R: Tibbles and Dataframes

Sakol Suethanapornkul

28 October 2020

Quiz

Use the following vectors to answer the questions below:

```
scores <- c(3, 12, 8, 2, 4, 11, 15, 19, 3, 7, 6, 9)

answer <- c(TRUE, FALSE, FALSE, TRUE, TRUE, FALSE)
```

[1] What kind of vector is `scores`? How many elements does it have?

[2] Check whether values of `scores` are either below 5 or above 10.

[3] Check whether values of `scores` are 11, 12, and 15.

[4] What kind of vector is `answer`? Find the average of `answer`.

[5] Bonus: Extract `scores` below 7?

Answers

Here are the answers to the first four questions:

[1] `typeof(scores)` and `length(scores)`

[2] `scores < 5 | scores > 10`

[3] `scores == 11 | scores == 12 | scores == 15`

[4] `typeof(answer)` and `mean(answer)`

Answers

But our answer to [3] is cumbersome and prone to error.

- If we want to check if `x` is one of the “things” in `y`, we can use `%in%`.
- So, a better answer is: `scores %in% c(11, 12, 15)`

Answers

So far, we have learned: `==` `!=` `<` `<=` `>` `>=` and `x %in% c()`

Answer to [5] needs a bit more explanation about indices.

Index & Subsetting

Say, we write:

```
scores <- c(3, 12, 8, 2, 4, 11, 15, 19, 3, 7, 6, 9)

scores < 7        #or check <- scores < 7
```

What do we get with this code?

Index & Subsetting

We'd like to extract `scores` below 7 from the vector (i.e., 3, 2, 4, 3, 6). We do not want `T` or `F` as an answer.

```
scores <- c(3, 12, 8, 2, 4, 11, 15, 19, 3, 7, 6, 9)
```

So, we'll need to talk about subsetting. Let's begin with subsetting with positive integers.

```
scores[1:2]

scores[c(1, 3, 5)]

scores[[2]]
```

Index & Subsetting

We can subset with a comparison function, which will keep values that are TRUE:

```
scores[scores < 7]
```

And if we need position numbers instead:

```
which(scores < 7)      #then

scores[which(scores < 7)]
```

Recap

What we have seen so far:

- `numbers <- c(1, 2, 5, 6)`; `numbers <- 1:10`
- `typeof(numbers)`; `length(numbers)`
- `numbers[1:2]`; `numbers[[2]]`
- `numbers >= 5`; `(numbers / 2) == 0 | numbers >= 3`
- `numbers %in% c(3, 5, 7)`
- `numbers * 3`

Recap

What we have seen so far:

- `numbers <- c(1, 2, 5, 6)`; `numbers <- 1:10`
- `typeof(numbers)`; `length(numbers)`
- `numbers[1:2]`; `numbers[[2]]`
- `numbers >= 5`; `(numbers / 2) == 0 | numbers >= 3`
- `numbers %in% c(3, 5, 7)`
- `numbers * 3`

Recap

What we have seen so far:

- `numbers <- c(1, 2, 5, 6)`; `numbers <- 1:10`
- `typeof(numbers)`; `length(numbers)`
- `numbers[1:2]`; `numbers[[2]]`
- `numbers >= 5`; `(numbers / 2) == 0 | numbers >= 3`
- `numbers %in% c(3, 5, 7)`
- `numbers * 3`

Recap

What we have seen so far:

- `numbers <- c(1, 2, 5, 6)`; `numbers <- 1:10`
- `typeof(numbers)`; `length(numbers)`
- `numbers[1:2]`; `numbers[[2]]`
- `numbers >= 5`; `(numbers / 2) == 0 | numbers >= 3`
- `numbers %in% c(3, 5, 7)`
- `numbers * 3`

Recap

What we have seen so far:

- `numbers <- c(1, 2, 5, 6)`; `numbers <- 1:10`
- `typeof(numbers)`; `length(numbers)`
- `numbers[1:2]`; `numbers[[2]]`
- `numbers >= 5`; `(numbers / 2) == 0 | numbers >= 3`
- `numbers %in% c(3, 5, 7)`
- `numbers * 3`

Recap

What we have seen so far:

- `numbers <- c(1, 2, 5, 6)`; `numbers <- 1:10`
- `typeof(numbers)`; `length(numbers)`
- `numbers[1:2]`; `numbers[[2]]`
- `numbers >= 5`; `(numbers / 2) == 0 | numbers >= 3`
- `numbers %in% c(3, 5, 7)`
- `numbers * 3`

Part I: The Basics

R contains numerous packages to perform various functions, and one of the first things we need to know is how to install external packages from CRAN:

```
install.packages("<package_name>")
```

For instance, to do data science, we'll need a package called `tidyverse`. We can install that on our laptop with:

```
install.packages("tidyverse")
```

Part I: The Basics

Once we have a package installed, we will need to load it with `library()`:

```
library(tidyverse)
```

EXERCISE: Try calling a package `lme4` with `library(lme4)`. What happened?

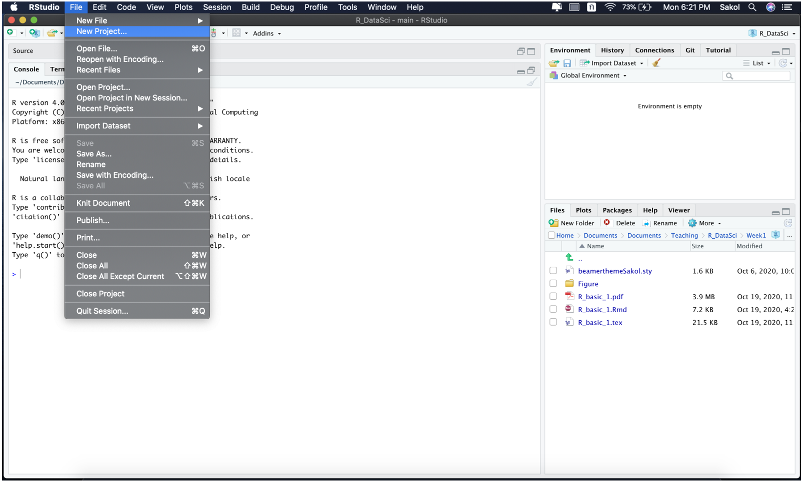
Part I: The Basics

Now that we have discussed `library()`, it's time we talk about our workflow!

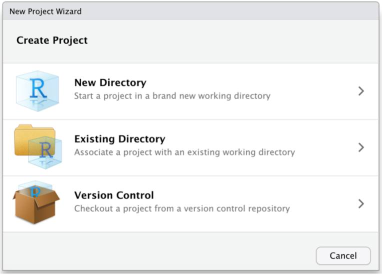
- Ideally, we want to call all the packages we need for our analysis once;
- We then want to run some analysis on our data and save all the results in one place;
- We also want to keep a record of what we have done in R, so we can recreate all these things later.

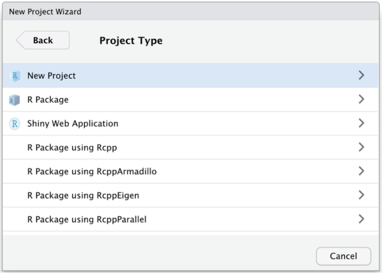
That can be achieved with an R project!

R project can be created right from RStudio:



R project can be created right from RStudio:





You can choose to place this newly-created R project in any folder you want. You can then check where your project lives with:

```
getwd()
```

Alternatively, you can see your current directory under Console.

EXERCISE: Why does this matter? What's the point of creating a project?

Run the following code (adapted from R for Data Science, p. 116):

```r
library(tidyverse)

ggplot(diamonds, aes(carat, price)) +
  geom_point(alpha = 0.2) +
  theme_bw()

ggsave("diamonds.png")


write_csv(diamonds, "diamonds.csv")
```

How can we foster this habit? A short answer is: create an R script!

An R Script, not the console, should be where your codes live.

- Begin with general comment about your script;
- Then, load all the packages you need in the preamble;
- Use comments to provide context for your analysis;
- Indent your codes and use new lines to your advantage.

Part II: Data frames & Tibbles

In most data analysis projects, we deal with data frames. Each data frame is **a named list of vectors**.

In data frames, the length of each element must be the same.

```
dat <- data.frame(Class  = c("A", "A", "A",
                             "B", "B", "B"),
                  Gender = c("M", "F", "M",
                             "F", "F", "M"),
                  Test   = c(15, 20, 21,
                             16, 17, 18)
                  )
```

Part II: Data frames & Tibbles

We can inspect the structure of a data frame with:

```
str(dat)
```

which lists on each line an object with a column name.

Another useful function, which provides a descriptive summary, is:

```
summary(dat)
```

And, finally, we can peak inside data frames:

```
head(dat, n = 3)
tail(dat, n = 3)
```

Part II: Data frames & Tibbles

We will work with data frames that come with the `tidyverse` package. Let's load two of the data sets:

```
data(diamonds)
data(mpg)
```

To find out more about these data sets, you can run `help()` such as `help(diamonds)`.

EXERCISE: Answer the following questions

[1] How many columns does `diamonds` have? How about `mpg`?

[2] In `diamonds`, what vector are `cut` and `color`?

[3] Figure out what `mpg` is about by running `help(mpg)`.

Part II: Data frames & Tibbles

If we create a data frame with `data.frame()` or read in external data with base R functions (e.g., `read.__`), we get a data frame.

Data frames have some undesirable behaviors. As shown in `diamonds`, some character vectors are converted to **factors**.

Part II: Data frames & Tibbles

```r
str(diamonds, vec.len = 0.5)
```

```
## tibble [53,940 x 10] (S3: tbl_df/tbl/data.frame)
##  $ carat  : num [1:53940] 0.23 ...
##  $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5
##  $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2
##  $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..:
##  $ depth  : num [1:53940] 61.5 ...
##  $ table  : num [1:53940] 55 ...
##  $ price  : int [1:53940] 326 ...
##  $ x      : num [1:53940] 3.95 ...
##  $ y      : num [1:53940] 3.98 ...
##  $ z      : num [1:53940] 2.43 ...
```

In this class, we will work with tibbles, which can be created with a function called `tibble()`:

```
dat <- tibble(Class  = c("A", "A", "A",
                         "B", "B", "B"),
              Gender = c("M", "F", "M",
                         "F", "F", "M"),
              Test   = c(15, 20, 21,
                         16, 17, 18)
              )
```

Right now, we will not dwell on why we should work with tibbles rather than data frames. We will instead focus on some helpful functions. Previously:

```
class <- c("A", "A", "A", "A", "A",
           "B", "B", "B", "B", "B",
           "C", "C", "C", "C", "C"
           )
```

This is tedious and error-prone. Instead:

```
class <- rep(c("A", "B", "C"), times = 4)

class <- rep(c("A", "B", "C"), each = 4)
```

You can `rep()` with various types of vectors.

```
a <- 1:5
b <- letters[1:5]

rep(a, each = 3)
rep(b, each = 2, times = 3)
```

Alternatively, you can generate a sequence of numbers with:

```
seq(from = 0, to = 1, by = 0.1)
seq(from = 0, to = 1, length.out = 100)
```

Wrap-up

We have discussed the following functions for data frames:

```
data.frame(a = , b = , ...)
str()
summary()
head()
tail()
```

And helpful functions to generate sequences for vectors:

```
seq()
rep()
```

Sandwiched in between are the basics:

```
install.packages()
library()
```

Homework

- First, create your own R project;
- Then, move some data file into that folder; and
- Last, create an R script for future analyses.