# R Basics

## Fundamental Techniques in Data Science

Kyle M. Lang

Department of Methodology & Statistics
Utrecht University

# Outline

The R Statistical Programming Language
- Open-Source Software
- What is R?
- Using R
- Project Management

Data I/O
- Writing Data

Functions

Iteration

# Attribution

This course was originally developed by Gerko Vink. You can access the original version of these materials on Dr. Vink's GitHub page: `https://github.com/gerkovink/fundamentals`. The course materials

have been (extensively) modified. Any errors or inaccuracies introduced via these modifications are fully my own responsibility and shall not be taken as representing the views and/or beliefs of Dr. Vink. You can see

Gerko's version of the course on his personal website: www.gerkovink.com/fundamentals

# What is “Open-Source”?

R is an open-source software project, but what does that mean?

- Source code is freely available to anyone who wants it.
  - Free Speech, not necessarily Free Beer
- Anyone can edit the original source code to suit their needs.
  - Ego-less programming
- Many open source programs are also “freeware” that are available free of charge.
  - R is both open-source and freeware

# What is R?

R is a holistic (open-source) software system for data analysis and statistical programming.

- R is an implementation of the S language.
  - Developed by John Chambers and colleagues
    - Becker and Chambers (1984)
    - Becker, Chambers, and Wilks (1988)
    - Chambers and Hastie (1992)
    - Chambers (1998)
- Introduced by Ihaka and Gentleman (1996).
  - Currently maintained by the *R Core Team*.
- Support by thousands of world-wide contributors.
  - Anyone can contribute an R package to the *Comprehensive R Archive Network* (CRAN)
  - Must conform to the licensing and packaging requirements.

# What is R?

I prefer to think about R as a *statistical programming language*, rather than as a data analysis program.

- R **IS NOT** its GUI (no matter which GUI you use).
- You can write R code in whatever program you like (e.g., RStudio, EMACS, VIM, Notepad, directly in the console/shell/command line).
- R can be used for basic (or advanced) data analysis, but its real strength is its flexible programming framework.
    - Tedious tasks can be automated.
    - Computationally demanding jobs can be run in parallel.
    - R-based research *wants* to be reproducible.
    - Analyses are automatically documented via their scripts.

# What is RStudio?

RStudio is an integrated development environment (IDE) for R.

- Adds a bunch of window dressing to R
- Also open-source
- Both free and paid versions

R and RStudio are independent entities.

- You do not need RStudio to work with R.
- You are analyzing your data with R, not RStudio
  - RStudio is just the interface through which you interact with R.

# Getting R

You can download R, for free, from the following web page:

- `https://www.r-project.org/`

Likewise, you can freely download RStudio via the following page:

- `https://www.rstudio.com/`

# How R Works

R is an interpreted programming language.

- The commands you enter into the R *Console* are executed immediately.
- You don't need to compile your code before running it.
- In this sense, interacting with R is similar to interacting with other syntax-based statistical packages (e.g., SAS, STATA, Mplus).

# How R Works

R mixes the *functional* and *object-oriented* programming paradigms.

### Functional

- R is designed to break down problems into functions.
- Every R function is a first-class object.
- R uses pass-by-value semantics.

### Object-Oriented

- Everything in R is an object.
- R functions work by creating and modifying R objects.
- The R workflow is organized by assigning objects to names.

# Interacting with R

When working with R, you will write *scripts* that contain all of the commands you want to execute.

- There is no “clicky-box” Tom-foolery in R.
- Your script can be run interactively or in “batch-mode”, as a self-contained program.

The primary purpose of the commands in your script will be to create and modify various objects (e.g., datasets, variables, function calls, graphical devices).

# Getting Help

Everything published on the Comprehensive R Archive Network (CRAN), and intended for R users, must be accompanied by a help file.

- If you know the name of the function (e.g., `anova()` ), then execute `?anova` or `help(anova)` .
- If you do not know the name of the function, type `??` followed by your search criterion.
  - For example, `??anova` returns a list of all help pages that contain the word "anova".

The internet can also tell you almost everything you'd like to know.

- Sites such as `http://www.stackoverflow.com` and `http://www.stackexchange.com` can be very helpful.
- If you google R-related issues, include "R" somewhere in your search string.

# Packages

Packages give R additional functionality.

- By default, some packages are included when you install R.
- These packages allow you to do common statistical analyses and data manipulation.
- Installing additional packages allows you to perform state-of-the-art statistical analyses.

# Packages

These packages are all developed by R users, so the throughput process is very timely.

- Newly developed functions and software are readily available
- Software implementations of new methods can be quickly disseminated
- This efficiency differs from other mainstream software (e.g., SPSS, SAS, MPlus) where new methodology may take years to be implemented.

A list of available packages can be found on CRAN.

# Installing & Loading Packages

Install a package (e.g., **mice**):

```
install.packages("mice")
```

There are two ways to load a package into R

```
library(stats)
require(stats)
```

# Project Management

Getting a handle on three key concepts will dramatically improve your data analytic life.

1. Working directories
2. Directory structures and file paths
3. RStudio projects

# Data I/O

# R Data & Workspaces

R has two native data formats.

```
## Load the built-in 'bfi' data from the 'psychTools' package
data(bfi, package = "psychTools")

## Access the documentation for the 'bfi' data
?psychTools::bfi

## Define the directory holding our data
dataDir <- "../../../data/"

## Load the 'boys' data from the R workspace
## '../../../data/boys.RData'
load(paste0(dataDir, "boys.RData"))

## Load the 'titanic' data stored in R data set
## '../../../data/titanic.rds'
titanic <- readRDS(paste0(dataDir, "titanic.rds"))
```

# Delimited Data Types

```r
## Load the 'diabetes' data from the tab-delimited file
## '../../../data/diabetes.txt'
diabetes <- read.table(paste0(dataDir, "diabetes.txt"),
                       header = TRUE,
                       sep = "\t")

## Load the 2017 UTMB data from the comma-separated file
## '../../../data/utmb_2017.csv'
utmb1 <- read.csv(paste0(dataDir, "utmb_2017.csv"))
```

NOTES:

- The `read.csv()` function assumes the values are separated by commas.
- For EU-formatted CSV files—with values delimited by semicolons—we can use the `read.csv2()` function.

# SPSS Data

Reading data in from other stats packages can be a bit tricky. If we want to read SAV files, there are two popular options:

- `foreign::read.spss()`
- `haven::read_spss()`

```
## Load the foreign package:
library(foreign)

## Use foreign::read.spss() to read '../../../data/mtcars.sav' into a list
mtcars1 <- read.spss(paste0(dataDir, "mtcars.sav"))

## Read '../../../data/mtcars.sav' as a data frame
mtcars2 <- read.spss(paste0(dataDir, "mtcars.sav"), to.data.frame = TRUE)

## Read '../../../data/mtcars.sav' without value labels
mtcars3 <- read.spss(paste0(dataDir, "mtcars.sav"),
                     to.data.frame = TRUE,
                     use.value.labels = FALSE)
```

# SPSS Data

```
## View the results:
mtcars1[1:3]

$mpg
 [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8
[12] 16.4 17.3 15.2 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5
[23] 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7 15.0 21.4

$cyl
 [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4
[29] 8 6 8 4

$disp
 [1] 160.0 160.0 108.0 258.0 360.0 225.0 360.0 146.7 140.8
[10] 167.6 167.6 275.8 275.8 275.8 472.0 460.0 440.0  78.7
[19]  75.7  71.1 120.1 318.0 304.0 350.0 400.0  79.0 120.3
[28]  95.1 351.0 145.0 301.0 121.0
```

# SPSS Data

```
head(mtcars2)

   mpg cyl disp  hp drat    wt  qsec       vs        am
1 21.0   6  160 110 3.90 2.620 16.46 V-Shaped    Manual
2 21.0   6  160 110 3.90 2.875 17.02 V-Shaped    Manual
3 22.8   4  108  93 3.85 2.320 18.61 Straight    Manual
4 21.4   6  258 110 3.08 3.215 19.44 Straight Automatic
5 18.7   8  360 175 3.15 3.440 17.02 V-Shaped Automatic
6 18.1   6  225 105 2.76 3.460 20.22 Straight Automatic
  gear carb
1    4    4
2    4    4
3    4    1
4    3    1
5    3    2
6    3    1
```

# SPSS Data

```
head(mtcars3)

   mpg cyl disp  hp drat    wt  qsec vs am gear carb
1 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
2 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
3 22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
4 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
5 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
6 18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

# SPSS Data

```
## Load the packages:
library(haven)
library(labelled)

## Use haven::read_spss() to read '../../../data/mtcars.sav' into a tibble
mtcars4 <- read_spss(paste0(dataDir, "mtcars.sav"))

head(mtcars4)

# A tibble: 6 x 11
    mpg   cyl  disp    hp  drat    wt  qsec vs        am
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl+lb> <dbl+l>
1  21       6   160   110  3.9   2.62  16.5 0 [V-Sh~ 1 [Man~
2  21       6   160   110  3.9   2.88  17.0 0 [V-Sh~ 1 [Man~
3  22.8     4   108    93  3.85  2.32  18.6 1 [Stra~ 1 [Man~
4  21.4     6   258   110  3.08  3.22  19.4 1 [Stra~ 0 [Aut~
5  18.7     8   360   175  3.15  3.44  17.0 0 [V-Sh~ 0 [Aut~
6  18.1     6   225   105  2.76  3.46  20.2 1 [Stra~ 0 [Aut~
# i 2 more variables: gear <dbl>, carb <dbl>
```

# SPSS Data

`haven::read_spss()` converts any SPSS variables with labels into labelled vectors.

- We can use the `labelled::unlabelled()` function to remove the value labels.

```
mtcars5 <- unlabelled(mtcars4)

head(mtcars5)

# A tibble: 6 x 11
    mpg   cyl  disp    hp  drat    wt  qsec vs       am
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <fct>    <fct>
1  21       6   160   110  3.9   2.62  16.5 V-Shaped Manual
2  21       6   160   110  3.9   2.88  17.0 V-Shaped Manual
3  22.8     4   108    93  3.85  2.32  18.6 Straight Manual
4  21.4     6   258   110  3.08  3.22  19.4 Straight Automa~
5  18.7     8   360   175  3.15  3.44  17.0 V-Shaped Automa~
6  18.1     6   225   105  2.76  3.46  20.2 Straight Automa~
# i 2 more variables: gear <dbl>, carb <dbl>
```

# SPSS Data

```
mtcars4$am[1:20]

<labelled<double>[20]>: Transmission type
 [1] 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1

Labels:
 value     label
     0 Automatic
     1    Manual

mtcars5$am[1:20]

 [1] Manual    Manual    Manual    Automatic Automatic
 [6] Automatic Automatic Automatic Automatic Automatic
[11] Automatic Automatic Automatic Automatic Automatic
[16] Automatic Automatic Manual    Manual    Manual
Levels: Automatic Manual
```

# Excel Data

We have two good options for loading data from Excel spreadsheets:

- `readxl::read_excel()`
- `openxlsx::read.xlsx()`

```
## Load the packages:
library(readxl)
library(openxlsx)

## Use the readxl::read_excel() function to read the data from the 'titanic'
## sheet of the Excel workbook stored at '../../../data/example_data.xlsx'
titanic2 <- read_excel(paste0(dataDir, "example_data.xlsx"),
                       sheet = "titanic")

## Use the openxlsx::read.xlsx() function to read the data from the 'titanic'
## sheet of the Excel workbook stored at '../../../data/example_data.xlsx'
titanic3 <- read.xlsx(paste0(dataDir, "example_data.xlsx"),
                      sheet = "titanic")
```

# Excel Data

```
## Check the results from read_excel():
str(titanic2)

tibble [887 x 8] (S3: tbl_df/tbl/data.frame)
 $ survived          : chr [1:887] "no" "yes" "yes" "yes" ...
 $ class             : chr [1:887] "3rd" "1st" "3rd" "1st" ...
 $ name              : chr [1:887] "Mr. Owen Harris Braund" "Mrs. John Bradley (Flo
 $ sex               : chr [1:887] "male" "female" "female" "female" ...
 $ age               : num [1:887] 22 38 26 35 35 27 54 2 27 14 ...
 $ siblings_spouses: num [1:887] 1 1 0 1 0 0 0 3 0 1 ...
 $ parents_children: num [1:887] 0 0 0 0 0 0 0 1 2 0 ...
 $ fare              : num [1:887] 7.25 71.28 7.92 53.1 8.05 ...
```

# Excel Data

```
## Check the results from read.xlsx():
str(titanic3)

'data.frame': 887 obs. of  8 variables:
 $ survived         : chr  "no" "yes" "yes" "yes" ...
 $ class            : chr  "3rd" "1st" "3rd" "1st" ...
 $ name             : chr  "Mr. Owen Harris Braund" "Mrs. John Bradley (Florence B
 $ sex              : chr  "male" "female" "female" "female" ...
 $ age              : num  22 38 26 35 35 27 54 2 27 14 ...
 $ siblings_spouses: num  1 1 0 1 0 0 0 3 0 1 ...
 $ parents_children: num  0 0 0 0 0 0 0 1 2 0 ...
 $ fare             : num  7.25 71.28 7.92 53.1 8.05 ...

## Compare:
all.equal(as.data.frame(titanic2), titanic3)

[1] TRUE
```

# Workspaces & Delimited Data

All of the data reading functions we saw earlier have complementary data writing versions.

```
## The save() function writes an R workspace to disk
save(boys, file = paste0(dataDir, "tmp.RData"))

## For delimited text files and RDS data, the write.table(), write.csv(), and
## saveRDS() function do what you'd expect
write.table(boys,
            paste0(dataDir, "boys.txt"),
            row.names = FALSE,
            sep = "\t",
            na = "-999")

write.csv2(boys, paste0(dataDir, "boys.csv"), row.names = FALSE, na = "")

saveRDS(boys, paste0(dataDir, "boys.rds"))
```

# SPSS Data

To write SPSS data, the best option is the `haven::write_sav()` function.

```
write_sav(mtcars2, paste0(dataDir, "mctars2.sav"))
```

`write_sav()` will preserve label information provided by factor variables and the 'haven_labelled' class.

# Excel Data

The **openxlsx** package provides a powerful toolkit for programmatically building Excel workbooks in R and saving the results.

- Of course, it also works for simple data writing tasks.

```
## Use the openxlsx::write.xlsx() function to write the 'diabetes' data to an
## XLSX workbook
write.xlsx(diabetes, paste0(dataDir, "diabetes.xlsx"), overwrite = TRUE)

## Use the openxlsx::write.xlsx() function to write each data frame in a list

## to a separate sheet of an XLSX workbook
write.xlsx(list(titanic = titanic, diabetes = diabetes, mtcars = mtcars),
           paste0(dataDir, "example_data.xlsx"),
           overwrite = TRUE)
```

# FUNCTIONS

# R Functions

Functions are the foundation of R programming.

- Other than data objects, almost everything else that you interact with when using R is a function.
- Any R command written as a word followed by parentheses, `()` , is a function.
  - `mean()`
  - `library()`
  - `mutate()`
- Infix operators are aliased functions.
  - `<-`
  - `+` , `-` , `*`
  - `>` , `<` , `==`

# User-Defined Functions

We can define our own functions using the `function()` function.

```
square <- function(x) {
    out <- x^2
    out
}
```

After defining a function, we call it in the usual way.

```
square(5)

[1] 25
```

One-line functions don't need braces.

```
square <- function(x) x^2
square(5)

[1] 25
```

# User-Defined Functions

Function arguments are not strictly typed.

```
square(1:5)

[1]  1  4  9 16 25

square(pi)

[1] 9.869604

square(TRUE)

[1] 1
```

But there are limits.

```
square("bob") # But one can only try so hard

Error in x^2:  non-numeric argument to binary operator
```

# User-Defined Functions

Functions can take multiple arguments.

```
mod <- function(x, y) x %% y
mod(10, 3)

[1] 1
```

Sometimes it's useful to specify a list of arguments.

```
getLsBeta <- function(datList) {
    X <- datList$X
    y <- datList$y

    solve(crossprod(X)) %*% t(X) %*% y
}
```

# User-Defined Functions

```
X        <- matrix(runif(500), ncol = 5)
datList <- list(y = X %*% rep(0.5, 5), X = X)

getLsBeta(datList = datList)

     [,1]
[1,]  0.5
[2,]  0.5
[3,]  0.5
[4,]  0.5
[5,]  0.5
```

# User-Defined Functions

Functions are first-class objects in R.

- We can treat functions like any other R object.

R views an unevaluated function as an object with type ”closure”.

```
class(getLsBeta)

[1] "function"

typeof(getLsBeta)

[1] "closure"
```

An evaluated functions is equivalent to the objects it returns.

```
class(getLsBeta(datList))

[1] "matrix" "array"

typeof(getLsBeta(datList))

[1] "double"
```

# Nested Functions

We can use functions as arguments to other operations and functions.

```r
fun1 <- function(x, y) x + y

## What will this command return?
fun1(1, fun1(1, 1))
```

```
[1] 3
```

Why would we care?

```r
s2 <- var(runif(100))
x  <- rnorm(100, 0, sqrt(s2))
```

# Nested Functions

```r
X[1:8, ]
```

```
          [,1]       [,2]       [,3]      [,4]       [,5]
[1,] 0.52431382 0.67136447 0.28228726 0.7148383 0.54204681
[2,] 0.01926742 0.11693762 0.09148502 0.6929171 0.88371944
[3,] 0.05100735 0.18432074 0.43547799 0.6097462 0.09026598
[4,] 0.60566972 0.12944127 0.21000143 0.2441917 0.68141473
[5,] 0.48737303 0.94030405 0.23988619 0.4915910 0.36353771
[6,] 0.19941958 0.96670678 0.11455820 0.1243947 0.24253273
[7,] 0.95507804 0.38705829 0.49733535 0.2968470 0.81001800
[8,] 0.11093197 0.07731757 0.84923006 0.8653987 0.61914193
```

```r
c(1, 3, 6:9, 12)
```

```
[1]  1  3  6  7  8  9 12
```

# ITERATION

# Loops

There are three types of loops in R: *for*, *while*, and *until*.

- You'll rarely use anything but the for loop.
- So, we won't discuss while or until loops.

A *for loop* is defined as follows

```
for(INDEX in RANGE) { Stuff To Do with the Current INDEX Value }
```

# Loops

For example, the following loop will sum the numbers from 1 to 100.

```
val <- 0
for(i in 1:100) {
    val <- val + i
}

val

[1] 5050
```

# Loops

This loop will compute the mean of every column in the 'mtcars' data.

```
means <- rep(0, ncol(mtcars))
for(j in 1:ncol(mtcars)) {
    means[j] <- mean(mtcars[ , j])
}

means

 [1]  20.090625   6.187500 230.721875 146.687500   3.596563
 [6]   3.217250  17.848750   0.437500   0.406250   3.687500
[11]   2.812500
```

# Loops

Loops are often one of the least efficient solutions in R

```
n <- 1e8

t0 <- system.time({
    val0 <- 0
    for(i in 1:n) val0 <- val0 + i
})

t1 <- system.time(
    val1 <- sum(1:n)
)
```

# Loops

Both approaches produce the same answer.

```
val0 - val1

[1] 0
```

But the loop is many times slower.

```
t0

   user  system elapsed
  1.267   0.000   1.267

t1

   user  system elapsed
      0       0       0
```

# Loops

There is often a built in routine for what you are trying to accomplish with the loop.

```
## The appropriate way to get variable means:
colMeans(mtcars)

       mpg        cyl       disp         hp       drat 
 20.090625   6.187500 230.721875 146.687500   3.596563 
        wt       qsec         vs         am       gear 
  3.217250  17.848750   0.437500   0.406250   3.687500 
      carb 
  2.812500 
```

# Apply Statements

In R, some flavor of *apply statement* is often preferred to a loop.

- Apply statements broadcast some operation across the elements of a data object.
- Apply statements can take advantage of internal optimizations that loops can't use.

There are many flavors of apply statement in R, but the three most common are:

- `apply()`
- `lapply()`
- `sapply()` .

# Apply Statements

Apply statements generally take one of two forms:

```
[x]apply(DATA, MARGIN, FUNCTION, ...)

[x]apply(DATA, FUNCTION, ...)
```

# Apply Examples

```
## Load some example data:
data(mtcars)

## Subset the data:
dat1 <- mtcars[1:5, 1:3]

## Find the range of each row:
apply(dat1, 1, range)

     Mazda RX4 Mazda RX4 Wag Datsun 710 Hornet 4 Drive
[1,]         6             6          4              6
[2,]       160           160        108            258
     Hornet Sportabout
[1,]                 8
[2,]               360
```

# Apply Examples

```
## Find the maximum value in each column:
apply(dat1, 2, max)

  mpg   cyl  disp
 22.8   8.0 360.0

## Subtract 1 from every cell:
apply(dat1, 1:2, function(x) x - 1)

                  mpg cyl disp
Mazda RX4         20.0   5  159
Mazda RX4 Wag     20.0   5  159
Datsun 710        21.8   3  107
Hornet 4 Drive    20.4   5  257
Hornet Sportabout 17.7   7  359
```

# Apply Examples

```
## Create a toy list:
l1 <- list()
for(i in 1:3) l1[[i]] <- runif(10)

## Find the mean of each list entry:
lapply(l1, mean)

[[1]]
[1] 0.526697

[[2]]
[1] 0.4020885

[[3]]
[1] 0.607818

## Same as above, but return the result as a vector:
sapply(l1, mean)

[1] 0.5266970 0.4020885 0.6078180
```

# Apply Examples

```
## Find the range of each list entry:
lapply(l1, range)

[[1]]
[1] 0.04395916 0.99350611

[[2]]
[1] 0.002797563 0.821082495

[[3]]
[1] 0.09926892 0.90430843

sapply(l1, range)

           [,1]        [,2]       [,3]
[1,] 0.04395916 0.002797563 0.09926892
[2,] 0.99350611 0.821082495 0.90430843
```

# Apply Examples

```
lapply(dat1, max)

$mpg
[1] 22.8

$cyl
[1] 8

$disp
[1] 360

sapply(dat1, max)

  mpg   cyl  disp
 22.8   8.0 360.0
```

# Some Programming Tips

You can save yourself a great deal of heartache by following a few simple guidelines.

- Keep your code tidy.
- Use comments to clarify what you are doing.
- When working with functions in RStudio, use the TAB key to quickly access the documentation of the function's arguments.
- Give your R scripts and objects meaningful names.
- Use a consistent directory structure and RStudio projects.

# General Style Advice

Use common sense and BE CONSISTENT.

- Browse the tidyverse style guide.
  - The point of style guidelines is to enforce a common vocabulary.
  - You want people to concentrate on *what* you're saying, not *how* you're saying it.
- If the code you add to a project/codebase looks drastically different from the extant code, the incongruity will confuse readers and collaborators.

Spacing and whitespace are your friends.

- `a<-c(1,2,3,4,5)`
- `a <- c(1, 2, 3, 4, 5)`
- At least put spaces around assignment operators and after every comma!

# References


Becker, R. A., & Chambers, J. M. (1984). *S: an interactive environment for data analysis and graphics*. Monterey, CA: Wadsworth and Brooks/Cole.

Becker, R. A., Chambers, J. M., & Wilks, A. R. (1988). *The new S language*. London: Chapman & Hall.

Chambers, J. M. (1998). *Programming with data: A guide to the S language*. New York: Springer Science & Business Media.

Chambers, J. M., & Hastie, T. J. (1992). *Statistical models in s*. London: Chapman & Hall.

Ihaka, R., & Gentleman, R. (1996). R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, *5*(3), 299–314.