Week2: R Basics

Hakan Mehmetcik

Introduction to R and RStudio

R is a powerful and versatile programming language and environment for statistical computing and data analysis. RStudio is a popular integrated development environment (IDE) that provides a user-friendly interface for working with R.

The R language is a free, open-source software environment for statistical computing and graphics. Download and install **R** from the Comprehensive **R** Archive Network (CRAN, http://www.r-project.org)

RStudio is an open-source integrated development environment (IDE) for R created by Posit that adds many features and productivity tools for R. . Download and install (RStudio) from https://posit.co/products/open-source/rstudio.

R Basics

Mastering R (or any other programming language) essentially consists in solving two interrelated tasks:

1. Defining various types of data as *objects*!
2. Manipulating these objects by using *functions*!

Here, we introduce both of these tasks. Later, we will see that analyzing data often involves creating and manipulating additional data structures and that the activity of *programming* usually involves creating our own processes and functions.

> *To understand computations in R, two slogans are helpful:*
> *- Everything that exists is* ***an object.***
> *- Everything that happens is* ***a function*** *call.*
>
> John Chambers

Using R as a Calculator

You can use R as a calculator to perform basic arithmetic operations. Just type in your calculations, and R will provide the results.

```
# Addition

3 + 5
```

[1] 8

```
# Subtraction

10 - 2
```

[1] 8

```
# Multiplication

4 * 7
```

[1] 28

```
# Division

15 / 3
```

[1] 5

Some additional computing numbers:

```
x <- 5
y <- 2

+ x      # keeping sign
```

[1] 5

```
#> [1] 5
- y      # reversing sign
```

[1] -2

```
#> [1] -2
x + y    # addition
```

[1] 7

```
#> [1] 7
x - y    # subtraction
```

[1] 3

```
#> [1] 3
x * y    # multiplication
```

[1] 10

```
#> [1] 10
x / y    # division
```

[1] 2.5

```
#> [1] 2.5
x ^ y    # exponentiation
```

[1] 25

```
#> [1] 25
x %/% y  # integer division
```

[1] 2

```
#> [1] 2
x %% y   # remainder of integer division (x mod y)
```

[1] 1

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

When an arithmetic expression contains more than one operator, the issue of *operator precedence* arises. When combining different operators, R uses the precedence rules of the so-called "BEDMAS" order:

- **B**rackets `()`,
- **E**xponents `^`,
- **D**ivision `/` and **M**ultiplication `*`,
- **A**ddition `+` and **S**ubtraction `-`

Creating New Objects

In simple term, in R, you can create and store values **as objects** and the main type of object used in R for representing data is **a *vector***. Vectors come in different data types and shapes and have some special properties that we need to know in order to use them in a productive fashion.

Defining an object in R is done using the assignment operator `<-`. You can name the object on the left and assign a value or expression to it on the right in the following format:

`obj_name <- value`

```
# Creating objects

lg <- TRUE
n1 <- 1
n2 <- 2L
cr <- "hi"

x <- 3 * 4

y <- 3 + 4

z <- 12 / 6

a <- x * y - z
```

To determine the *type* of these objects, we can evaluate the `typeof()` function on each of them:

```
typeof(lg)
```

[1] “logical”

```
#> [1] "logical"
typeof(n1)
```

[1] "double"

```
#> [1] "double"
typeof(n2)
```

[1] "integer"

```
#> [1] "integer"
typeof(cr)
```

[1] "character"

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

Naming objects

Naming objects (both data objects and functions) is an art in itself. A good general recommendation is to always aim for consistency and clarity. This may sound trivial, but if you ever tried to understand someone else's code — including your own from a while ago — it is astonishing how hard it actually is.

Here are some generic recommendations (some of which may be personal preferences):

- Always aim for short but clear and descriptive names:
  - data objects can be abstract (e.g., `abc`, `t_1`, `v_output`) or short words or abbreviations (e.g., `data`, `cur_df`),
  - functions should be verbs (like `print()`) or composita (e.g., `plot_bar()`, `write_data()`).
- Honor existing conventions (e.g., using `v` for vectors, `i` and `j` for indices, `x` and `y` for coordinates, `n` or `N` for sample or population sizes, …).
- Create new conventions when this promotes consistency (e.g., giving objects that belong together similar names, or calling all functions that plot something with `plot_...()`, that compute something with `comp_...()`, etc.).
- Use only lowercase letters and numbers for names (as they are easy to type — and absolutely avoid all special characters, as they may not exist or look very different on other people's computers),
- Use `snake_case` for combined names, rather than `camelCase`, and — perhaps most importantly —
- Break any of those rules if there are good (i.e., justifiable) reasons for this.

Functions

Objects come in different data "types" (e.g., character, numeric, logical) and "shapes" (short or long), and — like any other data object — are manipulated by suitable "functions". R has a vast collection of built-in functions, each with specific purposes. Functions are called by their names followed by arguments in parentheses.

To do something with these objects, we can apply other functions to them:

```
!lg         # negate a logical value
```

[1] FALSE

```
#> [1] FALSE
n1          # print an object's current value
```

[1] 1

```
#> [1] 1
n1 + n2     # add 2 numeric objects
```

[1] 3

```
#> [1] 3
nchar(cr)  # number of characters
```

[1] 2

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

```
# Using built-in functions

seq(from = 10, to = 68, by = 3) # Sequence of numbers
```

[1] 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67

```
rep(3, 6) # Repeat a value
```

[1] 3 3 3 3 3 3

You can also create your own functions in R.

```
# Defining a custom function

ifnegative <- function(x) {

if (x < 0)

print("negative")

else

print("positive")

}

# Calling the custom function

ifnegative(-5)
```

[1] "negative"

```
ifnegative(2)
```

[1] "positive"

Doing statistics in R essentially means to apply statistical functions to data objects. The following basic functions examine and describe a numeric data object **nums**:

```
# Define a numeric vector:
nums <- c(-10, 0, 2, 4, 6)

# basic functions:
length(nums) # nr. of elements
```

[1] 5

```
#> [1] 5
min(nums)    # minimum
```

[1] -10

```
#> [1] -10
max(nums)    # maximum
```

[1] 6

```
#> [1] 6
range(nums)  # min - max
```

[1] -10 6

```
#> [1] -10   6

# aggregation functions:
sum(nums)   # sum
```

[1] 2

```
#> [1] 2
mean(nums)  # mean
```

[1] 0.4

```
#> [1] 0.4
var(nums)   # variance
```

[1] 38.8

```
#> [1] 38.8
sd(nums)    # standard deviation
```

[1] 6.228965

```
#> [1] 6.228965
```

Operators

R supports various operators for arithmetic, comparison, and logical operations. Here are some commonly used operators:

Arithmetic Operators

- `+` (Addition)
- `-` (Subtraction)
- `*` (Multiplication)
- `/` (Division)
- `^` or `**` (Exponentiation)
- `%%` (Modulus)

Comparison Operators

- `<` (Less than)
- `>` (Greater than)
- `<=` (Less than or equal to)
- `=` (Greater than or equal to)
- `==` (Equal)
- `!=` (Not equal)

Logical Operators

- `!` (NOT)
- `|` (OR)
- `&` (AND)
- `isTRUE` (Test if an expression is TRUE)
- isFALSE (Test if an expression is FALSE)

The `|` is an “or” operator that operates on each element of a vector, while the `||` is another “or” operator that stops evaluation the first time that the result is true!

Colon Operator

The colon `:` operator is used to create sequences of numbers, making it helpful for creating numeric vectors.

```r
# Creating a sequence of numbers

1:10 # Generates numbers from 1 to 10
```

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

Checker `%in%` Operator

The `%in%` operator checks if elements belong to a vector and is often used for data manipulation.

```r
# Checking if elements belong to a vector

a <- c(1, 2, 3, 4, 5)

b <- c(3, 4, 5, 6, 7)

a %in% b # Check if elements in 'a' belong to 'b'
```

[1] FALSE FALSE TRUE TRUE TRUE

Data Types in R

For any data object, we distinguish between its *shape* and its *type*. The *shape* of an object mostly depends on its *structure*. Overall, the following data *types* are the one you probably encounter!

1. numbers (of type *integer* or *double*)
2. text or string data (of type *character*)
3. logical values (aka. Boolean values, of type *logical*)
4. dates and times (with various data types)

Numeric: These are numeric values (e.g., 3.14, 42).

```r
x <- c(1, 2, 3, 4.5, 6.7)
```

Character: These are text values (e.g., "apple," "banana").

```
y <- c("apple", "banana", "cherry")
```

Logical: These are binary values (e.g., TRUE or FALSE).

```
z <- c(TRUE, FALSE, TRUE, FALSE, TRUE)
```

Date: These are date values in specific format. Dates and times are more complicated data types — not because they are complicated *per se*, but because their definition and interpretation needs to account for a lot of context and conventions, plus some irregularities. At this early point in our R careers, we only need to know that such data types exist. Two particular functions allow to illustrate them and are quite useful, as they provide the current date and time:

```
Sys.Date()
```

[1] "2025-02-24"

```
#> [1] "2022-09-07"
Sys.time()
```

[1] "2025-02-24 13:06:45 +03"

```
#> [1] "2022-09-07 20:01:34 CEST"
```

To check the *type* of a data object, two elementary functions that can be applied to any R object are `typeof()` and `mode()`:

```
typeof(TRUE)
```

[1] "logical"

```
#> [1] "logical"
typeof(10L)
```

[1] "integer"

```
#> [1] "integer"
typeof(10)
```

[1] "double"

```
#> [1] "double"
typeof("oops")
```

[1] "character"

```
#> [1] "character"

mode(TRUE)
```

[1] "logical"

```
#> [1] "logical"
mode(10L)
```

[1] "numeric"

```
#> [1] "numeric"
mode(10)
```

[1] "numeric"

```
#> [1] "numeric"
mode("oops")
```

[1] "character"

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

Missing values

The final concept considered here is not a type of data, but a type of *value*. What happens when we do not know the value of a variable? In R, a lacking or missing value is represented by `NA`, which stands for *not available*, *not applicable*, or *missing value*:

```
# Assign a missing value:
ms <- NA
ms
```

[1] NA

```
#> [1] NA

# Data type?
typeof(ms)
```

[1] "logical"

```
#> [1] "logical"
mode(ms)
```

[1] "logical"

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

As missing values are quite common in real-world data, we need to know how to deal with them. The function `is.na()` allows us to test for a missing value:

```
is.na(12)
```

[1] FALSE

```
#> [1] FALSE
is.na(NA)
```

[1] TRUE

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

In R, `NA` values are typically "addictive" in the sense of creating more `NA` values when applying functions to them:

```
NA + 1
```

[1] NA

```
#> [1] NA
sum(1, 2, NA, 4)
```

[1] NA

```
#> [1] NA
```

but many functions have ways of instructing R to ignore missing values. For instance, many numeric functions accept a logical argument `na.rm` that remove any `NA` values:

```
sum(1, 2, NA, 4, na.rm = TRUE)
```

[1] 7

```
#> [1] 7
```

Data Structures

Table 1: R offers various data structures to store and manipulate data:

	Homogeneous	Heterogeneous
1d	Atomic Vector	List
2d	Matrix	Data Frame
nd	Array	

Atomic Vectors

- Atomic vectors store homogeneous data, meaning all elements are of the same data type.

```
# Create an atomic vector

var1 <- c(1, 2.5, 4, 6)
```

We have already seen that using the assignment operator `<-` creates new data objects and that the `c()` function combines (or concatenates) objects into vectors. When the objects being combined are already stored as vectors, we are actually creating longer vectors out of shorter ones:

```
# Combining scalar objects and vectors (into longer vectors):
v1 <- 1           # same as v1 <- c(1)
v2 <- c(2, 3)

v3 <- c(v1, v2, 4)  # but the result is only 1 vector, not 2 or 3:
v3
```

[1] 1 2 3 4

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

Lists

- Lists store heterogeneous data, allowing different data types in the same list.

```r
# Create a list

var2 <- list("name", 2, 3, c("item1", "item2"), 16, 75)
```

Matrices

- Matrices are two-dimensional data structures.

```r
# Create a matrix

a <- matrix(1:6, ncol = 3, nrow = 2)
```

- Data frames are a versatile and commonly used data structure in R. They are similar to matrices but can store both numeric and non-numeric data. Data frames are often used to store datasets. Data frames are particularly useful for working with tabular data, and you can perform a wide range of operations on them.

```r
# create a data frame
data <- data.frame(
  Name = c("Alice", "Bob", "Charlie"),
  Age = c(25, 30, 22),
  Score = c(95, 88, 73)
)
```

Get your data into R

R have multiple in-built data

```r
# to see which data you have
data()

# for instance, you can check out data
mtcars
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

Reading existing local data

```r
# read csv file
mydata <- read.csv("~/Desktop/mac_projects 2/data-science-with-R/data/cars.csv")
View(mydata)
# write csv
write.csv(mydata, file =
            "~/Desktop/mac_projects 2/data-science-with-R/data/mydata.csv")

# read csv-file from net
mydata <- read.csv(
  "https://gist.githubusercontent.com/MorganZhang100/0c489d1f376a04d5436a/raw/7c335ebe48e575
View(mydata)

# read excel
library(readxl)
mydata1 <- read_excel(
  "~/Desktop/mac_projects 2/data-science-with-R/data/datacom.xlsx")
View(mydata1)
```

Tip

You can use here() package in order to short-cut reading and writng data!
The "here" package in R is a useful tool for creating file paths in a project-agnostic way. It provides a simple and consistent method for specifying file paths in your R scripts and projects, making your code more portable and less prone to errors when you share it with others or move it between different machines.
Here are some key aspects and benefits of the "here" package:

1. **Platform-Independent Paths:**

   - "here" automatically generates file paths that are platform-independent. This means your code will work seamlessly on Windows, macOS, and Linux without the need to manually adjust path separators (e.g., using "\\" or "/" depending on the operating system).

2. **Project-Agnostic Paths:**

   - "here" is designed to work within R projects. It helps you create paths relative to the root directory of your project, making your code independent of the specific folder structure on your machine.

3. **Consistency:**

- By using “here” to specify file paths, you ensure consistency across your project. This reduces the likelihood of errors caused by incorrect or inconsistent path specifications.

4. **Easy Integration:**

- The “here” package can be seamlessly integrated into your R scripts and projects. You don’t need to install any additional dependencies or libraries.

Here’s how you can use the “here” package in R:
1. **Installation and Loading:**

- You can install the “here” package from CRAN using the following command:

```
# install.packages("here")
library(here)
```

2. **Creating & using Paths:**

- To create file paths relative to your project’s root directory, use the ‘here()‘ function. For example, to specify a path to a data file in your project, you can do the following:

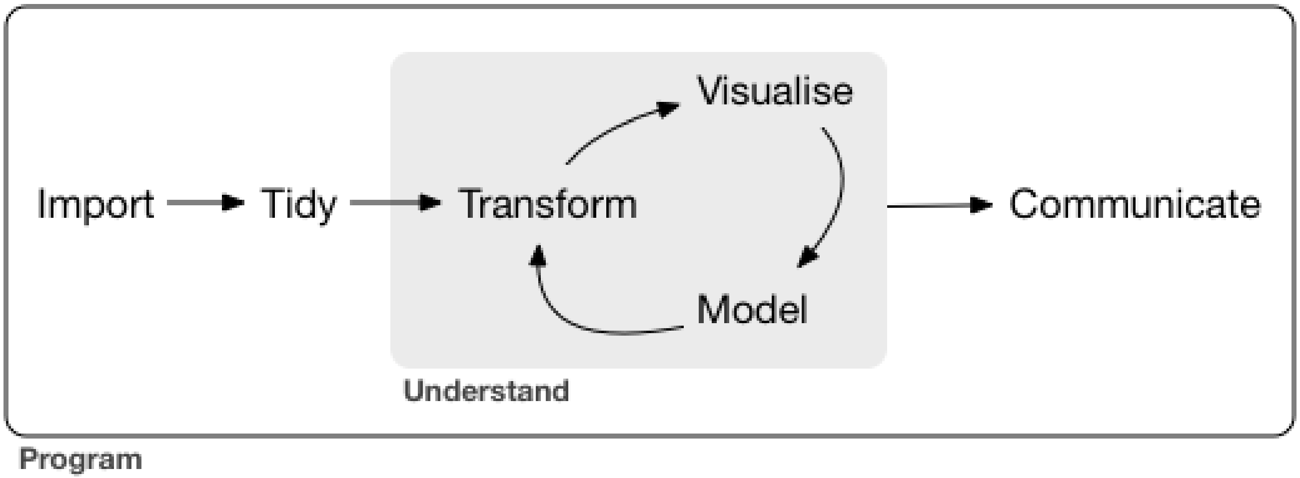
```
dpath <- "~/Desktop/mac_projects 2/data-science-with-R/data"
mydata2 <- read.csv(here(dpath, "/cars.csv"))
write.csv(mydata, here(dpath, "cars2.csv"))
```

The “here” package simplifies path management in R, making your code more robust and portable. It’s especially helpful in larger projects where consistent path specifications are essential for reproducibility and collaboration.

Tidy Data

First you must **import** your data into R. This typically means that you take data stored in a file, database, or web application programming interface (API), and load it into a data frame in R. If you can’t get your data into R, you can’t do data science on it!

Once you’ve imported your data, it is a good idea to **tidy** it. Tidying your data means storing it in a consistent form that matches the semantics of the dataset with the way it is stored. In brief, when your data is tidy, each column is a variable, and each row is an observation. Tidy data is important because the consistent structure lets you focus your struggle on questions about the data, not fighting to get the data into the right form for different functions.



Once you have tidy data, a common first step is to **transform** it. Transformation includes narrowing in on observations of interest (like all people in one city, or all data from the last year), creating new variables that are functions of existing variables (like computing speed from distance and time), and calculating a set of summary statistics (like counts or means). Together, tidying and transforming are called **wrangling**, because getting your data in a form that's natural to work with often feels like a fight!

Once you have tidy data with the variables you need, there are two main engines of knowledge generation: visualisation and modelling. These have complementary strengths and weaknesses so any real analysis will iterate between them many times.

Visualisation is a fundamentally human activity. A good visualisation will show you things that you did not expect, or raise new questions about the data. A good visualisation might also hint that you're asking the wrong question, or you need to collect different data. Visualisations can surprise you, but don't scale particularly well because they require a human to interpret them.

Models are complementary tools to visualisation. Once you have made your questions sufficiently precise, you can use a model to answer them. Models are a fundamentally mathematical or computational tool, so they generally scale well. Even when they don't, it's usually cheaper to buy more computers than it is to buy more brains! But every model makes assumptions, and by its very nature a model cannot question its own assumptions. That means a model cannot fundamentally surprise you.

The last step of data science is **communication**, an absolutely critical part of any data analysis project. It doesn't matter how well your models and visualisation have led you to understand the data unless you can also communicate your results to others.

Surrounding all these tools is **programming**. Programming is a cross-cutting tool that you use in every part of the project. You don't need to be an expert programmer to be a data scientist, but learning more about programming pays off because becoming a better programmer allows you to automate common tasks, and solve new problems with greater ease.

For more see: Modern Data Science with R - Appendix B — Introduction to R and RStudio (mdsr-book.github.io)