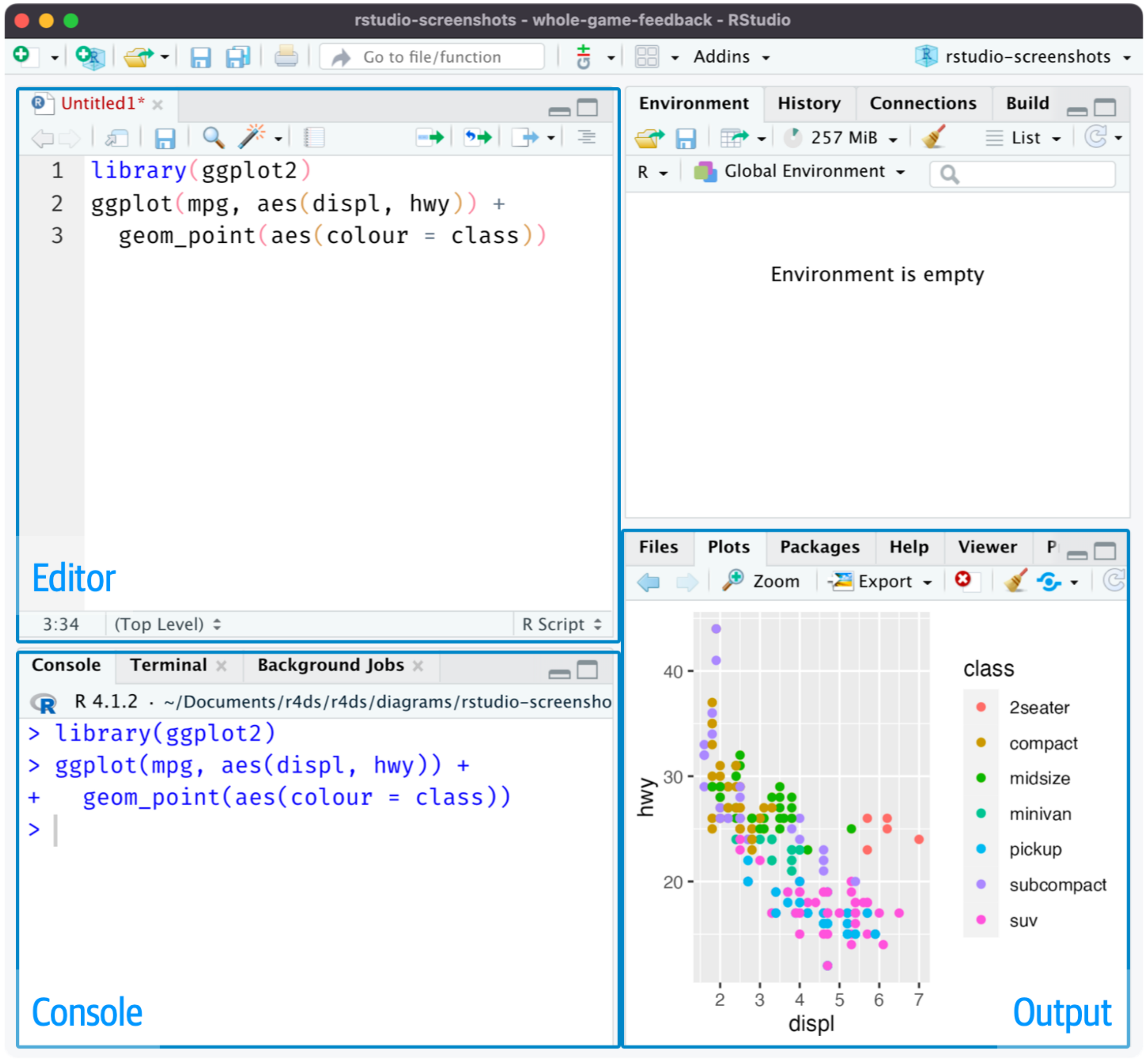
Exercise: Creating a R project

Task: Create a R project; name it “CIVL3530_tut_1”, and save it in the folder “CIVL3530_2024”.

Script editor

- The console is a great place to start coding and do some quick exploring in RStudio.
- The script editor should be a preferred place if you need to write lots of lines of script.
- The key to using the script editor effectively is to memorize one of the most important keyboard shortcuts: Cmd/Ctrl + Enter
- You can save your script as a script file to easily return to later.

Script editor



```
library(ggplot2)
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(colour = class))
```

```
R 4.1.2 · ~/Documents/r4ds/r4ds/diagrams/rstudio-screensho
> library(ggplot2)
> ggplot(mpg, aes(displ, hwy)) +
+   geom_point(aes(colour = class))
>
```

Importing data

- To import csv data, use *read_csv()* function in the **tidyverse** package.
- There are numerous methods for exporting R objects into other formats.
- Most time you can easily save your data file as a csv file, and then import it using *read_csv()*.

Import csv data

Please download the data file “pm_station_1014.csv” from the Blackboard, save it to your local computer. Then import it into R as a data frame named as *traffic_data*. In your script, **relative path** should be used.

```r
library(tidyverse)
station <- read_csv("pm_station_1014.csv")
```

Getting help

- Read the documents supplied with R: On the r-project webpage, you can find lots of useful documents, e.g., manuals on various aspects of R as a language, references for packages, frequently asked questions, resources, etc. It also provides a Search Engine for R.

```r
help.start() # access the help documents supplied with R.
```

```
If nothing happens, you should open
'http://127.0.0.1:23429/doc/html/index.html' yourself
```

Getting help

- Get help on a function: you can either use *help (function name)* or *?function name*. If you want to take a look at some examples on how to use a function, use *example(function name)*.

```r
help(mean)
?mean
args(mean)
```

```
function (x, ...)
NULL
```

```r
example(mean)
```

```

mean> x <- c(0:10, 50)

mean> xm <- mean(x)

mean> c(xm, mean(x, trim = 0.10))
[1] 8.75 5.50
```

Useful shortcuts

- Run current line/selection: Ctrl+Enter (Windows & Linux); Command+Enter (Mac)
- Navigate command history: Up/Down
- Insert chunk: Ctrl+Alt+I (Windows & Linux); Command+Option+I (Mac)
- Insert “<-”: Alt + -
- Insert pipe： Ctrl + Shift + M (Windows) or Cmd + Shift + M (Mac)
- Clear console: Ctrl+L
- Quit Session: Ctrl+Q (Windows & Linux); Command+Q (Mac)

File naming

Three important principles for file naming:

- File names should be machine readable: avoid spaces, symbols, and special characters. Don’t rely on case sensitivity to distinguish files.
- File names should be human readable: use file names to describe what’s in the file.
- File names should play well with default ordering: start file names with numbers so that alphabetical sorting puts them in the order they get used.

Exercise

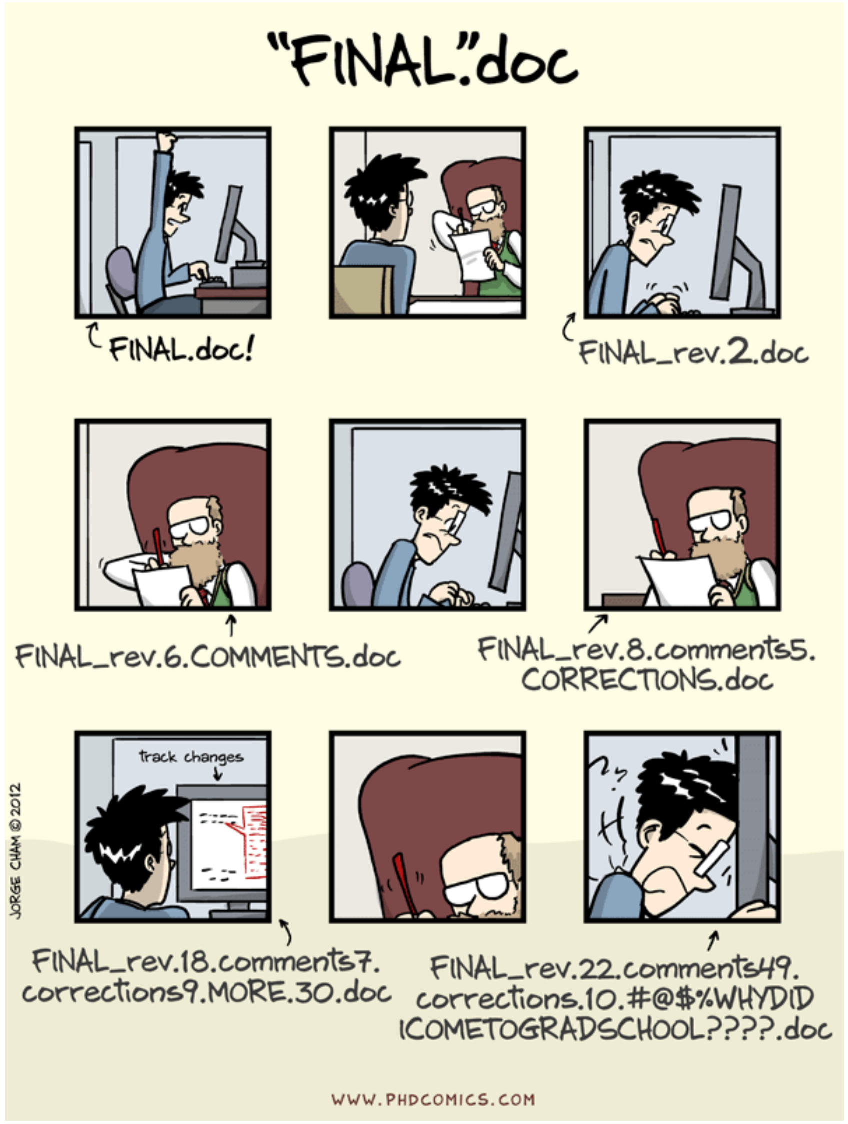
Suppose these files in a folder on your computer. Identify the issues in the file names.

```
alternative model.R
code for exploratory analysis.r
finalreport.qmd
FinalReport.qmd
fig 1.png
Figure_02.png
model_first_try.R
run-first.r
temp.txt
```

Solution

There are a variety of problems here: it’s hard to find which file to run first, file names contain spaces, there are two files with the same name but different capitalization (`finalreport` vs.`FinalReport`), and some names don’t describe their contents (`run-first` and `temp`).

Bad practice



Good practice

```
01-load-data.R
02-exploratory-analysis.R
03-model-approach-1.R
04-model-approach-2.R
fig-01.png
fig-02.png
report-2022-03-20.qmd
report-2022-04-02.qmd
report-draft-notes.txt
```

Good practice

- Numbering the key scripts make it obvious in which order to run them and a consistent naming scheme makes it easier to see what varies;
- The figures are labelled similarly;
- The reports are distinguished by dates included in the file names, and
- temp is renamed to report-draft-notes to better describe its contents.

Code style

- variable names should use only lowercase letters, numbers, and _. *Use* _ to separate words within a name.

```r
library(nycflights13)
# Strive for:
short_flights <- flights |> filter(air_time < 60)

# Avoid:
SHORTFLIGHTS <- flights |> filter(air_time < 60)
```

Code style

- As a general rule of thumb, it's better to prefer long, descriptive names that are easy to understand rather than concise names that are fast to type.
- If you have a bunch of names for related things, do your best to be consistent.
- Put spaces on either side of mathematical operators apart from ^ (i.e. +, -, ==, <, …), and around the assignment operator (<-).

```r
# Strive for
z <- (a + b)^2 / d
# Avoid
z<-( a + b ) ^ 2/d
```

Code style

- Don’t put spaces inside or outside parentheses for regular function calls. Always put a space after a comma, just like in standard English.

```r
# Strive for
mean(x, na.rm = TRUE)
# Avoid
mean (x ,na.rm=TRUE)
```

pipes

- A "pipeline": the process of chaining together multiple operations or functions using the pipe operator *%>%* (in the **magrittr** package, which is part of the **tidyverse**) or |> in R base.
- The pipe operator takes the result of the expression on its left-hand side and passes it as the first argument to the function on its right-hand side. This allows you to chain multiple functions together in a more readable and concise manner.

pipes

```
# Without a pipeline

result <- my_function(another_function(yet_another_function(data)))

# With a pipeline

result <- data %>% yet_another_function() %>% another_function() %>%
my_function()
```

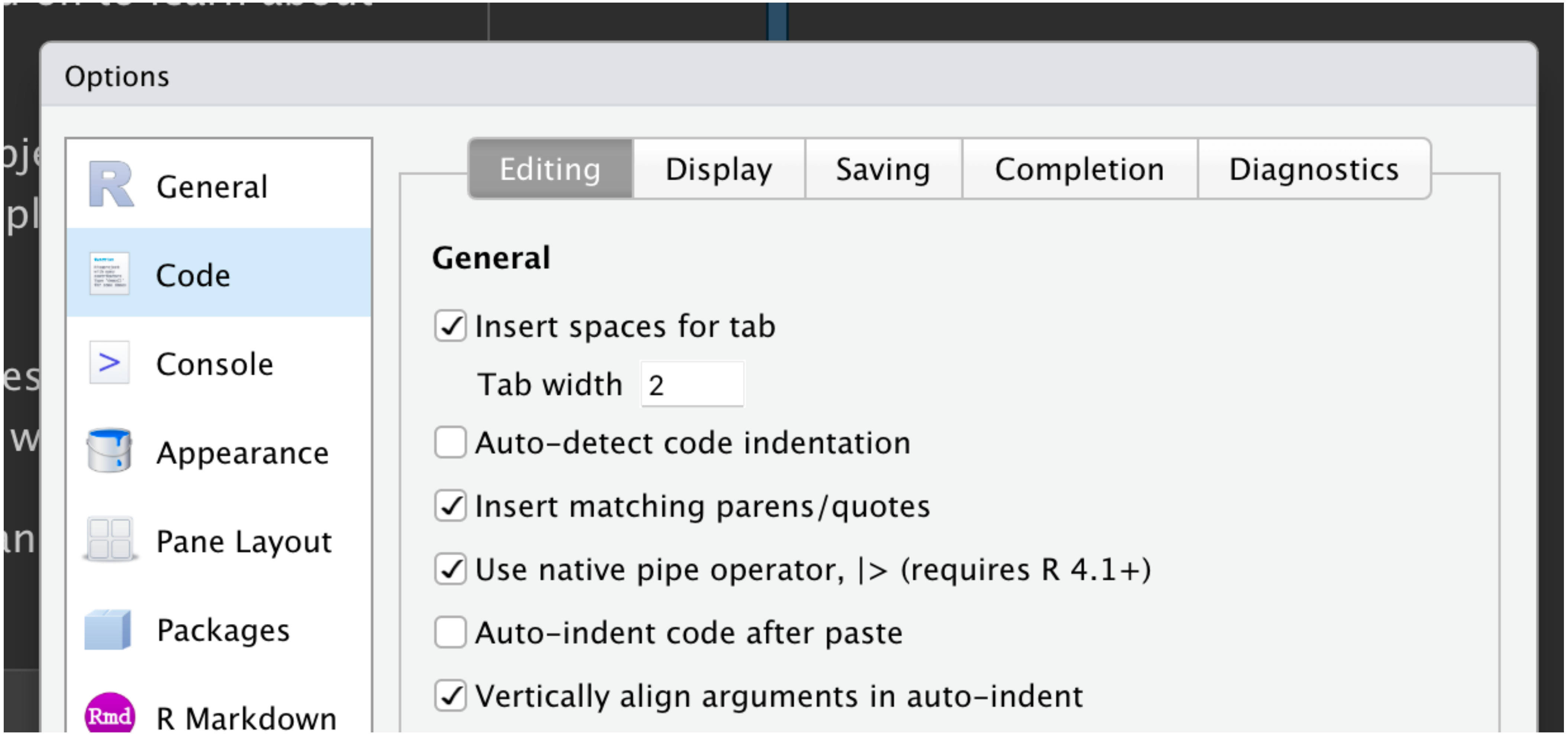
Benefits of the pipeline

- Improved readability: The pipeline allows you to read code from left to right, following the flow of operations.
- Modular code: Each step in the pipeline corresponds to a specific operation, making the code modular and easier to maintain.
- Reduced nesting: The pipeline reduces the need for nested function calls, leading to cleaner code.

The base pipe is preferred

- |> and %>% behave identically. However, |> is part of base R, and always available for you to use, even when you ere not using the tidyverse.
- To add the pipe to your code, you can use the built-in keyboard shortcut Ctrl/Cmd + Shift + M. You'll need to make one change to your RStudio options to use |> instead of %>% as shown in the figure below(make sure the "Use native pipe operator" option is checked.).

pipes



General

- [✓] Insert spaces for tab
  - Tab width 2
- [ ] Auto-detect code indentation
- [✓] Insert matching parens/quotes
- [✓] Use native pipe operator, |> (requires R 4.1+)
- [ ] Auto-indent code after paste
- [✓] Vertically align arguments in auto-indent

Good practices of using the base pipe (1)

|> should always have a space before it and should typically be the last thing on a line. This makes it easier to add new steps, rearrange existing steps, and modify elements within a step. The easiest way to pronounce the pipe is “then”.

```r
# Strive for
flights |>
  filter(!is.na(arr_delay), !is.na(tailnum)) |>
  count(dest)
# Avoid
flights|>filter(!is.na(arr_delay), !is.na(tailnum))|>count(dest)
```

Good practices of using the base pipe (2)

If the function you’re piping into has named arguments (like *mutate()* or *summarize()*), put each argument on a new line. If the function doesn’t have named arguments (like *select()* or *filter()*), keep everything on one line unless it doesn’t fit, in which case you should put each argument on its own line.

Good practices of using the base pipe (2)

```r
# Strive for
flights |>
  group_by(tailnum) |>
  summarize(
    delay = mean(arr_delay, na.rm = TRUE),
    n = n()
  )
# Avoid
flights |>
  group_by(
    tailnum
  ) |>
  summarize(delay = mean(arr_delay, na.rm = TRUE), n = n())
```

Good practices of using the base pipe (3)

After the first step of the pipeline, indent each line by two spaces. RStudio will automatically put the spaces in for you after a line break following a `|>` . If you’re putting each argument on its own line, indent by an extra two spaces. Make sure `)` is on its own line, and un-indented to match the horizontal position of the function name.

Good practices of using the base pipe (3)

```r
# Strive for
flights |>
  group_by(tailnum) |>
  summarize(
    delay = mean(arr_delay, na.rm = TRUE),
    n = n()
  )
# Avoid
flights|>
  group_by(tailnum) |>
  summarize(
             delay = mean(arr_delay, na.rm = TRUE),
             n = n()
           )
flights|>
  group_by(tailnum) |>
  summarize(
  delay = mean(arr_delay, na.rm = TRUE), n = n()
```

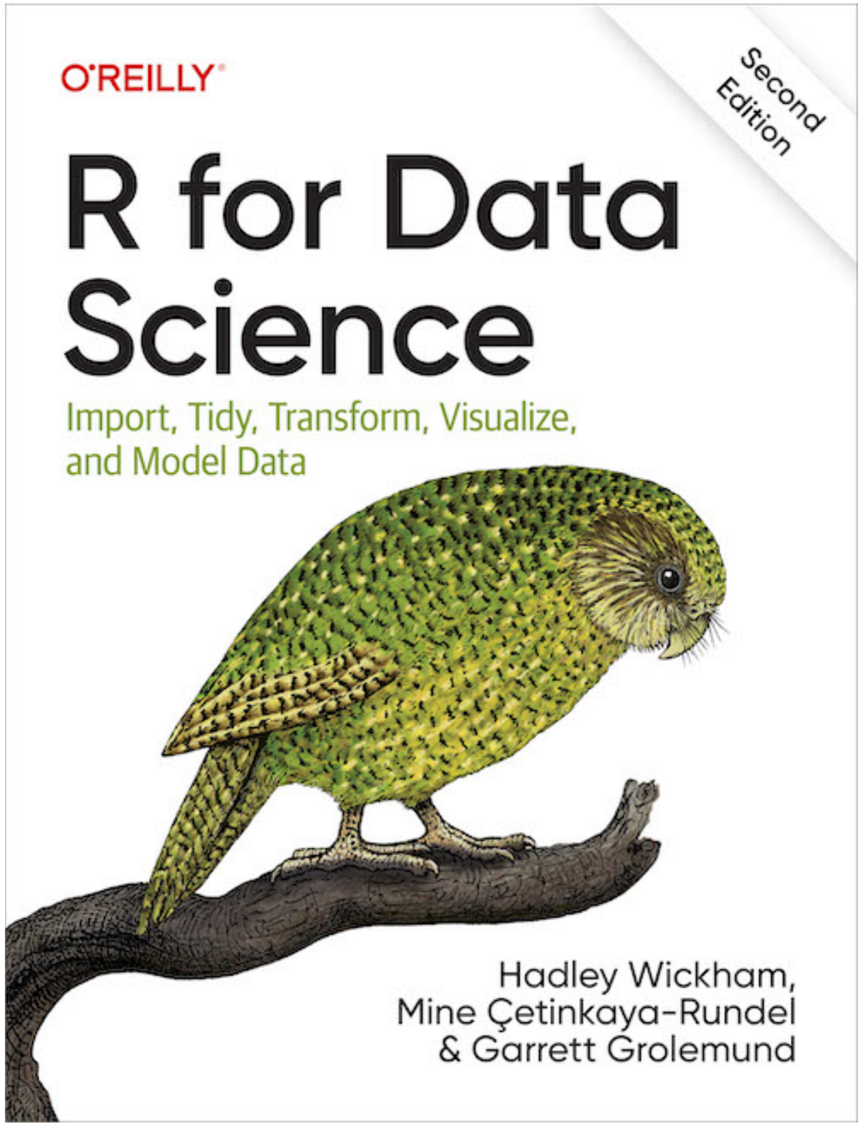
Exercise

Restyle the following pipelines following the guidelines above.

```
#|eval:false
flights|>filter(dest=="IAH")|>group_by(year,month,day)|>summarize(n
delay=mean(arr_delay,na.rm=TRUE))|>filter(n>10)
```

```
# A tibble: 365 × 5
# Groups:   year, month [12]
    year month   day     n delay
   <int> <int> <int> <int> <dbl>
 1  2013     1     1    20 17.8 
 2  2013     1     2    20  7   
 3  2013     1     3    19 18.3 
 4  2013     1     4    20 -3.2 
 5  2013     1     5    13 20.2 
 6  2013     1     6    18  9.28
 7  2013     1     7    19 -7.74
 8  2013     1     8    19  7.79
 9  2013     1     9    19 18.1 
10  2013     1    10    19  6.68
# ℹ 355 more rows
```

Reference and recommended reading


O'REILLY
Second Edition
R for Data Science
Import, Tidy, Transform, Visualize, and Model Data
Hadley Wickham, Mine Çetinkaya-Rundel & Garrett Grolemund