

1. Practical Introduction to R

Bernd Wurth

Table of contents

1	Introduction	1
2	Executing R Scripts: Full vs. Partial Execution	1
2.1	Full Execution of an R Script	1
2.2	Partial Execution of an R Script	2
2.3	Best Practices for Script Organization	2
3	Basic R Syntax	3
4	Objects and Variable Assignment	3
5	Data Types in R	3
5.1	Numeric	4
5.2	Integer	4
5.3	Character	4
5.4	Logical	4
5.5	Complex	5
5.6	Raw	5
6	Operators in R	5
7	Basic Data Structures	5
7.1	Vectors	5
7.2	Matrices	7
7.3	Lists	8
7.4	Data Frames	9
8	Functions in R	9
9	Control Structures	9
10	Reading and Writing Data	10
11	Basic Data Manipulation	10
12	Introduction to Basic Plotting	11

1 Introduction

This introduction covers the basics of R programming. As you progress, you'll discover more advanced features and packages that extend R's capabilities even further. Remember to use the help function (`?function_name`) to learn more about specific functions and their usage.

2 Executing R Scripts: Full vs. Partial Execution

2.1 Full Execution of an R Script

Running a script from top to bottom is useful when you want to execute all the code at once. This is typically done after you've written and verified the entire script.

Steps:

1. Save your script with a `.R` extension, such as `my_script.R`.
2. Use the `source()` function in R to execute the entire script:

```
source("my_script.R")
```

3. Alternatively, in RStudio:

- Click the Source button at the top of the script editor.
- Use the shortcut `Ctrl + Shift + S` (Windows) or `Cmd + Shift + S` (Mac).

Example script:

```
# Define a function
add_numbers <- function(x, y) {
  return(x + y)
}

# Perform calculations
result <- add_numbers(5, 3)
print(result)

# Generate a sequence
seq_data <- seq(1, 10, by = 2)
print(seq_data)
```

2.2 Partial Execution of an R Script

Running parts of your script manually is useful during the development and debugging process. This allows you to test specific sections without executing the entire script.

Steps in RStudio:

1. Highlight the portion of the script you want to run.
2. Press `Ctrl + Enter` (Windows) or `Cmd + Enter` (Mac) to run the selected lines.
3. If you want to run only the current line, place the cursor on that line and press the same shortcut.

```
# Define a function
add_numbers <- function(x, y) {
  return(x + y)
}

# Highlight and run the following line:
# result <- add_numbers(5, 3)

# Debugging this line separately:
print(result)

# Highlight and run this block to test sequences:
seq_data <- seq(1, 10, by = 2)
print(seq_data)
```

2.3 Best Practices for Script Organization

Divide your script into sections using comments:

```
# Section 1: Load libraries
library(ggplot2)

# Section 2: Load data
data <- read.csv("data.csv")

# Section 3: Analysis
summary(data)
```

Use **descriptive variable and function names** to make scripts easier to understand.

Keep your script modular—write functions for reusable code blocks.

Include comments to explain complex logic or calculations.

3 Basic R Syntax

R is case-sensitive and uses the `<-` operator for assignment (though `=` can also be used). Comments start with `#`.

```
# This is a comment
x <- 5 # Assign the value 5 to x
y = 10 # This also works, but <- is more common in R
```

4 Objects and Variable Assignment

In R, you can assign values to variables using the assignment operator `<-`:

```
my_variable <- 42
my_name <- "Alice"
```

You can view the contents of a variable by typing its name:

```
my_variable
my_name
```

Note

R is case sensitive, meaning that uppercase and lowercase letters are treated as different characters. In the example above, typing `My_variable` would result in an error. Below is another example:

```
x <- 10

x # will print the number 5
X # will return an error
```

5 Data Types in R

Understanding data types is fundamental when working with R. Data types define the kind of data you can store and manipulate. In R, there are six primary data types:

1. **Numeric:** Decimal (real) numbers.
2. **Integer:** Whole numbers.
3. **Character:** Text or string data.
4. **Logical:** TRUE or FALSE values (boolean).
5. **Complex:** Complex numbers (rarely used in business applications).
6. **Raw:** Raw bytes (advanced usage, rarely needed).

Examples of the fundamental data types are provided below. You can also convert data types by using functions like `as.numeric()`, `as.character()`, or `as.logical()`

```
# Converting a numeric to a character
numeric_value <- 123
character_value <- as.character(numeric_value)
typeof(character_value) # Output: "character"
```

Converting values is important when working with mixed data (e.g., reading in text files, user inputs, categorical data). The function `as.character(numeric_value)` converts a numeric value 123 into a character string “123”. R will now treat this number as text and you will not be able to use this variable for calculations. This can be useful or appropriate when numbers are used as labels (e.g., phone numbers, IDs) rather than mathematical values or when using functions that require specific data types (e.g., `paste()` expects characters), among others.

i Note

In R, `class()` and `typeof()` are both used to inspect the properties of an object, but they serve different purposes.

- `class()` tells you what **kind of object** it is in terms of R's functionality (often human-readable and associated with the object's use like `data.frame`, `matrix`, `factor`).
- `typeof()` tells you the **low-level data type** the object is stored as in memory (usually in more technical terms like `double`, `integer`, or `list`).

```
# Example 1:
x <- data.frame(a = 1:3, b = 4:6)
class(x) # Output: "data.frame"
typeof(x) # Output: "list" (because data frames are stored as lists internally)

# Example 2:
x <- c(1, 2, 3) # A numeric vector
class(x) # Output: "numeric" (functional description)
typeof(x) # Output: "double" (internal storage type)
```

You can also check specific data types by using `is.numeric()`, `is.character()`, `is.logical()`, etc., which will return a logical output.

5.1 Numeric

Numeric data includes real numbers (decimal values).

```
# Assigning a numeric value
salary <- 55000.75

# Displaying the type of data
typeof(salary) # Output: "double"
```

5.2 Integer

Integers are whole numbers. To explicitly define an integer, add an L after the number.

```
# Assigning an integer value
age <- 30L

# Displaying the type of data
typeof(age) # Output: "integer"
```

5.3 Character

Character data represents text or strings. In R, strings are enclosed in double (") or single (') quotes.

```
# Assigning a character string
name <- "John Doe"

# Displaying the type of data
typeof(name) # Output: "character"
```

5.4 Logical

Logical values represent TRUE or FALSE. They are useful for decision-making and logical comparisons.

```
# Assigning logical values
is_graduate <- TRUE

# Displaying the type of data
typeof(is_graduate) # Output: "logical"
```

5.5 Complex

Complex numbers consist of a real and an imaginary part. Business students will rarely use them.

```
# Assigning a complex number
z <- 2 + 3i

# Displaying the type of data
typeof(z) # Output: "complex"
```

5.6 Raw

Raw data represents bytes. This is an advanced data type, generally not required for business applications.

```
# Creating raw data
r <- charToRaw("ABC")

# Displaying the type of data
typeof(r) # Output: "raw"
```

6 Operators in R

R supports various types of operators:

1. Arithmetic: +, -, *, /, ^ (exponent), %% (modulus)
2. Relational: <, >, <=, >=, ==, !=
3. Logical: & (and), | (or), ! (not)

```
x <- 10
y <- 3

x + y
x > y
(x > 5) & (y < 5)
```

7 Basic Data Structures

In addition to the fundamental data types, R also has several important structured data types:

1. **Vectors**: One-dimensional arrays that can hold data of the same type
2. **Matrices**: Two-dimensional arrays with data of the same type
3. **Lists**: Can hold elements of different types
4. **Data Frames**: Two-dimensional arrays that can hold different types of data

7.1 Vectors

Vectors are one of the most fundamental data structures in R, designed to store a sequence of elements of the same type (e.g., numeric, character, logical). They are extensively used in R for data manipulation, analysis, and computations due to their simplicity and efficiency. A vector can be created using the `c()` function, which combines individual elements into a single structure. For example, `c(1, 2, 3)` creates a numeric vector containing three elements. R performs operations on vectors element-wise, making it easy to perform tasks such as arithmetic, logical comparisons, and indexing. Vectors serve as building blocks for more complex data structures like matrices, lists, and data frames, making them indispensable in R programming for both basic and advanced applications.

```
# Numeric vector
prices <- c(100.5, 200.75, 300.25) # c() is used to create both numeric and character vectors

# Character vector
products <- c("Laptop", "Tablet", "Smartphone")

# Logical vector
available <- c(TRUE, FALSE, TRUE)
```

```
# Mixed vectors
mixed <- c("g",1,3,"m")

# Checking the type of a vector
typeof(prices) # Output: "double"
typeof(mixed)  # Output: "character", if there are a mixture of strings and numbers the numbers will be
```

We can easily create vectors with repeating sequences of elements:

```
seq_vector1 <- seq(1,4, by=0.5)      # creates a vector going from 1 to 4 in steps of 0.5
seq_vector2 <- seq(1,4, length.out=10) # creates a vector of evenly spaced numbers from 1 to 4 with len
seq_vector3 <- 1:4                    # creates a vector from 1 to 4 in steps of 1

rep_vector1 <- rep(1, times=4)        # repeats the value 1 4 times
rep_vector2 <- rep(d, times=4)        # repeats the vector d 4 times
rep_vector3 <- rep(d, each=4)         # repeats each value in d 4 times
```

We can also access individual values from each vector using single square brackets.

```
# Define vector
seq_vector1 <- seq(1,4, by=0.5)

seq_vector1[2] # Output: "1.5"
seq_vector1[3] # Output: "2"
```

We can perform arithmetic operations with two numeric vectors in R, such as `a + b`, the operation is applied **component-wise**. This means each element in vector `d` is added to the corresponding element in vector `e`. If the vectors are of unequal length, R **recycles** elements of the shorter vector until it matches the length of the longer vector, with a warning if the lengths are not multiples of each other.

```
a <- c(1, 2, 3)
b <- c(4, 5, 6)
a + b # Output: c(5, 7, 9) (1+4, 2+5, 3+6)
```

When working with vectors of strings, the `paste()` function in R is a versatile tool. It is primarily used to concatenate elements of one or more vectors into strings, offering flexibility to combine text data either **component-wise** or by **collapsing** all elements into a single string. This makes `paste()` particularly useful for tasks such as creating descriptive labels, formatting output, or preparing data for presentation.

Key Arguments:

- **sep**: Specifies the separator to place between concatenated elements. Default is a single space (" ").
- **collapse**: Combines all elements of a single vector into one string, using the specified delimiter. If not provided, each concatenated result remains as a separate string.

When you have two or more vectors and want to combine corresponding elements (component-wise concatenation):

```
# Example vectors
products <- c("Laptop", "Tablet", "Smartphone")
prices <- c("1000", "500", "750")

# Combine product names and prices
result <- paste(products, prices, sep=" - $")
print(result) # Output: "Laptop - $1000" "Tablet - $500" "Smartphone - $750"
```

When you want to combine all elements of a single vector into one string:

```
# Example vector
items <- c("Apple", "Banana", "Cherry")

# Collapse all items into a single string
result <- paste(items, collapse=", ")
print(result) # Output: "Apple, Banana, Cherry"
```

7.2 Matrices

Matrices in R are two-dimensional data structures that store elements of the same type (e.g., numeric, character, or logical) in a grid format with rows and columns. They are ideal for mathematical computations, linear algebra operations, and organizing data with fixed dimensions. Matrices are created using the `matrix()` function, where you specify the data, number of rows (`nrow`), and columns (`ncol`). Operations like addition, subtraction, or multiplication can be applied element-wise or across rows and columns, and advanced matrix-specific operations (e.g., transposition, matrix multiplication) are supported. Matrices are often used in business applications like modeling financial data, performing statistical analyses, or visualizing multidimensional datasets.

```
# Creating a matrix
sales <- matrix(c(10, 20, 30, 40), nrow = 2, ncol = 2)

# Displaying the matrix
print(sales)

# Checking the type
typeof(sales) # Output: "double"
```

Element-wise multiplication performs operations on corresponding elements of two matrices of the same dimensions.

```
# Create two matrices
A <- matrix(c(1, 2, 3, 4), nrow = 2, ncol = 2)
B <- matrix(c(5, 6, 7, 8), nrow = 2, ncol = 2)

# Element-wise multiplication
C <- A * B
print(C)

# Output:
#      [,1] [,2]
# [1,]    5  21
# [2,]   12  32
```

In the example above, the vector `c(1, 2, 3, 4)` is filled **column-wise** by default into a 2×2 matrix. This results in:

```
A =
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

The expression `C <- A * B` performs element-wise multiplication, meaning each corresponding element in A and B is multiplied together.

```
C =
     [,1] [,2]
[1,]  1*5 3*7 => [1,]    5  21
[2,]  2*6 4*8 => [2,]   12  32
```

You can also apply operations **across rows or columns** using functions like `apply()` or `colSums()/rowSums()`.

```
# Create a matrix
M <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3)

# Sum across rows
row_totals <- rowSums(M)
print(row_totals)
# Output: [1] 9 12

# Sum across columns
col_totals <- colSums(M)
print(col_totals)
# Output: [1] 3 7 11
```

Transposition swaps rows and columns of a matrix. Use the `t()` function for this operation.

```
# Transpose a matrix
M <- matrix(c(1, 2, 3, 4), nrow = 2, ncol = 2)
print(M)

# Output:
#      [,1] [,2]
# [1,]    1    3
# [2,]    2    4

transposed <- t(M)
print(transposed)

# Output:
#      [,1] [,2]
# [1,]    1    2
# [2,]    3    4
```

Matrix multiplication (dot product) is performed using the `%*%` operator. This is different from element-wise multiplication and follows the rules of linear algebra.

```
# Create two matrices
A <- matrix(c(1, 2, 3, 4), nrow = 2, ncol = 2)
B <- matrix(c(5, 6, 7, 8), nrow = 2, ncol = 2)

# Matrix multiplication
C <- A %*% B
print(C)

# Output:
#      [,1] [,2]
# [1,]   23   31
# [2,]   34   46
```

There are further advanced operations, including the **determinant** of a square matrix, which can be calculated using `det()`.

```
det_A <- det(A)
print(det_A)
# Output: -2 (for the given matrix A)
```

The **inverse** of a square matrix (if it exists) can be calculated using `solve()`.

```
# Inverse of a matrix
inverse_A <- solve(A)
print(inverse_A)
# Output:
#      [,1] [,2]
# [1,]   -2  1.5
# [2,]    1 -0.5
```

Eigenvalues and eigenvectors can be calculated using the `eigen()` function.

```
eigen_A <- eigen(A)
print(eigen_A$values) # Eigenvalues
print(eigen_A$vectors) # Eigenvectors
```

7.3 Lists

Lists in R are highly versatile data structures that can store elements of **different types**, such as vectors, matrices, data frames, and even other lists. This flexibility makes lists ideal for organizing and managing complex data, such as combining related datasets or storing model outputs. Lists are created using the `list()` function, where each element can be named for easy reference. They are particularly useful in business analytics

for grouping data with varied structures, such as customer demographics, sales figures, and statistical results. Accessing elements in a list is done using `$` (by name) or double square brackets `[[]]` (by index). Lists are widely used in R for functions that return multiple results, such as regression models or simulation outputs, providing a structured yet flexible way to handle diverse data.

```
# Creating a list
employee <- list(name = "John Doe", age = 30L, salary = 55000.75, active = TRUE)

# Accessing elements
employee$name # Output: "John Doe"
```

7.4 Data Frames

Data frames in R are one of the most widely used data structures for storing and analyzing tabular data. They organize data into rows and columns, where each column can have a different data type (e.g., numeric, character, or logical), making them ideal for real-world datasets like business transactions or survey results. Data frames are created using the `data.frame()` function or imported from external files like CSVs or Excel sheets. They support powerful indexing and manipulation capabilities, allowing users to filter, summarize, and transform data efficiently. Data frames are fundamental for business analytics and statistical modeling, providing a structured and intuitive way to handle datasets in R.

```
# Creating a data frame
df <- data.frame(
  Product = c("Laptop", "Tablet", "Smartphone"),
  Price = c(1000, 500, 750),
  InStock = c(TRUE, TRUE, FALSE)
)

# Displaying the data frame
print(df)
```

8 Functions in R

Functions in R are reusable blocks of code designed to perform specific tasks, making programming more efficient and organized. They help reduce repetition, improve readability, and facilitate debugging by encapsulating logic into a single unit. Functions are essential for scalable and modular programming in R. R comes with many built-in functions for tasks like data manipulation, statistical analysis, and visualization (e.g., `mean()`, `summary()`).

```
# Using a built-in function
mean(c(1, 2, 3, 4, 5))
```

Users can also create custom functions using the `function()` keyword, specifying input arguments and defining the operations to perform. Custom functions are especially useful for automating repetitive tasks or implementing specialized calculations. To create a function, assign it to a name and define its arguments and body, returning the desired output. For example:

```
# Creating a custom function
square <- function(x) {
  return(x^2)
}

square(4)
```

9 Control Structures

Control structures like if-else statements and loops in R are fundamental tools for directing the flow of a program based on conditions and for performing repetitive tasks.

If-else statements allow conditional execution of code, enabling decisions to be made based on logical tests.

```
# If-else statement
x <- 10
```

```
if (x > 5) {
  print("x is greater than 5")
} else {
  print("x is not greater than 5")
}
```

Loops, such as `for` and `while`, are used for repetitive operations. **For** loops allow iteration over a sequence, such as a vector, list, or range. They are used for tasks where operations need to be applied to each element in a sequence. Examples include iterating over rows in a dataset, applying calculations to each column, or generating repeated output.

```
# Example of a for loop
for (i in 1:5) {
  print(i^2)
}
```

While loops continue to execute a block of code as long as a specified condition is true. They are used for tasks where the number of iterations is not predefined but depends on a dynamic condition. Examples include repeatedly performing calculations until a threshold is reached or a condition changes.

```
# While loop
i <- 1
while (i <= 5) {
  print(i^2)
  i <- i + 1
}
```

10 Reading and Writing Data

R can read data from various file formats. Here's an example with CSV:

```
# Reading a CSV file
# Assuming you have a file named "data.csv" in your working directory
data <- read.csv("data.csv")

# Writing a CSV file
write.csv(df, "output.csv", row.names = FALSE)
```

For this example, you will need to create a “data.csv” file in your working directory or adjust the file path accordingly.

11 Basic Data Manipulation

R provides many functions for manipulating data:

```
# Assuming we're using the 'df' data frame from earlier

# Selecting a column
df$Product

# Filtering rows
df[df$Price > 600, ]

# Adding a new column and printing the new dataset
df$HighEndt <- df$Price >= 999
print(df)

# Summarizing data
summary(df)
```

12 Introduction to Basic Plotting

R has powerful plotting capabilities. Here's a simple example:

```
# Create some data
x <- 1:10
y <- x^2

# Create a scatter plot
plot(x, y, main = "Square Function", xlab = "x", ylab = "y")

# Add a line
lines(x, y, col = "red")
```

We will explore more advanced plotting with the `ggplot` package later.