R Projects for Math 3080

Alvaro Arias

2024-01-22

Contents

Chapter 1

Introduction to R

We start reviewing fundamental concepts of the R language. We assume that you have downloaded and installed RStudio.

Base R refers to the core functionality and built-in functions of the R programming language without any additional packages or libraries. It provides essential data structures (vectors, matrices, data frames, lists), control structures (loops, conditionals), and basic statistical functions.

Libraries, or packages, in R are extensions that add specialized functions and capabilities beyond what's available in base R. They are developed by the R community to address specific tasks or domains. These libraries enhance R's functionality and make it a powerful tool for a wide range of data analysis and statistical tasks.

1.1 Variables and Data Types

In R, variables store data and are assigned using `<-`. There are various types, including numeric (real or complex numbers), character (strings, written between quotes), and logical or boolean (TRUE or FALSE)

```
x <- 2
y <- "I am a string"
z <- TRUE

print(x)
#> [1] 2
print(y)
#> [1] "I am a string"
print(z)
#> [1] TRUE
```

1.2 Logical Operators and Conditionals

The logical operators perform logical operations on boolean values. They include:

- AND that is represented by $\wedge$ in logic and by `&` in R.
- OR that is represented by $\vee$ in logic and by `|` in R.
- NOT that is represented by $\neg$ in logic and by `!` in R.

AND and OR are formally defined by the truth tables below. Note that AND is TRUE only when both values are true, and OR is TRUE when at least one value is true.

p	q	$p \wedge q$	$p \vee q$
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE

NOT is defined in the obvious way: $\neg p$ is FALSE if p is TRUE and it is TRUE if p is FALSE.

In R, `==` is used to check if the objects are equal and `!=` is used to check if the objects aren't equal. For example, `1==2` is FALSE and `1!=2` is TRUE. The comparison symbols `<` and `>` have the same meaning we use in math. Less than or equal $\leq$ is `<=` and greater than or equal $\geq$ is `>=`.

Conditionals in R are structures that allow us to execute different blocks of code based on specified conditions. The main conditional statement is the **if statement**, which can be extended to an **if else statement** and **else if clauses**. They have the following syntax:

```
if (condition) {
  # Code to execute when the condition is TRUE
}
```

```
if (condition) {
  # Code to execute when the condition is TRUE
} else {
  # Code to execute when the condition is FALSE
}
```

```
if (condition1) {
  # Code to execute when condition1 is TRUE
} else if (condition2) {
  # Code to execute when condition1 is FALSE and condition2 is TRUE
} else {
  # Code to execute when both condition1 and condition2 are FALSE
```

```
}
```

1.3 Functions

The functions in R have the following syntax:

```
function_name <- function(arg1, arg2, ...) {
  # Code defining the function's behavior
  return(result)
}
```

The functions f and g below demonstrate simple examples. The function g also illustrates the use of conditionals.

$$f(x) = 2x^2 - 3x - 1$$

$$g(x) = \begin{cases} 2x + 1 & \text{if } x < 0 \\ x^2 & \text{otherwise} \end{cases}$$

```
f <- function(x){
  return(2*x^2-3*x-1)
}

g <- function(x) {
  if (x < 0) {
    result <- 2 * x+1
  } else {
    result <- x^2
  }
  return(result)
}
```

We can evaluate the functions.

```
f(0)
#> [1] -1
g(-2)
#> [1] -3
```

1.4 Vectors and Operations

Vectors are fundamental data structures in R that provide a flexible way to work with and manipulate sequences of values.

In R, vectors are one-dimensional arrays that store elements of the *same data type*. Vectors are often created using the `c( )` function and they can be used to represent situations, like a box with 3 red balls, 2 yellow ones and 1 green.

```
numeric_vector <- c(1,2,3,4,5)
box <- c("red","red","red","yellow","yellow","green")
```

The ***colon*** operator `:` is used to create sequences of numbers with steps of length 1 or -1. The basic syntax is `start:end`. Here are a few examples:

```
1:10
#>  [1]  1  2  3  4  5  6  7  8  9 10
20:12
#> [1] 20 19 18 17 16 15 14 13 12
3:8
#> [1] 3 4 5 6 7 8
```

1.5 Indexing and Subsets

Elements in a vector are accessed using square brackets and indices. ***Indexing starts at 1 in R,*** which is different from most programming languages, where indexing typically starts at 0.

```
x <- box[3]
x
#> [1] "red"
```

We can also select a subset of elements using index vectors or boolean vectors.

```
box[2:4]
#> [1] "red"    "red"    "yellow"
box[c(3,4,6)]
#> [1] "red"    "yellow" "green"
numbers <- 1:6
numbers[c(T,F,T,F,T,F)]
#> [1] 1 3 5
```

1.6 Vectorized Operations

R supports *vectorized operations*, allowing us to perform operations on entire vectors without the need for explicit loops. Recall that `numeric_vector <- c(1,2,3,4,5)`.

```
numeric_vector*(-3)
#> [1]  -3  -6  -9 -12 -15
numeric_vector^3
#> [1]   1   8  27  64 125
numeric_vector+numeric_vector
#> [1]  2  4  6  8 10
```

Vectorized operation also supports logical operations

```
numeric_vector>2
#> [1] FALSE FALSE  TRUE  TRUE  TRUE
box == "yellow"
#> [1] FALSE FALSE FALSE  TRUE  TRUE FALSE
```

We can combine these features to select objects satisfying certain properties

Example 1.1. Suppose that `x<-c(2,1,4,1,5,1,6,2)`. Find the elements that are larger than 2

```
x<-c(2,1,4,1,5,1,6,2)
x
#> [1] 2 1 4 1 5 1 6 2
y <- x>2
y
#> [1] FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
x[y]
#> [1] 4 5 6
```

If we want to know the indices that have values greater than 2, we use the `which()` function

```
which(y)
#> [1] 3 5 7
```

We can select objects using more complicated expressions that involve logical operators.

Example 1.2. Suppose that `x<-c(2,1,4,1,5,1,6,2)`. Find the elements that are larger than 1 and less than 6

```
x
#> [1] 2 1 4 1 5 1 6 2
y <- (x>1)&(x<6) #Recall that & is the AND operator in R
y
#> [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE  TRUE
x[y]
#> [1] 2 4 5 2
```

1.6.1 Vectorization of functions

Since many built-in functions in R can be applied directly to vectors, functions like $f(x) = 2x^2 - 3x - 1$, that only use these operations, can be applied to vectors as well

```
x <- c(-1,0,1,2)
f(x)
#> [1]  4 -1 -2  1
```

However, this does not work with functions that have conditionals, like $g(x) = \begin{cases} 2x+1 & \text{if } x < 0 \\ x^2 & \text{otherwise.} \end{cases}$

```
x <- c(-1,0,1,2)
g(x)
#> Warning in if (x < 0) {: the condition has length > 1 and
#> only the first element will be used
#> [1] -1  1  3  5
```

Notice that g evaluated all values using $2x + 1$.

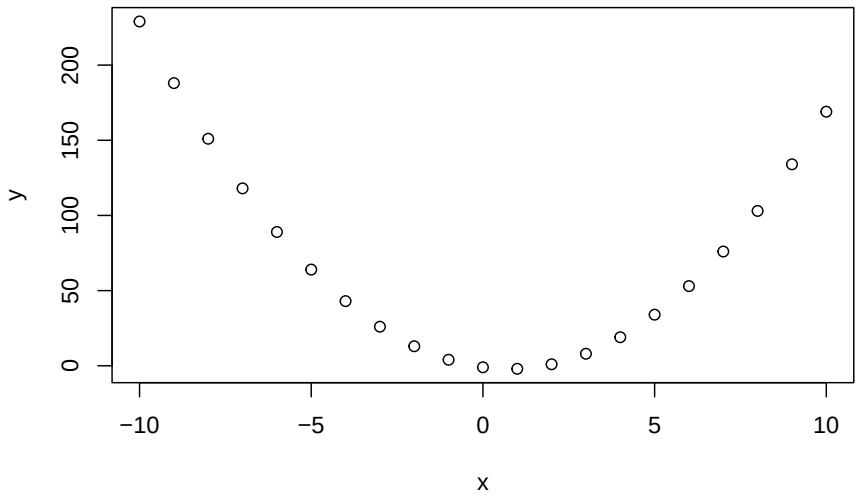
R has several ways to vectorize functions like g. One such solution is the `sapply()` from base R.

```
x <- c(-1,0,1,2)
sapply(x,g)
#> [1] -1  0  1  4
```

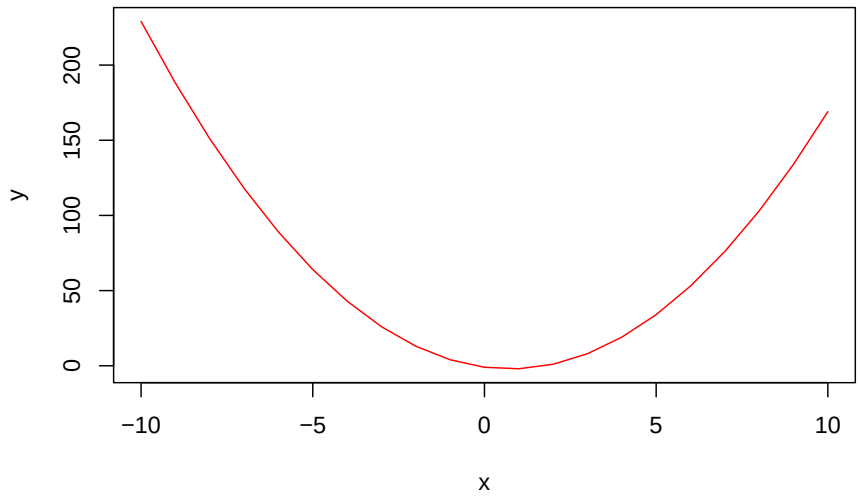
1.6.2 Graphs of Functions

Vectorization in R simplifies the process of generating plots when creating visualizations in R.

```
x <- -10:10
y <- f(x)
plot(x,y)
```



```r
# The optional arguments "type" and "col" allow us to change the graph
plot(x,y,type="l", col="red")
```



Chapter 2

Simulations

R is widely used for simulations due to its extensive set of statistical and mathematical functions, ease of coding, and strong support for random number generation. R's flexibility allows researchers and analysts to implement various simulation models.

In this class we will use the **sample** function and the **replicate** function extensively.

2.1 The sample function

The `sample()` function in R is used to randomly sample elements of a vector. The basic syntax is: for a vector `x`, `sample(x,size,replace=FALSE,prob=NULL)`

The arguments: *size*, *replace* and *prob* are optional. If they are not included, size is the length of the vector, replace is `FALSE` and prob is `NULL`. In this case, we simply get a random permutation of the vector. For example

```
sample(1:6)
#> [1] 1 5 4 2 3 6
```

The sample function can be used to simulate several experiments

Getting objects out of a Box

Suppose that we have a box with 3 red balls, 2 yellow ones and 1 green. We represent the box by the vector

```
box <- c("red","red","red","yellow","yellow","green")
```

1. To randomly select one ball from the box we write `sample(box,1)`

2. To randomly select two balls from the box we write `sample(box,2)`. Notice that this doesn't include replacement. We either take two balls, or we take one ball and then another one.
3. To select two balls with replacement means that we select one ball. We look at it. Put it back in the box and then select a second one at random. To do this we write `sample(box,2,replace=T)`.

Rolling dice

1. To roll a die we write `sample(1:6,1)`
2. To roll two dice we write `sample(1:6,2,replace=TRUE)`. Notice that we need to add `replace=TRUE` because we can get the same number.

Selecting a birthday at random

1. To select a birthday at random we write `sample(1:365,1)`. We ignore leap years and assume that birthdays are described by numbers from 1 to 365. January 1st is 1. January 2nd is 2, etc.
2. To select 10 birthdays at random, we write `sample(1:365,10,replace=T)`. Notice that we need to add the replace parameter since birthdays can be the same.
3. If we want to select 10 **different** birthdays at random, we write `sample(1:365,10)`.

Getting objects out of a Box using the prob Argument

The following is an alternative way to select one ball at random from a box that contains 3 red balls, 2 yellow ones and 1 green.

```
sample(c("red","yellow","green"),1,prob = c(3,2,1))
#> [1] "yellow"
```

The optional *prob* argument is assigned a vector of weights for obtaining the elements of the vector being sampled. They do not need to sum to one.

Simulating a Test with False Positives and False Negatives

Consider a diagnostic test for a specific medical condition. The test identifies individuals with the disease 90% of the time. However, it also wrongly identifies individuals without the disease as positive 8% of the time. This test has a 10% false negative rate and a 8% false positive rate.

Let's simulate this test in R. We start by agreeing that the input is 0 or 1 (0 means that the person doesn't have the disease and 1 means that the person has the disease) and that the output is also 0 and 1 (0 means the test is negative and 1 means the test is positive).

```
test <- function(x){
  if (x==1) {
    value = sample(c(0,1),1,prob = c(0.1,0.9))
  } else {
    value = sample(c(0,1),1,prob = c(0.92,0.08))
  }
  return(value)
}
```

Let's see the results of the test on 40 **healthy** people. We use the `rep( )` function. Notice that the function `test` has conditionals. To vectorize it, we use the `sapply` base R function as was described before

```
healthy <- rep(0,40)
healthy
#>  [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
#> [29] 0 0 0 0 0 0 0 0 0 0 0 0
resultsH <- sapply(healthy, test)
resultsH
#>  [1] 0 0 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
#> [29] 0 0 0 0 0 0 0 0 0 0 0 0
```

And now let's see the results on 40 **sick** people

```
sick <- rep(1,30)
sick
#>  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
#> [29] 1 1
resultsS <- sapply(sick, test)
resultsS
#>  [1] 1 1 0 1 1 1 1 1 1 1 1 1 0 1 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1
#> [29] 1 1
```

2.2 The Replicate Function

The `replicate( )` function in R is used to replicate the execution of an expression or a function multiple times. The basic syntax is `replicate(n,expression)`. For example

1. To roll a die 10 times we write `replicate(10,sample(1:6,1))`. We can also do this using the sample function.
2. To select 1 ball from the box 10 times, we write `replicate(10,sample(box,1))`.

2.3 Simulating Experiments

We combine the sample function, the replicate function and vector functions to simulate more complicated experiments. We illustrate this with these examples:

Example 2.1. Roll a die 20 times and count the number of 6's.

First we roll a die 20 times

```
values <- sample(1:6,20,replace = T)
values
#>  [1] 4 2 5 3 1 3 4 3 2 2 5 3 3 6 3 6 4 6 2 1
```

Then we check the values that are equal to 6

```
values == 6
#>  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#> [10] FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE
#> [19] FALSE FALSE
```

R treats `FALSE` values as 0's and `TRUE` values as 1's. If we sum these values we find out the number of 6's

```
sum(values == 6)
#> [1] 3
```

Example 2.2. Run the previous experiment 100 times and find the average.

We use the replicate function to run the experiment 100 times and then we use the `mean()` function to find the average

```
exper <-replicate(100,sum(sample(1:6,20,replace=T) == 6))
exper
#>   [1] 3 2 2 4 6 2 0 5 5 1 4 4 5 2 2 1 4 3 2 6 2 2 2 3 0 1 6
#>  [28] 5 3 5 3 1 5 4 4 3 2 2 4 2 0 4 1 2 3 4 7 3 4 1 3 4 2 2
#>  [55] 4 3 4 5 3 4 5 4 5 3 5 2 3 1 2 5 2 3 4 2 5 2 2 3 2 3 2
#>  [82] 3 0 5 2 4 2 3 4 3 5 3 3 2 3 1 5 3 5 4
mean(exper)
#> [1] 3.1
```

To simplifying reading the code or to handle more complex situations, we can wrap the expression in curly braces to compute it.

```
replicate(100,{
  values <- sample(1:6,20,replace=T)
  sum(values == 6)
})
#>   [1] 3 4 2 1 4 2 3 0 5 4 2 5 4 3 3 3 3 3 2 6 3 3 7 5 2 2
#>  [28] 4 5 4 3 4 3 2 2 3 0 5 5 2 6 2 2 3 3 4 0 5 5 2 2 1 4 4
#>  [55] 6 2 3 6 3 2 2 1 4 3 3 2 1 1 2 2 2 4 1 5 3 2 4 5 1 4 5
#>  [82] 4 5 1 5 3 7 2 8 4 3 4 6 3 2 3 4 6 1 3
```

By default, the function replicates the value of the last line

```
replicate(100,{
  values <- sample(1:6,20,replace=T)
  sum(values == 6)
  8
})
#>   [1] 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
#>  [28] 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
#>  [55] 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
#>  [82] 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
```

Chapter 3

Project 1

Use R to simulate the following:

1. Roll a dice 4 times and find the sum
2. Select 4 different numbers between 1 and 100. Find the maximum value, the minimum value, and the difference.
3. Roll 3 dice 100 times. Find how many times the sum of the dice is at least 10.
4. Permute the letters of the word SIMULATE 100 times and find out how many times the first letter is an S and the last letter is an E. That is, we are looking for words of the form S******E.
5. Repeat the previous problem but find out how many times the first letter is a vowel.
6. A box has 4 red balls and 6 blue balls. Select 5 balls 10000 times and find out how many times we get exactly 3 red balls.
7. In a popular carnival game, players roll three dice. If at least one of them is a 6, they win \$1; otherwise, they lose \$1. They argue that this is a fair game. Each die has a 1/6 chance of being a 6, and the probability of having at least one 6 in three dice rolls is $1/6 + 1/6 + 1/6 = 1/2$. Simulate the game in R. Play it 1000 times and find out how much money you won or lost.

Chapter 4

Introduction to Data Frame

4.1 Data Frames

Data frames in R are two-dimensional data structures, similar to spreadsheets. They are used to store and manipulate datasets. The key points are:

1. *Structure*: A data frame is a collection of vectors of equal length, where each vector represents a column. Recall that all the elements of a vectors have the same type.

2. *Columns*: Different columns can have different data types (numeric, character, factor, etc.).

3. *Rows*: Each row corresponds to a separate observation or record in the dataset.

4.1.1 Creating Data Frames

Data frames are created using the `data.frame()` function or by importing data sources.

One option is to define the column vectors and to combine them into a data frame using the `data.frame()` function as shown in this example:

```
name = c("John Smith","Jane Doe", "Mary Johnson")
a <- c(NaN,16,3)
b<- c(2,11,1)
df1 <- data.frame(name,a,b)
knitr::kable(df1)
```

name	a	b
John Smith	NaN	2
Jane Doe	16	11
Mary Johnson	3	1

We used the `knitr::kable()` function to have a nicer display.

Data frames can also be defined inside the data.frame() function directly, naming the columns and the values.

```
df2 <- data.frame(
  name = c("John Smith","Jane Doe","Mary Johnson","John Smith","Jane Doe","Mary Johnso
  treatment =c("a","a","a","b","b","b"),
  results = c(NaN,16,3,2,11,1)
)
knitr::kable(df2)
```

name	treatment	results
John Smith	a	NaN
Jane Doe	a	16
Mary Johnson	a	3
John Smith	b	2
Jane Doe	b	11
Mary Johnson	b	1

Notice that the data frames df1 and df2 have the same information. We will later learn that df2 is preferred over df1. We will learn how to convert a data frame like df1 into a data frame like df2. df2 has a "tidy" format that fits specially well with many libraries of R.

Importing data sets from R, head(), and tail()

R has several data sets that can be imported using the `data()` function. If you write data() inside an R block and run it, you will see a list of all data sets.

For illustration, we will import the iris data set that has 150 rows. We will use the `head()` function to see only the first 6 rows. The `tail()` function is similar, but it shows the last 6 rows by default.

```
data(iris)
head(iris)
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1          5.1         3.5          1.4         0.2  setosa
#> 2          4.9         3.0          1.4         0.2  setosa
#> 3          4.7         3.2          1.3         0.2  setosa
#> 4          4.6         3.1          1.5         0.2  setosa
#> 5          5.0         3.6          1.4         0.2  setosa
#> 6          5.4         3.9          1.7         0.4  setosa
```

Extracting columns from a data frame

The syntax to extract columns is data_frame$column_name. Recall that the columns of a data frame are vectors.

```
iris$Sepal.Length
#>   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8
#>  [14] 4.3 5.8 5.7 5.4 5.1 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0
#>  [27] 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0 5.5 4.9 4.4
#>  [40] 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4
#>  [53] 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6
#>  [66] 6.7 5.6 5.8 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7
#>  [79] 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5 5.5
#>  [92] 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3
#> [105] 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5
#> [118] 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2
#> [131] 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8
#> [144] 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

Some Data Frame Functions

The following is a list of some of the most useful data frame functions. In the next section we will import the library dplyr. This library handles functions 9-14 in a nicer way.

1. head(): Displays the first few rows of a data frame.
2. tail(): Displays the last few rows of a data frame.
3. str(): Shows the structure of a data frame.
4. summary(): Provides summary statistics for each column.
5. nrow(): Returns the number of rows in a data frame.
6. ncol(): Returns the number of columns in a data frame.
7. colnames(): Returns or sets the column names.
8. rownames(): Returns or sets the row names.
9. subset(): Subsets a data frame based on conditions. subset(df, condition)
10. select(): Chooses specific columns. select(df, column1, column2)
11. filter(): Filters rows based on conditions. filter(df, condition)
12. mutate(): Adds new variables or modifies existing ones. mutate(df, new_column = expression)
13. arrange(): Sorts rows based on one or more columns. arrange(df, column1, column2)
14. merge(): Combines two data frames by common columns. merge(df1, df2, by = "common_column")

We will illustrate some of these functions with the data frame iris.

```
str(iris) #str shows the structure of the data set.
#> 'data.frame':    150 obs. of  5 variables:
#>  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
```

```
#>  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
#>  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
#>  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
#>  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 .
```

```
summary(iris) # summary provides summary statistical for each column.
#>   Sepal.Length    Sepal.Width     Petal.Length
#>  Min.   :4.300   Min.   :2.000   Min.   :1.000
#>  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600
#>  Median :5.800   Median :3.000   Median :4.350
#>  Mean   :5.843   Mean   :3.057   Mean   :3.758
#>  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100
#>  Max.   :7.900   Max.   :4.400   Max.   :6.900
#>   Petal.Width          Species
#>  Min.   :0.100   setosa    :50
#>  1st Qu.:0.300   versicolor:50
#>  Median :1.300   virginica :50
#>  Mean   :1.199
#>  3rd Qu.:1.800
#>  Max.   :2.500
```

```
nrow(iris) # nrwon finds the number of rows
#> [1] 150
```

```
colnames(iris) # colnames finds the names of the columns
#> [1] "Sepal.Length" "Sepal.Width"  "Petal.Length"
#> [4] "Petal.Width"  "Species"
```

Chapter 5

DPLYR and Conditional Probability

dplyr is a popular R library used to manipulate data consistently and efficiently. It uses a set of functions, that they call "verbs". These functions are used to solve the most common data manipulation problems. The main verbs are:

- *mutate* to create new columns. The syntax is

```
mutate(data_frame, new_column_name = expression)
```

- *filter* to select rows. The syntax is

```
filter(data_frame,condition)
```

- *select* to select columns. The syntax is

```
select(data_frame, column1,column2,...,columnk)
```

- *arrange* reorders the rows. The syntax is

```
arange(data_frame,column1,columns2,...)
```

These functions are combined with group_by() and summarise() to easily perform group-wise operations and compute summary statistics within those groups.

To use a library we type library(name_of_library). If the library is not intalled, we install it through Packages in RStudio.

5.1 Conditional Probability - dice

Example 5.1. You roll two dice. Estimate the conditional probability that the first die is 6 if the sum is 10

Step 1: Select a large number of simulations and generate the dice data.

```
n <- 10000
die1 <- sample(1:6,n,replace = T)
die2 <- sample(1:6,n,replace = T)
```

Step 2: Make the data into a dataframe.

```
df <- data.frame(die1,die2)
head(df)
#>   die1 die2
#> 1    6    6
#> 2    2    5
#> 3    4    3
#> 4    1    1
#> 5    5    6
#> 6    2    2
```

Step 3: Use mutate to create a new column that includes the sum.

```
dfm <- mutate(df,sum = die1+die2)
head(dfm)
#>   die1 die2 sum
#> 1    6    6  12
#> 2    2    5   7
#> 3    4    3   7
#> 4    1    1   2
#> 5    5    6  11
#> 6    2    2   4
```

Step 4: Select the rows that add up to ten. This is the key point of conditional probability.

```
dff <- filter(dfm,sum == 10)
head(dff)
#>   die1 die2 sum
#> 1    5    5  10
#> 2    4    6  10
#> 3    6    4  10
#> 4    4    6  10
#> 5    6    4  10
#> 6    4    6  10
```

Step 5: Extract the column that has the first die, and find the proportion of sixes.

```
mean(dff$die1 == 6)
#> [1] 0.3277108
```

5.2 The Pipe Operator (%>%) in DPLYR

Saving new dataframes every time we perform new operations is tedious and it can be confusing. In dplyr, the pipe operator (%>%) passes the result of one operation as the first argument to another operation. This improves code readability by creating a concise syntax that chains multiple operations together.

For example, these four lines reproduce the previous operations and it adds a new column named ***experiment*** that checks if die1 is equal to 6.

```
df %>%
  mutate(sum = die1+die2) %>%
  filter(sum == 10) %>%
  mutate(experiment = die1 == 6) %>%
  head()
#>   die1 die2 sum experiment
#> 1    5    5  10      FALSE
#> 2    4    6  10      FALSE
#> 3    6    4  10       TRUE
#> 4    4    6  10      FALSE
#> 5    6    4  10       TRUE
#> 6    4    6  10      FALSE
```

5.3 Conditional Probability - test for disease

Example 5.2. A new test was developed for a disease that only 1% of the population has. The test identifies individuals with the disease 90% of the time. However, it also wrongly identifies individuals without the disease as positive 5% of the time. Estimate the probability that a person who tests positive for the disease actually has the disease.

Notice that this test has a 10% false negative rate and a 5% false positive rate.

We first choose a large number of simulations and then proceed to randomly assign individuals. Each person is assigned a value of 1 with a probability of 1%, indicating the presence of the disease, or 0 with a probability of 99%, signifying the absence of the disease.

```
n <- 100000
people <- sample(c(0,1),n,replace = T,prob = c(0.99,0.01))
```

Then we write the function that simulates the test for the disease.

```
test <- function(x){
  if (x==1) {
    value = sample(c(0,1),1,prob = c(0.1,0.9))
  } else {
    value = sample(c(0,1),1,prob = c(0.95,0.05))
```

```
  }
  return(value)
}
```

Now we apply the test to each person. Since the function has conditionals, we use the sapply() function.

```
results <- sapply(people, test)
```

```
df <- data.frame(people,results)
head(df)
#>   people results
#> 1      0       0
#> 2      0       0
#> 3      0       0
#> 4      0       0
#> 5      0       0
#> 6      0       0
```

Now we select the people who tested positive, and we use the base R function ***table*** to find out how many of them have the disease and how many don't.

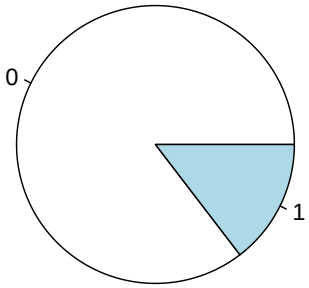
The function ***table*** counts the occurrences of each unique value in a given vector.

```
positive <- filter(df,results==1)
head(positive)
#>   people results
#> 1      0       1
#> 2      0       1
#> 3      0       1
#> 4      0       1
#> 5      0       1
#> 6      1       1
```

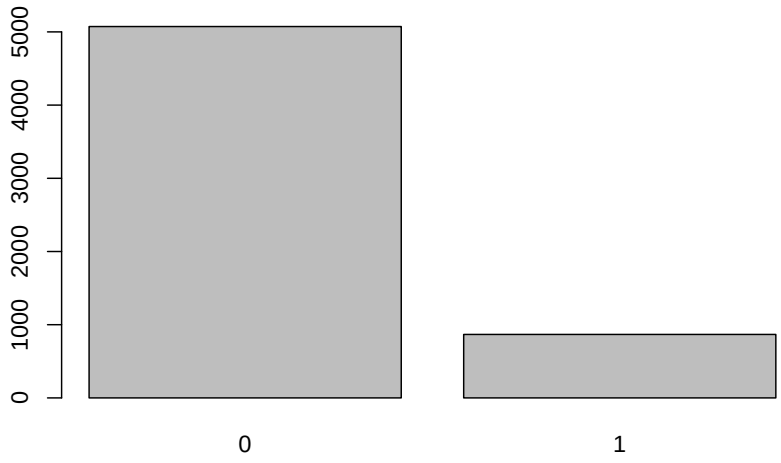
```
table(positive$people)
#>
#>    0    1
#> 5073  867
```

The output of the table function can be plot directly as a pie plot or as a bar plot.

```
pie(table(positive$people))
```

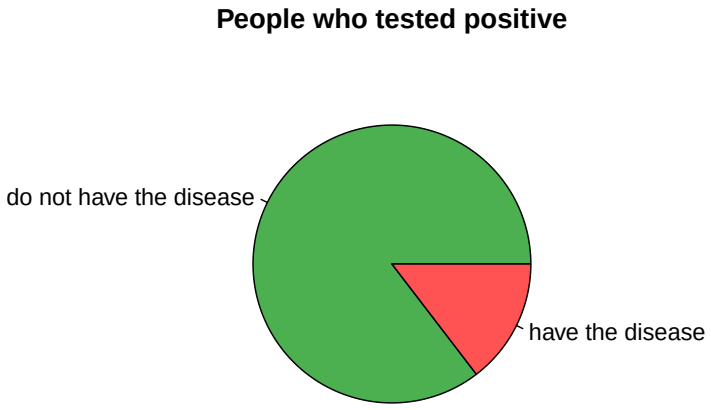



```
barplot(table(positive$people))
```

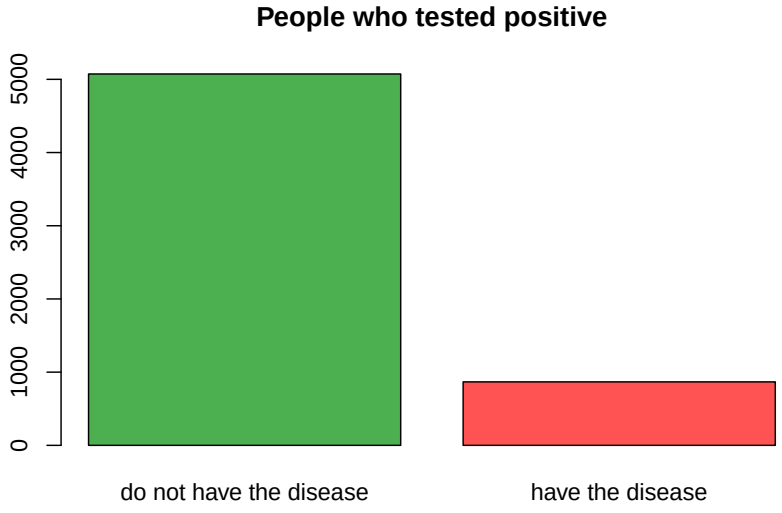


We can easily add titles and make the graphs more informative

```
pie(table(positive$people),main="People who tested positive", labels=c("do not have the
```




```
barplot(table(positive$people),main="People who tested positive", names.arg = c("do not
```



Chapter 6

Row Wise Operations

In R and in dplyr it is easier to perform column operations than row operations. We will illustrate this with the following data frame:

```
df
#>   x y  z
#> 1 5 1  9
#> 2 6 2 10
#> 3 7 3 11
#> 4 8 4 12
```

If we add the columns using the variables we get the right result.

```
mutate(df,sum = x+y+z)
#>   x y  z sum
#> 1 5 1  9  15
#> 2 6 2 10  18
#> 3 7 3 11  21
#> 4 8 4 12  24
```

On the other hand, if we use vectorized operators (i.e., we convert ‘`x,y,z` into a vector and add it), we get the wrong answer. R (or dplyr) computes the sum accross all the rows:

```
mutate(df, sum = sum(c(x,y,z)))
#>   x y  z sum
#> 1 5 1  9  78
#> 2 6 2 10  78
#> 3 7 3 11  78
#> 4 8 4 12  78
```

Something similar happens if we use functions.

The following block has four functions. We use the first two to compute the sum of the columns and the last two to select an element from each row. The first function computes the sum adding the variables. The second one vectorizes the variables and adds them. The third function vectorizes the variables and selects one at random, and the last one avoids vectorizing the variables with if then statements.

```
suma1 <- function(a,b,c){
  return(a+b+c)
}

suma2 <- function(vec){
  return(sum(vec))
}

choose_one1 <- function(a,b,c){
  return(sample(c(a,b,c),1))
}

choose_one2 <- function(a,b,c){
  n <- sample(1:3,1)
  if (n==1){
    return(a)
  } else if (n==2){
    return(b)
  } else {
    return(c)
  }
}
```

Then we add a column for each function

```
mutate(df,sum1 = suma1(x,y,z), sum2 = suma2(c(x,y,z)), choose_one1 = choose_one1(x,y,z
#>   x y  z sum1 sum2 choose_one1 choose_one2
#> 1 5 1  9   15   78           8           1
#> 2 6 2 10   18   78           8           2
#> 3 7 3 11   21   78           8           3
#> 4 8 4 12   24   78           8           4
```

As you see, the first sum works well but he second one doesn't. The other functions don't work well either. ***choose_one1*** selected one element from all the rows. ***choose_one2*** selected different elements, but all were chosen from the same column.

6.1 rowwise()

To perform row operations properly, we "group" the data across each row, using the function `rowwise()`. We illustrate it's use with the previous data frame and using the pipe operator. We start from the data frame, then we group the data frame by rows, and finally we add the four columns using the previous functions.

```
df %>%
  rowwise() %>%
  mutate(sum1 = suma1(x,y,z),sum2=suma2(c(x,y,z)),choose_one1 = choose_one1(x,y,z), choose_one2 =
#> # A tibble: 4 x 7
#> # Rowwise:
#>       x     y     z  sum1  sum2 choose_one1 choose_one2
#>   <dbl> <dbl> <dbl> <dbl> <dbl>       <dbl>       <dbl>
#> 1     5     1     9    15    15           9           9
#> 2     6     2    10    18    18           6          10
#> 3     7     3    11    21    21           3           3
#> 4     8     4    12    24    24           4           4
```

Notice that in this case, all the operations worked well. The last two columns are different because they were selected randomly. Nevertheless, each of them selects one element from each row.

6.2 ungroup()

The `rowwise()` operator does not change the data, but it groups it. If we look carefully at the output, we see that the table says "# Rowwise" to let us know that the data is grouped by rows. To remove the grouping, we add the `ungroup()` operator.

```
df %>%
  rowwise() %>%
  mutate(sum1 = suma1(x,y,z),sum2=suma2(c(x,y,z)),choose_one1 = choose_one1(x,y,z), choose_one2 =
  ungroup()
#> # A tibble: 4 x 7
#>       x     y     z  sum1  sum2 choose_one1 choose_one2
#>   <dbl> <dbl> <dbl> <dbl> <dbl>       <dbl>       <dbl>
#> 1     5     1     9    15    15           1           1
#> 2     6     2    10    18    18           2          10
#> 3     7     3    11    21    21           3           7
#> 4     8     4    12    24    24           4          12
```

Now the output of the table doesn't include "# Rowwise".

Chapter 7

Project 2

Use R to simulate the following problems. Use the conditional probability simulation examples as guide. 5.1 and 5.3

Problem 1

A cab was involved in a hit and run accident at night. Two cab companies, the Green and the Blue, operate in the city. You are given the following data:

- 85% of the cabs in the city are Green and 15% are Blue.
- A witness identified the cab as Blue.
- The court tested the reliability of the witness under the same circumstances that existed on the night of the accident and concluded that the witness correctly identified each one of the two colors 80% of the time and failed 20% of the time.

What is the probability that the cab involved in the accident was Blue rather than Green?

The next problems are variants of the boy-girl paradox. For each of them simulate a large number of families with two children using a data frame with two columns. The first column is the older kid (no twins) and the second one the younger one. Each row represents the children of the family. Each child has an equal chance of being either a boy or a girl.

Problem 2

Consider a family with two children. Given that the older one is a boy, what is the probability that both children are boys?

This problem has no ambiguity.

Problem 3

Consider a family with two children. Given that one of the children is a boy, what is the probability that both children are boys?

This problem can be ambiguous. How do we know that at least one of the children is a boy?

If we do know the gender of the children of all families, we get one answer. We select the families that have at least one boy and then check how many of them have two boys. Do this for this problem.

On the other hand, we may not know this for every family. Consider the following case:

Problem 4

Mr. Smith is the father of two. We meet him walking along the street with a young boy whom he proudly introduces as his son. What is the probability that Mr. Smith's other child is also a boy?

In this case, we know that Mr. Smith has at least one boy but the situation is different.

For this problem, assume that Mr. Smith was equally likely to select one of his two children to walk with him. Therefore, define a new column named **walking** that selects one of the two children in the family at random. To do this use the `rowwise()` operator described before. Then select the rows that have a "boy" in the **walking** column and check how many of these rows have two boys.

A variant of Problem 2 that makes the paradox more clear is: *Mr. Smith is the father of two. We meet him walking along the street with a young boy whom he proudly introduces as his older child. What is the probability that Mr. Smith's other child is also a boy?*