R Basics

Cecilia Sui and all other TAs

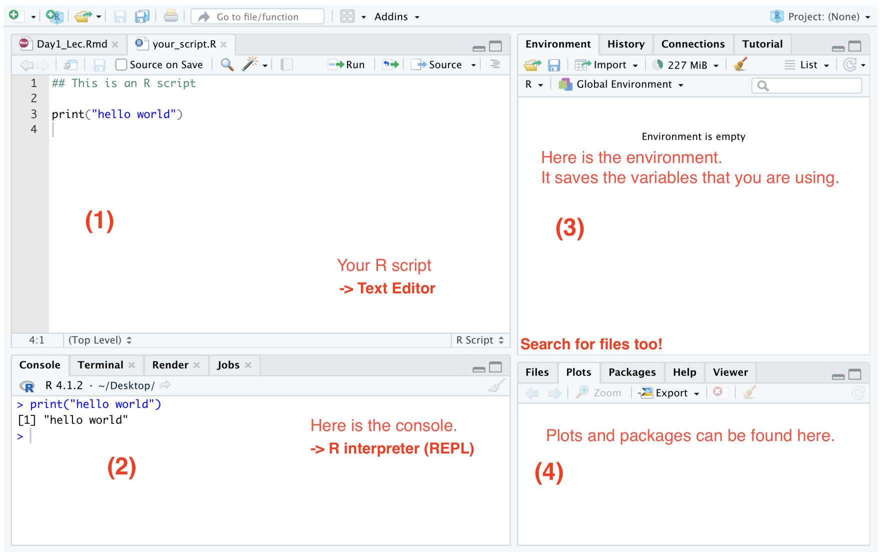
Contents

1 Why should we learn R?

- R is the language of data science both in academia and industry. (most commonly used)
- R is free and open-source.
- R is optimized for vector operations. (It means that you can easily go through an entire row or an entire table of data without having to write explicitly for loops.)
- R has over 10,000+ contributing packages. This makes it possible to do almost everything you need. (https://cran.r-project.org/web/packages/available_packages_by_name.html)
- R has a great community of programmers. (If you ever encounter an error or warning in your program, you will likely find solutions online just by copying and pasting your error messages into Google search bar.)

2 The Basics

This section provides a broad overview of the R language that will get you programming right away. Don't worry if you have never programmed before. This section will teach you everything you need to know to get started.



```
## This is an R script

print("hello world")
```

```
R 4.1.2 · ~/Desktop/
> print("hello world")
[1] "hello world"
>
```

Figure 1: R Studio Overview

2.1 The R Studio User Interface

R is a programming language that you can use to communicate with your computer, and RStudio is an integrated development environment (IDE) designed for R.

The RStudio interface is simple. There are **4 panes**:

1. The upper left pane (1) is a **text editor**

- You can write down your code here.

2. The lower left pane (2) is a **console**

- This is where R interpreter resides.
- You can execute lines of code here.
- For example, if you type 1 + 1 and hit Enter, RStudio will display:

```r
1 + 1
```

```
## [1] 2
```

3. The upper right pane (3) is an **environment**

- Summarizes global (by default) variables that you have created / imported.
- Take a glimpse of each variable by clicking the arrow button on the left.

4. The lower right pane (4) is a **browser**

- Files
- Plots
- Packages
- Help documents

2.2 R Console

Let's first play with R Console. As mentioned above R console is the R itself and we can execute code by typing on this console screen directly. The normal prompt > denotes that R is ready for the new input.

2.3 R as a calculator: Basic numerical operations

Once you get the hang of the command line, you can easily do anything in R that you would do with a calculator. For example, you could do some basic "arithmetic":

```
2 * 3 # asterisk
```

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

```
4 - 1 # subtraction
```

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

```
6 / 4 # slash
```

```
## [1] 1.5
```

```
2 ^ 3 # caret
```

```
## [1] 8
```

```
2 ** 3
```

```
## [1] 8
```

```
# '^'(2,3)

3 + 3 * 20
```

```
## [1] 63
```

```
(3+3) * 20
```

```
## [1] 120
```

R follows the order of operations, where precedence follows the BEDMAS order: Brackets `()`, Exponents `^`, Division `/` and Multiplication `*`, Addition `+` and Subtraction `-`.

2.3.1 Bracketed Numbers in Console Output

You might notice that a [1] appears next to your results. R is just letting you know that this line begins with the first value in your results. Some commands return more than one value, and their results may fill up multiple lines. For example, the command 100:130 returns 31 values; it creates a sequence of integers from 100 to 130. Notice that new bracketed numbers appear at the start of the first and second lines of output. These numbers just mean that the first line begins with the 1st value in the result, and the second line begins with the 20th value.

You can mostly ignore the numbers that appear in brackets:

```
# The "colon operator (:)" returns every integer between two integers.
# It is an easy way to create a sequence of numbers.
100:130
```

```
##  [1] 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118
## [20] 119 120 121 122 123 124 125 126 127 128 129 130
```

2.3.2 Incomplete Commands

If you type an incomplete command and press Enter, R will display a `+` prompt, which means R is waiting for you to type the rest of your command. Either finish the command or hit Escape to start over:

console example

```
> 5 -
+
+ 1
[1] 4
```

2.3.3 Errors and Warnings

- When R recognizes **Errors**, then it **stops** executing the code and will return an error message.
- Don't panic, most times the fixes are easy. Unfortunately, we cannot cover all the errors you will encounter in this course, but here is an example.

```
> 3 % 5
Error: unexpected input in "3 % 5"
>
# Here the user is trying to do modulus in R, but fails to use the correct
# operator for modulus %%.
```

- When you encounter **Warnings**, R usually finishes execution, but she wants to tell you that there are somethings you might want to take a look. Here is an example.

```
setwd("~/")
Warning message:
In in_dir(input_dir(), expr) :
  You changed the working directory to ~/ (probably via setwd()). It will be restored to the default wo
```

- The error comes out because in R markdown, `setwd` only works for a specific code chunk. So the warning says, even if we use it, the working directory for the other code chunks will remain to be the default (= where the R markdown file is).

2.4 Comments in R

- Comments are like helping text in your R program and they are ignored by the interpreter while executing your actual program, i.e. comments will not be run or executed.
- A single comment is written using `#` in the beginning of the statement.
- There is no multiline comments support.

```
# This is a comment.
## This is also a comment.
### This is still a comment.
### Comment again. #######


# line 1
# line 2
```

- Multiple Comments: Block select the lines and then,
  - On Windows and Linux, Ctrl + Shift + C.
  - On macOS, Cmd + Shift + C.

2.5 In-class exercises:

That's the basic interface for executing R code in RStudio. Let's try doing these simple tasks. If you execute everything correctly, you should end up with the same number that you started with:

1. Choose any number and add 2 to it.
2. Multiply the result by 3.
3. Subtract 6 from the answer.

4. Divide what you get by 3.

```
3 + 2
```

```
## [1] 5
```

```
5 * 3
```

```
## [1] 15
```

```
15 - 6
```

```
## [1] 9
```

```
9 / 3
```

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

```
6 %% 4 # modulus
```

```
## [1] 2
```

```
6 %/% 4 # quotient
```

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

3 R Objects

In this section, we will learn about different R objects by using it to assemble a deck of 52 playing cards.

We will start by building simple R objects that represent playing cards and then work our way up to a full-blown table of data. In short, we will build the equivalent of an Excel spreadsheet from scratch. When we are finished, our deck of cards will look something like this:"

```
 face   suit value
 king spades    13
queen spades    12
 jack spades    11
  ten spades    10
 nine spades     9
eight spades     8
...
```

This exercise will teach you how R stores data, and how you can assemble or disassemble your own data sets. You will also learn about the various types of objects available for you to use in R (not all R objects are the same!).

We will start with the very basics. The most simple type of object in R is an atomic vector. Atomic vectors are not nuclear powered, but they are very simple and they do show up everywhere. If you look closely enough, you will see that most structures in R are built from atomic vectors.

3.1 Creating an Object

Until now, we only deal with the numbers by itself; if we want `3 + 4`, we typed it explicitly. However, it would be great if we can give numbers some names that can carry meanings. We can achieve this by creating and assigning values to objects / variables. Let's start by creating `a` that has value of `3`.

```
a <- 3
print(a)
```

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

```r
print(class(a))
```

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

In R, you can assign values by using two operators: `<-` and `=`. However, `=` is usually reserved to be used inside functions. You can use `=` to assign values and usually this does not create any problems, but `<-` is the standard way.

Also, as you have noticed, R is smart enough to detect the type of the data, so you do not need to specify it. Speaking of data types, let's dive into them.

3.2 Five Basic Data Types

There are **5** major data types in R. That said, most data you encounter will fall into these five categories. You can check the type of any object using the `class()` function.

3.2.1 Numeric (Double)

This is the default for any number. This includes decimals (floats).

```r
weight <- 70.25
print(weight)
```

```
## [1] 70.25
```

```r
print(class(weight))
```

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

3.2.2 Integer

Whole numbers are contained as integer. In R, you must add `L` suffix to force numbers to be integer. Without `L` suffix, R will treat whole numbers as numeric

```r
# Naive assignment - Numeric
height_num <- 6
print(class(height_num))
```

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

```r
# Force it to be integer
height_int <- 6L
print(class(height_int))
```

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

3.2.3 Character (Strings)

Text data is stored in Character type. Characters and Strings should always be wrapped by "double quotes" or 'single quotes'. It does not matter which quotes you use, but use it consistently. Many researchers seems to prefer double quotes.

```r
city <- "St. Louis"
print(class(city))
```

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

```
city <- 'St. Louis'
print(class(city))
```

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

3.2.4 Logical (Boolean)

There are two values: `TRUE` and `FALSE`. `TRUE` has numeric value of 1 and `FALSE` has numeric value of 0. You can use `T` and `F` instead, but it is always recommended to use full `TRUE` and `FALSE`.

```
print(city == "New York")
```

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

```
print(city == "St. Louis")
```

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

```
isNY <- city == "New York"
isSTL <- city == "St. Louis"

print(isNY + isSTL)
```

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

3.2.5 Factor (categorical values)

Special types used to represent "categories" (e.g., "High", "Medium", "Low" or "Male", "Female"). They look like strings but are stored as integers with labels.

```
gender <- factor(c("Male", "Female", "Female"))
class(gender) # Returns "factor"
```

```
## [1] "factor"
```

```
levels(gender) # Shows the unique categories
```

```
## [1] "Female" "Male"
```

3.3 Data Structures

3.3.1 Vector

Vector is the most import data structure in R. R is specialized in vector level operations, so if you need calculations with a large set of numbers, transform it into a vector may speed things up.

Vector is a *list* of elements of the *same type*. The basic way to create it is using `c()` function.

```
# A numeric vector
heights <- c(150, 162, 180, 155)

# A character vector
colors <- c("red", "blue", "green")
```

```r
# IMPORTANT: If you mix types, R will "coerce" them to the most flexible type (usually character)
mixed <- c(1, "apple", TRUE)
class(mixed)
```

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

3.3.2 List

Unlike vectors, lists do not group together individual values; **lists group together R objects**, such as atomic vectors and other lists. For example, you can make a list that contains a numeric vector of length 31 in its first element, a character vector of length 1 in its second element, and a new list of length 2 in its third element.

To do this, use the list function. `list()` creates a list the same way `c()` creates a vector. Separate each element in the list with a comma:

```r
list1 <- list(100:130, "R", list(TRUE, FALSE))
list1
```

```
## [[1]]
##  [1] 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118
## [20] 119 120 121 122 123 124 125 126 127 128 129 130
##
## [[2]]
## [1] "R"
##
## [[3]]
## [[3]][[1]]
## [1] TRUE
##
## [[3]][[2]]
## [1] FALSE
```

```r
list1[[3]][2]
```

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

```r
# list1[[1]]
# list1[[2]][2] # NA
```

The **double-bracketed indices** tell you which *element* of the list is being displayed. The **single-bracket indices** tell you which *sub-element* of an element is being displayed.

For example, [[1]] [1] 100 is the first subelement of the first element in the list. [[2]] [1] "R" is the first sub-element of the second element in the list. This two-system notation arises because each element of a list can be any R object, including a new vector (or list) with its own indices.

Lists are a basic type of object in R, on par with atomic vectors. Like atomic vectors, they are used as building blocks to create many more sophisticated types of R objects.

As you can imagine, the structure of lists can become quite complicated, but this flexibility makes lists a useful all-purpose storage tool in R: you can group together anything with a list.

3.4 In-class exercises:

1. Use a list to store a single playing card, like the ace of hearts, which has a point value of one. The list should save the face of the card, the suit, and the point value in separate elements.

```r
card <- list("ace", "hearts", 1)
card
```

```
## [[1]]
## [1] "ace"
##
## [[2]]
## [1] "hearts"
##
## [[3]]
## [1] 1
```

3.4.1 Dataframe

A data frame is a collection of vectors of the **same length**. This is how 99% of data analysis is done. They are far and away the most useful storage structure for "data analysis", and they provide an ideal way to store an entire deck of cards. You can think of a data frame as R's equivalent to the Excel spreadsheet because it stores data in a similar format.

Data frames group vectors together into a "two-dimensional table". Each vector becomes a column in the table. As a result, each column of a data frame can contain a different type of data; but within a column, every cell must be the same type of data.

Creating a data frame by hand takes a lot of typing, but you can do it (if you like) with the data.frame function. Give data.frame any number of vectors, each separated with a comma. Each vector should be set equal to a name that describes the vector. data.frame will turn each vector into a column of the new data frame:

```r
df <- data.frame(HappyFace = c("ace", "two", "six"),
                 Suit = c("clubs", "clubs", "clubs"),
                 Value = c(1, 2, 3))
print(df)
```

```
##   HappyFace  Suit Value
## 1       ace clubs     1
## 2       two clubs     2
## 3       six clubs     3
```

You'll need to make sure that each vector is the same length (or can be made so with R's recycling rules) as data frames cannot combine columns of different lengths.

If you look at the type of a data frame, you will see that it is a list. In fact, each data frame is a list with class data.frame. You can see what types of objects are grouped together by a list (or data frame) with the `str()` function:

```r
typeof(df)
```

```
## [1] "list"
```

```r
class(df)
```

```
## [1] "data.frame"
```

```r
str(df) #structure
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ HappyFace: chr  "ace" "two" "six"
##  $ Suit     : chr  "clubs" "clubs" "clubs"
##  $ Value    : num  1 2 3
```

```r
# ?str
```

You could create this data frame with data.frame, but look at the typing involved! You need to write three vectors, each with 52 elements:

```r
deck <- data.frame(
  face = c("king", "queen", "jack", "ten", "nine", "eight", "seven", "six",
    "five", "four", "three", "two", "ace", "king", "queen", "jack", "ten",
    "nine", "eight", "seven", "six", "five", "four", "three", "two", "ace",
    "king", "queen", "jack", "ten", "nine", "eight", "seven", "six", "five",
    "four", "three", "two", "ace", "king", "queen", "jack", "ten", "nine",
    "eight", "seven", "six", "five", "four", "three", "two", "ace"),
  suit = c("spades", "spades", "spades", "spades", "spades", "spades",
    "spades", "spades", "spades", "spades", "spades", "spades", "spades",
    "clubs", "clubs", "clubs", "clubs", "clubs", "clubs", "clubs", "clubs",
    "clubs", "clubs", "clubs", "clubs", "clubs", "diamonds", "diamonds",
    "diamonds", "diamonds", "diamonds", "diamonds", "diamonds", "diamonds",
    "diamonds", "diamonds", "diamonds", "diamonds", "diamonds", "hearts",
    "hearts", "hearts", "hearts", "hearts", "hearts", "hearts", "hearts",
    "hearts", "hearts", "hearts", "hearts", "hearts"),
  value = c(13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 13, 12, 11, 10, 9, 8,
    7, 6, 5, 4, 3, 2, 1, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 13, 12, 11,
    10, 9, 8, 7, 6, 5, 4, 3, 2, 1)
)
# View(deck)
## You can run View(deck) to invoke the data viewer in R.
```

You should avoid typing large data sets in by hand whenever possible. Typing can lead to typos and errors. It is always better to acquire large data sets as a computer file. We will cover loading data from files and saving to files in tomorrow's content.

3.5 In-class exercises:

1. Create an empty dataframe named df.
2. Create a dataframe using the four given vectors. Name the dataframe df.

```r
# 2.
name = c('Anastasia', 'Dima', 'Katherine', 'James', 'Emily', 'Michael',
         'Matthew', 'Laura', 'Kevin', 'Jonas')
score = c(12.5, 9, 16.5, 12, 9, 20, 14.5, 13.5, 8, 19)
attempts = c(1, 3, 2, 3, 2, 3, 1, 1, 2, 1)
qualify = c('yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no', 'no', 'yes')

df <- data.frame(cbind(name,score,attempts,qualify))
df <- data.frame(name,score,attempts,qualify)

# View(df)
```

3. Get the structure of df.
4. Open df in the data viewer in R.

3.5.1 Matrices

Matrices store values in a two-dimensional array, just like a matrix from linear algebra that we learned in Math Modeling. To create one, first give matrix an atomic vector to reorganize into a matrix. Then, define how many rows should be in the matrix by setting the nrow argument to a number. matrix will organize your vector of values into a matrix with the specified number of rows. Alternatively, you can set the ncol argument, which tells R how many columns to include in the matrix:

```
die = 1:6
die
```

```
## [1] 1 2 3 4 5 6
```

```
m <- matrix(die, nrow = 2, byrow = FALSE)
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

matrix will fill up the matrix column by column by default, but you can fill the matrix row by row if you include the argument byrow = TRUE:

```
m <- matrix(die, nrow = 2, byrow = TRUE)
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

You can also create a matrix by providing the numbers directly like this:

```
matrix(1,3,2) # This creates a 3 by 2 unit matrix
```

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

```
# (value, row, col)
matrix(0,4,3) # This creates a 4 by 3 zero matrix
```

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

```

A <- matrix(c(2,3,-2,1,2,2),3,2)
A
```

3.5.1.1 Checking the Dimension

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

```
dim(A)
```

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

```

3 * A
```

3.5.1.2 Multiplication by a Scalar

```
##      [,1] [,2]
## [1,]    6    3
## [2,]    9    6
## [3,]   -6    6
```

```
B <- matrix(c(1,4,-2,1,2,1),3,2)
A
```

3.5.1.3 Matrix Addition & Subtraction

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

```
B
```

```
##      [,1] [,2]
## [1,]    1    1
## [2,]    4    2
## [3,]   -2    1
```

```
A - B
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]   -1    0
## [3,]    0    1
```

```
A + B
```

```
##      [,1] [,2]
## [1,]    3    2
## [2,]    7    4
## [3,]   -4    3
```

```
C <- matrix(c(2,-2,1,2,3,1),2,3)
A #  3 2
```

3.5.1.4 Matrix Multiplication

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

```
C # 2 3
```

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

```
dim(C)
```

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

```
dim(A)
```

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

```
C %*% A
```

```
##      [,1] [,2]
## [1,]    1   10
## [2,]    0    4
```

```
A %*% C
```

```
##      [,1] [,2] [,3]
## [1,]    2    4    7
## [2,]    2    7   11
## [3,]   -8    2   -4
```

```

A
```

3.5.1.5 Transpose of a Matrix

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

```
t(A)
```

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

```
t(t(A))
```

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

```

D <- matrix(c(4,4,-2,2,6,2,2,8,4),3,3)
D
```

3.5.1.6 Inverse of a Matrix

```
##      [,1] [,2] [,3]
## [1,]    4    2    2
## [2,]    4    6    8
## [3,]   -2    2    4
```

```
solve(D)
```

```
##      [,1] [,2] [,3]
## [1,]  1.0 -0.5  0.5
## [2,] -4.0  2.5 -3.0
## [3,]  2.5 -1.5  2.0
```

```

D
```

3.5.1.7 Determinant of a Matrix

```
##      [,1] [,2] [,3]
## [1,]    4    2    2
## [2,]    4    6    8
## [3,]   -2    2    4
```

```
det(D)
```

```
## [1] 8
```

```

A
```

3.5.1.8 Horizontal Concatenation

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

```
B
```

```
##      [,1] [,2]
## [1,]    1    1
## [2,]    4    2
## [3,]   -2    1
```

```
cbind(A,B)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    1    1    1
## [2,]    3    2    4    2
## [3,]   -2    2   -2    1
```

```

rbind(A,B)
```

3.5.1.9 Vertical Concatenation

```
##      [,1] [,2]
## [1,]    2    1
## [2,]    3    2
## [3,]   -2    2
## [4,]    1    1
## [5,]    4    2
## [6,]   -2    1
```

3.6 In-class exercises:

1. Create a 3 by 3 identity matrix I.
2. Create another 3 by 3 matrix M with numbers 1 through 9, i.e. the first row should be 1, 2, 3, etc.
3. Print the diagonal of M.
4. Multiply M by 10, and save the new matrix as N.
5. Find the determinant of M.
6. Transpose M.
7. Does M have an inverse? If so, find the inverse of M. If not, explain why.

3.7 Common Attributes

An attribute is a piece of information that you can attach to an atomic vector or any R object. The attribute won't affect any of the values in the object, and it will not appear when you display your object. You can think of an attribute as "metadata"; it is just a convenient place to put information associated with an object. R will normally ignore this metadata, but some R functions will check for specific attributes. These functions may use the attributes to do special things with the data.

You can see which attributes an object has with attributes. `attributes()` will return NULL if an object has no attributes. An atomic vector, like die, won't have any attributes unless you give it some:

```
die <- 1:6
attributes(die)
```

```
## NULL
```

```
# R uses NULL to represent the null set, an empty object.
# NULL is often returned by functions whose values are undefined.
# You can create a NULL object by typing NULL in capital letters.
```

3.7.1 Names

The most common attributes to give an atomic vector are names, dimensions (dim), and classes. Each of these attributes has its own helper function that you can use to give attributes to an object. You can also use the helper functions to look up the value of these attributes for objects that already have them. For example, you can look up the value of the names attribute of die with names:

```
names(die)
```

```
## NULL
```

NULL means that die does not have a names attribute.

You can given a "names" attribute to die like this:

```
# The vector should include one name for each element in die
names(die) <- c("one", "two", "three", "four", "five", "six")

# Now die has a names attribute
attributes(die)
```

```
## $names
## [1] "one"   "two"   "three" "four"  "five"  "six"
```

```
die
```

```
##   one   two three  four  five   six
##     1     2     3     4     5     6
```

```
# View(die)
```

To remove the names attribute, set it to NULL:

```
names(die) <- NULL
names(die)
```

```
## NULL
```

```
die
```

```
## [1] 1 2 3 4 5 6
```

```r
# View(die)
```

3.7.2 Dimension

Atomic vectors are 1-dimensional. You can get the length of them via the function `length()`.

```r
length(die)
```

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

You can transform an atomic vector into an n-dimensional array by giving it a dimensions attribute with `dim()`.

To do this, set the dim attribute to a numeric vector of length n. R will reorganize the elements of the vector into n dimensions. Each dimension will have as many rows or columns as the nth value of the dim vector. For example, you can reorganize die into a 2×3 matrix (which has 2 rows and 3 columns):

```r
dim(die) <- c(2, 3) # values by default added column-wise
die # 1, 2, 3, 4, 5, 6
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

R will always use the first value in dim for the number of rows and the second value for the number of columns. In general, rows always come first in R operations that deal with both rows and columns.

You don't have much control over how R reorganizes the values into rows and columns. For example, R always fills up each matrix by columns, instead of by rows. In the next section, we will talk about how to customize the process.

4 Functions

Function is a set of instructions that performs a specific task, typically taking input (known as arguments), processing these inputs, and returning an output. Arguments is a value that you pass to a function when you call it. The function then uses this value as input to perform its operations. Think of it like giving ingredients to a chef; the ingredients (arguments) are what the chef (function) needs to make a meal (perform a task or calculation). R comes with many functions that you can use to do sophisticated tasks like random sampling. For example, you can round a number with the **round()** function, or calculate its factorial with the **factorial()** function. Using a function is pretty simple: just write the name of the function and then the data you want the function to operate on in parentheses.

```r
# The following two functions come with base R.
# You do not need to install and load new packages to use them.
round(3.1415)
```

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

```r
factorial(3)
```

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

The data that you pass into the function is called the function's argument. The argument can be raw data, an R object, or even the results of another R function. In this last case, R will work **from the innermost**

function to the outermost.

```
die <- 1:6
die
```

```
## [1] 1 2 3 4 5 6
```

```
mean(1:6)
```

```
## [1] 3.5
```

```
mean(die)
```

```
## [1] 3.5
```

```
round(mean(die))
```

```
## [1] 4
```

```
ceiling(mean(die)) # rounds up
```

```
## [1] 4
```

```
floor(mean(die)) # rounds down
```

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

```
ceiling(4.000001)
```

```
## [1] 5
```

```
floor(3.99999999)
```

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

```
round(3.1415926535, numbers = 1)
# Error in round(3.1415, numbers = 3) : unused argument (numbers = 3)
?round
round(x = 3.1415926535, digits = 3)
```

4.1 Arguments

Often, the name of the first argument is not very descriptive, and it is usually obvious what the first piece of data refers to anyways.

But how do you know which argument names to use? If you try to use a name that a function does not expect, you will likely get an error.

If you are not sure which names to use with a function, you can look up the function's arguments with **args()**. To do this, simply plug in the name of the function into the parentheses. For example, you can see that the round function takes two arguments, one named x and one named digits:

```
args(round)
```

```
## function (x, digits = 0, ...)
## NULL
```

```
round(3.14, digits = 1)
```

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

```
args(ceiling)
```

```
## function (x)
## NULL
```

```
args(floor)
```

```
## function (x)
## NULL
```

Did you notice that `args()` shows that the digits argument of round is already set to 0?

Frequently, an R function will take optional arguments like digits. These arguments are considered optional because they come with a **default value**. You can pass a new value to an optional argument if you want, and R will use the default value if you do not.

For example, `round()` will round your number to 0 digits past the decimal point by default. To override the default, you can always specify your own argument value. For example,

```
round(3.1415)
```

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

```
round(3.1415, digits = 2)
```

```
## [1] 3.14
```

When you first start to learn R, it is recommended that you write out the **names of all your arguments** when you do a function call with multiple arguments. This not only helps you to learn the functions and their arguments better and faster, but also helps others to understand your code more easily. Moreover, you might not always remember correctly how the arguments should be ordered when first using these functions. Writing out the argument names helps **prevent errors**.

After using R for a while, you might notice that it is often for R users to skip the names for the first one or two arguments, and only write out the names for the rest of arguments.

If you do not write out the names of your arguments, R will match your values to the arguments in your function by order. You can learn about the correct order by **running the args() function, hovering over the function call, or using the help pages** .

As you provide more arguments, it becomes more likely that your order and R's order may **not align**. As a result, values may get passed to the wrong argument. Argument names prevent this. R will always match a value to its argument name, no matter where it appears in the order of arguments.

For example, if we reverse the order of arguments in the function call above, it is still equivalent.

```
round(digits = 2, x = 3.1415) # reversed order, specify argument name
```

```
## [1] 3.14
```