R Programming
Module 1: Basics of R Programming

Johns Hopkins University

July 6, 2024

Contents

1 Introduction

In this course, we will:

- Learn how to program in R;
- Learn how to use R for effective data analysis;
- Learn how to install and configure software necessary for a statistical programming environment;
- Discuss generic programming language concepts as they are implemented in a high-level statistical language;
- Cover practical issues in statistical computing, including:
  - Reading data into R
  - Accessing R packages
  - Writing R functions
  - Debugging
  - Organising and commenting R code.

2 Data Types

2.1 Objects and attributes

2.1.1 Objects

- R has five atomic (basic) classes of objects:
  1. Character
  2. Numeric
  3. Integer
  4. Complex
  5. Logical
- The most basic object is a **vector**. A vector can only contain objects of the same class. One exception to this is a **list**, which is represented as a vector but can contain objects of different classes.
- Empty vectors can be created using the **vector()** function.

2.1.2 Numbers

- Numbers in R are generally treated as numeric objects - double precision real numbers;
- If you explicitly want an integer, you need to specify the L suffix;
- Example: Entering 1 gives you a numeric object; entering 1L explicitly gives you an integer;
- There is also a special number Inf which represents infinity; e.g., 1/0. Inf can be used in ordinary calculations, e.g., 1/Inf is 0;
- The value NaN represents an undefined value ("not a number"); e.g., 0/0. NaN can also be thought of as a missing value.

2.1.3 Attributes

- Every object in R can also have **attributes**, which we can think of as as named list of arbitrary metadata;
- Two attributes are particularly important: (1) the **dimension** attribute, which turns vectors into matrices and arrays, (2) the **class** attribute, which powers the S3 object system - this includes important vectors such as factors, date and times, data frames, and tibbles;
- Other attributes include: names and dimnames, length, and other user-defined attributes/metadata;
- Attributes of an object can be accessed using the **attributes()** function.

2.2 Vectors and lists

2.2.1 Creating vectors

The c() function can be used to create vectors of objects:

```r
a <- c(0.5, 0.6)
a

## [1] 0.5 0.6

class(a) # Numeric

## [1] "numeric"

b <- c(TRUE, FALSE)
b

## [1]  TRUE FALSE

class(b) # Logical

## [1] "logical"

c <- c(T, F)
c

## [1]  TRUE FALSE

class(c) # Logical

## [1] "logical"

d <- c("a", "b", "c", "d")
d

## [1] "a" "b" "c" "d"

class(d) # Character

## [1] "character"

e <- c(9:29)
e

##  [1]  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

class(e) # Integer

## [1] "integer"

f <- c(1 + 0i, 2 + 4i)
f

## [1] 1+0i 2+4i

class(f) # Complex

## [1] "complex"
```

We can also use the vector() function to create vectors:

```r
x <- vector("numeric", length = 10) # Two arguments: mode (class) and length.
x

##  [1] 0 0 0 0 0 0 0 0 0 0
```

2.2.2 Mixing objects

What would happen if we create the following vectors?

```r
y1 <- c(1.7, "a")
y1

## [1] "1.7" "a"

class(y1)

## [1] "character"

y2 <- c(TRUE, 2)
y2

## [1] 1 2

class(y2)
```

```
## [1] "numeric"

y3 <- c("a", TRUE)
y3

## [1] "a"    "TRUE"

class(y3)

## [1] "character"
```

When objects of different classes are mixed in a vector, **coercion** occurs so that every element in the vector is of the **same** class.

2.2.3 Explicit coercion

Objects can be explicitly coerced from one class to another usin the as.* functions:

```
x <- 0:6
class(x)

## [1] "integer"

as.numeric(x)

## [1] 0 1 2 3 4 5 6

as.logical(x)

## [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE

as.character(x)

## [1] "0" "1" "2" "3" "4" "5" "6"
```

Nonsensical coercion results in NA values:

```
x <- c("a", "b", "c")
as.numeric(x)

## Warning: NAs introduced by coercion

## [1] NA NA NA

as.logical(x)

## [1] NA NA NA

as.complex(x)

## Warning: NAs introduced by coercion

## [1] NA NA NA
```

2.2.4 Lists

Lists are a special type of vector that can contain elements of different classes. Lists are a very important data type which we should familiarise ourselves with. Elements of lists can be accessed using double square brackets [[]], while elements of other vectors can be accessed using single square brackets [].

```
x <- list(1, "a", TRUE, 1 + 4i)
x[[1]]

## [1] 1

x[[2]]

## [1] "a"

x[[3]]

## [1] TRUE

x[[4]]

## [1] 1+4i
```

2.3 Matrices

Matrices are vectors with a *dimension* attribute. The dimension attribute is itself an integer vector of length 2 (nrow, ncol).

```
m <- matrix(nrow = 2, ncol = 3)
m
```

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

```
dim(m)
```

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

```
attributes(m)
```

```
## $dim
## [1] 2 3
```

Matrices are constructed *column-wise*, so entries can be thought of starting in the “upper left” corner and running down the columns.

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

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

Matrices can also be created directly from vectors by adding a dimension attribute:

```
m <- 1:10
m
```

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

```
dim(m) <- c(2, 5)
m
```

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

Matrices can also be created by column-binding or row-binding using the cbind() and rbind() functions, respectively:

```
x <- 1:3
y <- 10:12
cbind(x, y)
```

```
##      x  y
## [1,] 1 10
## [2,] 2 11
## [3,] 3 12
```

```
rbind(x,y)
```

```
##   [,1] [,2] [,3]
## x    1    2    3
## y   10   11   12
```

2.4 Factors

- Factors are used to represent categorical data. Factors can be unordered or ordered. One can think of a factor as an integer vector where each integer has a *label*;
- Factors are treated specially by modelling functions like lm() and glm();
- Using factors with labels is *better* than using integers because factors are self-describing; having a variable that has values “Male” and “Female” is better than a variable that has values 1 and 2.

Factors can be created using the **factor()** function:

```
x <- factor(c("yes", "yes", "no", "yes", "no"))
x

## [1] yes yes no  yes no
## Levels: no yes

table(x)

## x
##  no yes
##   2   3

unclass(x)

## [1] 2 2 1 2 1
## attr(,"levels")
## [1] "no"  "yes"

attr(x, "levels")

## [1] "no"  "yes"
```

The order of the levels can be set using the **levels()** argument in the factor() function. This can be important in linear modelling because the first level is used as the baseline level.

```
x <- factor(c("yes", "yes", "no", "yes", "no"),
            levels = c("yes", "no"))
x

## [1] yes yes no  yes no
## Levels: yes no
```

2.5 Missing values

Missing values are denoted by **NA** or **NaN** for undefined mathematical operations.

- **is.na()** is used to test objects if they are NA;
- **is.nan()** is used to test for NaN;
- NA values have a **class** also, so there are integer NA, character NA, logical NA, etc.;
- A NaN value is also NA but a NA value is not always a NaN value.

```
x <- c(1, 2, NA, 10, 3)
is.na(x)

## [1] FALSE FALSE  TRUE FALSE FALSE

is.nan(x)

## [1] FALSE FALSE FALSE FALSE FALSE

x <- c(1, 2, NaN, NA, 4)
is.na(x)

## [1] FALSE FALSE  TRUE  TRUE FALSE

is.nan(x)

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

2.6 Data Frames

Data frames are used to store **tabular** data.

- Data frames are represented as a special type of list where every element of the list must have the same length;
- Each element of the list can be thought of as a column and the length of each element of the list of the number of rows;
- Unlike matrices, data frames can store different classes of objects in each column (similar to lists); elements in a matrix must be of the same class;
- Data frames also have a special attribute called **row.names**;
- Data frames are usually created by calling **read.table()** or **read.csv()**.

We can also create data frames using the **data.frame()** function:

```
x <- data.frame(particle = c("a", "b", "c", "d"), energy = c(100, 200, 300, 400))
x

##   particle energy
## 1        a    100
## 2        b    200
## 3        c    300
## 4        d    400

nrow(x)

## [1] 4

ncol(x)

## [1] 2
```

2.7 Names attribute

All R objects can have names, which is very useful for writing readable code and self-describing objects.

```
x <- 1:3
names(x)

## NULL

names(x) <- c("electron", "photon", "positron")
x

## electron   photon positron
##        1        2        3

names(x)

## [1] "electron" "photon"   "positron"
```

Lists can have names:

```
x <- list(a = 1, b = 2, c = 3)
x

## $a
## [1] 1
##
## $b
## [1] 2
##
## $c
## [1] 3
```

Matrices can have names:

```
m <- matrix(1:4, nrow = 2, ncol = 2)
m

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

dimnames(m) <- list(c("a", "b"), c("c", "d"))
m

##   c d
## a 1 3
## b 2 4
```

3 Reading data

There are a few principal functions used for reading data into R:

- **read.table()** and **read.csv()** for reading tabular data;
- **readLines()** for reading lines of a text file;

- **source()** and **dget()** for reading in R code files;
- **load()** for reading in saved workspaces;
- **unserialize()** for reading single R objects in binary form.

3.1 Reading tabular data

The read.table() function is one of the most commonly used functions for reading data into R. It has a few important arguments:

- **file**: The name of the file, or a connection (e.g. ODBC);
- **header**: Logical argument indicating if the file has a header line;
- **sep**: A string indicating how the columns are separated;
- **colClasses**: A character vector indicating the class of each column in the dataset;
- **nrows**: The number of rows in the dataset;
- **comment.char**: A character string indicating the comment character;
- **skip**: The number of lines to skip from the beginning;
- **stringsasFactors**: Logical argument specifying if character variables should be coded as factors.

For small to moderately sized datasets, we can usually call the **read.table()** function without specifiying other arguments. This would lead to R to automatically:

- Skip lines beginning with a #;
- Determine how many rows there are (and how much memory needs to be allocated);
- Determine what type of variable is in each column of the table.

If we explicitly state the parameters for each argument, then R run will faster and more efficiently, though this is not necessary for smaller datasets. N.B. The **read.csv()** function is identical to **read.table()** except that the default separator is a **comma**.

3.2 Reading large tables

With much larger datasets, doing the following things will make your life easier and will prevent R from choking:

- Read the help page for read.table, which contains many hints;
- Estimate the amount of memory required to store your dataset - if the dataset is larger than the amount of RAM on your computer, you can probably stop right here;
- Set comment.char = “” if there are no commented lines in your file.

Another important step is to use the **colClasses** argument:

- Specifying the parameter of the ColClasses argument, rather than using the default, can make the read.table() function run much (about 2 times) faster;
- To use this option, you must know the class of each column in your data frame;
- If, for example, all the columns are of class numeric, then you can just set colClasses = “numeric”.

A quick way to determine the classes of each column is the following:

```r
# initial <- read.table("sample_data.txt", nrows = 100)
# classes <- sapply(initial, class)
# tabAll <- read.table("sample_data.txt", colClasses = classes)
```

We could also specify the parameter to the **nrows** argument. This doesn’t make R run faster but it helps with memory usage. In general, when using R with larger datasets, it is useful to know some details about your system, including:

- How much memory (RAM) is available?
- What other applications are in use?
- Are there other users logged into the same system?
- What operating system is being used?
- Is the operating system 32-bit (x86) or 64-bit (x64)?

3.2.1 Calculating memory requirements

Suppose we have a data frame with 1.5 million rows and 120 columns, all of which are numeric data. To determine how much memory is required to store this data frame, we can do the following set of calculations:

```r
# Calculate the number of bytes based on the fact that each number requires 8 bytes of RAM.
bytes <- 1500000 * 120 * 8
bytes_unit <- "bytes"
bytes_formatted <- format(bytes, scientific = FALSE, big.mark = ",")
formatted_bytes <- sprintf("%s %s", bytes_formatted, bytes_unit)
formatted_bytes
```

```
## [1] "1,440,000,000 bytes"
```

```r
# Convert bytes to megabytes (MB) - there are 2^20 bytes/MB.
megabytes <- bytes/2^20
megabytes_unit <- "MB"
megabytes_formatted <- format(megabytes, scientific = FALSE, big.mark = ",")
formatted_megabytes <- sprintf("%s %s", megabytes_formatted, megabytes_unit)
formatted_megabytes
```

```
## [1] "1,373.291 MB"
```

```r
# Convert MB to GB - divide by 1000.
ram_required <- megabytes/1000
ram_unit <- "GB"
ram_required_formatted <- format(ram_required, scientific = FALSE, big.mark = ",")
formatted_ram_required <- sprintf("%s %s", ram_required_formatted, ram_unit)
formatted_ram_required
```

```
## [1] "1.373291 GB"
```