# Data Analysis with R for Software Engineers (Intro)

## About R Programming Language

Before diving into the R language, it is important to mention about other Statistical Software Packages. At the end of the day, R is not a general purpose programming language, but rather a programming language that its aimed to help for statistical analysis.

### R vs Other Statistical Software Packages

#### SPSS

SPSS (originally "Statistical Package for the Social Sciences") is mostly used in Social Sciences and uses a graphical user interface (GUI), though historically, it was a command-line program. It was one of the earliest statistical packages, with the first release being in 1968. SPSS was acquired in 2009 by IBM, and between 2009 and 2010, the product was referred to as PASW (Predictive Analytics SoftWare). As of January 2010, it became "SPSS: An IBM Company".

#### SAS

The SAS (previously "Statistical Analysis System") system dominates the commercial market. It was developed at North Carolina State University from 1966 until 1976, when SAS Institute was incorporated. SAS is a really expensive software. The pharmaceutical industry uses almost exclusively SAS.

#### Stata

Stata is a more recent statistical package with the first version being released in 1985. Since then, it has become increasingly popular in the areas of epidemiology and economics, and probably now rivals SPSS and SAS in it user base.

**S**

S is a statistical programming language developed between 1975 and 1976 primarily by John Chambers and (in earlier versions) Rick Becker and Allan Wilks of Bell Laboratories. The aim of the language, as expressed by John Chambers, was "to turn ideas into software, quickly and faithfully."

The S engine was licensed to and finally purchased by Insightful (now acquired by TIBCO) in 1988, which sells a value-added version called S-Plus containing a graphical user interface. S-Plus used to dominate the high-end market (academic and industrial research).

**R**

R is an alternative implementation of the S programming language combined with lexical scoping semantics, started in 1991 by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand. The first official release came in 1995. R is currently now maintained by the R Core Team. R is a dynamically typed and interpreted programming language. At its heart, R is a functional programming language, but the R system includes some support for object-oriented programming (OOP). The official R software environment is written primarily in C, Fortran, and R itself, and is freely available under the GNU General Public License.

R is comparable to SAS, SPSS, and Stata, but R is available to users at no charge under a free software license. R is more procedural-code oriented than either SAS or SPSS. R generally processes data in-memory, which limits its usefulness in processing extremely large files.

## R vs Other Programming Languages

R is usually compared to other languages in the Data Science world, it is particularly compared to Python, and more recently to Julia.

### R vs Python

We will not discuss much about the differences between R and Python, there is already too much controversy, and a lot of information is already online. However, it is worth noting that one of the main advantages from R over Python is its ease to use for time series and forecasting.

### R vs Julia

The differences from R vs Julia are much more interesting. To start with, Julia is a compiled language, whereas Python and R are interpreted. Julia is a multiple-paradigm (fully imperative, partially functional, and partially object-oriented) programming language designed for scientific and technical (read numerical) computing. I have read that Julia gives you C-like speed without optimization and hand-crafted profiling techniques, which makes it really interesting. Julia was created in 2009 by a four-person team and unveiled to the public in 2012, which makes it a relatively new language. Julia's syntax is similar to Python's—terse, but also expressive and powerful.

## RStudio

RStudio is an integrated development environment (IDE) to help you be more productive with R. It not only provides an editor for us to create and edit our R code but also provides many other useful tools which we will see later. The newest version of RStudio (1.4.11) improved support for Python, including an environment pane for Python and visualization of Python objects. This means that you can interchange data objects between R and Python in the same session.

```
# {r cars}
cars <- head(mtcars)
```

```
# {python}
import pprint
pp = pprint.PrettyPrinter(width=41, compact=True)

def print_df(df):
    pp.pprint(df)

print_df(r.cars)
```

```
## {'am': [1.0, 1.0, 1.0, 0.0, 0.0, 0.0],
##  'carb': [4.0, 4.0, 1.0, 1.0, 2.0, 1.0],
##  'cyl': [6.0, 6.0, 4.0, 6.0, 8.0, 6.0],
##  'disp': [160.0, 160.0, 108.0, 258.0,
##           360.0, 225.0],
##  'drat': [3.9, 3.9, 3.85, 3.08, 3.15,
##           2.76],
##  'gear': [4.0, 4.0, 4.0, 3.0, 3.0, 3.0],
##  'hp': [110.0, 110.0, 93.0, 110.0, 175.0,
##         105.0],
##  'mpg': [21.0, 21.0, 22.8, 21.4, 18.7,
##          18.1],
##  'qsec': [16.46, 17.02, 18.61, 19.44,
##           17.02, 20.22],
##  'vs': [0.0, 0.0, 1.0, 1.0, 0.0, 1.0],
##  'wt': [2.62, 2.875, 2.32, 3.215, 3.44,
##         3.46]}
```

# Data Types

Atomic data types in R can be divided into:

- `numeric` (9.2, 11, 232)
- `integer` (1L, 10L, 100L, where the letter "L" declares this as an integer)
- `complex` (1 + 3i, where "i" is the imaginary part)
- `character/string` ("l", "R is the best", "TRUE", "9.2")
- `logical/booleans` (TRUE or FALSE, T or F)

```r
x <- 10.5
class(x)
# [1] "numeric"

x <- 1000L
class(x)
# [1] "integer"

x <- 9i + 3
class(x)
# [1] "complex"

x <- "R is exciting"
class(x)
# [1] "character"

x <- TRUE
class(x)
# [1] "logical"
```

## Data Structures

- Vectors
- Matrices
- Lists
- Data Frames
- Factors

### Vectors

```
x <- c(0.5, 0.6)      ## numeric
str(x)
# num [1:2] 0.5 0.6

x <- c(TRUE, FALSE)   ## logical
str(x)
# logi [1:2] TRUE FALSE

x <- c("a", "b", "c") ## character
str(x)
# chr [1:3] "a" "b" "c"

x <- 9:29
str(x)
# int [1:21] 9 10 11 12 13 14 15 16 17 18 ...
```

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

```
x <- c(TRUE, 1, "a")   ## character
str(x)
# chr [1:3] "TRUE" "1" "a"

x <- c(TRUE, 2)        ## numeric
# num [1:2] 1 2
```

You can also coerce into a different data type if you want:

```
x <- c(1, 2, 3)
str(x)
# num [1:3] 1 2 3

str(as.character(x))
# chr [1:3] "1" "2" "3"
```

### Matrices

Matrices are vectors with a dimension attribute. Internally, R stores matrices in column-major mode, unlike C or Java where it would be stored in what is called "row-major mode", i.e. the rows would be stacked on top of each other.

```
x <- matrix(1, nrow = 2, ncol = 2)
x
#      [,1] [,2]
# [1,]    1    1
# [2,]    1    1

dim(x)
# [1] 2 2

matrix(1:6, ncol=2)
#      [,1] [,2]
# [1,]    1    4
# [2,]    2    5
# [3,]    3    6
```

You can combine two or more vectors into a matrix:

```
x <- 2:4
y <- 1:3
z <- 10:12

# column bind
cbind(x, y, z)
#      x y  z
# [1,] 2 1 10
# [2,] 3 2 11
# [3,] 4 3 12

# row bind
m <- rbind(x, y, z)
#   [,1] [,2] [,3]
# x    2    3    4
```

```
# y    1    2    3
# z   10   11   12

# Transpose of a matrix
t(m)
#      x y  z
# [1,] 2 1 10
# [2,] 3 2 11
# [3,] 4 3 12

# Matrices can only have one type, so data will be coerced
# when combining vectors from different types
rbind(x, y, z=c("a", "b", "c"))
#   [,1] [,2] [,3]
# x "2"  "3"  "4"
# y "1"  "2"  "3"
# z "a"  "b"  "c"

m <- matrix(1:9, ncol=3)
m
#      [,1] [,2] [,3]
# [1,]    1    4    7
# [2,]    2    5    8
# [3,]    3    6    9

col(m)
#      [,1] [,2] [,3]
# [1,]    1    2    3
# [2,]    1    2    3
# [3,]    1    2    3

row(m)
#      [,1] [,2] [,3]
# [1,]    1    1    1
# [2,]    2    2    2
# [3,]    3    3    3
```

```r
m[row(m) == col(m)]
# [1] 1 5 9
```

### Lists

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

```r
x <- list(1, "abc", name=TRUE)
x
# [[1]]
# [1] 1

# [[2]]
# [1] "abc"

# $name
# [1] TRUE

x[1]
# [[1]]
# [1] 1

x[[1]]
# [1] 1

x["name"]
# $name
# [1] TRUE

x[["name"]]
# [1] TRUE

x$name
# [1] TRUE
```

### Data Frames

Data frames are similar to matrices, but they can have different types of data for different columns.

```r
df <- data.frame(x = 1:3, y = c("a", "b", "c"))
df
#   x y
# 1 1 a
# 2 2 b
# 3 3 c

# print the structure
str(df)
# 'data.frame': 3 obs. of  2 variables:
#  $ x: int  1 2 3
#  $ y: Factor w/ 3 levels "a","b","c": 1 2 3

# print the colnames
names(df)
# [1] "x" "y"

colnames(df)
# [1] "x" "y"

# access to the x column
df$x
# [1] 1 2 3

# number of columns
ncol(df)
# [1] 2

# number of rows
nrow(df)
# [1] 3
```

```r
# you can rename the colnames of a dataframe
colnames(df) <- c("foo", "bar")
df
#   foo bar
# 1   1   a
# 2   2   b
# 3   3   c
```

### Factors

Factors are used to represent categorical data. They can be unordered or ordered.

```r
x <- factor(c("good", "bad", "bad", "good"))
x
# [1] good bad  bad  good
# Levels: bad good

levels(x)
# [1] "bad"  "good"

unclass(x)
# [1] 2 1 1 2
# attr(,"levels")
# [1] "bad"  "good"
```

## Missing Data

Missing values are denoted by NA. You can test objects if they are NA using `is.na()`. `NA` values have a class as well, so they are integer NA, character NA, etc.

```r
x <- c("abc", NA, 2, 4, TRUE)
is.na(x)
# [1] FALSE  TRUE FALSE FALSE FALSE

x <- 1:5
mean(x)

# careful when applying certain functions to vectors with NA
```

```r
x <- c(1:5, NA)
x
# [1]  1  2  3  4  5 NA

mean(x)
# [1] NA

mean(x, na.rm = TRUE)
# [1] 3
```

## How to create a vector

Let's say you want to create a vector `x = [2, 10, 11, 1, 23]`. The easiest way to create this vector is through the `c` function:

```r
x <- c(2, 10, 11, 1, 23)
```

In case you want to know what's the current value of variable x, you can either just add the command `x`. Alternative, you can use `print(x)`:

```r
x # this will print the current value of x
## [1]  2 10 11  1 23

print(x) # this will also print it
## [1]  2 10 11  1 23
```

Now, let's say you want to create a vector with a sequence of numbers, such as creating a vector with the numbers from 1 to 10. There are different ways of accomplishing this, such as:

```r
x <- 1:10
x
##  [1]  1  2  3  4  5  6  7  8  9 10
```

In this case `1:10` denotes a way of creating a sequence of numbers from 1 to 10. You can also specify it in decreasing order, in case you want to create a sequence from 10 to 1:

```r
x <- 10:1
x
##  [1] 10  9  8  7  6  5  4  3  2  1
```

You can add to vectors together with the `c()` function:

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

```
x <- c(10, 100, 1000)
y <- c(1, 2, 3)
z <- c(x, y, x)
z
## [1]   10  100 1000    1    23   10  100 1000
```

You can also use a different function to create this vector, with the function `seq`:

```
x <- seq(1:10)
x
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Alternatively, you can use different syntax:

```
x <- seq(from=1, to=10)
x
##  [1]  1  2  3  4  5  6  7  8  9 10
```

So this line `x <- seq(1:10)` is equivalent to `x <- seq(from=1, to=10)`. The latter has some advantages, such as defining other parameters. Let's say that in the sequence you want to have numbers increasing by 2:

```
x <- seq(from=1, to=10, by=2)
x
## [1] 1 3 5 7 9
```

Or, you want exactly 7 numbers:

```
x <- seq(from=1, to=10, length.out=7)
x
## [1]  1.0  2.5  4.0  5.5  7.0  8.5 10.0
```

R will automatically create the equally spaced sequence in the [1, 10] bounds. To see more parameters of seq function, check the R help (`?seq`).

There are other way of specifying vectors, with the funcion `rep`. Let's say you want to

create a vector of length 10, all of them being 1:

```
x <- rep(1, 10)
x
##  [1] 1 1 1 1 1 1 1 1 1 1
```

There are other types of sequences you can do with rep, such as:

```
rep(1:3)
## [1] 1 2 3

rep(1:3, times=3)
## [1] 1 2 3 1 2 3 1 2 3

rep(1:3, each=3)
## [1] 1 1 1 2 2 2 3 3 3
```

# Differences between R and other Programming Languages

## Vector indices in R start with 1, instead of 0

```
x <- c(2, 10, 11, 1, 23)
x[1]
## [1] 2
```

## Assignment using <- instead =

Even though assigning a value using = will work, the <- symbol is the preferred by the R ecosystem. You can also use the symbol -> to assign a value from left to right, but it is not considered a good practice.

```
x <- "hello"
x
## [1] "hello"

x = "world"
x
## [1] "world"

x -> y
y
## [1] "world"
```

## & and | instead of && and ||

The shorter ones are vectorized, meaning they can return a vector, like this:

```
((-2:2) >= 0) & ((-2:2) <= 0)
# [1] FALSE FALSE  TRUE FALSE FALSE
```

The longer form evaluates left to right examining only the first element of each vector, so the above gives

```r
((-2:2) >= 0) && ((-2:2) <= 0)
# [1] FALSE
```

See section 8.2.17 in The R Inferno, titled "and and andand" for more information in this StackOverflow question.

## if vs ifelse

I still remember the hour that I spent looking at the code and trying to figure out what was wrong. `if` will work normally but does not support vectorized operations.

```r
# normal operation
if (10 > 5) {
  print("yes, 10 is > than 5")
}
# [1] "yes, 10 is > than 5"

# this one will throw a warning since only the first element of the vector
# (10) will be evaluated
if (c(10, 2) > 5) {
  print(TRUE)
} else {
  print(FALSE)
}
# [1] TRUE
## Warning in if (c(10, 2) > 5) {:
## the condition has length > 1 and only the first## element will be used

# this will be TRUE for the first element of the vector,
# and FALSE for the second
ifelse (c(10, 2, 50) > 5, TRUE, FALSE)
# [1]  TRUE FALSE TRUE
```

## Packages

R packages are extensions to the R statistical programming language. R packages contain code, data, and documentation in a standardized collection format that can be installed by users of R, typically via a centralized software repository such as CRAN (the Comprehensive R Archive Network). CRAN was officially announced 23 April 1997. It is a network of ftp and web servers around the world that store identical, up-to-date, versions of code and documentation for R. It is not that easy to get a package submitted to the CRAN, so if you develop your own package, other developers will be able to also download them from a remote installation, such as Github, using something like: `remotes::install_github("user/reponame")`, using the `remotes` package. At the time of this writing, there seems to be around ~17000 packages on the CRAN. You can find the increase of submitted packages over time here.

When you install a package using `install.packages()`, the needed resources will be downloaded locally. You can see the placement of such libraries using the function `.libPaths()`.

For example:

```
# checking the current package folder
.libPaths()
# [1] "/Library/Frameworks/R.framework/Versions/3.6/Resources/library"
```

You can change the default folder using `.libPaths(c("/home/myfolder"))`, or setting the `R_LIBS_USER` environment variable inside a `.Renviron` file. See more information here. Alternatively, you can use the `Sys.setenv()` function. To see what `R_LIBS_USER` is set to, you can use `?Sys.getenv()`. Note that if the folder from `R_LIBS_USER` does not exist, it will be ignored (see here).

For more information, see this documentation and this one.

## Loaded Libraries

Every time you restart RStudio, a new session will be created, which means that only default libraries will be loaded. If you want to use a function from an specific, you'll need to load it thorugh the `library()` function. To see which packages are attached to the session, you can use the function `sessionInfo()`.

```r
# load data.table package
library(data.table)

# check into the session
sessionInfo()
```

```
## R version 3.6.2 (2019-12-12)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS Catalina 10.15.6
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/3.6/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/3.6/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] data.table_1.12.8
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_1.0.5      lattice_0.20-38 digest_0.6.25   grid_3.6.2
##  [5] jsonlite_1.7.1  magrittr_1.5    evaluate_0.14   rlang_0.4.7
##  [9] stringi_1.4.6   Matrix_1.2-18   reticulate_1.18 rmarkdown_2.6
## [13] tools_3.6.2     stringr_1.4.0   xfun_0.19       yaml_2.2.1
## [17] compiler_3.6.2  htmltools_0.5.0 knitr_1.29
```

## R Built-in Datasets

Try to start a new session in R (using command line), or through RStudio and type `cars`. You will a dataframe with the `speed` and `dist` columns. Now type `iris`. You will see a dataset with some columns like `Sepal.Length`, `Sepal.Width` and `Species`. You may be wondering where these things are coming from.

R comes with several built-in data sets, which are generally used as demo data for playing with R functions. The `iris` dataset is very popular in the Data Science world and it even has its own Wikipedia page! You can read more about it here.

To see all the built in datasets within R, you can type `data()`.