Introduction to R

Camilo Abbate and Sofia Olguin

8/22/2025

[1]

1 Loading a Dataset

In this part, we will use data obtained from the replication package of Carneiro et al. (2021).[2]

2 Pipes (Ctrl + Shift + M)

Pipes are from the **magrittr** package, and **tidyverse** loads them automatically. It is used to chain a sequence of calculations or operations. Consider that as saying "then do this."

```
## Use the head function to get a snapshot of the data


# Without a pipe
head(dataset)
```

```
# Using a pipe
dataset %>%
  head()
```

[1]Instructors: Camilo Abbate and Sofia Olguin. This note was prepared for the 2025 UCSB Math Camp for Ph.D. students in Economics. It integrates materials from past instructors, including Will Heo and ChienHsun Lin, as well as sources like Hadley Wickham and Garrett Grolemund's *R for Data Science*, *DataCamp* and Michael Topper and Danny Klinenberg's "Data Wrangling for Economists".

[2]Carneiro, J., Cole, M. A., and Strobl, E. (2021). The effects of air pollution on students' cognitive performance: evidence from brazilian university entrance tests. Journal of the Association of Environmental and Resource Economists, 8(6):1051–1077.

3 Summary Statistics with Tidyverse Package

3.1 Exploring the data

```
## View only one data column (female)
dataset$female # dummy variable
```

```
## Mean with and without NA removal
mean(dataset$female)
```

```
## [1] 0.5704658
```

```
mean(dataset$female, na.rm=T) # the option na.rm removes NAs when calculating the mean
```

```
## [1] 0.5704658
```

```
## Using pipe and storing the value
avg_female <- dataset %>%
  pull(female) %>% # extract the column as a vector
  mean(na.rm=T) # calculate mean
```

3.2 Summary Statistics with summarize()

The summarize() function in the dplyr package allows us to quickly and efficiently generate statistics over columns. It allows you to compute values such as mean, standard deviation, and median, as well as give the generated columns specific names. Each call to summarize() delivers a tibble, a modern data frame with improved formatting.

```
# Mean of the female column
dataset %>%
  summarize(avg_female = mean(female, na.rm=T))
```

```r
# Average and Standard Deviation of math scores
dataset %>%
  summarize(avg_math_score = mean(score_math, na.rm=T),
            sd_math_score = sd(score_math, na.rm=T))
```

```r
# Average and Standard Deviation of math scores by gender (function group_by)
dataset %>%
  group_by(female) %>%
  summarize(avg_math_score = mean(score_math, na.rm=T),
            sd_math_score = sd(score_math, na.rm=T))
```

```r
# Round statistics to match the paper
dataset %>%
  group_by(female) %>%
  summarize(avg_math_score = mean(score_math, na.rm=T),
            sd_math_score = sd(score_math, na.rm=T)) %>%
  mutate(avg_math_score = round(avg_math_score, 0),
         sd_math_score = round(sd_math_score, 1))
```

3.3 Quick summary with summary()

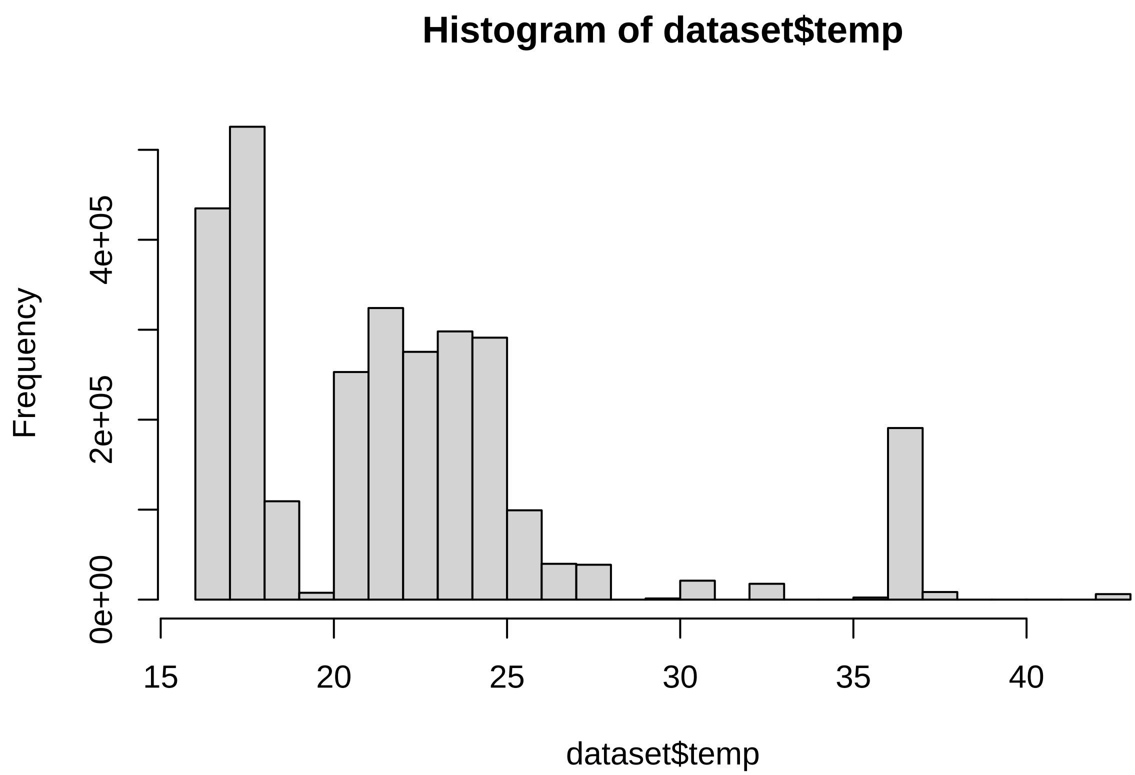
```r
summary(dataset$score_math)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     0.0   422.8   489.4   511.3   581.8   999.9
```

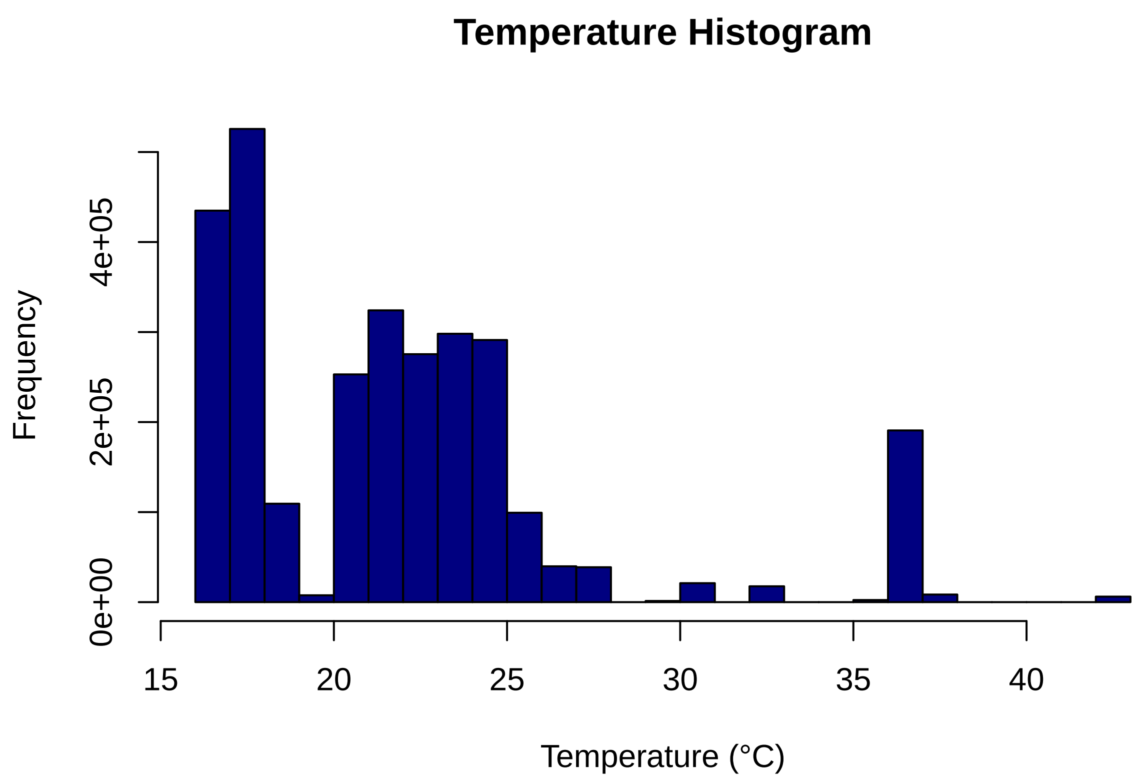
3.4 Histogram

We will produce histograms of the variable temperature (temp) using the base R histogram and the package ggplot2 that is part of the library tidyverse.

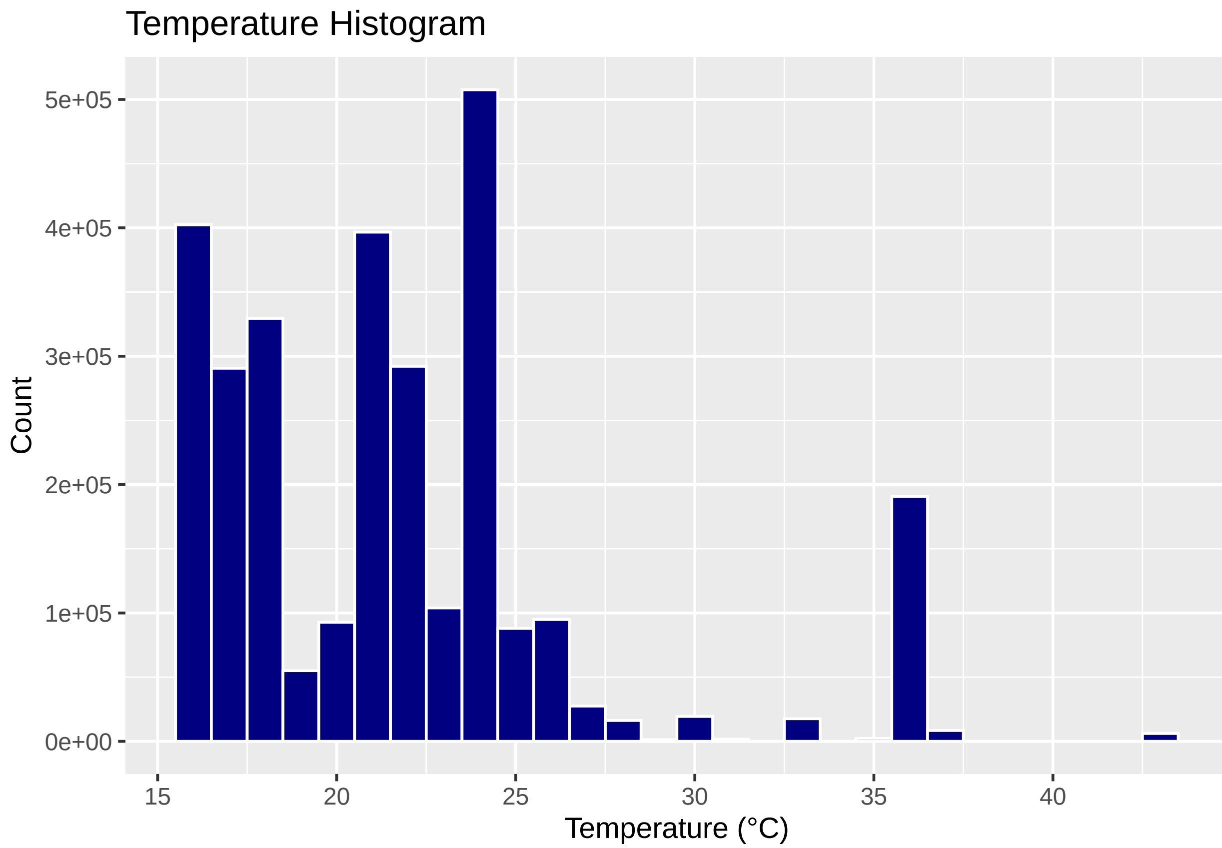
```r
# Base R Histogram

hist(dataset$temp)
```




```r
hist(dataset$temp,

     main = "Temperature Histogram",

     xlab = "Temperature (\u00B0C)",

     col = "navy")
```




```r
# Histogram using ggplot2
hist_tmp <- dataset %>%
  ggplot(aes(x = temp)) +
  geom_histogram(binwidth = 1, fill = "navy", color = "white") +
  labs(title = "Temperature Histogram",
       x = "Temperature (\u00B0C)",
       y = "Count")
hist_tmp
```



4 Vector and Matrix

Every vector has two key properties, type and length.There are various types of vectors: logical, integer, double, charater, complex, raw and list. Integer and double vectors are called numeric vectors.

```
typeof(letters)
```

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

```
typeof(1:10)
```

```
## [1] "integer"
```

```
x <- list("a","b",1:10) #list can contain another lists.
length(x)
```

```
## [1] 3
```

```
#Logical
1:10 %% 3 == 0 #divided by 3
```

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

For numeric vectors, integers have NA while doubles have four: NA, NaN, Inf and -Inf. NaN, Inf and -Inf can arise during division:

```
c(-1, 1, 0)/0
```

```
## [1] -Inf  Inf  NaN
```

You can coerce one type of vector to another. Explicit coercion happens when using as.logical(), as.integer(), as.double() or as.character(). Implicit coercion example:

```
x <- sample(20,100,replace=TRUE)
y <- x>10 #TRUE is converted to 1 and FALSE is converted to 0
sum(y)
```

```
## [1] 62
```

```
mean(y)
```

```
## [1] 0.62
```

4.1 Test functions: is_type()

```
is_integer(x)
```

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

```r
is_integer(y)
```

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

```r
is_logical(y)
```

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

4.2 Naming vectors

```r
c(x=1, y=2, z=4)
```

```
## x y z
## 1 2 4
```

```r
set_names(1:3, c("a", "b", "c"))
```

```
## a b c
## 1 2 3
```

4.3 Subsetting

```r
x <- c("one", "two", "three", "four", "five")
x[c(3, 2, 5)]
```

```
## [1] "three" "two"   "five"
```

```r
x <- c(abc = 1, def = 2, xyz = 5)
x[c("xyz", "def")]
```

```
## xyz def
##   5   2
```

4.4 Matrix

The syntax for matrix: matrix(value, nrow, ncol, byrow, dimnames) byrow is TRUE or FALSE. If TRUE, the elements of the matrix are arranged in the row, whereas FALSE will arrange the element by column-wise.

```r
mat = matrix(1:10,nrow = 2, ncol = 5,byrow = TRUE)
print(mat)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
```

```r
mat = matrix(1:10,nrow = 5, ncol = 2,byrow = F)
print(mat)
```

```
##      [,1] [,2]
## [1,]    1    6
## [2,]    2    7
## [3,]    3    8
## [4,]    4    9
## [5,]    5   10
```

```r
mat <- matrix(1:10,  nrow = 2,dimnames = list(c("r1","r2"), c("c1","c2","c3","c4","c5")))
mat
```

```
##    c1 c2 c3 c4 c5
## r1  1  3  5  7  9
## r2  2  4  6  8 10
```

```r
mat[2,3]
```

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

4.5 Multiplication of Matrix

```
mat1<- matrix(c(2,4,6,8), nrow = 2,ncol =2)
print(mat1)
```

```
##      [,1] [,2]
## [1,]    2    6
## [2,]    4    8
```

```
mat2 <- matrix(c(14,16,18,20), nrow = 2,ncol=2)
print(mat2)
```

```
##      [,1] [,2]
## [1,]   14   18
## [2,]   16   20
```

```
mat1*mat2
```

```
##      [,1] [,2]
## [1,]   28  108
## [2,]   64  160
```

```
mat1%*%mat2
```

```
##      [,1] [,2]
## [1,]  124  156
## [2,]  184  232
```

5 If..else

```
#simplest example
a <- 10
b <- 200
```

```r
if (b > a) {
  print("b is greater than a")
}
```

```
## [1] "b is greater than a"
```

```r
#using else if
a <- 10
b <- 10

if (b > a) {
  print("b is greater than a")
} else if (a == b) {
  print ("a and b are equal")
}
```

```
## [1] "a and b are equal"
```

```r
if (b > a) {
  print("b is greater than a")
} else if (a < b) {
  print("b is smaller than a")
} else {
  print("a and b are equal")
}
```

```
## [1] "a and b are equal"
```

```r
#ifelse
ifelse(b==a, "a and b are equal", "a and b are different")
```

```
## [1] "a and b are equal"
```

6 Loops

Just as other programming languages, you can declare loops to do repetitive works for you. There are two main classes of loop environments: `for` and `while`.

6.1 `for`

`for` executes a set of commands for certain times.

```
nums <- numeric(15)
# create a vector with 15 numeric elements (default at 0).

for(x in 1:10){
  nums[x] = x-1
  #make the xth element of nums x-1
}

nums
```

```
##  [1] 0 1 2 3 4 5 6 7 8 9 0 0 0 0 0
```

```
for (y in 1:length(nums)){
  nums[y] = 15-y
}

nums
```

```
##  [1] 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
```

```
num_matrix <- matrix(nums, nrow = 3, ncol = 5)

#Create transpose matrix with nested for loops
trans_matrix <- matrix(numeric(15), nrow = 5, ncol = 3)
```

```r
for (x in 1:ncol(num_matrix)){
  for (y in 1:nrow(num_matrix)){
    trans_matrix[x,y] <- num_matrix[y,x]
  }
}

trans_matrix
```

```
##      [,1] [,2] [,3]
## [1,]   14   13   12
## [2,]   11   10    9
## [3,]    8    7    6
## [4,]    5    4    3
## [5,]    2    1    0
```

```r
for (x in 1:length(num_matrix)){
  old_v = num_matrix[x]
  if (mod(num_matrix[x],5)==0){
    num_matrix[x] = 100
  }else{
    next #break and escape from the loop
  }
  print(paste("replaced the element ",old_v))
}
```

```
## [1] "replaced the element  10"
## [1] "replaced the element  5"
## [1] "replaced the element  0"
```

```r
num_matrix
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   14   11    8  100    2
```

```
## [2,]   13  100    7    4    1
## [3,]   12    9    6    3  100
```

```
for (x in 1:length(num_matrix)){
  if (num_matrix[x]>=7){
    num_matrix[x] = sqrt(num_matrix[x]-7)
  }else{
    print("I cannot conduct this manipulation: there exists an element in negative domain.")
    break #break and escape from the loop
  }
}
```

```
## [1] "I cannot conduct this manipulation: there exists an element in negative domain."
```

```
num_matrix
```

```
##          [,1]     [,2] [,3] [,4] [,5]
## [1,] 2.645751 2.000000    1  100    2
## [2,] 2.449490 9.643651    0    4    1
## [3,] 2.236068 1.414214    6    3  100
```

6.2 while

`while` is a loop defined by a condition: while the condition is met, the commands are executed; otherwise break the loop. Be very careful that when the condition for `while` is always true, the loop does not stop automatically.

```
truth_v <- TRUE
while_num <- 1

while (truth_v){
  while_num <- while_num + 1
  truth_v <- (while_num<4)
  print(c(while_num, truth_v))
}
```

```
## [1] 2 1
## [1] 3 1
## [1] 4 0
```

```r
# Equivalently,

while_num <- 1

while (while_num<4){
  while_num <- while_num + 1
  truth_v <- (while_num<4)
  print(c(while_num, truth_v))
}
```

```
## [1] 2 1
## [1] 3 1
## [1] 4 0
```

You can also use `next` and `break` for `while`.

7 `apply` Functions

Because of the fundamental data structure of R, it is more efficient to run the for loops with apply function.

```r
# make your matrix a data frame
## Note: You can still use matrix as the input of apply
data<-as.data.frame(num_matrix)

data
```

```r
apply(data,MARGIN=1,FUN=sum)
```

```
## [1] 107.64575  17.09314 112.65028
```

```
apply(data,MARGIN=2,FUN=sum)
```

```
##         V1         V2         V3         V4         V5
##   7.331309  13.057864   7.000000 107.000000 103.000000
```

```
sum <- c()
for (i in 1:ncol(data)){
  sum <- c(sum, sum(data[,i]))
}
sum
```

```
## [1]   7.331309  13.057864   7.000000 107.000000 103.000000
```

You can also use `lapply` to run the apply function on the columns. The “l’ ’ means that the output of `lapply` is a list.

```
max_baby <- lapply(babynames, max)


max_baby
```

```
## $year
## [1] 2017
##
## $sex
## [1] "M"
##
## $name
## [1] "Zzyzx"
##
## $n
## [1] 99686
##
## $prop
## [1] 0.08154561
```

```
max_baby$year
```

```
## [1] 2017
```

8 Functions

You can define your own functions.

```
# Basic syntax: function(argument1,argument2=default value) { commands }

transpose_mat <- function(MAT){
  for (x in 1:ncol(MAT)){
    for (y in 1:nrow(MAT)){
      trans_matrix[x,y] <- MAT[y,x]
    }
  }
  return(trans_matrix) #use return() to specify the output of the function.
}

transpose_mat(num_matrix)
```

```
##           [,1]     [,2]       [,3]
## [1,]   2.645751 2.449490   2.236068
## [2,]   2.000000 9.643651   1.414214
## [3,]   1.000000 0.000000   6.000000
## [4,] 100.000000 4.000000   3.000000
## [5,]   2.000000 1.000000 100.000000
```

```
min_col_mat <- function(MAT,col=1){
  min_n <- min(MAT[,col])
  return(min_n)
}
```

```r
min_col_mat(num_matrix)
```

```
## [1] 2.236068
```

```r
min_col_mat(num_matrix,col=2)
```

```
## [1] 1.414214
```

```r
min_col_mat(num_matrix,3)
```

```
## [1] 0
```

```r
min_max_mat <- function(MAT){
  max_x<-c()
  for (x in 1:ncol(MAT)){
      max_x <- c(max_x, max(MAT[,x]))
  }
  min_max <- min(max_x)
  return(min_max)
}


min_max_mat(num_matrix)
```

```
## [1] 2.645751
```

```r
min_max_mat(transpose_mat(num_matrix))
```

```
## [1] 9.643651
```

Defining functions has a least two obvious advantages. Firstly, it keeps your work space clean; you won't need to create a lot of variables that is only going to be used only once. Secondly, functions can be integrated with dplyr functions and loops.

9 R Markdown

To produce a complete report containing all text, code, and results, click “Knit” or press Cmd/Ctrl + Shift + K.

9.1 Headers

A single hashtag creates a first level header and as the number of hashtags increases, the size of the header decreases.

9.2 Italic and Bold text

Two stars in each side are used for bold texts and one star in each side is used for Italic text. **This is bold** and *This is italic*

9.3 List

Each bullet point begins with one asterisk. Leave a blank line before the first bullet.

- First bullet
    - First sub-bullet
        - sub-sub bullet
- Second bullet

To make a numbered list, use 1 instead of asterisks.

1. First numbered item
2. Second numbered item

9.4 Line breaks

Line ends with
two or more spaces for the end of line.

9.5 Links

Use a plain http address or add a link to a phrase.

http://www.githup.com

Github

9.6 Mathmatical formula

Use dollar signs as in LaTeX.

$$y = x + 10$$

9.7 Showing codes and results in the document

Before we begin, this document uses the following global chunk setting:

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

This means that both the code and its output will be shown for every chunk. You can override this behavior in individual chunks using the options below inside the brackets and after r:

- *Default*: shows codes and results at the same time.

```
mat1
```

```
##      [,1] [,2]
## [1,]    2    6
## [2,]    4    8
```

- *Hiding results*: add the argument eval = FALSE to prevent the code from running.

```
mat1
```

- *Hiding codes*: add the argument echo = FALSE to hide the code but show the output.

```
##      [,1] [,2]
## [1,]    2    6
## [2,]    4    8
```

- *Hiding both code and results*: add the argument include = FALSE to hide everything in the final document.