R

Fabrice Rossi

CEREMADE
Université Paris Dauphine

2020

Outline

Introduction

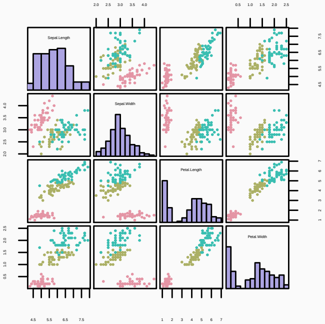
Main concepts

Data Management in R

R

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

- R is a programming language and an environment for statistical computing and visualization
- R is a multiplatform free software
- R can be extended by thousands of packages
- R provides state-of-the-art implementation of myriads of statistical, data mining and machine learning algorithms



R

Pros

- open source
- full-fledged programming language
- strong support from multiple compagnies
- broad coverage of data science, statistics, etc.
- high performance packages
- report generation and high quality graphics

Cons

- limited point-and-click support
- peculiar language with specific constructs
- naive code has low performances
- way too many packages

Recommended installs

- R `https://www.r-project.org/`
- Rstudio `https://www.rstudio.com/`
  - standard IDE for R
  - open source (desktop version)
  - numerous integrated tools
- tidyverse `https://www.tidyverse.org/`
  - collection of data science oriented packages
  - try to enforce good practices
  - associated book `https://r4ds.had.co.nz/`

Outline

Introduction

Main concepts

Data Management in R

Programming Language

Definition

- a formal language with a strict mathematical definition
- defines syntactically correct programs
- associated to a semantics
  - (formal) model of the computer
  - effects of a program on the model

Programming Language

Definition

- a formal language with a strict mathematical definition
- defines syntactically correct programs
- associated to a semantics
  - (formal) model of the computer
  - effects of a program on the model

In other words...

- a programming language can be used to write programs $\simeq$ texts
- a programming language has a strict syntax
  - lexical aspects $\simeq$ word spelling
  - grammatical aspects $\simeq$ sentence level
- when a program follows the syntax, it has a proper meaning i.e. an effect on the computer on which it runs

A computer

Turing Machine

- standard mathematical model
- too low level to a daily use

Other models

- data oriented models
- a model of the data
- together with a model of the execution of a program
  - effects of instructions on the data $\simeq$ sentence level
  - global flow and organization on a program $\simeq$ text level
- include input/output aspects

Interactive mode

Standard program execution

- a program is written in a file (or a set of files)
- in some languages the file can be translated to a more efficient language
- the file (or its translation) is executed on a computer

Console/Shell

- some languages have an associated “console” or “shell” (e.g. Python and R)
- one can type interactively program sentences and get associated results
- simplifies learning and testing

R console

- R provides a console for interactive use
- in general integrated in a specific window of a programming environment (Rstudio)
- can be launched from the command line (`R`)
- command prompt `>`

R console

- R provides a console for interactive use
- in general integrated in a specific window of a programming environment (Rstudio)
- can be launched from the command line (`R`)
- command prompt >

```
|>
```

R console

- R provides a console for interactive use
- in general integrated in a specific window of a programming environment (Rstudio)
- can be launched from the command line (`R`)
- command prompt >

```
> 2 + 2
```

R console

- R provides a console for interactive use
- in general integrated in a specific window of a programming environment (Rstudio)
- can be launched from the command line (`R`)
- command prompt `>`

```
> 2 + 2
[1] 4
>
```

R console

- R provides a console for interactive use
- in general integrated in a specific window of a programming environment (Rstudio)
- can be launched from the command line (`R`)
- command prompt >

```
> 2 + 2
[1] 4
> 4 ^ 3
```

R console

- R provides a console for interactive use
- in general integrated in a specific window of a programming environment (Rstudio)
- can be launched from the command line (`R`)
- command prompt >

```
> 2 + 2
[1] 4
> 4 ^ 3
[1] 64
>
```

R console

- R provides a console for interactive use
- in general integrated in a specific window of a programming environment (Rstudio)
- can be launched from the command line (`R`)
- command prompt >

```
> 2 + 2
[1] 4
> 4 ^ 3
[1] 64
>
```

Warning

The behavior of a program in the console is not exactly the same as the behavior of a program outside of the console

Using the console

Console presentation

In the slides

- code to type given directly
- outputs given as comments ####

```
4 + 2 * 5
## [1] 14
```

Integrated help

- `?foo` gives the documentation of `foo`
- direct access in Rstudio

R console as a calculator

Numerical

```r
2 + 3.5
## [1] 5.5

4 - 2/3
## [1] 3.333333

5%/%2
## [1] 2

5%%2
## [1] 1

7^2
## [1] 49

sqrt(3)
## [1] 1.732051

sin(0.5 * pi)
## [1] 1
```

Logical

```r
5 > 5
## [1] FALSE

sqrt(5) <= 3
## [1] TRUE

5 == 5
## [1] TRUE

(2 > 3) | (3 > 2)
## [1] TRUE

(5 > 0) & (5^2 > 20)
## [1] TRUE
```

R data model

R understands

- numerical values
  - both integers `12`
  - and decimal numbers `1.4e-5`
- logical values **`TRUE`** and **`FALSE`**
- names such `sin` and **`pi`**

Object oriented

- R handles *objects*
- vectors, matrices, lists, data frames, etc.
- each object has a class which specifies
  - the possible values for objects of the class
  - the operations than can be performed on the objects
- values are by default put into vectors

Variables and vectors

Variables

- a variable is a name for an object
- assignment with = or <-
- integrated "calculator"
- standard behavior

```
a <- 2
b <- a + 3
a <- 1
a
## [1] 1

b
## [1] 5
```

Vector

- the main object type
- `c(2, 3, 4)`: a vector with 3 coordinates
- `[t]`: access/modification operator (numbering starts at 1)

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

x[2]
## [1] 3

x[1] <- x[1] + 7
x
## [1] 9 3 4
```

Vector indexing

Sequences

- `a:b` vector of integers from `a` to `b` (included)
- `seq(a, b)` same vector
- `seq(a, b, by)` vector of integers `a`, `a+by`, up to `b`

```
1:3
## [1] 1 2 3

-2:2
## [1] -2 -1  0  1  2

-1:-5
## [1] -1 -2 -3 -4 -5

seq(1, 4, 2)
## [1] 1 3

seq(2, -3, -2)
## [1]  2  0 -2
```

Vector indexing

- a vector can be indexed by a vector of integers
- `x[y]` gives the vector of values from `x` at the positions selected by `y`

```
x <- 5:10
x
## [1]  5  6  7  8  9 10

x[c(1, 2)]
## [1] 5 6

x[seq(1, 5, 2)]
## [1] 5 7 9
```

- negative indexes exclude elements

```
x[c(-3, -4)]
## [1]  5  6  9 10
```

Functions and packages

Principle

- R is mostly extended via packages that provide functions
- a function has a name, takes parameters and generally returns a result
- numerous built-in functions (`seq`)

Some built-in functions

- `library(foo)`: loads package `foo`
- `class(x)`: gives the class of an object

```r
class(c(1.5, 2, 3))
## [1] "numeric"
```

More built-in functions

- `length(x)`: length of a vector
- `sum(x)`, `mean(x)`, `median(x)`, `sd(x)`, etc.: statistics on a vector

```r
y <- c(2, -3, 1, 4, 5)
length(y)
## [1] 5

sum(y)
## [1] 9

mean(y)
## [1] 1.8

median(y)
## [1] 2

sd(y)
## [1] 3.114482
```

Function parameters

Parameters

- R functions have frequently many parameters
- default values are provided
- how to override the defaults?

Positional parameters

- `seq(from = 1, to = 1, by, length.out, ...)`
- during a call, parameters are considered in order
- in `seq(0)`, `from` takes the value 0
- in `seq(2, 4)`, `from` is 2 and `to` is 4

```
seq(-1)
## [1]  1  0 -1

seq(2, -4)
## [1]  2  1  0 -1 -2 -3 -4
```

Named parameters

Limitation

- many parameters with default values
- what about overriding only the last one?

Named parameters

- after positional parameters, one can give *named* parameters
- direct assignment for those parameters
- in `seq(10, by=-2)`, `from=10`, `to=1` (default value) and `by=-2`

```
seq(10, by = -2)
## [1] 10  8  6  4  2

seq(by = 3, to = 10)
## [1]  1  4  7 10
```

Types

Base types

- vectors contain values of the same *type*
- standard types:
  - logical (TRUE, FALSE)
  - integer
  - double (real numbers)
  - complex
  - character
- special values
  - **NA**: missing value, not a number
  - **NULL**: no value
- `typeof` function

Examples

```
class(c(1, 2))
## [1] "numeric"

typeof(c(1, 2))
## [1] "double"

typeof(c(1L, 2L))
## [1] "integer"

class(c("abcd", "efgh"))
## [1] "character"

class(c(TRUE, TRUE, FALSE))
## [1] "logical"

class(c(1 + (0+2i), 0+4i, -2.75 -
    (0+1i)))
## [1] "complex"
```

Factors

Nominal variables

- a.k.a. categorical variables
- finite number of possible values (e.g. gender)
- sometimes ordered

Factor

- R representation of nominal variables
- the `levels` are the values

Examples

```
x <- factor(c("A", "B", "A", "A"),
    levels = c("A", "B", "C"))
x
## [1] A B A A
## Levels: A B C

as.numeric(x)
## [1] 1 2 1 1

x[3] <- "D"

## Warning in `[<-.factor`(`*tmp*`,
3, value = "D"):  invalid factor
level, NA generated

x
## [1] A    B    <NA> A
## Levels: A B C
```

Outline

Introduction

Main concepts

Data Management in R

- Data Frame
- Data import
- Data transformation

Data Frame

Tabular data

- standard data representation
- each row is an object
- each column is a variable
- R version: a data frame (of class `data.frame`)

Example

```r
data(iris)  # loads the standard iris dataset
class(iris)
## [1] "data.frame"

head(iris)
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

Data Frame

Global operations

- `View(iris)`: spreadsheet like view of the data frame
- dimensions with `dim`
- names of the variables with `names`
- statistical summary

Example

```
dim(iris)
## [1] 150   5

names(iris)
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"

summary(iris)
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width          Species
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100   setosa    :50
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300   versicolor:50
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300   virginica :50
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
```

Data Frame

Columns

- columns are vectors (single type per column)
- named and positional access

```
names(iris)
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length"
## [4] "Petal.Width"  "Species"

iris[[1]][1:5]
## [1] 5.1 4.9 4.7 4.6 5.0

iris$Sepal.Width[2:4]
## [1] 3.0 3.2 3.1

iris[["Species"]][10:13]
## [1] setosa setosa setosa setosa
## Levels: setosa versicolor virginica
```

Tibble

Modern data frames

- replacement for `data.frame`: does less but better
- part of the tidyverse
- package `tibble`, class `tbl_df`, creation `tibble`, conversion `as_tibble`

Example

```
iris_tbl <- as_tibble(iris)
iris_tbl
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length
##           <dbl>       <dbl>        <dbl>
##  1          5.1         3.5          1.4
##  2          4.9         3            1.4
##  3          4.7         3.2          1.3
##  4          4.6         3.1          1.5
##  5          5           3.6          1.4
##  6          5.4         3.9          1.7
##  7          4.6         3.4          1.4
##  8          5           3.4          1.5
##  9          4.4         2.9          1.4
## 10          4.9         3.1          1.5
## # ... with 140 more rows, and 2 more variables:
## #   Petal.Width <dbl>, Species <fct>
```

Reccurring example

Bank dataset

- sources
  - https://archive.ics.uci.edu/ml/datasets/Bank%2BMarketing
  - http://hdl.handle.net/1822/14838
- data types
  - age: integer
  - balance: integer
  - education: categorical semi ordered
  - most of the others: categorical with some binary

Importing data

CSV files

- comma-separated values `2.5,-3.2,A`
- de factor standard (but poor quality)
- local variants (e.g. French!)

R import

- use the readr package
- `read_csv`, `read_csv2`, `read_tsv`, `read_delim`
- load a `tbl` (tibble)

Example

```
library(readr)
bank <-
  read_delim("../data/bank.csv",";")
class(bank)
## [1] "spec_tbl_df" "tbl_df"
## [3] "tbl"         "data.frame"

names(bank)
##  [1] "age"       "job"
##  [3] "marital"   "education"
##  [5] "default"   "balance"
##  [7] "housing"   "loan"
##  [9] "contact"   "day"
## [11] "month"     "duration"
## [13] "campaign"  "pdays"
## [15] "previous"  "poutcome"
## [17] "y"

class(bank$age)
## [1] "numeric"

class(bank$job)
## [1] "character"
```

Importing data

Builtin CSV import

- functions `read.csv`, `read.csv2`, `read.table`
- numerous limitations
  - (very) slow and memory hungry
  - no support for “fancy” variable names
  - automatic conversion to factors (sometimes useful!)

Example

```r
bank <-
  read.table("../data/bank.csv",
             header=TRUE,sep=";")
class(bank)
## [1] "data.frame"

names(bank)
##  [1] "age"       "job"
##  [3] "marital"   "education"
##  [5] "default"   "balance"
##  [7] "housing"   "loan"
##  [9] "contact"   "day"
## [11] "month"     "duration"
## [13] "campaign"  "pdays"
## [15] "previous"  "poutcome"
## [17] "y"

class(bank$age)
## [1] "integer"

class(bank$job)
## [1] "character"
```

Importing data

Other formats

- numerous other input/output formats (and data sources)
- native R: rds files `saveRDS` `readRDS`
- R/python compatible: feather
- SPSS/SAS/Stata: haven
- excel: readxl
- database connection: DBI
- OLAP connection (windows specific)

Example

```r
library(readxl)
bank <-
  read_excel("../data/bank.xlsx")
class(bank)
## [1] "tbl_df"     "tbl"
## [3] "data.frame"

names(bank)
##  [1] "age"        "job"
##  [3] "marital"    "education"
##  [5] "default"    "balance"
##  [7] "housing"    "loan"
##  [9] "contact"    "day"
## [11] "month"      "duration"
## [13] "campaign"   "pdays"
## [15] "previous"   "poutcome"
## [17] "y"

class(bank$age)
## [1] "numeric"

class(bank$job)
## [1] "character"
```

Sub-setting

Using dplyr (and magrittr for the pipe operator)

Data subset

- `filter` function
- keeps in a tibble rows that match the conditions

Example (standard syntax)

```
library(dplyr)
bank <- read_delim("../data/bank.csv",";")
dim(bank)
## [1] 4521   17

subbank <- filter(bank, marital=="married", age>40,
                  education=="secondary")
dim(subbank)
## [1] 731  17
```

Sub-setting

Using dplyr (and magrittr for the pipe operator)

Data subset

- `filter` function
- keeps in a tibble rows that match the conditions

Example (pipe operator)

```
library(dplyr)
bank <- read_delim("../data/bank.csv",";")
dim(bank)
## [1] 4521   17

subbank <- bank %>% filter(marital=="married", age>40,
                           education=="secondary")
dim(subbank)
## [1] 731  17
```

Column oriented subsetting

- two main motivations
  - to restrict the data set to variable types compatible with some technique
  - to restrict the data set to meaningful variables for an automated analysis (e.g. clustering or predictive modeling)
- simple declarative approach

Example

```
bank %>% select(age, balance, day, duration) %>% print(n = 6)
## # A tibble: 4,521 x 4
##     age balance   day duration
##   <dbl>   <dbl> <dbl>    <dbl>
## 1    30    1787    19       79
## 2    33    4789    11      220
## 3    35    1350    16      185
## 4    30    1476     3      199
## 5    59       0     5      226
## 6    35     747    23      141
## # ... with 4,515 more rows
```

Selecting and extracting variables

Dropping some variables

- `select` can be passed variable names prefixed by the − operator
- this removes those variables from the tibble

Example

```
print(dim(bank))
## [1] 4521   17

print(dim(bank %>% select(-age, -y)))
## [1] 4521   15
```

Extracting a variable

- `select(tbl, X)` is a tibble
- `pull(tbl, X)` extracts the columns in its native class

Example

```
print(class(bank %>% select(age)))
## [1] "tbl_df"
## [2] "tbl"
## [3] "data.frame"

print(class(bank %>% pull(age)))
## [1] "numeric"
```

Creating new variables

Principle

- a form of row oriented calculation
- create a new variable using the existing ones
- e.g. duration from starting and ending times

Support

- numerous statistical summary functions (column oriented)
- column oriented arithmetic (e.g. sum of columns)
- column oriented logical operations (e.g. comparison)
- function application (e.g. to each row)

Example

Bank data set

Binary variable telling whether some client has some characteristics

- more than average mean annual balance
- at least one loan

```
bank %>% mutate(moreavg = balance > mean(balance)) %>% select(moreavg,
    balance) %>% print(n = 2)
## # A tibble: 4,521 x 2
##   moreavg balance
##   <lgl>     <dbl>
## 1 TRUE       1787
## 2 TRUE       4789
## # ... with 4,519 more rows

bank %>% mutate(oneormoreloan = loan == "yes" | housing ==
    "yes") %>% select(oneormoreloan, loan, housing) %>%
    print(n = 2)
## # A tibble: 4,521 x 3
##   oneormoreloan loan  housing
##   <lgl>         <chr> <chr>
## 1 FALSE         no    no
## 2 TRUE          yes   yes
## # ... with 4,519 more rows
```

Creating new variables

Functions

- `mutate`: creates new variables and adds them to the data frame
- `transmute`: creates new variables and drops other variables
- `rename`

Example

```
bank %>% transmute(age > 40, oneormoreloan = loan == "yes" |
    housing == "yes") %>% rename(`At least one loan` = oneormoreloan) %>%
    print(n = 3)
## # A tibble: 4,521 x 2
##   `age > 40` `At least one loan`
##   <lgl>      <lgl>
## 1 FALSE      FALSE
## 2 FALSE      TRUE
## 3 FALSE      TRUE
## # ... with 4,518 more rows
```

Numerical summaries

Single value summary

- `summarise` function
- computes a value that summarises a variable
- e.g.
  - `mean`
  - `median`
  - `min`
  - etc.

Examples

```
bank %>%
    summarise(avg_balance=mean(balance),
              avg_age=mean(age))
## # A tibble: 1 x 2
##   avg_balance avg_age
##         <dbl>   <dbl>
## 1       1423.    41.2

bank %>%
    filter(marital=="married",
           education=="secondary") %>%
    summarise(avg_balance=mean(balance),
              avg_age=mean(age))
## # A tibble: 1 x 2
##   avg_balance avg_age
##         <dbl>   <dbl>
## 1       1273.    42.4
```

Conditional analysis

Finding dependencies and links

One of the main goal of data analysis, e.g.

- predictive models: links between target variables and explanatory variables
- frequent patterns: variables that are frequently non zero at the same time
- etc.

Conditional summaries

- chose one variable
- for each possible value of the variable
  - find all corresponding objects in the data set
  - compute a summary of the other variables on this subset

Conditional summaries

Conditional analysis

- group rows by values on some variables with `group_by`
- then `summarise` each group

Example

```
bank %>% group_by(marital) %>%
    summarise(nb = n(), avg_balance = mean(balance), avg_age = mean(age))
## # A tibble: 3 x 4
##   marital     nb avg_balance avg_age
##   <chr>    <int>       <dbl>   <dbl>
## 1 divorced   528       1122.    45.5
## 2 married   2797       1463.    43.5
## 3 single    1196       1460.    33.9
```

General mechanism

X	Y
2	A
1	B
4	B
3	B
1	B
2	C
1	A
3	C
3	A
2	A
2	A
3	A
3	B
1	A
1	A

Split

Y	X
A	2
A	1
A	3
A	2
A	2
A	3
A	1
A	1

Y	X
B	1
B	4
B	3
B	1
B	3

Y	X
C	2
C	3

Apply

Y	Sum
A	15

Y	Sum
B	12

Y	Sum
C	5

Combine

Y	Sum
A	15
B	12
C	5

Grouping on more than one variable

Principle

- groups are computed using all possible combinations of values of the grouping variables
- apply and combine work as for a single variable

Example

```
bank %>% group_by(marital, housing) %>%
    summarise(nb = n(), avg_balance = mean(balance), avg_age = mean(age))
## # A tibble: 6 x 5
## # Groups:   marital [3]
##   marital  housing    nb avg_balance avg_age
##   <chr>    <chr>   <int>       <dbl>   <dbl>
## 1 divorced no        230       1085.    48.0
## 2 divorced yes       298       1151.    43.6
## 3 married  no       1172       1766.    47.3
## 4 married  yes      1625       1245.    40.7
## 5 single   no        560       1448.    33.8
## 6 single   yes       636       1471.    34.0
```

Group by subtleties

Understanding group_by

- group_by adds to a tibble groups organized in layers
  - each grouping variable corresponds to a layer
  - layers order reproduces variables order in group_by
- most tibble operations take groups into account
- summarise consume a group layer (the last one, but see next slide)

Example

```
bank %>% group_by(marital) %>% mutate(nb = n()) %>% select(marital,
    nb) %>% print(n = 4)
## # A tibble: 4,521 x 2
## # Groups:   marital [3]
##   marital    nb
##   <chr>   <int>
## 1 married  2797
## 2 married  2797
## 3 single   1196
## 4 married  2797
## # ... with 4,517 more rows
```

Group by subtleties

Functions that preserve groups

- column/variable oriented functions preserve groups (e.g. `mutate` and co)
- `filter` preserve groups
- aggregates are group aware
  - e.g. `sum` computes the sum group by group
  - each row of group receives the same aggregate value

`summarise`

- aggregates are computed for each group
- the resulting tibble has one row per group
- it is grouped over all the variables except the last one: the last layer is removed (`dplyr<1.0`)
- the `.groups` parameter enables user control on the resulting grouping (experimental in `dplyr 1.0.0`)

Example

Original group by

```
bank %>% group_by(housing, loan, y) %>% print(n = 2)
## # A tibble: 4,521 x 17
## # Groups:   housing, loan, y [8]
##     age job   marital education default balance housing
##   <dbl> <chr> <chr>   <chr>     <chr>     <dbl> <chr>
## 1    30 unem~ married primary   no         1787 no
## 2    33 serv~ married secondary no         4789 yes
## # ... with 4,519 more rows, and 10 more variables:
## #   loan <chr>, contact <chr>, day <dbl>, month <chr>,
## #   duration <dbl>, campaign <dbl>, pdays <dbl>,
## #   previous <dbl>, poutcome <chr>, y <chr>
```

- 3 binary variables ⇒ 8 groups
- 3 layers: housing, loan and y

Example

Counting

```
bank %>% group_by(housing, loan, y) %>% summarise(n = n())
## # A tibble: 8 x 4
## # Groups:   housing, loan [4]
##   housing loan  y         n
##   <chr>   <chr> <chr> <int>
## 1 no      no    no     1394
## 2 no      no    yes     283
## 3 no      yes   no      267
## 4 no      yes   yes      18
## 5 yes     no    no     1958
## 6 yes     no    yes     195
## 7 yes     yes   no      381
## 8 yes     yes   yes      25
```

- 2 layers: housing, loan
- 2 binary variables $\Rightarrow$ 4 groups

Example

Counting and summing

```
bank %>% group_by(housing, loan, y) %>% summarise(n = n()) %>%
    mutate(nb = sum(n))
## # A tibble: 8 x 5
## # Groups:   housing, loan [4]
##   housing loan  y         n    nb
##   <chr>   <chr> <chr> <int> <int>
## 1 no      no    no     1394  1677
## 2 no      no    yes     283  1677
## 3 no      yes   no      267   285
## 4 no      yes   yes      18   285
## 5 yes     no    no     1958  2153
## 6 yes     no    yes     195  2153
## 7 yes     yes   no      381   406
## 8 yes     yes   yes      25   406
```

- 2 layers: housing, loan
- 2 binary variables $\Rightarrow$ 4 groups
- `sum` applies to the housing$\times$loan grouping

Example

Conditional distribution of y

```
bank %>% group_by(housing, loan, y) %>% summarise(n = n()) %>%
    mutate(freq = n/sum(n))
## # A tibble: 8 x 5
## # Groups:   housing, loan [4]
##   housing loan  y         n   freq
##   <chr>   <chr> <chr> <int>  <dbl>
## 1 no      no    no     1394 0.831
## 2 no      no    yes     283 0.169
## 3 no      yes   no      267 0.937
## 4 no      yes   yes      18 0.0632
## 5 yes     no    no     1958 0.909
## 6 yes     no    yes     195 0.0906
## 7 yes     yes   no      381 0.938
## 8 yes     yes   yes      25 0.0616
```

- 2 layers: housing, loan
- 2 binary variables $\Rightarrow$ 4 groups
- the frequency of y is computed in each group

Limitation of `summarise` prior dplyr 1.0

Multiple values in an aggregate

- in `dplyr<1.0`
- an aggregation function cannot return multiple values
- for instance

  ```
  bank %>% group_by(marital) %>% summarise(q_age = quantile(age))
  ```

  fails with the message

  ```
  Error: Column `q_age` must be length 1 (a summary value), not 5
  ```

Workaround for dplyr < 1.0

```
bank %>% group_by(marital) %>%
    summarise(q_age=list(enframe(quantile(age)))) %>%
        unnest(q_age) %>% rename(quantile=name,age=value)
## # A tibble: 15 x 3
##    marital  quantile   age
##    <chr>    <chr>    <dbl>
##  1 divorced 0%          26
##  2 divorced 25%         37
##  3 divorced 50%         45
##  4 divorced 75%         53
##  5 divorced 100%        84
##  6 married  0%          23
##  7 married  25%         35
##  8 married  50%         42
##  9 married  75%         51
## 10 married  100%        87
## 11 single   0%          19
## 12 single   25%         29
## 13 single   50%         32
## 14 single   75%         37
## 15 single   100%        69
```

Since dplyr 1.0.0

Multiple values in an aggregate

- no particular problem
- vector values are handled as multiple single values: one row per value in the result

Example

```
bank %>% group_by(marital) %>% summarise(q_age = quantile(age,
    probs = c(0.25, 0.5, 0.75)))
## # A tibble: 9 x 2
## # Groups:   marital [3]
##   marital  q_age
##   <chr>    <dbl>
## 1 divorced    37
## 2 divorced    45
## 3 divorced    53
## 4 married     35
## 5 married     42
## 6 married     51
## 7 single      29
## 8 single      32
## 9 single      37
```

Mutliple columns in `summarise`

Data frame like results

- aggregate functions can return data frames
- each column corresponds to a column in the result table
- multiple rows are handled as in the case of vector valued aggregates

Example

```
bank %>% group_by(marital) %>% summarise(tibble(q_age = quantile(age,
   probs = c(0.25, 0.5, 0.75)), probs = c(0.25, 0.5, 0.75)))
## # A tibble: 9 x 3
## # Groups:   marital [3]
##   marital  q_age probs
##   <chr>    <dbl> <dbl>
## 1 divorced    37  0.25
## 2 divorced    45  0.5
## 3 divorced    53  0.75
## 4 married     35  0.25
## 5 married     42  0.5
## 6 married     51  0.75
## 7 single      29  0.25
## 8 single      32  0.5
## 9 single      37  0.75
```

Multidimensional analysis

Dimensions

- variables with finite number of values
- each cell summarise the original data for a given combination of the values of the dimensions
- this is exactly `group_by`

Measures

- variables with numerical values
- aggregated in each cell
- this is `summarise`

Multidimensional analysis

Dimensions

- variables with finite number of values
- each cell summarise the original data for a given combination of the values of the dimensions
- this is exactly `group_by`

Measures

- variables with numerical values
- aggregated in each cell
- this is `summarise`

Tidy data

- but the results is not a pivot table
- data scientist keep data tidy
  - each column is a variable
  - each row is in object
- `group_by %>% summarise`
  - rows: group
  - column: original variables + aggregated values
  - from objects to groups
- tidyr allows to switch from tidy data to untidy data (and vice versa)

Spreading data

From tidy data to pivot table

- `pivot_wider` function (`spread` in older versions)
- operates on two variables in the original table: a key and a value
- each value taken by the key becomes a column
- the value variable is used to fill the column

Original table

X	Y	Z
1	A	2
1	B	3
2	A	4
2	B	5

Spread table

Y is the key, Z is the value

X	A	B
1	2	3
2	4	5

Example

Principle

- `names_from`: the key variable(s)
- `values_from`: the value variable

Tidy MDA

```
bank %>% group_by(marital, education) %>%
    summarise(nb = n())
## # A tibble: 12 x 3
## # Groups:   marital [3]
##    marital  education    nb
##    <chr>    <chr>     <int>
##  1 divorced primary      79
##  2 divorced secondary   270
##  3 divorced tertiary    155
##  4 divorced unknown      24
##  5 married  primary     526
##  6 married  secondary  1427
##  7 married  tertiary    727
##  8 married  unknown     117
##  9 single   primary      73
## 10 single   secondary   609
## 11 single   tertiary    468
## 12 single   unknown      46
```

Pivot table

```
bank %>% group_by(marital, education) %>%
    summarise(nb = n()) %>%
    pivot_wider(names_from=marital,
                values_from=nb)
## # A tibble: 4 x 4
##   education divorced married single
##   <chr>        <int>   <int>  <int>
## 1 primary         79     526     73
## 2 secondary      270    1427    609
## 3 tertiary       155     727    468
## 4 unknown         24     117     46
```

Example

Tidy MDA

```
bank %>% group_by(housing, loan,
    y) %>% summarise(n = n()) %>%
    mutate(freq = n/sum(n)) %>%
    select(-n)
## # A tibble: 8 x 4
## # Groups:   housing, loan [4]
##   housing loan  y       freq
##   <chr>   <chr> <chr>  <dbl>
## 1 no      no    no    0.831
## 2 no      no    yes   0.169
## 3 no      yes   no    0.937
## 4 no      yes   yes   0.0632
## 5 yes     no    no    0.909
## 6 yes     no    yes   0.0906
## 7 yes     yes   no    0.938
## 8 yes     yes   yes   0.0616
```

Pivot table

```
bank %>% group_by(housing, loan,
    y) %>% summarise(n = n()) %>%
    mutate(freq = n/sum(n)) %>%
    select(-n) %>% pivot_wider(names_from = y,
    values_from = freq)
## # A tibble: 4 x 4
## # Groups:   housing, loan [4]
##   housing loan     no    yes
##   <chr>   <chr> <dbl>  <dbl>
## 1 no      no    0.831 0.169
## 2 no      yes   0.937 0.0632
## 3 yes     no    0.909 0.0906
## 4 yes     yes   0.938 0.0616
```

Contingency table

Cross tabulation of two variables

```
bank %>% group_by(marital, education) %>%
    summarise(nb = n()) %>%
    pivot_wider(names_from=marital,
                values_from=nb) %>%
    select(-education)
## # A tibble: 4 x 3
##   divorced married single
##      <int>   <int>  <int>
## 1       79     526     73
## 2      270    1427    609
## 3      155     727    468
## 4       24     117     46
```

Dependency tests

- feed the contingency table to a dependency test
- e.g. the χ-squared test

```
bank %>% group_by(marital, education) %>%
    summarise(nb = n()) %>%
    pivot_wider(names_from=marital,
                values_from=nb) %>%
    select(-education) %>%
    chisq.test()
##
##  Pearson's Chi-squared
##  test
##
## data:  .
## X-squared = 139.09, df =
## 6, p-value < 2.2e-16
```

Multi-valued summaries

Tidy MDA

```
bank %>% group_by(marital) %>%
    summarise(tibble(q_age = quantile(age),
        probs = c(0, 0.25, 0.5,
            0.75, 1)))
## # A tibble: 15 x 3
## # Groups:   marital [3]
##    marital  q_age probs
##    <chr>    <dbl> <dbl>
##  1 divorced    26  0
##  2 divorced    37  0.25
##  3 divorced    45  0.5
##  4 divorced    53  0.75
##  5 divorced    84  1
##  6 married     23  0
##  7 married     35  0.25
##  8 married     42  0.5
##  9 married     51  0.75
## 10 married     87  1
## 11 single      19  0
## 12 single      29  0.25
## 13 single      32  0.5
## 14 single      37  0.75
## 15 single      69  1
```

Pivot table

```
bank %>% group_by(marital) %>%
    summarise(tibble(q_age=quantile(age),
            probs=c(0,0.25,0.5,0.75,1))) %>%
    pivot_wider(names_from=probs,
            values_from=q_age)
## # A tibble: 3 x 6
## # Groups:   marital [3]
##   marital   `0` `0.25` `0.5` `0.75`
##   <chr>   <dbl>  <dbl> <dbl>  <dbl>
## 1 divorc~    26     37    45     53
## 2 married    23     35    42     51
## 3 single     19     29    32     37
## # ... with 1 more variable: `1` <dbl>
```

Spreading

Spreading to Tidy

- spreading data can improve the tidiness
- when an object is described by several rows

Spreading

Spreading to Tidy

- spreading data can improve the tidiness
- when an object is described by several rows

Untidy

- flow (in number of persons) in and out a building
- direction encoded in the flow variable
- time periods should be the objects

Example

```
calit
## # A tibble: 10,080 x 4
##     flow date       time   count
##    <int> <date>     <time> <dbl>
##  1     7 2005-07-24 00:00      0
##  2     9 2005-07-24 00:00      0
##  3     7 2005-07-24 00:30      1
##  4     9 2005-07-24 00:30      0
##  5     7 2005-07-24 01:00      0
##  6     9 2005-07-24 01:00      0
##  7     7 2005-07-24 01:30      0
##  8     9 2005-07-24 01:30      0
##  9     7 2005-07-24 02:00      0
## 10     9 2005-07-24 02:00      0
## # ... with 10,070 more rows
```

Spreading

Spread version

```
tcalit <- calit %>%
    pivot_wider(names_from=flow,values_from=count) %>%
    rename(flowin = `7`, flowout = `9`)
tcalit
## # A tibble: 5,040 x 4
##    date       time   flowin flowout
##    <date>     <time>  <dbl>   <dbl>
##  1 2005-07-24 00:00       0       0
##  2 2005-07-24 00:30       1       0
##  3 2005-07-24 01:00       0       0
##  4 2005-07-24 01:30       0       0
##  5 2005-07-24 02:00       0       0
##  6 2005-07-24 02:30       2       0
##  7 2005-07-24 03:00       0       0
##  8 2005-07-24 03:30       0       0
##  9 2005-07-24 04:00       0       0
## 10 2005-07-24 04:30       0       0
## # ... with 5,030 more rows
```

Gathering data

Pivot longer

- `pivot_longer` is the reverse operation of `pivot_wider` (`gather` in older versions)
- it reduces the number of columns by encoding them as a series of rows and two new columns/variables
- the new key variable encode the gathered column while the new value variable contains the original value

Original table

Gather X, Y and Z

W	X	Y	Z
a	1	2	3
b	5	6	7

Spread table

W	K	V
a	X	1
a	Y	2
a	Z	3
b	X	5
b	Y	6
b	Z	7

Example

Wide data

- weekly product sales
- one row per product, one column per week: product view

```
prodperweek
## # A tibble: 811 x 53
##    Product_Code    W0    W1    W2    W3    W4    W5    W6    W7    W8    W9   W10   W11
##    <chr>        <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1 P1              11    12    10     8    13    12    14    21     6    14    11    14
##  2 P2               7     6     3     2     7     1     6     3     3     3     2     2
##  3 P3               7    11     8     9    10     8     7    13    12     6    14     9
##  4 P4              12     8    13     5     9     6     9    13    13    11     8     4
##  5 P5               8     5    13    11     6     7     9    14     9     9    11    18
##  6 P6               3     3     2     7     6     3     8     6     6     3     1     1
##  7 P7               4     8     3     7     8     7     2     3    10     3     5     2
##  8 P8               8     6    10     9     6     8     7     5    10    10     8     8
##  9 P9              14     9    10     7    11    15    12     7    13    12    15    15
## 10 P10             22    19    19    29    20    16    26    20    24    20    31    22
## # ... with 801 more rows, and 40 more variables: W12 <dbl>, W13 <dbl>, W14 <dbl>,
## #   W15 <dbl>, W16 <dbl>, W17 <dbl>, W18 <dbl>, W19 <dbl>, W20 <dbl>, W21 <dbl>,
## #   W22 <dbl>, W23 <dbl>, W24 <dbl>, W25 <dbl>, W26 <dbl>, W27 <dbl>, W28 <dbl>,
## #   W29 <dbl>, W30 <dbl>, W31 <dbl>, W32 <dbl>, W33 <dbl>, W34 <dbl>, W35 <dbl>,
## #   W36 <dbl>, W37 <dbl>, W38 <dbl>, W39 <dbl>, W40 <dbl>, W41 <dbl>, W42 <dbl>,
## #   W43 <dbl>, W44 <dbl>, W45 <dbl>, W46 <dbl>, W47 <dbl>, W48 <dbl>, W49 <dbl>,
## #   W50 <dbl>, W51 <dbl>
```

Example

Gather the weeks

- gather all the columns except the product one
- object: (product, week)

```
prodperweek %>%
    pivot_longer(!Product_Code,
                 names_to="Week",
                 values_to="Sales")
## # A tibble: 42,172 x 3
##    Product_Code Week  Sales
##    <chr>        <chr> <dbl>
##  1 P1           W0       11
##  2 P1           W1       12
##  3 P1           W2       10
##  4 P1           W3        8
##  5 P1           W4       13
##  6 P1           W5       12
##  7 P1           W6       14
##  8 P1           W7       21
##  9 P1           W8        6
## 10 P1           W9       14
## # ... with 42,162 more rows
```

Example

Gather the weeks

- gather all the columns except the product one
- object: (product, week)

```
prodperweek %>%
    pivot_longer(!Product_Code,
                 names_to="Week",
                 values_to="Sales")
## # A tibble: 42,172 x 3
##    Product_Code Week  Sales
##    <chr>        <chr> <dbl>
##  1 P1           W0       11
##  2 P1           W1       12
##  3 P1           W2       10
##  4 P1           W3        8
##  5 P1           W4       13
##  6 P1           W5       12
##  7 P1           W6       14
##  8 P1           W7       21
##  9 P1           W8        6
## 10 P1           W9       14
## # ... with 42,162 more rows
```

Recoding

```
prodperweek %>%
    pivot_longer(!Product_Code,
                 names_to="Week",
                 values_to="Sales") %>%
    mutate(Week=parse_number(Week))
## # A tibble: 42,172 x 3
##    Product_Code  Week Sales
##    <chr>        <dbl> <dbl>
##  1 P1               0    11
##  2 P1               1    12
##  3 P1               2    10
##  4 P1               3     8
##  5 P1               4    13
##  6 P1               5    12
##  7 P1               6    14
##  8 P1               7    21
##  9 P1               8     6
## 10 P1               9    14
## # ... with 42,162 more rows
```

Example

Spreading again

- Week point of view

```
prodperweek %>% pivot_longer(!Product_Code, names_to = "Week",
    values_to = "Sales") %>% mutate(Week = parse_number(Week)) %>%
    pivot_wider(names_from = Product_Code, values_from = Sales) %>%
    print(10, n_extra = 0)
## # A tibble: 52 x 812
##     Week    P1    P2    P3    P4    P5    P6    P7    P8
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1     0    11     7     7    12     8     3     4     8
##  2     1    12     6    11     8     5     3     8     6
##  3     2    10     3     8    13    13     2     3    10
##  4     3     8     2     9     5    11     7     7     9
##  5     4    13     7    10     9     6     6     8     6
##  6     5    12     1     8     6     7     3     7     8
##  7     6    14     6     7     9     9     8     2     7
##  8     7    21     3    13    13    14     6     3     5
##  9     8     6     3    12    13     9     6    10    10
## 10     9    14     3     6    11     9     3     3    10
## # ... with 42 more rows, and 803 more variables
```

R

- R is with Python the *de facto* standard for data science
- R can be extended by thousands of packages
- R can be use to implement extremely efficient data processing pipelines on large scale data
- R support several usage level from basic scripting to advanced programming
- ongoing efforts to simplify R use (e.g. blueSky statistics and jamovi)

For more...

R for Data Science

Licence

Version

Last git commit: 2020-10-06
By: Fabrice Rossi (Fabrice.Rossi@apiacoa.org)
Git hash: d3eaab94717e93344a41efec611d1abf20ef8708

Changelog

- Septembre 2020:
  - added tibbles
  - updated to dplyr 1.0
  - updated to tidyr 1.0
- January 2020:
  - improved the coverage of `group_by`
  - added variable oriented operations
  - added data source
- September 2019: initial version