# An introduction to programming in R

Sahir Bhatnagar and Yi Yang[1]

July 8 – 11, 2018

[1] https://github.com/sahirbhatnagar/npu

# Notice #1

- This is an **introduction** to the R language

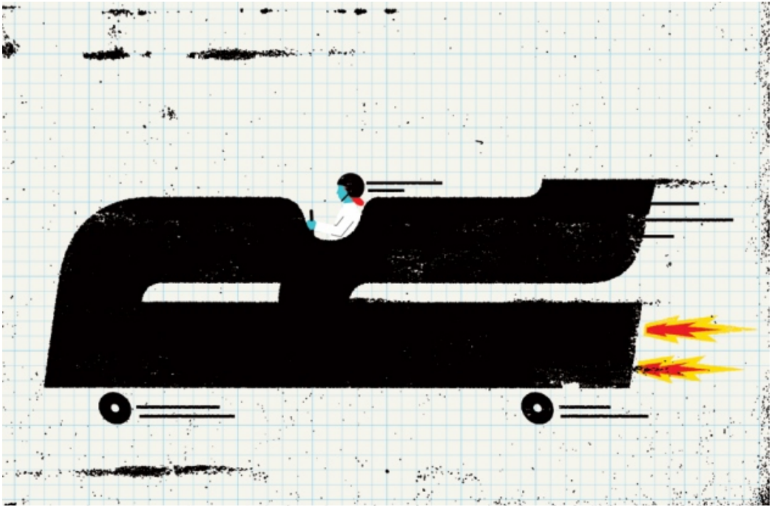
# Notice #1

- This is an **introduction** to the R language
- Feel free to ask quesitons

# Notice #2

*We do not have any commercial affiliations with these software*

# Let's Begin



## After this workshop you should be able to:

- Understand, create and modify the 4 main objects in R (`vector`, `data.frame`, `matrix`, `list`)
- Use basic functions
- Import a dataset from an external file
- Create a plot

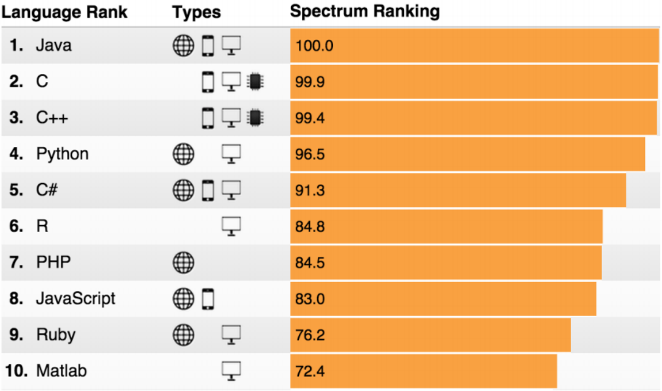
# 1. Background on the R language

## Objectives of this section

1. Understand the advantages of R
2. Know it's characteristics
3. Start an R session and execute some basic commands
4. Create, modify and save an R script

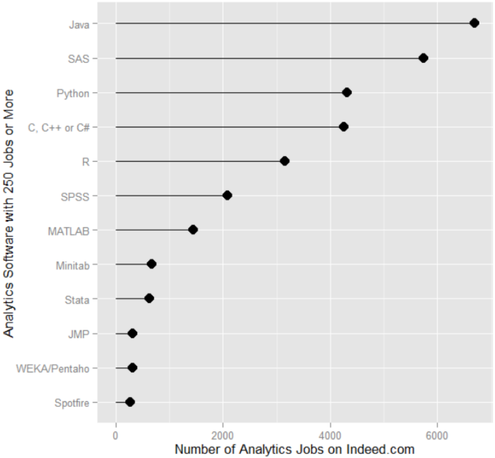
# The popularity of R

# The rise of popularity



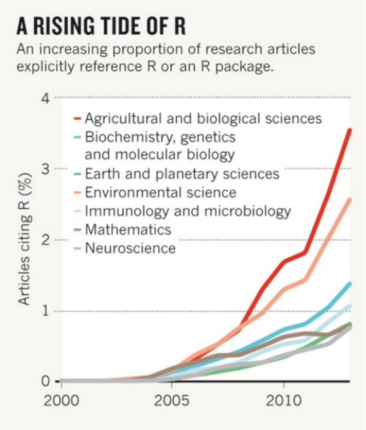

The best programming languages in 2015 according to IEEE Spectrum

# Number of Jobs




reference: http://r4stats.com/articles/popularity/

# Used in many domains




Published in *Nature*

# Powerful tool to analyze data

- Several resources for state-of-the-art statistical computing
- Powerful graphing system
- Integrate your R code into web applications
- Ensure the reproducibility of your analyses

# A Brief History

# Before R there was S by John M. Chambers

Figure 1: S, is a language developped at Bell Laboratories in the 1970s by a group of researchers led by John M. Chambers

# Creators of R

Figure 2: Inspired by S, Ross Ihaka (left) and Robert Gentleman (right) from the University of Auckland in New Zeland launched the first version of R in 1996

# Characteristics of R

# Object oriented programming language



- This makes it easy to find and reuse the results of your analyses

# Object oriented programming language

- This makes it easy to find and reuse the results of your analyses
- A function can complete several tasks

# An interpreted language

- An interpreted language: does not require compiling a program into machine-language instructions
- A compiled language: `C`, `C++`, `JAVA`
- More accessible than a compiled language → which allows economists, ecologists, biologists, engineers, statisticians, epidemiologists, etc. to use R

# An interpreted language

- An interpreted language: does not require compiling a program into machine-language instructions
- A compiled language: `C`, `C++`, `JAVA`
- More accessible than a compiled language → which allows economists, ecologists, biologists, engineers, statisticians, epidemiologists, etc. to use R
- The program we run to use R is the interpreter

# An interpreted language

- An interpreted language: does not require compiling a program into machine-language instructions
- A compiled language: `C`, `C++`, `JAVA`
- More accessible than a compiled language → which allows economists, ecologists, biologists, engineers, statisticians, epidemiologists, etc. to use R
- The program we run to use R is the interpreter
- This interpreter takes commands in R and it will immediately run

# An interpreted language



- An interpreted language: does not require compiling a program into machine-language instructions
- A compiled language: `C`, `C++`, `JAVA`
- More accessible than a compiled language → which allows economists, ecologists, biologists, engineers, statisticians, epidemiologists, etc. to use R
- The program we run to use R is the interpreter
- This interpreter takes commands in R and it will immediately run
- Another example: code Python

# Open Source Software

- Active development for the creation of new tools in several fields
  - https://cran.r-project.org/web/views/
- Easily see other people's code with GitHub
  - http://www.r-pkg.org/
- Well-documented with a lot of free resources available on the internet
  - stackoverflow
  - http://www.rdocumentation.org/
  - http://www.r-bloggers.com/
  - twitter
  - R user groups
  - Google

# Statistical tool that optimizes the matrix approach

- The R language is based on the notion of vector, which simplifies mathematical calculations (not only computation but also writing)

# Statistical tool that optimizes the matrix approach

- The R language is based on the notion of vector, which simplifies mathematical calculations (not only computation but also writing)

- Reduces the use of iterative structures (loops `for`, `while`, etc.)

# Statistical tool that optimizes the matrix approach

- The R language is based on the notion of vector, which simplifies mathematical calculations (not only computation but also writing)
- Reduces the use of iterative structures (loops `for`, `while`, etc.)

**R code 1.1**

```
c(1,2,3) + c(4,5,6)

## [1] 5 7 9
```

# How to find help for a function



- `?name_of_the_function`

# How to find help for a function

- `?name_of_the_function`

R code 1.2

```
# find help for linear regression function 'lm'
?lm
```

# The help page - 2 main sections



- **Usage:** the name of the function, and all of its arguments and default values

# The help page - 2 main sections

- **Usage:** the name of the function, and all of its arguments and default values
- **Value:** the type of object returned and its contents

# Starting a session

R code 1.3

```r
# Start the interface for documentation
# and navigate the different resources
help.start()

# find help for the rnorm function
?rnorm

# Get the working directory
getwd()
```

R code 1.4

```r
# addition
39 + 3

# substraction
58 - 16

# multiplication
6 * 7

# division
8 / 3
```

R code 1.5

```
# Generate two random vectors of size 50
# from a standard normal distribution
x <- rnorm(50)
y <- rnorm(50)

# Plot the points (x, y)
plot(x, y)

# A histogram of x
hist(x)
```

R code 1.6

```r
# to see the contents of the x vector
x

# see the objects in your workspace
ls()

# delete the two vectors x and y
rm(x,y)

# see the contents of x
x

# see the objects in your workspace
ls()
```

R code 1.7

```
# generate a sequence 1, 2, ..., 20.
x <- 1:20

# create another vector as a function of x
y <- 2 * x + 3

# create a data.frame and see its contents
dt <- data.frame(x, y)
dt

# run a linear regression and see the
# results
fit <- lm(y ~ x, data = dt)
summary(fit)
```

R code 1.8

```
# The 'seq' function will generate more gen-
eral sequences
seq(from = -5, to = 10, by = 3)
seq(from = -5, length = 10)

#'rep' repeats values
rep(1, 5) # repeat 1, 5 times
rep(1:5, 5) # repeat the vec-
tor 1,...,5,  five times
rep(1:5, each = 5) # repeat each ele-
ment five times
```

R code 1.9

```
# vector arithmetic
v <- 1:12  # initialize a vector
v + 2 # add 2 to each element of the vector
v * -12:-1 # element-wise product
v + 1:3 # the shortest vector is recycled

# Generate random uniform(1,10) numbers
v <- runif(12, min = 1, max = 10); v

# You can place the call in parenthe-
ses to see the result
( v <- runif(12, min = 1, max = 10) )
```

R code 1.10

```
# trouver le répertoire où se trouve le
# jeux de données 'morley', qui est inclu avec
# l'installation de R
filepath <- system.file("data", "morley.tab",
            package="datasets")

# importer les données dans un objet appeller 'mm'
mm <- read.table(filepath)

# Graphique
plot(mm$Expt, mm$Speed,
main="Speed of Light Data", xlab="Experiment No.")

# Terminer la session
q()
```

# 2. Basics of R

## Objectives of this section

1. Understand what a function is and how to use it
2. Identify the main objects in R: `vector`, `matrix`, `data frame` and `list`
3. Create and manipulate these objects
4. Import some external datasets `.txt` et `.csv`
5. Install a *package*

# Functions

# Definition of a Function

- Instead of writing many repetitive lines of code, we call **functions** instead

# Definition of a Function

- Instead of writing many repetitive lines of code, we call **functions** instead
- A function is characterised by two components

# Definition of a Function

- Instead of writing many repetitive lines of code, we call **functions** instead
- A function is characterised by two components
  1. It's **name**: this name allows the user to call the function

# Definition of a Function

- Instead of writing many repetitive lines of code, we call **functions** instead

- A function is characterised by two components
  1. It's **name**: this name allows the user to call the function
  2. a **list of arguments**: this is the information the function needs to return a proper result

# Function syntax

- You have to write the name of the function followed by two parentheses

# Function syntax

- You have to write the name of the function followed by two parentheses

- The required arguments between these two parentheses are what the function requires to execute it:

```
name_of_the_function(arguments)
```

# Function syntax

- You have to write the name of the function followed by two parentheses

- The required arguments between these two parentheses are what the function requires to execute it:

  `name_of_the_function(arguments)`

## R code 2.1

```
# look at the help page for square root function
?sqrt

# we see that the sqrt function takes one argument
sqrt(49)
```

# Optional arguments

- Certain functions have optional arguments while others have no arguments at all: `getwd()`)

# Optional arguments

- Certain functions have optional arguments while others have no arguments at all: `getwd()`)
- The optional arguments have a default value

# Optional arguments

- Certain functions have optional arguments while others have no arguments at all: `getwd()`)
- The optional arguments have a default value
- If we do not specify the optional argument, then the function will use the default, else it will use the argument value specified

# Optional arguments

- Certain functions have optional arguments while others have no arguments at all: `getwd()`)
- The optional arguments have a default value
- If we do not specify the optional argument, then the function will use the default, else it will use the argument value specified

## R code 2.2

```
# look at the help for the logarithm function
?log

# we don't need to specify a value for the 2nd argument
log(2)

# we can specify the second argument if we want
log(2, base = exp(1))
```

# R Objects

# R Objects

Table 1: The principal R objects and their name

| dimension | same type[a] | different type[b] |
|---|---|---|
| 1d | Atomic vector `c()` | List `list()` |
| 2d | Matrix `matrix()` | Data frame `data.frame()` |

[a] all the elements have to be of the same type
[b] the elements can be of different types

# Atomic vectors

- In R, everything is a vector

# Atomic vectors

- In R, everything is a vector
- The function to create a vector is `c()` (concatenation)

# *Atomic vectors*

- In R, everything is a vector
- The function to create a vector is `c()` (concatenation)

**R code 2.3**

```
c(1, 2, 5)

## [1] 1 2 5
```

# *Atomic vectors*

- Frequently used *atomic vectors*:
  1. *double* (also called *numeric*)
  2. *integer*
  3. *character*
  4. *logical*

# *Atomic vectors*

- Frequently used *atomic vectors*:
  1. *double* (also called *numeric*)
  2. *integer*
  3. *character*
  4. *logical*

## R code 2.4

```
c(1, 2.5, 4.5) # numeric
c(1L, 6L, 10L) # integer
c("these are", "characters") #character
c(TRUE, FALSE, T, F) # logical
```

# Test

- `typeof()`: to find out the type of vector
- `is.character()`, `is.double()`, `is.integer()`, `is.logical()`, `is.atomic()`: for case specific types

# Test

- `typeof()`: to find out the type of vector
- `is.character()`, `is.double()`, `is.integer()`, `is.logical()`, `is.atomic()`: for case specific types

## R code 2.5

```
int_var <- c(1L, 6L, 10L)
typeof(int_var)

## [1] "integer"

is.integer(int_var)

## [1] TRUE

is.atomic(int_var)

## [1] TRUE
```

# Test

## R code 2.6

```
dbl_var <- c(1, 2.5, 4.5)
typeof(dbl_var)

## [1] "double"

is.double(dbl_var)

## [1] TRUE

is.atomic(dbl_var)

## [1] TRUE
```

# *Coercion*

- All the elements of an *atomic vector* must be of the same type

# *Coercion*

- All the elements of an *atomic vector* must be of the same type

- When you try to combine several types, it will convert everything to the most flexible type

# Coercion

- All the elements of an *atomic vector* must be of the same type
- When you try to combine several types, it will convert everything to the most flexible type
- From least flexible to most flexible:
  1. *logical*
  2. *integer*
  3. *double*
  4. *character*

## R code 2.7

```
# combine a character and an interger will pro-
duce what?
str(c("a", 1))

##  chr [1:2] "a" "1"
```

# Coercion

- Most mathematical operations will convert an *atomic vector* into a *double* or *integer*

## R code 2.8

```
x <- c(FALSE, FALSE, TRUE)
as.numeric(x)

## [1] 0 0 1

# Number of TRUE
sum(x)

## [1] 1

# Proportion of TRUE
mean(x)

## [1] 0.33
```

# *List*

- *lists* are different from *atomic vectors* because the elements will be of the same type
- The function to create a *list* is a `list()`

# *List*

- *lists* are different from *atomic vectors* because the elements will be of the same type
- The function to create a *list* is a `list()`

## R code 2.9

```
(x <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.3, 5.9)))

## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1]  TRUE FALSE  TRUE
##
## [[4]]
## [1] 2.3 5.9
```

# *Matrix*

- Matrices are nothing but vectors in 2 dimensions

# *Matrix*

- Matrices are nothing but vectors in 2 dimensions
- Used for mathematical computations (think of linear algebra courses)

# *Matrix*

- Matrices are nothing but vectors in 2 dimensions
- Used for mathematical computations (think of linear algebra courses)
- The function to create a *matrix* is `matrix()`

# *Matrix*

- Matrices are nothing but vectors in 2 dimensions
- Used for mathematical computations (think of linear algebra courses)
- The function to create a *matrix* is `matrix()`

## R code 2.10

```
# filled by column by defaults
matrix(c(1,2,3,4,5,6), nrow = 2, ncol = 3)

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

# Matrix

## R code 2.11

```
# automatically converted to same data type
matrix(c(1,2,3,"a","b","c"), nrow = 2, ncol = 3)

##      [,1] [,2] [,3]
## [1,] "1"  "3"  "b"
## [2,] "2"  "a"  "c"
```

# Data frame

- The *data frame* is a collection of vectors of different types

# *Data frame*

- The *data frame* is a collection of vectors of different types
- The function to create a *data frame* is `data.frame()`

# Data frame

- The *data frame* is a collection of vectors of different types
- The function to create a *data frame* is `data.frame()`

## R code 2.12

```
# an 'id'  column to identify the individuals
# an 'age' column for their age
data.frame(id = c("yi","yang","zhengzin"),
           age = c(37,34,32))

##          id age
## 1        yi  37
## 2      yang  34
## 3 zhengzin  32
```

# The difference between a *data frame* and a *matrix*?



- `matrix`: matrix calculations
- `data.frame`: all other analyses of different data types

# R Commands

# Expression and Assignment

1. An **expression** is immediately evaluated and the result is posted in the R console:

# Expression and Assignment

1. An **expression** is immediately evaluated and the result is posted in the R console:

R code 2.13

```
2 + 3

## [1] 5

pi

## [1] 3.1

cos(pi/4)

## [1] 0.71
```

# Expression and Assignment

2. When performing an **assignment**, an expression is evaluated, but the result is stored in an object and nothing is printed to the console
   - The assignment operator is <-
   - the two characters < and - must be placed one after another (no spaces):

# Expression and Assignment

3. When performing an **assignment**, an expression is evaluated, but the result is stored in an object and nothing is printed to the console
   - The assignment operator is <-
   - the two characters < and - must be placed one after another (no spaces):

### R code 2.14

```
a <- 5
a

## [1] 5

b <- a - 2
b

## [1] 3
```

# Expression and Assignment

4. To create an assignment and simultaneously print its result, you can place the expression in parentheses:

# Expression and Assignment

4. To create an assignment and simultaneously print its result, you can place the expression in parentheses:

### R code 2.15

```
(a <- 2 + 3)

## [1] 5
```

# Expression and Assignment

4. To create an assignment and simultaneously print its result, you can place the expression in parentheses:

**R code 2.15**

```
(a <- 2 + 3)

## [1] 5
```

5. The $=$ is valid, but is not recommended
   - it can cause confusion between the `name = value` in function calls

# Naming conventions for objects

- Characters are allowed for naming objects:
  1. lower case letters a–z
  2. upper case letters A–Z
  3. numbers 0–9,
  4. the period .
  5. underscore _

# Naming conventions

- R is case-sensitive, which means `foo`, `Foo` and `FOO` are three distinct objects

# Naming conventions

- R is case-sensitive, which means `foo`, `Foo` and `FOO` are three distinct objects
- Certain names are reserved for R functions, so its best to avoid them

```
c, q, t, C, D, I, diff, length, mean, pi,
range, var, sd
break, else, for, function, if, in, next,
repeat, return, while
TRUE, FALSE, T, F
Inf, NA, NaN, NULL
```

# Indexing a *vector*

- Indexing has two purposes

# Indexing a *vector*

- Indexing has two purposes
  1. extract elements
  2. replace elements

# Indexing a *vector*

- Indexing has two purposes
  1. extract elements
  2. replace elements

## R code 2.16

```
# create a vectir
x <- c(a = -1, b = 2, c = 8, d = 10)

# extract by position
x[1]

# extract by name
x["c"]

# remplace the second element by 5
x[2] <- 5
```

# Indexing a *data frame* and a *matrix*

- Indexing observations by rows and columns for the *data.frame* and *matrix*:

```
df[row, column]
```

# Indexing a *data frame* and a *matrix*

- Indexing observations by rows and columns for the *data.frame* and *matrix*:

  `df[row, column]`

## R code 2.17

```
# create a data frame
d <- data.frame(Noms = c("Pierre", "Jean", "Jacques"),
Age = c(42, 34, 19),
Fumeur = c(TRUE, TRUE, FALSE))

d[1, ] # first row

d[ ,1] # first column

d[3,2] # third row, second column
```

# Indexing a *list*

## R code 2.18

```
# create a list
x <- list(player = c("V", "C"),
                  score = c(10, 12))

# first element of the list
x[[1]]

# 1st element of the 2nd element of the list
x[[2]][1]
```

# Import data and code

Table 2: Functions for importing data and code

| | files | objects[c] | R script |
|---|---|---|---|
| import | `read.table()`[a]<br>`read.csv()`[b] | `load()` | `source()` |
| save | `write.table()`<br>`write.csv()` | `save()` | File -> Save As.. |

[a] value is separated by a space
[b] each value is separated by a comma
[c] objects in your working environment

# Import data

- You must specify where the data is located on your hard drive

# Import data

- You must specify where the data is located on your hard drive
  - `getwd()`: to know the working directory

# Import data

- You must specify where the data is located on your hard drive
    - `getwd()`: to know the working directory
    - `setwd()`: to modify the working directory

# Import data

- You must specify where the data is located on your hard drive
  - `getwd()`: to know the working directory
  - `setwd()`: to modify the working directory

## R code 2.19

```
# modify working directory
setwd("~/git_repositories/npu/data")

# import the files
# assign it to the 'lung' and 'admit' objects
lung <- read.csv("lung.csv")

admit <- read.table("admit.txt")
```

# Import code from an R Script

## R code 2.20

```r
# modify the working directory
setwd("~/git_repositories/npu/script")

# execute the commands of the RScript 'mtcars.R'
# and show the output
source("mtcars.R", echo = TRUE)

# save the results in an RData object
save(df, fit, file = "mtcars.RData")

# delete the R objects
rm(df, fit)

# import the R objects saved in 'mtcars.RData'
load("mtcars.RData")
```

# Instal *packages* in R

- A R **package** is a collection of functions, data and documentation that allows a user to perform other tasks.
- A list of these packages is available at `http://www.r-pkg.org/`

# Instal *packages* in R

- A R **package** is a collection of functions, data and documentation that allows a user to perform other tasks.
- A list of these packages is available at http://www.r-pkg.org/

R code 2.21

```
# install packages for reproducible code
install.packages(c("knitr","rmarkdown"))

# to get access to those functions
library(knitr)
library(rmarkdown)
```

# Summary

# Summary of basic R commands

Working Environment

```
sessionInfo() install.packages()
library()
setwd()
getwd()
rm()
ls()
```

Frequently used R objects

```
c()
data.frame()
matrix()
list()
```

Access to external data and scripts

```
read.table()
write.table()
load()
save()
source()
```

# 3. R Graphics

## Objectives of this section

1. Comprendre la syntaxe de la fonction `plot`
2. Créer des `boxplot`, `histogram` et `density` plot
3. Comment sauvegarder des graphiques

# La fonction `plot`

# Introduction aux graphiques en R

- R offre une variété de graphiques remarquables
- Pour avoir une petite idée des possibilités offertes, il suffit de taper la commande `demo(graphics)`

# La fonction `plot`

- La fonction `plot` peut prendre plusieurs arguments et types d'objets

# La fonction plot

- La fonction `plot` peut prendre plusieurs arguments et types d'objets

## R code 3.1

```
# Plot des valeurs d'un vecteur contre leurs in-
dices
# équivalent à plot(mtcars[,"mpg"])
plot(mtcars$mpg, xlab = "Index", ylab = "mpg",
    main = "Titre")

# Graphique des couples (x, y)
plot(mtcars$mpg, mtcars$disp, xlab = "mpg",
ylab = "disp", main = "mpg vs. disp")
```

# La fonction `plot`

- La fonction `plot` peut aussi prendre un objet du modèle linéaire comme argument

# La fonction `plot`

- La fonction `plot` peut aussi prendre un objet du modèle linéaire comme argument
- Ceci est un exemple qui démontre pourquoi R est un langage orientée objet

# La fonction plot

- La fonction `plot` peut aussi prendre un objet du modèle linéaire comme argument
- Ceci est un exemple qui démontre pourquoi R est un langage orientée objet

## R code 3.2

```
# importer 'mtcars.RData'
load("mtcars.RData")

# Graphiques des 4 diagnostiques
# du modèle linéaire
# placer dans 2 rangés et 2 colonnes
par(mfrow=c(2,2))
plot(fit)
```

# histogram et boxplot

# La fonction `histogram`

- Utile pour voir la distribution des données

- Pour les données continues et univariées

# La fonction histogram

- Utile pour voir la distribution des données
- Pour les données continues et univariées

## R code 3.3

```
# la taille de 237 étudiants disponibles dans le jeu
# de données 'survey' du library(MASS)
library(MASS)

# voir le nom des colonnes
names(survey)

# histogram de la taille et montrer la fréquence
# de chaque barre
hist(survey$Height, labels = TRUE)
```

# La fonction `boxplot`

- Utile pour voir la différence d'une variable continue parmi plusieurs groupes

# La fonction boxplot

- Utile pour voir la différence d'une variable continue parmi plusieurs groupes

## R code 3.4

```
# visualiser la différence de taille entre les
# hommes et les femmes dans le jeux
# de données 'survey' du library(MASS)

boxplot(survey$Height ~ survey$Sex,
        ylab = "Taille (cm)",
        col = c("lightpink","lightblue"))
```

# Sauvegarder les graphiques

- 2 fonctions principales: `pdf()` et `png()`
- l'argument de ces fonctions est le nom du fichier désiré

# Sauvegarder les graphiques

- 2 fonctions principales: `pdf()` et `png()`
- l'argument de ces fonctions est le nom du fichier désiré

## R code 3.5

```
# enregistrer dans le répertoire de tra-
vail courant
pdf("boxplot_survey.pdf")
boxplot(survey$Height ~ survey$Sex,
        ylab = "Taille (cm)")
dev.off()
```

# Sommaire

# Sommaire des commandes de graphiques

Fonctions pour créer les graphiques

```
plot()
hist()      box-
plot()
```

Arguments communs pour ces fonctions

```
xlab
ylab
main
type
col
```

Fonctions pour enregistrer les graphiques

```
pdf()
png()
dev.off()
```

# 4. Statistiques

# Moyenne, écart type

# Moyenne, variance, écart type, minimum, maximum

- fonctions principales: `mean()`, `var()`, `sd()`, `min()`, `max()`

# Moyenne, variance, écart type, minimum, maximum

- fonctions principales: `mean()`, `var()`, `sd()`, `min()`, `max()`
- la fonction `summary()` est utile pour calculer quelques statistiques de bases pour un *data frame*

# Moyenne, variance, écart type, minimum, maximum

- fonctions principales: `mean()`, `var()`, `sd()`, `min()`, `max()`
- la fonction `summary()` est utile pour calculer quelques statistiques de bases pour un *data frame*

## R code 4.1

```
# enregistrer dans le répertoire de tra-
vail courant
summary(mtcars)
```

# La fonction `apply`

- Pour calculer des statistiques plus compliquées sur un *data frame* (ou *matrix*), on utilise la fonction `apply`

# La fonction `apply`

- Pour calculer des statistiques plus compliquées sur un *data frame* (ou *matrix*), on utilise la fonction `apply`
- La fonction `apply` sert à appliquer une fonction quelconque sur une partie d'un *matrix* ou *data frame*

# La fonction `apply`

- La syntaxe de la fonction est la suivante:

```
apply(X, MARGIN, FUN)
```

- `X`: un matrix ou data frame
- `MARGIN`: 1 si l'on veut faire des calculs sur les rangées, 2 sur les colonnes
- `FUN`: est la fonction à appliquer

# La fonction apply

- La syntaxe de la fonction est la suivante:

  `apply(X, MARGIN, FUN)`

- `X`: un matrix ou data frame
- `MARGIN`: 1 si l'on veut faire des calculs sur les rangées, 2 sur les colonnes
- `FUN`: est la fonction à appliquer

## R code 4.2

```r
# variance par ligne
apply(mtcars, 1, var)
```

# La fonction apply

## R code 4.3

```
# variance par colonne
apply(mtcars, 2, var)

# écart type par colonne
apply(mtcars, 2, sd)

# minimum de chaque rangée
apply(mtcars, 1, min)


# maximum de chaque rangée
apply(mtcars, 1, max)
```

# Test t et analyse de variance

# Test t

- la fonction pour faire un test t est `t.test()`

# Test t

- la fonction pour faire un test t est `t.test()`

## R code 4.4

```
# la taille de 237 étudiants disponible dans le jeux
# de données 'survey' du library(MASS)
# est-ce qu'il y a une différence de taille entre
# les hommes et les femmes?

t.test(Height ~ Sex, data = survey)
```

# Analyse de variance



- La fonction pour l'analyse de variance est `aov`

# Analyse de variance

- La fonction pour l'analyse de variance est `aov`
- Prenons un jeux de données disponible dans R: `InsectSprays`

# Analyse de variance

- La fonction pour l'analyse de variance est `aov`
- Prenons un jeux de données disponible dans R: `InsectSprays`
- 6 insecticides ont été testés 12 fois en culture, la réponse observée étant le nombre d'insectes

# Analyse de variance

- La fonction pour l'analyse de variance est `aov`
- Prenons un jeux de données disponible dans R: `InsectSprays`
- 6 insecticides ont été testés 12 fois en culture, la réponse observée étant le nombre d'insectes
- Le but c'est de voir s'il y a une différence importante entre les 6 insecticides par rapport à le nombre d'insectes

# Analyse de variance



## R code 4.5

```
# boxplot pour voir la différence entre
# les groupes
boxplot(InsectSprays$count ~ InsectSprays$spray)

# Les résultats ne sont pas affichés, ceux-ci sont

# copiés dans un objet nommé aov.spray
aov.spray <- aov(count ~ spray, data = Insect-
Sprays)

# sommaire des résultats
summary(aov.spray)
```

# Modèles linéaires

# Les formules

- Les formules représentent un élément-clé des analyses statistiques avec R

# Les formules

- Les formules représentent un élément-clé des analyses statistiques avec R
- La notation utilisée est la même pour (presque) toutes les fonctions

# Les formules

- Les formules représentent un élément-clé des analyses statistiques avec R
- La notation utilisée est la même pour (presque) toutes les fonctions
- Une formule est typiquement de la forme

$$\texttt{y} \sim \texttt{model}$$

# Les formules

- Les formules représentent un élément-clé des analyses statistiques avec R
- La notation utilisée est la même pour (presque) toutes les fonctions
- Une formule est typiquement de la forme

$$\texttt{y} \sim \texttt{model}$$

1. `y`: est la réponse analysée
2. `model`: est un ensemble de termes pour lesquels les paramètres sont estimés

# Les formules

- Les formules représentent un élément-clé des analyses statistiques avec R
- La notation utilisée est la même pour (presque) toutes les fonctions
- Une formule est typiquement de la forme

$$\texttt{y} \sim \texttt{model}$$

  1. `y`: est la réponse analysée
  2. `model`: est un ensemble de termes pour lesquels les paramètres sont estimés

- Ces termes sont séparés par des symboles arithmétiques mais qui ont ici une signification particulière: `a+b`, `a*b`

# Modèles linéaires



## R code 4.6

```
# importer 'admit.txt'
admit <- read.table("admit.txt", header = TRUE)

# est-ce que gpa et rank sont reliés à gre
fit <- lm(gre ~ gpa+rank, data = admit)

# voir les résultats
summary(fit)
```

# GLM

## R code 4.7

```r
# régression logistique
fit.glm <- glm(admit ~ gre+gpa+rank, data = admit,
               family = binomial(link = "logit"))

# voir les résultats
summary(fit.glm)
```

# Sommaire

# Sommaire des commandes de statistiques

Fonctions pour des statistiques descriptives

```
mean()
var()
sd()
min()
max()   me-
dian()   ap-
ply()
```

Tests statistiques

```
aov()
t.test()
chisq.test()
```

Fonctions pour les modèles de régression

```
lm()
glm()
```

# 5. Créer des rapports

# Introduction

# Introduction

- http://rmarkdown.rstudio.com/
- http://shiny.rstudio.com/gallery/

# Session Information

- R version 3.5.0 (2018-04-23), `x86_64-w64-mingw32`
- Running under: `Windows 10 x64 (build 17134)`
- Matrix products: default
- Base packages: base, datasets, graphics, grDevices, methods, stats, utils
- Other packages: data.table 1.11.0, dplyr 0.7.4, ggplot2 2.2.1, knitr 1.20, xtable 1.8-2
- Loaded via a namespace (and not attached): assertthat 0.2.0, bindr 0.1.1, bindrcpp 0.2.2, colorspace 1.3-2, compiler 3.5.0, evaluate 0.10.1, formatR 1.5, glue 1.2.0, grid 3.5.0, gtable 0.2.0, highr 0.6, lazyeval 0.2.1, magrittr 1.5, munsell 0.4.3, pillar 1.2.2, pkgconfig 2.0.1, plyr 1.8.4, R6 2.2.2, Rcpp 0.12.16, rlang 0.2.0, scales 0.5.0, stringi 1.1.7, stringr 1.3.0, tibble 1.4.2, tools 3.5.0