Chapter 2 R Basics

This chapter introduces the core R concepts you will use throughout the course. The goal is not to be exhaustive, but to give you enough familiarity with R's basic building blocks so that later analytical methods make sense.

By the end of this chapter, you should be comfortable creating objects, working with vectors and data frames, indexing data, and performing simple data manipulation tasks.

2.1 Basics of R Commands

- R is case sensitive
- When using the console, use the keyboard `↑` and `↓` arrow keys to easily cycle through previous commands typed.
- When using the text editor (i.e., a script file) in the source pane, use the `Ctrl` + `Enter` (Windows/Linux) or `Cmd` + `Enter` (Mac) keyboard shortcut to submit a line of code directly to the console.
    - The entire line does **not** need to be highlighted; the cursor needs to be anywhere on the line to be submitted.
- When using the text editor/script file, the "#" symbol signifies a comment
    - Everything after is ignored
    - It can be on the same line:

```
x <- 100   # Assign 100 to x
```

- It can be on separate lines:

```r
# Assign 100 to x
x <- 100
```

2.2 Operators

Mathematical and logical operators are used frequently.

Table 2.1: R Operators

Description	Operator
Mathematical	
addition	+
subtraction	−
multiplication	*
division	/
exponentiation	^ or **
Logical	
less than	<
less than or equal to	<=
greater than	>
greater than or equal to	>=
exactly equal to	==
not equal to	!=
Not x	!x
x OR y	x|y
x AND y	x&y
test if X is TRUE	isTRUE(x)

2.3 Objects and assignment

In R, almost everything you work with is an **object**. Objects store values, data, or results from functions.

You create objects using the assignment operator `<-`. In an RStudio script file or in the console, you can use a keyboard shortcut to produce the assignment operator. For Windows/Linux, `Alt` + `-`. For Mac, `Option` + `-`.

```
x <- 10
x
```

```
[1] 10
```

You can overwrite objects by assigning a new value:

```
x <- 25
x
```

```
[1] 25
```

As stated before, object names are case sensitive

```
x <- 100
X
```

```
Error:
! object 'X' not found
```

2.4 Vectors

A **vector** is a collection of values of the same type.

2.4.1 Creating vectors

Vectors are aften created using the concatenate function, `c(item1, item2, ...)`

```
ages <- c(18, 21, 25, 30)
ages
```

```
[1] 18 21 25 30
```

Common vector types include:

- Numeric
- Character
- Logical

```
names <- c("Alex", "Jamie", "Taylor", "Pat")
passed <- c(TRUE, FALSE, TRUE, TRUE)
```

The class of a vector can be checked with the `class(ojbect_name)` or `str(object_name)` function.

```
class(ages)
```

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

```
str(names)
```

```
 chr [1:4] "Alex" "Jamie" "Taylor" "Pat"
```

```
class(passed)
```

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

Vectors can only hold a single class/type of value. When multiple classes are included, the values are coerced to the most general type.

```
mixed <- c(1, FALSE, 3.5, "Hello!")
mixed
```

```
[1] "1"      "FALSE"  "3.5"    "Hello!"
```

```
class(mixed)
```

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

The `c()` function can be used to add to existing vectors, or combine vectors. Type coercion will be applied as needed.

```
ages2 <- c(ages, 29, 24)
ages
```

```
[1] 18 21 25 30
```

```
ages2
```

```
[1] 18 21 25 30 29 24
```

```
ages_names <- c(ages, names)
ages_names
```

```
[1] "18"     "21"     "25"     "30"     "Alex"   "Jamie"  "Taylor" "Pat"
```

```
class(ages_names)
```

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

2.4.2 Vectorized operations

R is vectorized, meaning operations apply to all elements at once.

```
ages
```

```
[1] 18 21 25 30
```

```
ages + 1
```

```
[1] 19 22 26 31
```

```
ages * 2
```

```
[1] 36 42 50 60
```

2.4.3 Vector Length

The number of items in a vector can be checked with the `length(object_name)` function, but can also be seen using the `str(object_name)` function from earlier.

```
length(ages2)
```

```
[1] 6
```

```
str(ages2)
```

```
 num [1:6] 18 21 25 30 29 24
```

2.5 Data frames

A **data frame** is a table where:

- Each column is a variable
- Each row is an observation

Data frames are the most common way to handle data sets and to provide data to statistical functions.

Data frames can be created using the `data.frame(objects)` function:

```
# Creating a data frame all in one step
students <- data.frame(
  id = 1:4,
  age = c(18, 21, 25, 30),
  major = c("MKT", "FIN", "MKT", "MKT"))
students
```

```
  id age major
1  1  18   MKT
2  2  21   FIN
3  3  25   MKT
4  4  30   MKT
```

```r
# Creating a data frame by combining vectors
new_students <- data.frame(c(names,ages,passed))
new_students
```

```
   c.names..ages..passed.
1                    Alex
2                   Jamie
3                  Taylor
4                     Pat
5                      18
6                      21
7                      25
8                      30
9                    TRUE
10                  FALSE
11                   TRUE
12                   TRUE
```

You will work with data frames constantly in this course.

2.6 Indexing and sequencing

Indexing is used to obtain particular elements of a data structure (vectors, matrices, data frames). Sequences are useful for indexing and loops.

2.6.1 Indexing vectors

Use square brackets `[]` to select elements.

```r
ages[1]
```

```
[1] 18
```

```
ages[2:4]
```

```
[1] 21 25 30
```

Logical indexing is also common:

```
ages[ages > 21]
```

```
[1] 25 30
```

2.6.2 Sequencing

Use the `#:#` coding or the `seq(from = , to = , by = )` function to create a sequence.

```
1:10
```

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

```
seq(from = 0, to = 1, by = 0.2)
```

```
[1] 0.0 0.2 0.4 0.6 0.8 1.0
```

```
seq(0,100,10)
```

```
 [1]   0  10  20  30  40  50  60  70  80  90 100
```

2.7 Common functions

Functions take inputs (arguments) and return outputs.

Examples of commonly used functions:

```
mean(ages)
```

```
[1] 23.5
```

```
min(ages)
```

```
[1] 18
```

```
max(ages)
```

```
[1] 30
```

```
summary(ages)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  18.00   20.25   23.00   23.50   26.25   30.00 
```

To learn about a function:

```
?mean
```

2.8 Missing (and Other Interesting) Values

In R, missing values are assigned a special constant, `NA`.

`NA` is not a character value, but a type of its own. Any math performed on a value of `NA` becomes `NA`.

```
ages_missing <- c(ages, NA, NA)
ages_missing
```

```
[1] 18 21 25 30 NA NA
```

```
mean(ages_missing)
```

```
[1] NA
```

However, many commands contain a option, `na.rm=TRUE`, to ignore `NA` data when performing the function.

```
mean(ages_missing, na.rm=TRUE)
```

```
[1] 23.5
```

R also has special types for infinity, `Inf`, and undefined numbers (i.e., "not a number"), `NaN`. To see this in action, take the natural log, `log()`, of certain numbers. Notice that R provides a warning when the `NaN` is found.

```
log(-1)   # Not a number
```

```
Warning in log(-1): NaNs produced
```

```
[1] NaN
```

```
log(0)    # Infinity
```

```
[1] -Inf
```

2.9 Factors

Character data can be converted into nominal **factors** using the `as.factor(object_name)` function. Each unique character value will be a level of the factor. Behind the scenes, R stores the values as integers, with a separate list of labels. When the data type is set as a factor, R knows how to handle it appropriately in the model. The levels can be accessed with the `levels(object_name)` function.

```
school_year <- c("JR", "SR", "SR", "SO", "FR", "JR")
class(school_year)
```

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

```
school_year <- as.factor(school_year)
str(school_year)
```

```
 Factor w/ 4 levels "FR","JR","SO",..: 2 4 4 3 1 2
```

```
levels(school_year)
```

```
[1] "FR" "JR" "SO" "SR"
```

2.10 What’s next

In the next chapter, we focus on **using functions effectively in R**.

You will learn how to:

- pass arguments to functions,
- work with positional versus named arguments,
- understand default values, and
- read function documentation more efficiently.

These skills are essential for working with both built-in R functions and the custom functions provided in the **MKT4320BGSU** package.