Statistical Computing

Lecture 1: R basics

Yanfei Kang yanfeikang@buaa.edu.cn

School of Economics and Management Beihang University

Objectives

- Overview of R
- R nuts and bolts
- Getting data in and out of R
- Subsetting R objects

Overview of R

What is R?

- A freely available language and environment
- Statistical computing and graphics
- Linear and nonlinear modelling, statistical tests, time series analysis, classification, clustering, etc.

Installation

- Install R
- Install Rstudio

Why Rstudio?

- Syntax highlighting
- Able to evaluate R code
  - by line
  - by selection
  - entire file
- Command auto-completion

Design of the R System

- When you download R from CRAN, you get the "base" system - a substantial amount of functionality.

- 10,000 packages on CRAN that have been developed by users and programmers around the world.

- People often make packages available on their personal websites.

- There are a number of packages being developed on repositories like GitHub and BitBucket.

R Nuts and Bolts

Basic Operations

```
1 + 2 + 3
## [1] 6
1 + 2 * 3
## [1] 7

x <- 1
y <- 2
z <- c(x, y)
z
## [1] 1 2

exp(1)
## [1] 2.718282
cos(3.141593)
## [1] -1
log2(1)
## [1] 0
```

R Objects

R has five basic classes of objects:

1. character
2. numeric (real numbers)
3. integer
4. complex
5. logical (True/False)

Numbers

- Numbers in R are generally treated as numeric objects.
- Difference of `1` and `1L`?
- Special number `Inf`. Try `1/Inf`.
- `NaN`: an undefined value (not a number). Try `0/0`. It can also be thought of as a missing value.

Attributes

Attributes can be accessed by `attributes()`. Some examples of R object attributes are:

- names, dimnames
- dimensions (e.g. matrices, arrays)
- class (e.g. integer, numeric)
- length

Vectors

The `c()` function can be used to create vectors of objects by concatenating things together.

```r
x <- c(0.5, 0.6)  ## numeric
x <- c(TRUE, FALSE)  ## logical
x <- c(T, F)  ## logical
x <- c("a", "b", "c")  ## character
x <- 9:29  ## integer
x <- c(1 + (0+0i), 2 + (0+4i))  ## complex
```

You can also use the `vector()` function to initialize vectors.

```r
x <- vector("numeric", length = 10)
x
##  [1] 0 0 0 0 0 0 0 0 0 0
```

Matrices

```r
m <- matrix(c(1:6), 2, 3)
attributes(m)
## $dim
## [1] 2 3
dim(m)
## [1] 2 3
t(m)
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
## [3,]    5    6
m[1, 2]
## [1] 3
m[1, ]
## [1] 1 3 5
n <- matrix(c(8:13), 2, 3)
cbind(m, n)
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    3    5    8   10   12
## [2,]    2    4    6    9   11   13
rbind(m, n)
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
## [3,]    8   10   12
## [4,]    9   11   13
```

Lists

- Special data structure that matrix could not handle.
  - Data length are not the same.
  - Data type are not the same.

```r
l <- list(a = c(1, 2), b = "apple")
attributes(l)
## $names
## [1] "a" "b"
```

Factors

Factors are used to represent categorical data.

```
f <- factor(c("yes", "yes", "no", "yes", "no"))
attributes(f)
## $levels
## [1] "no"  "yes"
##
## $class
## [1] "factor"
```

Data Frames

- A special type of list.
- Unlike matrices – data frames can store different classes of objects in each column.
- They have column names and row names.

```
d <- data.frame(x = 1:10, y = letters[1:10])
attributes(d)
## $names
## [1] "x" "y"
##
## $class
## [1] "data.frame"
##
## $row.names
##  [1]  1  2  3  4  5  6  7  8  9 10
names(d)
## [1] "x" "y"
row.names(d)
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

Names

Names are very useful for writing readable code and self-describing objects.

```
x <- 1:3
names(x)
## NULL
names(x) <- c("New York", "Seattle", "Los Angeles")
x
##    New York     Seattle Los Angeles
##           1           2           3
names(x)
## [1] "New York"    "Seattle"     "Los Angeles"
```

Lists can also have names, which is often very useful.

```
x <- list(`Los Angeles` = 1, Boston = 2, London = 3)
x
## $`Los Angeles`
## [1] 1
##
```

```
## $Boston
## [1] 2
##
## $London
## [1] 3
names(x)
## [1] "Los Angeles" "Boston"      "London"
```

Getting Data in and out of R

Reading and Writing Data

There are a few principal functions reading data into R.

- `read.table`, `read.csv`, for reading tabular data
- `readLines`, for reading lines of a text file
- `source`, for reading in R code files (`inverse` of `dump`)
- `dget`, for reading in R code files (`inverse` of `dput`)
- `load`, for reading in saved workspaces

There are analogous functions for writing data to files.

- `write.table`, for writing tabular data to text files (i.e. CSV) or connections
- `writeLines`, for writing character data line-by-line to a file or connection
- `dump`, for dumping a textual representation of multiple R objects
- `dput`, for outputting a textual representation of an R object
- `save`, for saving an arbitrary number of R objects in binary format (possibly compressed) to a files

There are many R packages that have been developed to read in all kinds of other datasets (e.g., the `readr` package).

Subsetting R objects

How to Subset?

There are three operators that can be used to extract subsets of R objects.

- The `[` operator always returns an object of the same class as the original. It can be used to select multiple elements of an object

- The `[[` operator is used to extract elements of a list or a data frame. It can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame.

- The `$` operator is used to extract elements of a list or data frame by literal name. Its semantics are similar to that of `[[`.

Subsetting a Vector

Vectors are basic objects in R and they can be subsetted using the `[` operator.

```
x <- c("a", "b", "c", "c", "d", "a")
x[1]  ## Extract the first element
## [1] "a"
```

```
x[2]  ## Extract the second element
## [1] "b"
```

The [operator can be used to extract multiple elements of a vector by passing the operator an integer sequence. Here we extract the first four elements of the vector.

```
x[1:4]
## [1] "a" "b" "c" "c"
x[c(1, 3, 4)]
## [1] "a" "c" "c"
x[x > 2]
## [1] "a" "b" "c" "c" "d" "a"
```

Subsetting a Matrix

Matrices can be subsetted in the usual way with (i,j) type indices.

```
x <- matrix(1:6, 2, 3)
x
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

We can access the $(1, 2)$ or the $(2, 1)$ element of this matrix using the appropriate indices.

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

Indices can also be missing. This behavior is used to access entire rows or columns of a matrix.

```
x[1, ]  ## Extract the first row
## [1] 1 3 5
x[, 2]  ## Extract the second column
## [1] 3 4
```

Subsetting Lists

ists in R can be subsetted using all three of the operators mentioned above, and all three are used for different purposes.

```
x <- list(foo = 1:4, bar = 0.6)
x
## $foo
## [1] 1 2 3 4
##
## $bar
## [1] 0.6
```

The [[operator can be used to extract *single* elements from a list. Here we extract the first element of the list.

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

The `[[` operator can also use named indices so that you don't have to remember the exact ordering of every element of the list. You can also use the `$` operator to extract elements by name.

```
x[["bar"]]
## [1] 0.6
x$bar
## [1] 0.6
```

Subsetting Nested Elements of a List

The `[[` operator can take an integer sequence if you want to extract a nested element of a list.

```
x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
## Get the 3rd element of the 1st element
x[[c(1, 3)]]
## [1] 14
## Same as above
x[[1]][[3]]
## [1] 14
## 1st element of the 2nd element
x[[c(2, 1)]]
## [1] 3.14
```

Extracting Multiple Elements of a List

The `[` operator can be used to extract *multiple* elements from a list. For example, if you wanted to extract the first and third elements of a list, you would do the following

```
x <- list(foo = 1:4, bar = 0.6, baz = "hello")
x[c(1, 3)]
## $foo
## [1] 1 2 3 4
##
## $baz
## [1] "hello"
```

Note that `x[c(1, 3)]` is NOT the same as `x[[c(1, 3)]]`.

Remember that the `[` operator always returns an object of the same class as the original. Since the original object was a list, the `[` operator returns a list. In the above code, we returned a list with two elements (the first and the third).

Removing NA Values

A common task in data analysis is removing missing values (`NA`s).

```
x <- c(1, 2, NA, 4, NA, 5)
bad <- is.na(x)
print(bad)
## [1] FALSE FALSE  TRUE FALSE  TRUE FALSE
x[!bad]
## [1] 1 2 4 5
```

What if there are multiple R objects and you want to take the subset with no missing values in any of those objects?

```
head(airquality)
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
good <- complete.cases(airquality)
head(airquality[good, ])
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 7    23     299  8.6   65     5   7
## 8    19      99 13.8   59     5   8
```

Review of this lecture

- Overview of R
- R nuts and bolts
- Getting data in and out of R
- Subsetting R objects

Lab Session 1

Read and Write Data in R

You'll be working with swimming_pools.csv; it contains data on swimming pools in Brisbane, Australia (Source: data.gov.au). The file contains the column names in the first row. It uses a comma to separate values within rows.

1. Try `read.csv()` and `read.table()` to import "swimming_pools.csv" as a data frame with the name `pools`.
2. Try `write.table()`, `dput()`, and `save()` functions to write `pools` to files.
3. Restart R and read your saved data in R.
4. Practice subsetting of a data frame.

References

Chapters 3-10 of the book "R programming for data science".