Lab 1 Demo 1

Di Zhou

2/5/2021

Contents

Installing and using packages (setting up your environment)

1. After creating a `R Project` and a `.Rmd` file, you need to install and load necessary packages so that your script is replicatable.

2. You only need to install packages once. After they are install, you can simply load them to your environment in the future.

```
knitr::opts_chunk$set(echo = TRUE)

# Install package
# (after you install, you can delete the line below and keep only the 'library' line)
# install.packages(c("tidyverse", "gridExtra", "kableExtra"))

# Load package to environment
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.0 --

## v ggplot2 3.3.2     v purrr   0.3.4
## v tibble  3.0.4     v dplyr   1.0.2
## v tidyr   1.1.2     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.0

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Coding in R Markdown

1. Chunks with a *white* background are *text editor* chunks. You can incorporate text formatting using Markdown languages. Use this cheatsheet for formatting questions.

2. Chunks with a *grey* background are *coding* chunks. You will code in these chunks.
3. You can run your code by line, or by chunk. The output (if any) will be displayed after the current coding chunk.

Debug in R coding

If you come across an error message, debug your code by:

1. Reading documentation of the function/package you use: Type the package or function name you use in the **Help** tab in the lower right panel.
2. Google your error massage
3. Post your question on stack overflow

For more information, you can read this chapter on debugging in R.

Types of variables in R

1. Most common data types in R:

- **Logical** variable: `TRUE` (`T`) or `FALSE` (`F`)

- **Character** variable (think of the "categorical variables" we covered in lecture): a string, e.g. "hello world!", "college education", "female"

- **Numeric** variable:
  - **Integer** (think of the "discrete variables" we covered in lecture): e.g. 1L, 2L, ...
  - **Double** (think of the "continuous variables" we covered in lecture): e.g. 1.44, 3.14
  - R automatically converts between these two classes when needed for mathematical purposes.

2. Variable types matter when you use different functions in R. For example, you cannot perform arithmetic with character variables even if they appear to be numbers.
3. Check variable type using `class()` or `str()`
4. To create a variable, you give it a name first, then use either `<-` or `=` followed by the value you want to assign. E.g. `variable1 = "hello world!"`
5. It's preferable to **leave spaces around your `<-` or `=`** so that your code is easy to read.

```
# logical
TRUE
```

```
## [1] TRUE
```

```
FALSE
```

```
## [1] FALSE
```

```
str(T)
```

```
##  logi TRUE
```

```
str(TRUE)
```

```
##  logi TRUE
```

```
class(T)
```

```
## [1] "logical"
```

```r
class(TRUE)
```

```
## [1] "logical"
```

```r
# character
c1 = "1.1"
c2 = "2"

# numeric
n1 = 1.1
n2 = 2

# try run:
# c1 + c2 # this will throw an error message
n1 + n2
```

```
## [1] 3.1
```

```r
# check variable type
str(c1)
```

```
##  chr "1.1"
```

```r
str(n1)
```

```
##  num 1.1
```

```r
class(c1)
```

```
## [1] "character"
```

```r
class(n1)
```

```
## [1] "numeric"
```

```r
# you can also use is.xxx() to get a T/F for a particular data type:
is.numeric(c1)
```

```
## [1] FALSE
```

```r
is.numeric(n1)
```

```
## [1] TRUE
```

```r
is.character(c1)
```

```
## [1] TRUE
```

```r
is.character(n1)
```

```
## [1] FALSE
```

```r
is.integer(c1)
```

```
## [1] FALSE
```

```r
is.integer(n1)
```

```
## [1] FALSE
```

Types of data in R

1. Most common data types in R:

- **Vectors**: A collection of elements of the same data type, e.g. logical vector, character vector, numeric vector
- **Matrices**: A vector with two dimensions. Elements in a matrix must share the same variable type (numeric, character, etc.).
- **Arrays**: Arrays are similar to matrices but can have more than two dimensions
- **Data frames**: Similar to matrices but different columns can have different variables types (numeric, character, logical, etc.). There can also be columns that have *data structure* rather than *variables*, e.g. a column of lists, a column of data frames, a column of matrices, etc.

- **Lists**: An ordered collection of objects within no constraint on their variable or data types. E.g. a list of character, a list of numeric, a list of vector, a list of list, a list of a mix of logical variables, character variables, and dataframes.
- **Factors**: A vector that is ordered. It can organize a categorical variable in a particular order for your desired ranking/ordering needs: for example, you can change a vector of educational levels `c("high school graduate", "4-year college", "some college", "below high school")` to have an internal ranking `c("below high school", "high school graduate", "some college", "4-year college", )` so that when you plot, these categories are ordered.

2. Similar to variable types, data types matter when you use different functions in R. It also matters in terms of indexing and managing data.

3. Similar to variable, to create a data object, you give it a name first, then use either `<-` or `=` followed by the data object you want to assign. E.g. `vector1 = c(1, 2, 5.3, 6, -2, 4)`

4. It's preferable to **leave space after each comma ,** in your code to make it easy to read.

```r
# --------- Vector ---------
v1 = c(1, 2, 5.3, 6, -2, 4) # numeric vector
v2 = c("one", "two", "three") # character vector
v3 = c(TRUE, TRUE, TRUE, FALSE, TRUE, FALSE) #logical vector

v4 = vector(mode = "numeric", length = 10) # a vector of zeros
v4[5] = 8 # you can assign value by indexing the vector

v5 = seq(0.1, 1, 0.1) # seq() creates a regular sequence

# check if vector
is.vector(v1)
```

```
## [1] TRUE
```

```r
# check length of vector
length(v1)
```

```
## [1] 6
```

```r
# vector operation
v6 = v4 + v5



# --------- Matrices ---------
m1 = matrix(rep(1, 9), nrow = 3, ncol = 3)
m2 = matrix(seq(0.1, 1.5, 0.1), nrow = 3, ncol = 5)
# combine matrices
cbind(m1, m2)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,]    1    1    1  0.1  0.4  0.7  1.0  1.3
## [2,]    1    1    1  0.2  0.5  0.8  1.1  1.4
## [3,]    1    1    1  0.3  0.6  0.9  1.2  1.5
```

```
# scalar
0.5 * m1
```

```
##      [,1] [,2] [,3]
## [1,]  0.5  0.5  0.5
## [2,]  0.5  0.5  0.5
## [3,]  0.5  0.5  0.5
```

```
# transpose
t(m2)
```

```
##      [,1] [,2] [,3]
## [1,]  0.1  0.2  0.3
## [2,]  0.4  0.5  0.6
## [3,]  0.7  0.8  0.9
## [4,]  1.0  1.1  1.2
## [5,]  1.3  1.4  1.5
```

```
# diagonal elements
diag(m2)
```

```
## [1] 0.1 0.5 0.9
```

```
# matrix multiplication
m1 %*% m2
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]  0.6  1.5  2.4  3.3  4.2
## [2,]  0.6  1.5  2.4  3.3  4.2
## [3,]  0.6  1.5  2.4  3.3  4.2
```

```
# element-wise multiplication
m1 * m2[1:3, 1:3]
```

```
##      [,1] [,2] [,3]
## [1,]  0.1  0.4  0.7
## [2,]  0.2  0.5  0.8
## [3,]  0.3  0.6  0.9
```

```
# --------- Data frames ---------
var1 <- c(1, 2, 3, 4)
var2 <- c("red", "white", "red", NA)
var3 <- c(TRUE, TRUE, TRUE, FALSE)
mydf <- data.frame(var1, var2, var3)
names(mydf) <- c("ID", "Color", "Passed") # update variable names

# Base R methods of data frame indexing
mydf[1:2] # columns 1, 2 of data frame
```

```
##   ID Color
## 1  1   red
## 2  2 white
## 3  3   red
## 4  4  <NA>
```

```
mydf[c(1, 3), ] # row 1, 3 of data frame
```

```
##   ID Color Passed
## 1  1   red   TRUE
## 3  3   red   TRUE
```

```
mydf[c("ID", "Color")] # columns ID and Color from data frame
```

```
##   ID Color
## 1  1   red
## 2  2 white
## 3  3   red
## 4  4  <NA>
```

```
mydf$Color # variable Color in the data frame
```

```
## [1] "red"   "white" "red"   NA
```

```
# Convert a vector or matrix to df
as.data.frame(v1)
```

```
##     v1
## 1  1.0
## 2  2.0
## 3  5.3
## 4  6.0
## 5 -2.0
## 6  4.0
```

```
as.data.frame(m1)
```

```
##   V1 V2 V3
## 1  1  1  1
## 2  1  1  1
## 3  1  1  1
```

```
# --------- Lists ---------
l1 = list(n1, c1, v1, m1)
l1
```

```
## [[1]]
## [1] 1.1
##
## [[2]]
## [1] "1.1"
##
## [[3]]
## [1]  1.0  2.0  5.3  6.0 -2.0  4.0
##
## [[4]]
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    1    1    1
## [3,]    1    1    1
```

```
# List index
l1[4]
```

```
## [[1]]
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    1    1    1
## [3,]    1    1    1
```

```
l1[[4]]
```

```
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    1    1    1
## [3,]    1    1    1
```

```
l1[4][[1]]
```

```
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    1    1    1
## [3,]    1    1    1
```

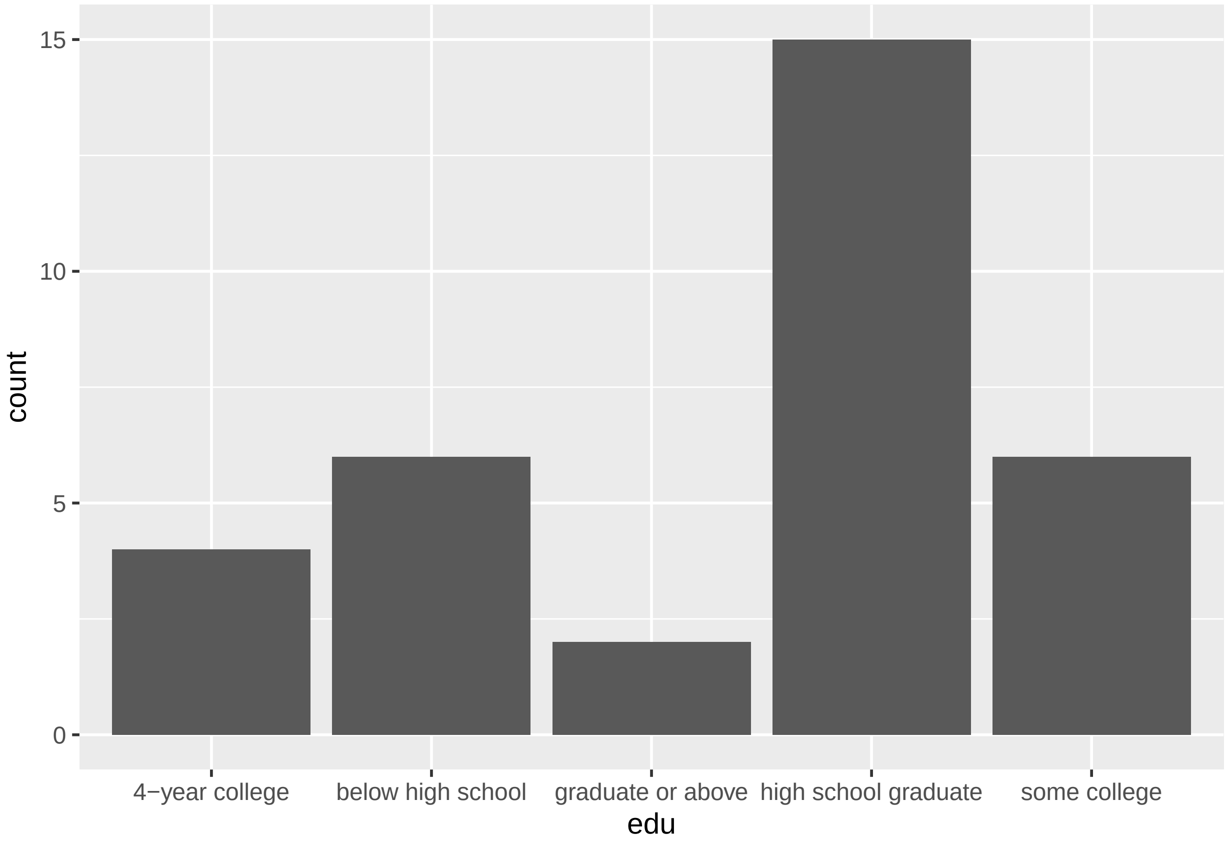
```
# Nested list
l2 = list(m2, l1)
l2
```

```
## [[1]]
##      [,1] [,2] [,3] [,4] [,5]
## [1,]  0.1  0.4  0.7  1.0  1.3
## [2,]  0.2  0.5  0.8  1.1  1.4
## [3,]  0.3  0.6  0.9  1.2  1.5
##
## [[2]]
## [[2]][[1]]
## [1] 1.1
##
## [[2]][[2]]
## [1] "1.1"
##
## [[2]][[3]]
## [1]  1.0  2.0  5.3  6.0 -2.0  4.0
##
## [[2]][[4]]
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    1    1    1
## [3,]    1    1    1
```

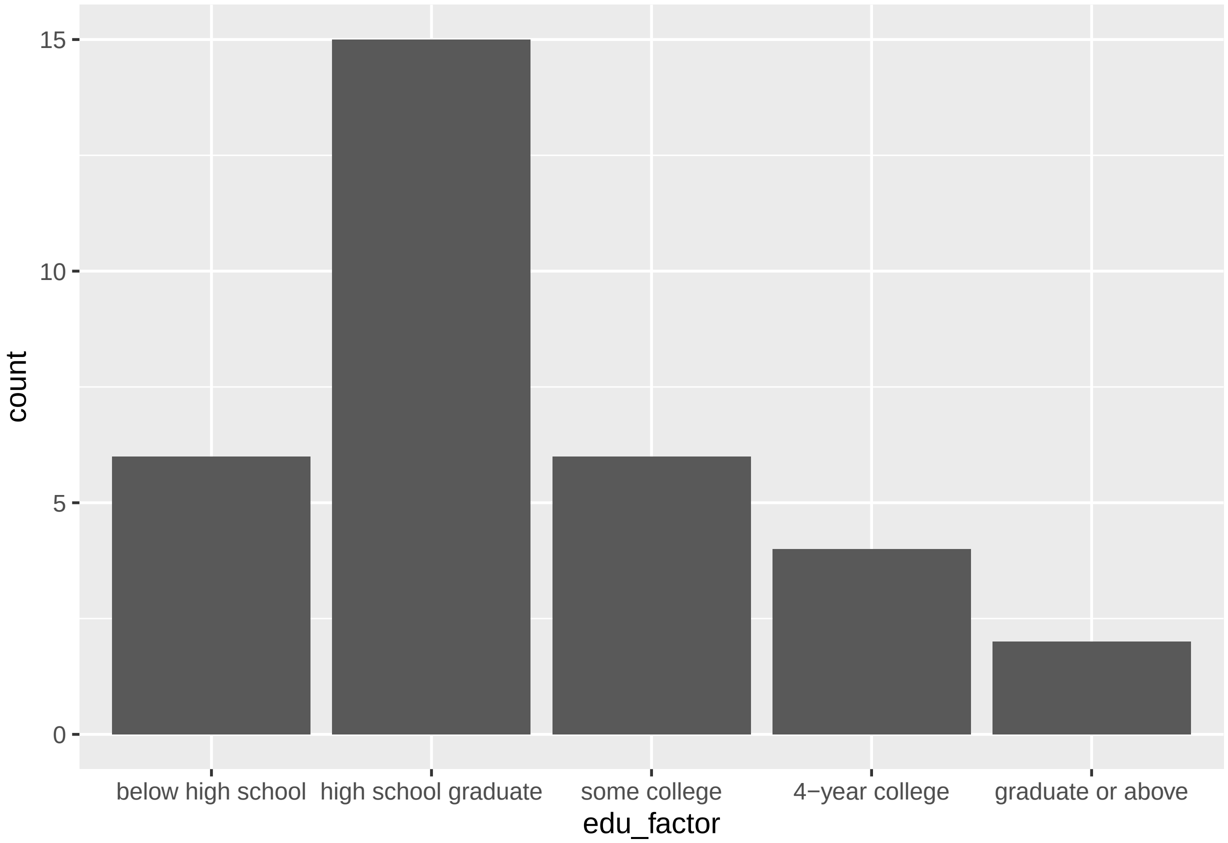
```
# --------- Factor ---------

# A vector of education levels, unordered
edu <- c(rep("high school graduate", 15),
         rep("4-year college", 4),
         rep("graduate or above", 2),
         rep("some college", 6),
         rep("below high school", 6))

# Plot its count: Default order is alphabetical
edu %>%
  as.data.frame() %>%
  ggplot(aes(x = edu)) +
  geom_bar()
```



```
# Covert vector to a factor with cutomized levels
edu_factor <- factor(edu,
                     levels = c("below high school",
                                "high school graduate" ,
                                "some college",
                                "4-year college",
                                "graduate or above"))
# Plot the counts again:
edu_factor %>%
  as.data.frame() %>%
  ggplot(aes(x = edu_factor)) +
  geom_bar()
```



What’s a function in R?

1. Functions execute certain tasks. For example, `class(x)` is a function that tells you the type of your input variable or data object.
2. A function is consist of a name, a set of arguments (input), and an output. The function `class(x)` has the function name **class**, and the input **x**, and will return an output -a character tells you the type of **x**.
3. You can write your own functions in R using the function syntax below:

```
your_function_name <- function(input){

  # A series of actions or operations of your function
  code
  code
  code

  return(output)

}
```

4. For example, write a function that add 1 to each value of an vector:

```r
# function
add_one <- function(vector){
  out_vector = vector + 1
  return(out_vector)
}
```

```r
# try:
add_one(v1)
```

```
## [1]  2.0  3.0  6.3  7.0 -1.0  5.0
```

Typing equations in R Markdown

1. For "displayed equations" (equations that will break your lines), use the double dollar sign `$$` to wrap your expression:

$$\overline{y} = \frac{y_1 + y_2 + y_3 + ... + y_n}{n} = \frac{\sum_{i=1}^n y_i}{n}$$

2. To type inline equations, use the dollar sign `$` to wrap your expression: $\hat{\mu} = \overline{y} = \frac{\sum_{i=1}^n y_i}{n}$.
3. **Do not leave a space between the `$$` (or `$`) and your mathematical notation!**
4. Hover over your R Markdown equation expressions, you can preview the equation you write.
5. You don't need to memorize all of the expressions. Google or refer to this guide when you work on mathematical equations.

Knitting R Markdown to HTML or PDF

Now let's try knit this R Markdown to HTML and PDF

1. Before you knit, always make sure you can run your code from beginning to end. You won't be able to knit if some codes throw error messages.
2. There are many options in R Markdown that helps you manipulate how you want your document be like. For example, you can hide the code chunk and only show the output by adding `echo = FALSE` in your code chunk options. You can also use `include = FALSE` to prevent the code AND its output to appear in your knitted document. You can also use `eval = FALSE` to prevent your code from running (but it will be displayed) in your knitted document.
3. For detailed documentation, see here.

Other advice

1. Managing your working environment:

- Keep an eye on objects in your environment. It's always easier to keep track of your work if you clear your environment every time you start a new task. In addition, by removing large data objects in your environment, R will run more smoothly.
- But make sure you save the important objects before you clear your environment. I always create a `temp_data` folder for temporary data, and use `save(object_name, file = "temp_data/object_name.RData")` to quickly save things I might need to use in the future.
- To clean the working environment, I often change the environment display from `list` to `Grid`, and select the objects I want to remove, then use the little broom logo in the Environment tab. You can also use `rm(list=ls())` to clear everything in your work space, but use this with caution!

2. In-line comments and other coding style suggestions:

- Make sure you comment your code (start comments with the `#` sign) **as detailed as you can**. It will help your grader and your reader and your(future)self to understand what's going on in the code.
- In general, it's better to be generous in spacing. Add spaces between numbers, =, commas, etc. You can start a new line after `,` in your code, so it's always better to break a very long line of code to multiple lines.
- R language is case-sensitive. Make sure you are using the correct function name, for example, `as.Date()` instead of `as.date()`.

- Unlike LaTeX, R Markdown doesn't have automatic spell check in the text editor chunks, but you can check that by clicking the "ABC" button to check your spellings.

3. Recommended reading: R for Data Science