R Programming

Furkan Oguz

4/20/2020

R Porgramming

Week 1: Getting Started and R Nuts and Bolts

Writing Code/Setting Your Working Directory

In R (not R Studio): How to set my working directory and how to edit R code files using the text editor.

You can find out what your working directory is by using following code:

```
getwd()
```

```
## [1] "/Users/furkanoguz/Desktop/R Datascience/datascience_course"
```

To change the working directory: Go to (above) Misc > Change Working Directory.

Code to list all files within directory:

```
dir()
```

```
## [1] "data.R"                      "datascience_course.Rproj"
## [3] "hw1_data.csv"                "Notices R Programming.Rmd"
## [5] "Notices-R-Programming.pdf" "Notices-R-Programming.Rmd"
## [7] "Week 1, Quiz.R"              "y.R"
```

When you want to look if there are objects in your work space:

```
ls()
```

```
## character(0)
```

You can open a text editor in R with the white paper sign. To run it you can copy the content and run on R or you can save the file as .r-file. To source the file you saved, you can source it within the directory with the following code:

```
## source("filename.R")
```

Learning Objectives

- Install R and RStudio software packages
- Download and install the swirl package for R
- Describe the history of the S and R programming lectures
- Describe the differences between atomic data types
- Execute basiv arithmetic operations
- Subset R objects using the "[", "[[" and "$" operators and logical vectors
- Describe the explicit coercion feature of R
- Remove missing (NA) values from a vector

Introduction

Course specializiation is to get the nuts and bolts of R programming!

Overview and History of R

What is R? It is a dialect of S. **What is S?** It is a language initiated as an internal statistical analysis environment (originally implemented as Fortran libraries -> Matlab) In earliest version did not contain functions for statistical modelling. Later it was rewritten in C to make it more compatibel. A later version (1998) was that one we use today.

S Philosohpy: Would enter language by beginning in an interactive environment. So sliding into programming!

Back to R: Invented in 1991. First public announcement 1993. In 1995 it is a free software!

Features of R:

- Syntax is very similar to S
- Semantics are superficially similar to S, but in reality quite different.
- Runs on almost any standard computing platform/OS
- Frequent releases and active development!
- Quite lean, functionality is divided into modular packages
- Graphicscapabilities very sophisitcated and better than most stat packages
- Useful for interactice work, but contains a powerful programming language for developing new tools (user -> programmer)
- Very active and vibrant user community (Stack Overflow)
- It's free

Free Software: * Freedom to run the program, for any purpose (freedom 0) * Freedom to study how the program works, and adapt it to your needs (freedom 1). Acces to the source code. * Freedom to redistribute copies so you can help your neighbour (freedom2) * Freedom to improve the program, and release your improvements to the public, so that the whole community benefits (freedom 3).

Drawbacks of R: * 40 year old technology * Little built-in support for dynamic or 3D graphis (but improved greatly) * Functionality is based on consumer demand and user contributions * Objects must be stored in physical memory (but great advancements) * Not ideal for all possible situations (but for all software packages same problem)

Design of the R System: It is divided into 2 conceptual parts:

- "Base" R system that you download from CRAN
- Everything else R functionality is divided into a number of packages:
- "Base" system contains base package with most fundamental functions
- Other packages contained in the "base" system include **utils, stats, datasets, graphics, grDevices, grid, methods, tools, parallel, compiler, splines, tcltk, stats4**
- Also "recommended" packages: **boot, class, cluster, codetools, foreign, KernSmooth, lattice, mgcv, nlme, rpart, survival, MASS, spatial, nnet, Matrix**
- Many other packages (4'000)

Getting Help

Email: r-help@r-project.org

R Console Input and Evaluation

Entering Input: Things we type in R prompt are called expressions. The sympobl looking like left-hand-arrow is actually the assignment operator.

```r
x <- 1
print(x)
```

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

```r
msg <- "hello"
## z <- ## Incomplete expression
```

This means tha value right is assigned to variable left. The last example is an error, because it is a comment which R doesn't use for computation.

Evaluation: When expression is entered in prompt, it is evaluated and the result of evaluated expression is returned. Result may be auto-printed.

```r
x <- 5 ## nothing printed
x ## auto-printing occurs (typing objects name!)
```

```
## [1] 5
```

```r
print(x) ## explicit printing
```

```
## [1] 5
```

Printing: The : operator is use to create integer sequences. In this case integer vector

```r
x <- 1:20
x
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

Data Types - R Objects and Attributes

Objects: All the things we that we encounter in R. There are different kind of data, but everything is an object. R has 5 basic or "atomic" classes of objects:

- character
- numeric (real numbers)
- integer
- complex
- logical (True/False)

The most basic object is a vector:

- Can only obtain objects of the same class
- BUT: the one exception is a *list*, which is represented as vector, but can contain objects of different classes (reason why usually used, you can even have a list within a list, very useful!)

How to create empty vector by using vector function:

```r
vector()
```

```
## logical(0)
```

Numbers: 3 Numbers are treated as numeric objects (double precision real numbers)

- If explicitly integers were wished, you need to specify the L suffix.
- Example: Entering 1 gives numeric object; entering 1L gives integer
- There is also a special number: Inf which is infinity like 1/0. It can be used in ordinary calculations like 1/Inf which is 0.
- The value NaN is undefined value ("not a number"), 0/0 is NaN. Can also be thaught as a missing value.

Attributes: Objects can have attributes

- names, dimnames
- dimensions (matrices, arrays so on)
- class
- length
- other user-defined attributes/metadata

Attributes of an object can be acceses with **attributes() function**!

Data Types - Vectors and Lists

Creating Vectors:

The **c() function** can also be used to create vectors and objects

```
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+0i, 2+4i) ##complex
```

Using the **vector()** function:

```
x <- vector("numeric", length=20)
x
```

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

Mixing Objects:

```
y <- c(1.7, "a") ##character
y <- c(TRUE, 2) ##numeric
y <- c("a", TRUE) ##character
```

When different objects are mixed in a vector, *coercion* occurs so that every element in the vector is of the same class!! (TRUE is a 1, and FALSE is a 0, they will be converted as a number)

Explicit Coercion: Objects can be explicitly coerced from one class to another using the as.*functions, if available.

```
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 NAS.

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

```
## Warning: NAs durch Umwandlung erzeugt
```

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

```
as.logical(x)
```

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

```
as.complex(x)
```

```
## Warning: NAs durch Umwandlung erzeugt
```

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

Lists: Lists are a special type of vector that can contain elements of different classes. Lists are a very important data type in R!!

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

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 1+4i
```

Lists have for index, always double brackets on them!

Data Types - Matrices

Matrices are vectors with a dimension attribute (itself an integer vector of length 2, **nrow** and **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
```

It 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
```

Can be created by *column-binding* or *row-binding* with **cbind()** and **rbind()**

```
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
```

Data Types -Factors

Factors are used to represent categorical data. 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 **lm()** and **glm()**
- Using factors with labels is *better* than using integers because factors are self-decribing; having a variable that has values “Male” and “Female” is better than a variable that has values 1 and 2.

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

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

```r
table(x)
```

```
## x
##  no yes
##   2   3
```

```r
unclass(x) ##brings an integer vector
```

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

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

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

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

Data Types - 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** etc.
- A **NaN** value is also a **NA** but the converse is not true

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

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

```r
is.nan(x)
```

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

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

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

```r
is.nan(x)
```

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

Data Types - Data Frames

Data frames are used to store tabular data

- They are represented as a special type of list where every element of the list has to 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 is the number of rows
- Unlike matrices, data frames can store different class of objects in each column (like lists); matrices must have every element be the same class
- Data frames also have a special attribute called **row.names**

- Are usually created by calling **read.table(*) **or** read.csv()**
- Can be converted to a matrix by calling **data.matrix()**

```
x <- data.frame(foo=1:4, bar=c(T, T, F, F))
x
```

```
##   foo   bar
## 1   1  TRUE
## 2   2  TRUE
## 3   3 FALSE
## 4   4 FALSE
```

```
nrow(x)
```

```
## [1] 4
```

```
ncol(x)
```

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

Data Types - Names Attribute

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

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

```
## NULL
```

```
names(x) <- c("foo", "bar", "norf")
x
```

```
##  foo  bar norf
##    1    2    3
```

```
names(x)
```

```
## [1] "foo"  "bar"  "norf"
```

Lists can also have names.

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

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

And matrices.

```
m <- matrix(1:4, nrow=2, ncol=2)
dimnames(m) <- list(c("a", "b"), c("c", "d"))
m
```

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

Data Types - Summary

- atomic classes: numeric, logical, character, integer, complex
- vectors, lists
- factors
- missing values
- data frames
- names

Reading Tabular 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 R code files (*inverse of dput*) (it's for reading R objects that have been dparsed into text files)
- **load**, for reading in saved workspaces
- **unserialize**, for reading single R objects in binary form

There are analogous functions for writing data to files:

- **write.table**
- **writeLines**
- **dump**
- **dput**
- **save**
- **serialize**

Reading Data Files with *read.table* (very important!!): It is the most commonly used functions for reading data. It has a few important arguments:

- **file**, the name of a file, or a connection
- **header**, logical 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
- **strinsAsFactors**, should character variables be coded as factors?

For small to moderately sized datasets, you can usually call read.table without specifying any other arguments

```
## data <-read.table("foo.txt")
```

R will automatically:

- skip lines that begin with a #
- figure out how many rows there are (and how much memory needs to be allocated)
- figure what type of variable is in each column of the table. Telling R all these things directly makes R run faster and more efficiently
- **read.csv** is identical to **read.table** except that the default separator is a comma (vice versia with space)

Reading Large Tables

With mnuch 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
- Make a rough calculation of the memory required to store your dataset. If the dataset is larger than the amount of RAM on your computer, you can probably stop right there.
- Set **comment.char** ="" if there are no commended lines in your files.

Reading in Larger Datasets with read.table

- Use the **colClasses** argument. Specifying this option instead of default makes it run faster, often twice as fast. For this you need to know the class of each column in your data frame. If all columns are "numeric" then you can set **colClasses = "numeric"**. A fast, dirty way to figure out the classes of each column is:

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

- set **nrows**. It doesn't make it run faster but it helps with memory usage. A mild overestimate is ok. We can use Unix tool **wc** to calculate the number of lines in a file.

Know the System When using R with larger datasets, it's useful to knew followin things about our system.

- How much memory is available
- What other applications are in use?
- Are there other users logged into the same system?
- What operating system?
- Is the OS32 or 64 bit?

Textual Data Formats

Textual Formats

- **dumping** and **dputing** are useful because the resulting textual format is edit-able, and in the case of corruption, potentially recoverable
- Unlike writing out a table or csv file, **dump** and **dput** preserve the metadata (sacrificing some readability), so that another user doesn't have to specify it all over again
- Textual formats can work much better with version control programs like subversion or git which can only track changes meaningfully in text files
- Textual formats can be longer-lived; if there is courruption somewhere in the file, it can be easier to fix the problem
- Textual formats adhere to the "Unix philosophy"
- Downside: Format is not very space-efficient…

dput-ing R Objects Another way to pass data around is by deparsing the R object with **dput** and reading it back in using **dget**

```
y <- data.frame(a=1, b="a")
dput(y)
```

```
## structure(list(a = 1, b = structure(1L, .Label = "a", class = "factor")), class = "data.frame", row.
## -1L))
```

```r
dput(y, file= "y.R")
new.y <- dget("y.R")
new.y
```

```
##   a b
## 1 1 a
```

Dumping R Objects Multiple objects can be deparsed using the dump function and read back in using **source**.

```r
x <- "foo"
y <- data.frame(a=1, b="a")
# dump(c("x", "y"), file="data.R")
y
```

```
##   a b
## 1 1 a
```

```r
x
```

```
## [1] "foo"
```

Connections: Interfaces to the Outside World

Data are read in using *connection* interfaces. Connections ca be made to files (most coomon) or to other more exotic things.

- **file**, opens a connection to a file
- **gzfile**, opens a connection to a file compressed with gzip
- **bzfile**, ioens a connection to a file compressed with bzip2
- **url**, opens a connection to a webpage

File Conneections

```r
str(file)
```

```
## function (description = "", open = "", blocking = TRUE, encoding = getOption("encoding"),
##     raw = FALSE, method = getOption("url.method", "default"))
```

- **description** is the name of the file
- **open** is a code indicating
  - "r" read only
  - "w" writing (and initializing a new file)
  - "a" appending
  - "rb", "wb", "ab" reading, writing or appending in the binary mode (Windows)

Connections In general, powerful tools that let you navigate files or other external objects. In practice, we often don't need to deal with the connection interface directly.

```r
## con <- file("foo.txt", "r")
## data <- read.csv(con)
## close con
```

is the same as

```r
## data <- read.csv("foo.txt")
```

Reading Lines of Text File

```r
## con <- gzfile("words.gz")
## x <- readLines(con, 10)
## x
```

writeLines takes a character vector and writes each element one line at a time to a text file

Also possible for webpages!

```r
con <- url("https://www.jhsph.edu", "r")
x <- readLines(con)
head(x)
```

```
## [1] "<!DOCTYPE html>"
## [2] "<html lang=\"en\">"
## [3] ""
## [4] "<head>"
## [5] "<meta charset=\"utf-8\" />"
## [6] "<title>Johns Hopkins Bloomberg School of Public Health</title>"
```

Subsetting - Basics

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

- [always returns an object of the same class as the original; can be used to select more than one element (there is one exception)
- [[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
- $ is used to extract elements of a list or data frame by name; semantics are similar to hat of [[

```r
x <- c("a", "b", "c", "c", "d", "a")
x[1]
```

```
## [1] "a"
```

```r
x[2]
```

```
## [1] "b"
```

```r
x[1:4]
```

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

```r
x[x > "a"]
```

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

```r
u <- x > "a" #It is a logical vector, which tells which elements are greater than "a"
u
```

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

```r
x[u]
```

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

Subsetting - Lists

```
# Subsetting Lists
x <- list(foo=1:4, bar=0.6)
x[1]
```

```
## $foo
## [1] 1 2 3 4
```

```
x[[1]]
```

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

```
x$bar
```

```
## [1] 0.6
```

```
x[["bar"]]
```

```
## [1] 0.6
```

```
x["bar"]
```

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

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

```
## $foo
## [1] 1 2 3 4
##
## $baz
## [1] "hello"
```

The [[operator can be used with *computed* indices; $ can only used with literal names

```
x <- list(foo=1:4, bar=0.6, baz="hello")
name <- "foo"
x[[name]] ## computed index for 'foo'
```

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

```
x$name ## element 'name' doesn't exist!
```

```
## NULL
```

```
x$foo ## element 'foo' does exist
```

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

```
x <- list(a=list(10, 12, 14), b=c(3.14, 2.81))
x[[c(1, 3)]]
```

```
## [1] 14
```

```
## x[[1]][[3]]
x[[c(2, 1)]]
```

```
## [1] 3.14
```

Subsetting - Matrices

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

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

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

```
x[2, 1]
```

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

```
x[1, ]
```

```
## [1] 1 3 5
```

```
x[, 2]
```

```
## [1] 3 4
```

Indices can also be missing. above

By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1x1 matrix. This behavior can be turned off by setting **drop=FALSE**

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

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

```
x[1, 2, drop=FALSE]
```

```
##      [,1]
## [1,]    3
```

Similarly subsetting a single column or row will give you a vector, not a matrix (by default).

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

```
## [1] 1 3 5
```

```
x[1, , drop=FALSE]
```

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

Subsetting - Partial Matching

Partial matching of names is allowed with [[and $

```
x <- list(aardvark=1:5)
x$a
```

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

```
x[["a"]]
```

```
## NULL
```

```
x[["a", exact=FALSE]]
```

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

To save a lot of typing and making it faster. You can shortcut with letters if only one letter within list is associated with an element!

Subsetting - Removing Missing Values

Removing NA Values A common task is to remove missing values (NA).

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

```
## [1] 1 2 4 5
```

What if there are multiple things and you want to take the subset with no missing values? work with *complete.cases* (logical vector!)!

```
x <- c(1, 2, NA, 4, NA, 5)
y <- c("a", "b", NA, "d", NA, "f")
good <- complete.cases(x, y)
good
```

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

```
x[good]
```

```
## [1] 1 2 4 5
```

```
y[good]
```

```
## [1] "a" "b" "d" "f"
```

Vectorized Operations

Many operations in R are *vectorized* making code more efficient, concise, and easier to read.

```
x <- 1:4; y <- 6:9
x+y
```

```
## [1]  7  9 11 13
```

```
x > 2
```

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

```
y==8
```

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

```
x*y
```

```
## [1]  6 14 24 36
```

```
x/y ## Not an Inverse!!!
```

```
## [1] 0.1666667 0.2857143 0.3750000 0.4444444
```

Vectorized Matrix Operations

```r
x <- matrix (1:4, 2, 2); y <- matrix(rep(10, 4), 2, 2)
x*y ##element-wise multiplication
```

```
##      [,1] [,2]
## [1,]   10   30
## [2,]   20   40
```

```r
x/y
```

```
##      [,1] [,2]
## [1,]  0.1  0.3
## [2,]  0.2  0.4
```

```r
x %*% y ## true matrix multiplication
```

```
##      [,1] [,2]
## [1,]   40   40
## [2,]   60   60
```