

Day 1: Basics of R

Qingyin Cai

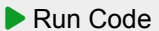


Department of Applied Economics
University of Minnesota

WebR Status

 Ready!

Slide Guide

- Click on the three horizontally stacked lines at the bottom left corner of the slide, then you will see the table of contents, and you can jump to the section you want to see.
- Hitting letter “o” on your keyboard and you will have a panel view of all the slides.
- You can directly write and run R code, and see the output on slides.
- When you want to execute (run) code, hit **command** + **enter** (Mac) or **Control** + **enter** (Windows) on your keyboard. Alternatively, you can click the “Run Code” button on the top left corner of the code chunk.

 Run Code  

```
1 # This is an example of R code chunk you will see in this slide deck.  
2 print("Let's get started!")
```

🎯 Learning Objectives

- To understand the R coding rules.
- To understand the basic types of data and structure in R, and to be able to manipulate them.
- To be able to use base R functions to do some mathematical calculations.
- To be able to create R projects and save and load data in R.

* Reference

- [Section 6: Workflow: scripts, R for Data Science](#)
- [Section 8: Workflow: projects, R for Data Science](#)

☰ Today's outline

1. General coding rules in R
2. Basic data types in R
3. Types of Data Structures in R
 - a. Vector (one-dimensional array)
 - b. Matrix (Two-dimensional array)
 - c. Data Frame
 - d. List
4. Matrix/Linear Algebra in R
5. Loading and Saving Data
6. Exercise problems
7. Appendix: Useful base-R functions

Before you start

- We'll cover many basic topics today.
- You don't need to memorize nor completely understand all the contents in this lecture.
- At the end of each section, I will include a summary of the key points you need to know. As long as you understand those key points, you are good to go.

General coding rules in R

General coding rules in R

Basics

- **R is object-oriented:** Everything in R is an “object” that you can name and reuse.
- **Creating objects:** Use `<-` or `=` to store information in objects.
 - Example: e.g., `x <- 1` assigns 1 to an object called `x`.
- **Objects can be overwritten:** If you use the same name twice, the new value replaces the old one.
- **View your objects:** Simply type the object name to see what’s stored inside.

Example

▶ Run Code



```
1 # assign value 1 to an object called "x"
2 x <- 1
3 # see what's inside object "x"
4 x
```

▶ Run Code



```
1 # assign the result of a product to an object called "y"
2 y <- 2 * 3
3
4 # store sum of x and y in z
5 z <- x + y
6 # see what's inside object "z"
7 z
```

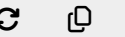
```
1 # take z, add 1, and store result back in z
2 z <- z + 1
3 # Now, the value stored in z is updated.
4 z
```


General coding rules in R

Object Naming

- Object names must start with a letter (not a number or symbol).
- Use underscores `_` or dots `.` to separate words in names.
- Choose descriptive names that tell you what the object contains.
 - Good: `student_age`, `exam_scores`
 - Avoid: `x`, `data1`, `thing`

▶ Run Code



```
1 # For example, this is not allowed. You will see an error.
2 1_test <- "R is fun"
```

▶ Run Code



```
1 # Instead, you can do this.
2 test_1 <- "R is fun"
3 test_1
```

General coding rules in R

Packages

- Packages provide extra functions beyond base R
- Install once: `install.packages("package_name")`
- Load every session: `library(package_name)`
- Troubleshooting: See `could not find function "xxxx"`? → Load the package!

Basic data types in R

Overview

Data types in R

These are the basic data elements in R.

Data Type	Description	Example
numeric	General number, can be integer or decimal.	5.2, 3L (the L makes it integer)
character	Text or string data.	"Hello, R!"
logical	Boolean values.	TRUE, FALSE
integer	Whole numbers.	2L, 100L
complex	Numbers with real and imaginary parts.	3 + 2i
raw	Raw bytes.	charToRaw("Hello")
factor	Categorical data. Can have ordered and unordered categories.	factor(c("low", "high", "medium"))

- Three main data types you'll use most often: `numeric`, `character`, and `logical`.
- Text must be in quotes:
 - Correct: `"Hello"` or `'Hello'`
 - Wrong: `Hello` (without quotes)

Overview

[How to check data types?](#)

Use `class()` or `is.XXX()` to examine the data types.

▶ Run Code



```
1 class(5.2)
2 class(2L)
3 class(TRUE)
```

Overview

Data Type Conversion

Convert between data types using `as.XXX()` functions:

- `as.numeric()` → converts to numbers
- `as.character()` → converts to text
- `as.factor()` → converts to categories

▶ Run Code



```
1 test_chr <- "5.2"  
2 class(test_chr)
```

▶ Run Code



```
1 # convert to numeric  
2 test_num <- as.numeric(test_chr)  
3 test_num  
4 class(test_num)
```

▶ Run Code



```
1 # convert from numeric to character type  
2 as.character(test_num)  
3  
4 # convert from numeric to factor type  
5 as.factor(test_num)
```

Logical values (a.k.a. Boolean values)

Basics

- Logical values are **TRUE**, **FALSE**, and **NA** (not available/undefined).
- They are often generated by **comparison operators**: **<**, **>**, **<=**, **>=**, **==**, **!=**.
- **Logical operators** include **&** (and), **|** (or), and **!** (not).
- Every comparison evaluates to **TRUE**, **FALSE**, or **NA**.
- When treated as numbers, **TRUE** equals **1** and **FALSE** equals **0**.
- Logical values can be used as **indices** to subset vectors or data.

Logical values (a.k.a. Boolean values)

Example

▶ Run Code



```
1  #--- true or false ---#
2  5 == 5
3  5 != 4
4  5 > 4
5  5 >= 4
6  5 < 4
7  5 <= 4
8
9  5 == 5 & 5 != 4
10 5 == 5 & 5 < 4
11 5 == 5 | 5 < 4
12
13 TRUE + TRUE
14 TRUE + FALSE
```


Summary

Key points

- R defines several basic data types, including `numeric`, `character`, and `logical`.
- Use the `class()` function to check the data type of an object.
- Use `as.XXX()` functions to convert an object from one type to another.
- Logical values play an important role in many R operations.

Types of Data Structures in R

Types of Data Structures in R

R provides several types of data structures for storing data.

Data Structure	Description	Creation Function	Example
Vector	One-dimensional; Holds elements of the same type .	<code>c()</code>	<code>c(1, 2, 3, 4)</code>
Matrix	Two-dimensional; Holds elements of the same type .	<code>matrix()</code>	<code>matrix(1:4, ncol=2)</code>
Array	Multi-dimensional; Holds elements of the same type .	<code>array()</code>	<code>array(c(1:12), dim = c(2, 3, 2))</code>
List	Can hold elements of different types .	<code>list()</code>	<code>list(name="John", age=30, scores=c(85, 90, 92))</code>
Data Frame	Like a table; Each column can hold different data types. This is the most common data structure.	<code>data.frame()</code>	<code>data.frame(name=c("John", "Jane"), age=c(30, 25))</code>

Vector (one-dimensional array)

Basics

- A vector object is a collection of elements of the same type.
- Vectors can contain numbers, characters, or logical values.
- Use `c()` to create a vector or to combine vectors (`c` stands for combine).

Basic syntax

```
1 c(element1, element2, element3, ...)
```

You can name each element in a vector:

```
1 c(x1 = element1, x2 = element2, x3 = element3, ...)
```

Vector (one-dimensional array)

Example

▶ Run Code



```
1 # Empty vector
2 c()
```

▶ Run Code



```
1 # Create a numeric vector
2 x <- c(1, 2, 3)
3 x
```

▶ Run Code



```
1 # Combine another numeric vector and x
2 y <- c(x, c(4, 5))
3 y
```

▶ Run Code



```
1 # Create a character vector
2 z <- c("a", "b", "c")
3
4 # See what happens when you combine numeric and character vector
5 c(x, z) #numeric is always coerced to character (power relationship, character > numeric > logical)
```

Vector: How to manipulate?

Indexing

Basics

- Use square brackets `[]` to extract one or more elements from a vector by their position.
- If a vector has names, you can extract elements using their names.
- To update an element, assign a new value to the position (or name) you want to change.

Example

▶ Run Code



```
1 # --- Create a numeric vector --- #  
2 x <- c(x1 = 5, x2 = -8, x3 = 2, x4 = -1)  
3 x
```

▶ Run Code



```
1 # --- Get the 1nd and 2nd element of x --- #  
2 index_vec <- c(1, 2)  
3 x[index_vec] #or simply you can do x[c(1, 2)]
```

▶ Run Code



```
1 # --- Get x1 and x2 --- #  
2 index_vec <- c("x1", "x2")  
3 x[index_vec] #or simply you can do x[c("x1", "x2")]
```

▶ Run Code



```
1 # --- Modify --- #
```

```
2 x[c(1, 2)] <- c(100, 200)
```

```
3 x
```

Vector: How to manipulate?

Logical Vectors

- A logical vector contains only logical values (**TRUE** and **FALSE**).
- Logical vectors can be used as index vectors: only elements matching **TRUE** are returned.

Example

▶ Run Code



```
1 # --- Create a numeric vector --- #
2 x <- c(5, -8, 2, -1)
3
4 # === For example, let's get the positive elements === #
5 # create a logical vector (condition)
6 y <- x > 0
7 # Let's see what's inside y
8 y
```

▶ Run Code



```
1 # subset the data
2 x[y] #or you can simply do x[x>0]
```


In-class Exercise

The following code randomly samples 30 numbers from a uniform distribution between 0 and 1, and stores the result in `x`.

▶ Run Code

```
1 # Run this code to work on the exercise problems.  
2 set.seed(3746)  
3 x <- runif(n = 30, min = 0, max = 1)  
4 x # see what's inside x
```

Questions

▶ Run Code

```
1 # Q1: Extract the 10th and the 15th elements of `x`.  
2 # Write your answer here
```

▶ Run Code

```
1 # Q2: Extract elements larger than $0.5$.
```

▶ Run Code

```
1 # Q3: Replace the 10th and the 15th elements of `x` to 0.
```

▶ Run Code

```
1 # Q4: If an element of `x` is larger than $0.9$, replace it with $1$.
```

Matrix (Two-dimensional array)

Basics

- A matrix is a collection of elements of the same type arranged in rows and columns (essentially a vector with an added dimension attribute).
- In practice, matrices are less common for real-world data storage and are used mainly for linear algebra operations.
- Use the `matrix()` function to create a matrix.

Syntax

```
1 matrix(data = vector_data, nrow = number_of_rows, ncol = number_of_column, byrow = FALSE)
```

- You need to specify the `vector_data` and the `number_of_rows` and `number_of_columns`.
- If the length of `vector_data` is a multiple of `number_of_columns` (or `number_of_rows`), R fills in the other dimension automatically.
- By default, values are filled by column. Use `byrow = TRUE` to fill by row.

Matrix (Two-dimensional array)

Example 1

▶ Run Code



```
1 # Create a numeric matrix
2 m_num <- matrix(1:6, nrow = 3)
3 m_num
4
5 # use dim() to see the dimension of the matrix
6 dim(m_num)
```

▶ Run Code



```
1 # Create a numeric matrix
2 m_num <- matrix(1:6, nrow = 3, byrow = TRUE)
3 m_num
```

▶ Run Code



```
1 # Create a matrix of characters
2 m_chr <- matrix(c("a", "b" , "c", "d", "e", "f"), nrow = 3)
3 m_chr
```

Matrix (Two-dimensional array)

Example 2

You can also create a matrix by combining multiple vectors using `cbind()` or `rbind()` functions.

- `rbind()` function combines vectors by row.
- `cbind()` function combines vectors by column.

▶ Run Code



```
1  vec_a <- 1:4
2  vec_b <- 4:7
3
4  mat1 <- cbind(vec_a, vec_b)
5  mat1
6
7  mat2 <- rbind(vec_a, vec_b)
8  mat2
```

Matrix: How to manipulate

Indexing

- You can access matrix elements with `[]`.
- Specify the row index and column index: `[row, col]`.
- Leave one index blank to select an entire row or column.

Example

▶ Run Code



```
1 # --- Create a matrix of numbers --- #
2 m_num <- matrix(1:6, nrow = 3)
3 m_num
4
5 # --- Get the elements in the 1st row and 2nd column --- #
6 m_num[1, 2]
```

▶ Run Code



```
1 # --- Get the 1st row --- #
2 m_num[1, ]
3
4 # --- Get the the first two rows --- #
5 m_num[1:2, ]
```

▶ Run Code



```
1 # --- Modify a specific element --- #
```

```
2 m_num[1, 2] <- 100
```

```
3 m_num
```

Matrix: How to manipulate

Miscellaneous

You can add column names and row names to a matrix using `colnames()` and `rownames()` functions. If a matrix has column names and row names, you can use the names as the index.

▶ Run Code



```
1 # Create a matrix of numbers
2 m_num <- matrix(1:6, nrow = 3)
3
4 # Add column names
5 colnames(m_num) <- c("A", "B")
6 m_num
7
8 # Add row names
9 rownames(m_num) <- c("a", "b", "c")
10
11 # --- Get the value of row "c" and column "B" --- #
12 m_num["c", "B"]
```

▶ Run Code



```
1 # --- Get the value of row "c" and column "B" --- #
2 m_num["c", "B"]
```

Matrix: Exercise Problem (Optional)

Use the following matrix:

▶ Run Code

```
1 set.seed(3746)
2 num <- runif(n = 30, min = 0, max = 1)
3 mat <- matrix(data = num, nrow = 6)
4 colnames(mat) <- c("A", "B", "C", "D", "E")
5 rownames(mat) <- c("a", "b", "c", "d", "e", "f")
6 mat # see what's inside mat
```

Questions

▶ Run Code

```
1 # Q1: Extract the element in the 2nd row and 3rd column.
2 # Write your code here
```

▶ Run Code

```
1 # Q2: Extract the 2nd row.
```

▶ Run Code

```
1 # Q3: Subset the rows where column "A" is larger than 0.5. (Use logical indexing).
```


Data Frame

Basics

- A `data.frame` class object is similar to a matrix, but each column can store a different data type.
- It is designed for tabular data, which makes it the most common structure in real-world datasets.

Syntax

```
1 data.frame(column_1 = vector_1, column_2 = vector_2)
```

Example

▶ Run Code



```
1 # create a data.frame
2 df_student <-
3   data.frame(
4     Name = c("Alice", "Ben", "Clara"),
5     Age = c(20, 22, 21),
6     Major = c("Economics", "Statistics", "Biology")
7   )
8 df_student
```

- If you do not provide column names, R automatically assigns default names (e.g., `X1`, `X2`, `X3`).

Data Frame

Indexing

- You can access elements of a `data.frame` using square brackets `[]`.
- Specify the row and column index, similar to a matrix.
- Indexing options include:
 - **Positional index** (e.g., `df[1, 2]`)
 - **Column names** (e.g., `df[, "Age"]`)
 - **Logical vectors** (e.g., `df[df$Age > 20,]`)

Run Code



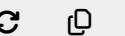
```
1 # extract the elements in the 1st row
2 df_student[1, ]
3
4 # extract the elements in the Name and Major columns
5 df_student[, c("Name", "Major")]
6
7 # find students who are older than 20
8 df_student[df_student$Age > 20, "Name"] # student$Age > 20 returns a logical vector
```

Data Frame

\$ and [[]] operator

- You can extract a single column from a `data.frame` using the `$` or `[[]]` operator.
- `$` and `[[]]` can only return **one column at a time** as a vector, while `[]` can select multiple columns.
- Type `?"$"`, `? "["`, and `? "[" [` in the Console for details.
- Inside `[[]]`, provide the column name as a character (e.g., `df[["Age"]]`).
- Why this matters: many R functions (`mean()`, `sum()`, `sqrt()`, etc.) work on **vectors**, and `$ / [[]]` are the fastest way to extract a vector for calculations.

▶ Run Code



```
1 df_student[["Age"]] #returns a vector
2
3 df_student$Age #returns a vector
```

Data Frame

Adding and Removing Columns

You can add a new column to a `data.frame` object using the `$` operator.

Syntax

```
1 data_frame$new_column <- vector_data
```

- A new column added to a `data.frame` must have the same length as the number of rows.
- If the length does not match, R will **recycle** the values to fill the column.

▶ Run Code



```
1 # Add a new column with graduation years
2 df_student$GraduationYear <- c(2025, 2024, 2026)
```

▶ Run Code



```
1 # Overwrite the column with a single value (recycled across rows)
2 df_student$GraduationYear <- 2025
3
4 # Add a new column with the same value for all rows
5 df_student$Semester <- "Fall"
```

▶ Run Code



```
1 # --- Remove the Semester column --- #
2 df_student$Semester <- NULL
```

Data Frame

Miscellaneous

▶ Run Code

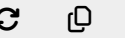


```
1 # Column names
2 names(df_student) # or colnames(df_student)
3
4 # Change the column names to lower case
5 names(df_student) <- tolower(names(df_student))
```

In-class Exercise

We will use the built-in dataset `mtcars` for this exercise. Run the following code to load the data.

▶ Run Code



```
1 # --- Load data --- #
2 data(mtcars)
3 ?mtcars # to see the description of the yield_data
4
5 # --- Take a look at the data --- #
6 # head() function shows the first several rows of the data
7 head(mtcars)
```

Questions

▶ Run Code



```
1 # Q1: Extract the rows corresponding to cars with row numbers 1, 5, and 10 using numeric indexing.
2 # Write your code here
```

▶ Run Code



```
1 # Q2: Add a new column to the `mtcars` data frame called `power_to_weight_ratio`, calculated as the ratio
  of horsepower (`hp`) to weight (`wt`).
```

▶ Run Code



```
1 # Q3: Create a new data frame called `efficient_cars` that contains cars with `mpg > 20` and
  `power_to_weight_ratio < 5`.
```

▶ Run Code



```
1 # Q4: *(Optional)* Sort the `efficient_cars` data frame by the `power_to_weight_ratio` column in ascending
  order and display the result.
```

```
2 # Hint 1: Use the `order()` function to sort the data frame
```

```
2 # Hint 1: Use the `order()` function to sort the data frame.  
3 # Hint 2: Use `order(efficient_cars$power_to_weight_ratio)` as an index vector.
```

with() and within()

- The `with()` function evaluates an expression **inside a data frame**.
 - Example: `with(df_student, mean(Age))` instead of `mean(df_student$Age)`
- The `within()` function is similar, but it allows you to **modify the data frame** directly.
 - Example: `df_student <- within(df_student, { GPA2 <- GPA^2 })`
- Using these functions helps avoid repeatedly typing the data frame name and `$`.

Example

▶ Run Code



```
1 # --- Plot --- #
2 plot(x = as.factor(df_student$major), y = df_student$age)
3
4 # instead, you can do:
5 with(df_student, plot(x = as.factor(major), y = age))
6
7 with(df_student, mean(age)) # same as mean(df_student$age)
```

▶ Run Code



```
1 # --- Add multiple columns --- #
2 df_student$graduation_year <- c(2025, 2024, 2026)
3 df_student$semester <- c("Fall", "Spring", "Fall")
4
5 # instead, you can do
6 new_df_student <- within(df_student, {
7   graduation_year <- c(2025, 2024, 2026)
```



```
8 semester <- c("Fall", "Spring", "Fall")  
9  })
```

List

Basics

- A list in R can store elements of different types and sizes: numbers, characters, vectors, matrices, data frames, or even other lists.
- A list is a flexible container that can hold any combination of data structures.
- Use the `list()` function to create a list.

▶ Run Code



```
1 list_a <- list(1, 2, "3", 4)
2 list_a
3
4 # You can even store `data.frame` in a list.
5 example_list <-
6   list(
7     num = 1:3,
8     df  = df_student
9   )
```

List

Indexing

- You can access list elements using `$`, `[]`, or `[[]]`.
- `[]` returns a list containing the selected elements.
- `[[]]` returns a single element itself (not wrapped in a list).
- `$` is shorthand for `[[]]`, but it only works if the list elements are named.

▶ Run Code



```
1 # --- Using [ ] returns a list ---  
2 example_list[1]          # or example_list["num"], still a list containing 'numbers'
```

▶ Run Code



```
1 # --- Using [[ ]] returns the element itself ---  
2 example_list[[1]]        # or example_list[["num"]], returns the vector 1:3  
3 example_list[["num"]]    # same as above
```

▶ Run Code



```
1 # --- Try with the data frame element ---  
2 example_list[2]          # list containing the data frame  
3 example_list[[2]]        # the data frame itself
```

Summary

Key points

- Know how to create the main data structures in R: `vector`, `matrix`, `data.frame`, and `list`.
 - Vectors and matrices store **one data type**.
 - Data frames and lists can store **different data types**.
- Learn how to access, subset, and modify elements using indexing.
 - Indexing can be **positional**, **logical**, or **by name**.
 - Operators include `[]`, `$`, and `[[]]`.

Matrix/Linear Algebra in R

Basic arithmetics

- You do not need to memorize the operators for remainder and quotient.

Run Code



```
1 #--- addition ---#
2 2 + 3
3 #--- subtraction ---#
4 6 - 2
5 #--- multiplications ---#
6 6 * 2
7 #--- exponentiation ---#
8 2 ^ 3
9 #--- division ---#
10 6 / 2
11 #--- remainder ---#
12 9 %% 4
13 #--- quotient ---#
14 9 %/% 4
```

Matrix/Linear Algebra in R

Vector calculation

- Arithmetic operations on vectors are performed **element-wise**.
- This means the operation is applied to elements in the **same position** of each vector.

▶ Run Code



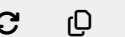
```
1 a <- c(1, 3, 2)
2 b <- c(5, 7, 6)
3
4 # --- Addition --- #
5 a + 1
6 a + b
```

▶ Run Code



```
1 # --- Subtraction --- #
2 a - 1
3 b - a
```

▶ Run Code



```
1 # --- Multiplication --- #
2 a*2
3 a*b
```

Matrix/Linear Algebra in R

Matrix calculation

- By default, `*` does element-wise multiplication.
- To perform true **matrix multiplication**, use the `%*%` operator.

▶ Run Code



```
1 mat_a <- matrix(1:4, nrow = 2)
2 mat_b <- matrix(4:7, nrow = 2)
3
4 #--- Matrix Addition and Subtraction ---#
5 mat_a + mat_b
6 mat_b - mat_a
```

▶ Run Code



```
1 # --- Matrix Multiplication using %*% operator --- #
2 mat_a %*% mat_b
3
4 mat_a * mat_b # element-wise multiplication
```

▶ Run Code



```
1 # --- Matrix Transpose --- #
2 t(mat_a)
```

Loading and Saving Data in R

R base functions for data import and export

- Like other softwares (e.g., Stata, Excel) do, R has two native data formats: `.Rdata` (or `.rdata`) and `.Rds` (or `.rds`).
 - `.Rdata` is used to save multiple R objects.
 - `.Rds` is used to save a single R object.

`.Rdata` format

- Load data:

```
load("path_to_Rdata_file")
```

- Save data:

```
save(object_name1, object_name2, file =  
"path_to_Rdata_file")
```

`.Rds` format

- Load data:

```
readRDS("path_to_Rds_file")
```

- Save data:

```
saveRDS(object_name, file = "path_to_Rds_file")
```

Setting the working directory

Basics

To access to the data file, you need to provide the path to the file (the location of the data file).

Example

Suppose that I want to load `data_example.rds` in the Data folder. On my computer, the full path (i.e., absolute path) to the file is `/Users/qingyin/Dropbox/Teaching/R_Review_2025/Data/data_example.rds`.

```
1 # this code only works in my local machine
2 df_example <- readRDS(file = "/Users/qingyin/Dropbox/Teaching/R_Review_2025/Data/data_example.rds")
```

Why avoid hard-coding full paths?

- Typing the full file path every time is cumbersome and slows you down.
- Hard-coded paths make your code **less portable**:
 - Team members may have different folder structures.
 - Code that works on your computer might fail on theirs.

Setting the working directory

Working Directory

- The **working directory** is the folder where R looks for files to load and saves files you create.
- Check the current working directory with `getwd()`.
- By default, R uses your **home directory** (or the project folder if you're in an R Project).

Setting the working directory

setwd()

- If you often import or save data in a specific folder, it helps to set that folder as the **working directory**.
- Use `setwd()` to change the working directory:

Example

In my case, I set the working directory to the `R_Review_2025` folder.

```
1 setwd("/Users/qingyin/Dropbox/Teaching/R_Review_2025")
```

Now, R will look for the data file in the `R_Review_2025` folder by default. So, I can load the data using *relative path*, not *absolute path*.

```
1 df_example <- readRDS(file = "Data/data_example.rds")
```

Problems

- `setwd()` still relies on an **absolute path**, which can vary across people.
 - e.g., one person saves files in Dropbox, another in Google Drive).
- This means `setwd()` does **not fully solve the collaboration problem**.
 - code may still break if teammates have different folder structures.

Setting the working directory

[R project](#)

[What is it?](#)

[Let's create a project!](#)

[Load the data](#)

“R experts keep all the files associated with a project together — input data, R scripts, analytical results, figures. This is such a wise and common practice that RStudio has built-in support for this via **projects**.” - [R for Data Science Ch 8.4](#)

RStudio Projects

- An RStudio project is a way to organize your work.
- When you open a Project, R automatically sets the working directory to the folder containing the `.Rproj` file — no need for `setwd()`.
- As long as the folder structure inside the Project is consistent, you can share code with teammates and relative paths will work for everyone.

Loading data other than .Rds (.rds) format

Basics

- R can load data from various formats including `.csv`, `.xls(x)`, and `.dta`.
- There exists many functions that can help you to load data:
 - `read.csv()` to read a `.csv` file.
 - `read_excel()` from the `readxl` package to read data sheets from an `.xls(x)` file.
 - `read.dta13()` function from the `readstata13` package to read a STATA data file (`.dta`).

Use `import()` function of the `rio` package

- But `import()` function from the `rio` package might be the most convenient one to load various format of data.
 - Unlike, `read.csv()` and `read.dta13()` which specialize in reading a specific type of file, `import()` can load data from various sources.

Loading data other than .Rds (.rds) format

Let's do it

In **Data** folder, **data_example** data is saved with three different formats: **data_example.csv**, **data_example.dta**, and **data_example.xlsx**. Let's load the data using **import()** function on your Rstudio.

▶ Run Code



```
1 # If you don't have the rio package, install it by running the following code:
2 # install.packages("rio")
3 library(rio)
```

Saving the data

- You can save data in many formats (`.csv`, `.dta`, `.xlsx`, etc.).
- **But** unless you need compatibility with other software, it's best to save data in `.rds`.
 - How: `saveRDS(object_name, path_to_save)`

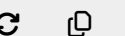
Why prefer `.rds`?

- Designed for R — no reason to use another format if you work only in R.
- **Faster and more efficient** for saving and loading.
- Produces **smaller file sizes** compared to `.csv` or `.xlsx` when data gets larger.
- (Try it! Check the size of the `data_example` dataset saved in different formats.)

Let's try!

- Load the `data_example` data in the `Data` folder.

▶ Run Code



```
1 df_example <- readRDS("Data/data_example.rds")
2 # see the first several rows
3 head(df_example)
```


Summary

Key points

- Rstudio project (`.Rproj`) is a useful tool to organize your work. As long as the folder structure under the `.Rproj` is the same, you can share the code involving data loading with your team members.
- To load data:
 - use `readRDS()` function for `.Rds` (`.rds`) format.
 - you can use `import()` function from the `rio` package for various format.
- To save the data, it is recommended to use `.rds` format and use `saveRDS()` function.

After-class Exercise Problems

Exercise Problems 1: Vector

Problems

1. Create a sequence of numbers from 20 to 50 and name it `x`. Let's change the numbers that are multiples of 3 to 0.
2. `sample()` is commonly used in Monte Carlo simulation in econometrics. Run the following code to create `r`. What does it do? Use `?sample` to find out what the function does.

▶ Run Code



```
1 set.seed(12345) #don't worry about this
2 r <- sample(1:100, size=20, replace = TRUE)
```

3. Find the value of mean and SD of vector `r` without using `mean()` and `sd()`.
4. Figure out which position contains the maximum value of vector `r`. (use `which()` function. Run `?which()` to find out what the function does.).
5. Extract the values of `r` that are larger than 50.
6. Extract the values of `r` that are larger than 40 and smaller than 60.
7. Extract the values of `r` that are smaller than 20 or larger than 70.

Exercise Problems 1: Vector

Answers

Run Code



```
1  # === Part 1 === #
2  x <- 20:50
3  # using `:` operator is the most basic way to create a sequence of numbers, but it only works with integer
   numbers with a step of 1.
4  # seq() function is more flexible. For example, you can create a sequence of numbers, , incremented by 0.5.
5  # x <- seq(from = 20, to = 50, by = 0.5)
6  x[x %% 3 == 0] <- 0
7
8  # === Part 2 === #
9  # In this code, sample() function creates a random sample of numbers with size 20 (size=20) from a range 1
   to 100 (x = 1:100) allowing replacement (replace = TRUE).
10
11 # === Part 3 === #
12 # mean
13 mean_r <- sum(r) / length(r)
14 # SD
15 sd_r <- sqrt(sum((r - mean_r)^2) / (length(r) - 1))
16
17 # === Part 4 === #
18 max_index <- which(r == max(r))
19
20 # === Part 5 === #
```

```
21  r_50 <- r[r > 50]
22
23  # === Part 6 === #
24  r_40_60 <- r[r > 40 & r < 60]
25
26  # === Part 7 === #
27  r_20_70 <- r[r < 20 | r > 70]
```

Exercise Problem 2: Data Frame

Problems

1. Load the file `nscg17tiny.dta`. You can find the data in the `Data` folder.
 - This data is a subset of the National Survey of College Graduates (NSCG) 2017, which collects data on the educational and occupational characteristics of college graduates in the United States.
2. Each row corresponds to a unique respondent. Let's create a new column called "ID". There are various ways to create an ID column. Here, let's create an ID column that starts from 1 and increments by 1 for each row.
3. To take a quick look at the summary statistics of a specific column, `summary()` function is useful. Use `summary()` to create a table of the descriptive statistics for `hrswk`. You'll provide `hrswk` column to `summary()` as a vector.
4. Create a new variable in your data that represents the z-score of the hours worked (use `hrswk` variable).
, where \bar{x} , s , and n .
 $Z_i = \frac{x_i - \bar{x}}{s/\sqrt{n}}$
5. Calculate the share of observations in your data sample with above average hours worked.

Exercise Problem 2: Data Frame

Answer

Run Code



```
1  # === Part 1 === #
2  nscg17 <- rio::import("Data/nscg17tiny.dta")
3
4  # === Part 2 === #
5  nscg17$ID <- 1:nrow(nscg17)
6
7  # === Part 3 === #
8  summary(nscg17$salary)
9
10 # === Part 4 === #
11 nscg17$z_hrswk <- (nscg17$hrswk - mean(nscg17$hrswk)) / sd(nscg17$hrswk)
12 # or using with() function, you can write the code more concisely
13 # nscg17$z_hrswk2 <- with(nscg17, (hrswk - mean(hrswk)) / sd(hrswk))
14
15 # Note: For part 2 and 3, you can use within() function to create new columns more concisely.
16 # nscg17 <-
17 #   within(
18 #     nscg17, {
19 #       ID <- 1:nrow(nscg17)
20 #       z_hrswk <- (hrswk - mean(hrswk)) / sd(hrswk)
21 #     }
22
```

```
23 # === Part 5 === #
24 # create a logical vector that indicates whether the hours worked is above average
25 above_avg_hrswk <- with(nscg17, z_hrswk > mean(z_hrswk)) # you can get the same result by using `hrswk`.
26 # subset the data
27 nscg17_above_avg_hrswk <- nscg17[above_avg_hrswk, ]
28 # calculate the share of observations with above average hours worked
29 share_above_avg_hrswk <- nrow(nscg17_above_avg_hrswk) / nrow(nscg17)
30 share_above_avg_hrswk
```


Appendix: Useful base-R functions

Appendix: A List of Useful R Built-in Functions

For data manipulation

Function	Description
<code>length()</code>	get the length of the vector and list object
<code>nrow()</code> , <code>ncol()</code>	get the number of rows or columns
<code>dim()</code>	get the dimension of the data
<code>rbind()</code> , <code>cbind()</code>	Combine R Objects by rows or columns
<code>colMeans()</code> , <code>rowMeans()</code>	calculate the mean of each column or row
<code>with</code> and <code>within()</code>	You don't need to use ` —

Appendix: A List of Useful R Built-in Functions

For numerical manipulation

Function	Description
<code>sum()</code> , <code>mean()</code> , <code>var()</code> , <code>sd()</code> , <code>cov()</code> , <code>cor()</code> , <code>max()</code> , <code>min()</code> , <code>abs()</code> , <code>round()</code>	
<code>log()</code> and <code>exp()</code>	Logarithms and Exponentials
<code>sqrt()</code>	Computes the square root of the specified float value.
<code>seq()</code>	Generate a sequence of numbers
<code>sample()</code>	randomly sample from a vector
<code>rnorm()</code>	generate random numbers from normal distribution
<code>runif()</code>	generate random numbers from uniform distribution