R Basics

Marco Torchiano

Version 1.1.0 - March 2021

License

Introduction

What is R?

http://cran.r-project.org/

R is a free software environment for statistical computing and graphics.
Available on several different platform

Basic features

- CLI
  - Command Line Interface
  - Immediate evaluation of expression
- Scripts
- Extensive help system https://www.rseek.org/
- Large resource set online https://stackoverflow.com/

IDE

- Several graphical front-ends (GUI)
- RStudio is a full IDE for R

  http://www.rstudio.com
- Also cloud version: https://rstudio.cloud

R Console

Basic text based REPL

- Read: from the user keyboard
  - Or from a script
- Evaluate the R language expression
- Print the result of the evaluation
- Loop
  - Until `quit()`

Objects are stored in a common environment

R environment

- Shared global memory space where all objects stored
  - Variables
  - Functions
- Can be inspected at any time
- Every time a command assign a value to a variable, it is placed inside the environment
- All valuee in the environment are available to any later statement

R script

- A text file containing commands intended to be executed as a whole
- It is possible to execute the statements one by one
  - The result is the same
- Execution means taking a statement from the script instead of reading it from the keyboard
  - Accesses the global environment

R package

- Library of functions designed to work together
  - Include documentation
- Can be installed from R official repository (CRAN)
  - From CLI: `install.packages("ggplot2")`
  - From GUI: Tools > Install packages…
- Must be loaded before use
  - `library("ggplot2")`

R help

- All built-in and package functions are documented
- Help system is integrated in
  - Console
    - Help on function `? log`
    - Search for topic: `?? logarithm`
  - R Studio
    - Help pane

R Language

R elements

- Statements
    - assignment
    - expression
    - control
- Functions
- Variables
- Data types
    - Primitive
    - Compound

Statements

Statements can be terminated by

- a *new-line* : most common
- a `;`
    - to avoid ambiguities
    - to put multiple statements on a single line

Comments

On any line from `#` until end of line is considered comments. Typical usage:

- `#` as first caracter: comment line
- `#` after statement: comment specific statement

```
#--------------- Define constants --------------
#
PI <- 7/22 # a reasonable approximation
```


Assignment

The global environment stores objects, e.g. values

Operator `<-` is used to store an object with a name

```
answer <- "fortytwo"
```

Variables are not *typed*

- i.e. you can (re-)assign any type of value

  ```
  answer <- 42
  ```

Assignment

Assignment operator `<-` copies the value of an expression into the environment and assign a name

- Operator `=` can be used instead
- Non recommended to avoid confusion

An assignment overwrites the value previously linked to that name

- Be careful with names

Names

- Variable names
  - Must start with a letter,
  - Can’t contain spaces
- Style recommendations:
  - Use lowercase characters
  - Use an underscore (`_`) to separate words
  - Avoid using names that are predefined

Expression

- When an expression is entered, R evaluates it and prints the result
- Uses the names to retrieve values from the environment

```
answer
```

```
## [1] 42
```

- it is possible to explicitly force printing an expression with `print()`

```
print( (answer / 3) %% 11)
```

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


Primitive types

`numeric`

- Default type (also for integer values)
- Uses standard IEEE-754 (ISO/IEC 60559)
  - E.g., `1.2` , `1`

`integer`

- Used to force integer arithmetic
- Suffix letter “`L`” to force integer
  - E.g., `42L`

Primitive types

`complex`

- Allow complex number operations

```
sqrt( -1 + 0i )
```

```
## [1] 0+1i
```

`logical`

- Keywords: `TRUE` | `FALSE`
- Also predefined variables: `T` and `F`

Primitive types

`character`

- String of characters
- Can be described using both
  - `'single'` and
  - `"double"` quotes

Data types

Type-related functions:

- Type of variable: `class(x)`
- Check type: `is.`*`type`*`(x)`
- Conversion: `as.`*`type`*`(x)`

Special values

- `NA` : is generally interpreted as a missing, does not exist
  - Stands for Not Available
  - tested with `is.na()`
- `NULL` : is for empty object
  - tested with `is.null()`
- `NaN` : the result is not a number, e.g. `log(-1)`
  - Stands for Not-a-N
  - tested with `is.nan()`
- Inf numeric infinity ∞, e.g. `1/0`

Operators

- Arithmetic on numeric: `+`, `-`, `*`, `/`, `^`
  - integer `%%` (modulo)
- Comparison: `==`, `!=`, `<`, `<=`, `>`, `>=`
  - works also on strings

Character operations

- `nchar()` : lenght of the string
- `paste( ..., sep=" ")`: concatenates with separator
  - `paste0(...)`: no separator, i.e. `sep=""`

```
nchar("Visualization")
```

```
## [1] 13
```

```
paste("Visualization","of","Quantitative","Informatic
```

```
## [1] "Visualization of Quantitative Information"
```

Character operations

- `substr()` : extract and replaces portion of a string

```
title <- "Visualization of Quantitative Information"
substr(title, 15, 16)
```

```
## [1] "of"
```

```
substr(title, 15, 16) <- "OF"
title
```

```
## [1] "Visualization OF Quantitative Information"
```


Block statements

A series of statements can be gathered in a block using the `{` … `}` syntax.

- they are treated as a single (compound) statement
- non new environment is created

Block statement are used as branches or bodies of structured control statements.

Control statements

- `if( cond )` .. `else`
- `while( cond )`
- `for( var in seq )`

Conditional

Use the usual syntax: `if(cond)` … `else` …

- else clause is optional

```
a <- 10
if( a < 0){
  "negative"
}else{
  "positive"
}
```

```
## [1] "positive"
```

While loop

Use the `while(cond)` … syntax

```
a <- 10
while( a > 1){
  print(a)
  a <- a / 2;
}
```

```
## [1] 10
## [1] 5
## [1] 2.5
## [1] 1.25
```


Functions definition

Using the keyword `function`

```
percentage <- function(part,whole){
  part/whole*100
}
```

- return evaluation of last expression
- or can use `return()` statement

Can provide default values:

```
percentage <- function(part=1, whole=1){
  return( part/whole*100 )
}
```

Function invocation

Usual invocation (positional)

```
percentage(3, 4)
```

```
## [1] 75
```

Named arguments:

```
percentage(whole=4, part=3)
```

```
## [1] 75
```

Leverage default values:

```
percentage(part=0.75)
```

```
## [1] 75
```


Exercise 1

Define a function `pythagoras()` accepting three values (a,b,c) one of which can be missing and is computed using the Pythagorean theorem.

```
pythagoras(3,4)
```

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

```
pythagoras(c=5,a=3)
```

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

Vectors

Vectors

- All values in R are considered as vectors
  - Possibly with dimension 1 for scalar values
- When printed
  - if spread on many lines, the index of the first element printed on the line is shown in `[]`
  - for a scalar, `[1]` is shown indicating the index of the first and only element
- All elements in a vector must have the same type
  - Type coercion can be applied

Vector creation

- With *combine* function `c()` by enumeration of elements

```
v <- c(2,4,5)
```

  - Remember also scalars are vector: `1 == c(1)`

- With `vector()` function, with type and length, creates a zero-ed vector

```
w <- vector("numeric",3)
w
```

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


Ranges

Range operator `:` generates an integer vector

```
1:3
```

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

- equivalent to

```
c(1L, 2L, 3L)
```

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

Vector operations

- Merging:

```
c( 1:3, 7:9)
```

```
## [1] 1 2 3 7 8 9
```

  - Type coercion can be applied

- Length with function `length()`

```
length( 1:10 )
```

```
## [1] 10
```


Vector operations

Arithmetic operators

- Pair-wise on same-index elements

```
1:3 + 3:1
```

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

- Recycling if different size

```
1:3 + 1
```

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

  - Longest length must be multiple of shortest

Empty vectors

- Using primitive types function to create empty (typed) vectors

```
empty_numeric <- vector("numeric",0)
length(empty_numeric)
```

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

```
empty_numeric
```

```
## numeric(0)
```

- The combine function without arguments gives `NULL`
  - The reason is that no type is specified

Vector access

- Operator `[]`
- Uses an index to access an element

In R, indexes start at 1!!!

```
s = c("aa", "bb", "cc", "dd", "ee")
s[1]
```

```
## [1] "aa"
```

Vector access

- With index 0 returns an empty vector

```
s[0]
```

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

- Out of bound returns `NA`

```
s[6]
```

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


Vector slicing

Slicing allows extracting a subset of the vector elements

- Using a vector of indexes

```
s[ c(1,3) ]
```

```
## [1] "aa" "cc"
```

- Indexes can be repeated

```
s[c(5,1,1)]
```

```
## [1] "ee" "aa" "aa"
```

Vector slicing

- Using a vector of logicals

```
l <- c(TRUE, FALSE, FALSE, FALSE, TRUE)
s[ l ]
```

```
## [1] "aa" "ee"
```


For-Loops

For-loop sintax: `for(` `variable` `in` `vector` `)`

- in each iteration the `variable` will assume all the consecutive values in the `vector`.

```
min <- 100;
for( d in c(7,2,5,10,20,12,3) ){
  if( d < min)
    min <- d
}
min
```

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

For-Loops

Iteration on a vector can be implemented also with an index over a range

```
min <- 100;
numbers <- c(7,2,5,10,20,12,3)
for( i in 1:length(numbers) ){
  if( numbers[i] < min)
    min <- numbers[i]
}
min
```

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


Loop control

- `break` : steps out of the loop skipp rest of body
- `next` : skips remaining of the body and start new iteration

Named vectors

- Elements of a vector can be named

```
days<-c(Jan=31, Feb=28, Mar=31, Apr=30, May=31, Jun
        Jul=31, Aug=31, Sep=30, Oct=31, Nov=30, Dec
```

- When printed, names are reported above the values

```
days
```

```
## Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
##  31  28  31  30  31  30  31  31  30  31  30  31
```


Named vectors

- Names can be used instead of indexes

```
days["Feb"]
```

```
## Feb
##  28
```

- Also for slicing purposes

```
days[ c("Feb","Dec") ]
```

```
## Feb Dec
##  28  31
```

Named vectors

Function `names()` access names

- Allows getting and setting names

```
names(days)
```

```
##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Au
##  [9] "Sep" "Oct" "Nov" "Dec"
```

```
triplet <- 1:3
names(triplet) <- c("one","two","three")
triplet
```

```
##   one   two three
##     1     2     3
```


Exercise 2

Modify the `pythagoras()` function so that it returns a vector with three elements named ‘a’, ‘b’, and ‘c’ according to the Pythagorean theorem.

```
pythagoras(3,4)
```

```
## a b c
## 3 4 5
```

```
pythagoras(c=5,a=4)
```

```
## a b c
## 4 3 5
```

Character vector

- `strsplit(s, split)`: creates a a list of vectors of strings by splittig at given separator

```
strsplit(title, " ")
```

```
## [[1]]
## [1] "Visualization" "OF"                    "Quantitative"
## [4] "Information"
```


Character vector

Function `paste( ..., sep=" ", collapse)`:

- first concatenates strings at corresponding indexes (w/recycling) with separator
- then concatenates elements of the resulting vector

```
paste(1:3,c("one","two","three"),".")
```

```
## [1] "1 one ."   "2 two ."   "3 three ."
```

```
paste(1:3,c("one","two","three"),".", collapse=" - ")
```

```
## [1] "1 one . - 2 two . - 3 three ."
```

Sequences

Function `seq(from,to,by,lenght.out)` allows different combination of arguments

```
seq(1,10)   # by=1
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
seq(1,10,by=3)
```

```
## [1]  1  4  7 10
```

```
seq(1,10,length.out=4)
```

```
## [1]  1  4  7 10
```

```
seq(1,length.out=10) # by=1
```


Type coercion

When putting values of different type in the same vector they are (silently) coerced to the same type

- the most general type among the elements is used
- `character` > `complex` > `numeric` > `integer` > `logical`

```
c( 3, "two", TRUE)
```

```
## [1] "3"    "two"  "TRUE"
```

```
c( 22/7, 42L, FALSE)
```

```
## [1]  3.142857 42.000000  0.000000
```

Type coercion

Coercion is performed using the conversion functions `as.`*`type`*`()`

Not always conversion is possible, in such cases `NA` is produced

```
as.numeric(c("1","b","3.2"))
```

```
## Warning: NAs introduced by coercion
```

```
## [1] 1.0  NA 3.2
```

```
as.logical(c("true","FALSE","T","V","0"))
```

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


Logical to Numeric

When summing, logicals are coerced to integers

- `TRUE` → 1, `FALSE` → 0

```
thirty <- days == 30
thirty
```

```
##   Jan   Feb   Mar   Apr   May   Jun   Jul   Aug   T
## FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE  T
##   Oct   Nov   Dec
## FALSE  TRUE FALSE
```

```
sum(thirty) # how many (coercion: T->1 F->0)
```

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

Filtering vectors with logicals

```
names(days)[thirty]
```

```
## [1] "Apr" "Jun" "Sep" "Nov"
```


Filtering vectors with indexes

Function `which()` returns indexes of element satisfying condition (==`TRUE`)

```
thirty.ix <- which( days==30 )
thirty.ix
```

```
## Apr Jun Sep Nov
##   4   6   9  11
```

```
names(days)[thirty.ix]
```

```
## [1] "Apr" "Jun" "Sep" "Nov"
```

Sorting

Data in a vector can be sorted using function `sort()`

- Note: the original array is not modified

```
numbers <- c(3, 7, 14, 2, 5, 8)
sort(numbers)
```

```
## [1]  2  3  5  7  8 14
```

```
words <- c("There", "must", "be", "some", "kind",
           "of", "way", "out", "of", "here")
sort(words)
```

```
##  [1] "be"    "here"  "kind"  "must"  "of"    "of"
##  [7] "out"   "some"  "There" "way"
```


Ordering the indexes

Function **`order()`** sorts the **indexes** based on the value of the corresponding elements

- the first element of the result contains the index of the smallest element
- slicing with the ordered indexes gives a sorted vector

```
order(numbers)
```

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

```
numbers[ order(numbers) ] # slicing in order
```

```
## [1]  2  3  5  7  8 14
```

Ranking

Function `rank()` computes the ranks of the corresponding elements

```
r <- rank(numbers)
names(r) <- numbers
r
```

```
##  3  7 14  2  5  8
##  2  4  6  1  3  5
```


Matching

Operator `%in%` finds which element of left-hand vector are present in the right-hand one.

```
c("John","Jane","Mike","Iris") %in% c("Jane","Iris","
```

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

Vectorization

Often it is useful to apply a function to all elements in a vector

A *vectorized* function is one that can apply the same operation to all elements of it argument

- It is much easier to use and more efficient
- Most builtin functions are vectorized

Vectorization vs. loops

Specific functions are not always vectorized

```
score_to_grade <- function(score){
  if(score<17.5) "Failed"
  else if(score>=30.5) "30L"
  else round(score)
}
scores <-c(15,24.3,32,27.5)
score_to_grade(scores)
```

```
## Warning in if (score < 17.5) "Failed" else if (scc
## 30.5) "30L" else round(score): the condition has l
## > 1 and only the first element will be used
```

```
## [1] "Failed"
```

Vectorization vs. loops

A loop can be used to apply to all elements

```
grades <- numeric( length(scores) )
for( i in 1:length(grades)){
  grades[i] = score_to_grade(scores[i])
}
grades
```

```
## [1] "Failed" "24"     "30L"    "28"
```


Vectorization functionals

A *functional* is a function that applies another function

Functional `sapply()`:

- takes a vector and a function
- applies the function to all elements of the vector
- collects the results into a vector

```
grades <- sapply(scores,score_to_grade)
grades
```

```
## [1] "Failed" "24"     "30L"    "28"
```

Composed data types

Matrix

Construction:

```
matrix(1:9, 3, 3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
A <- matrix(1:9, 3, 3, byrow=TRUE);  A
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

Matrix indexing

Indexes start at 1, like vectors.

```
A[2, 3] # single cell
```

```
## [1] 6
```

```
A[2,  ] # row
```

```
## [1] 4 5 6
```

```
A[ , 3] # column
```

```
## [1] 3 6 9
```


Matrix indexing

```
A[2,3] <- 66; A
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5   66
## [3,]    7    8    9
```

Matrix transposition

```
B <- t(A);  B
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3   66    9
```


Matrix Composition

```
cbind(A,B) # column-wise
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    2    3    1    4    7
## [2,]    4    5   66    2    5    8
## [3,]    7    8    9    3   66    9
```

```
rbind(A,B) # row-wise
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5   66
## [3,]    7    8    9
## [4,]    1    4    7
## [5,]    2    5    8
## [6,]    3   66    9
```

List

An array whose element can be of different types

- both primitive and compound types

Construction:

```
l <- list(c(1,2),"a"); l
```

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


List named members

Usually list members are named

```
l <- list( n=c(1,2), char="a") ; l
```

```
## $n
## [1] 1 2
##
## $char
## [1] "a"
```

```
names(l)
```

```
## [1] "n"    "char"
```

List access

Access to a member uses the *accessor* operator `$`, or the element indexing operator `[[`.

```
l$n; l[["n"]] ; l[[1]]
```

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

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

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


List access

Access operators can be used to change and existing element or to add a new one if the name is not present

```
l$char = "B"
l$logicals = c( TRUE, FALSE, TRUE)
l
```

```
## $n
## [1] 1 2
##
## $char
## [1] "B"
##
## $logicals
## [1]  TRUE FALSE  TRUE
```

List slicing

Slicing return a subset of the list:

```
l[2]
```

```
## $char
## [1] "B"
```

Indexing returns the element

```
l[[2]]
```

```
## [1] "B"
```


Exercise 3

Modify the `pythagoras()` function so that it accepts a list with two elements named ‘a’, ‘b’, or ‘c’ and computes the missing one, according to the Pythagorean theorem.

```
pythagoras.list(list(a=3,b=4))
```

```
## $a
## [1] 3
##
## $b
## [1] 4
##
## $c
## [1] 5
```

Factor

Represent nominal variables

- Internally stored as integer vector
- created using the `factor()` function

```
f = factor( c("Red", "Green", "Blue", "Blue",
              "Red", "Red"))
f
```

```
## [1] Red   Green Blue  Blue  Red   Red  
## Levels: Blue Green Red
```


Factor

Levels:

```
levels(f)
```

```
## [1] "Blue"  "Green" "Red"
```

Frequencies:

```
table(f)
```

```
## f
##  Blue Green   Red 
##     2     1     3
```

Ordered factors

```r
f = factor( c("L", "M", "L", "H", "L", "H", "L"),
            levels=c("L","M","H"), ordered=T)
f
```

```
## [1] L M L H L H L
## Levels: L < M < H
```


Dataframe

It is the main data structure used to represent tabular datasets.

- Most data is processed in the form of dataframes
- Most I/O of data handle dataframes
- It is a list of vectors of equal length
- Typical semantic
    - each row is case or observation
    - each column is an attribute or variable

Dataframe

Construction:

```
courses <- data.frame(
  code = c("15AHM","12BHD","16ACF",
           "01PNN", "01RKC","17AXO"),
  course= c("Chemistry","Computer science","Calculus
            "Free Credits","Linear Algebra","Physics
  semester = c(1,1,1,2,2,2),
  credits = c(8,8,10,6,10,10)
)
```


Dataframe example

code	course	semester	credits
15AHM	Chemistry	1	8
12BHD	Computer science	1	8
16ACF	Calculus I	1	10
01PNN	Free Credits	2	6
01RKC	Linear Algebra	2	10
17AXO	Physics I	2	10

Dataframe indexing

Column (attribute/variable) selection is usually performed with the *accessor* operator `$`

- list-specific syntax can be used also

```
courses$credits; courses[[4]]; courses[["credits"]]
```

```
## [1]  8  8 10  6 10 10
```

```
## [1]  8  8 10  6 10 10
```

```
## [1]  8  8 10  6 10 10
```


Dataframe indexing and slicing

Cell indexing is similar to matrixes

```
courses[2,2]
```

```
## [1] "Computer science"
```

Dataframe slicing works like lists

```
courses[c("semester", "credits")]
```

```
##   semester credits
## 1        1       8
## 2        1       8
## 3        1      10
## 4        2       6
## 5        2      10
## 6        2      10
```

Slicing dataframe by row

```
courses[c(1,3,6) , ]
```

```
##    code     course semester credits
## 1 15AHM  Chemistry        1       8
## 3 16ACF Calculus I        1      10
## 6 17AXO  Physics I        2      10
```


Sorting a dataframe

Order and slice

```
ord <- order( - courses$credits) # - means descending
courses[ord,]
```

```
##    code           course semester credits
## 3 16ACF       Calculus I        1      10
## 5 01RKC   Linear Algebra        2      10
## 6 17AXO        Physics I        2      10
## 1 15AHM        Chemistry        1       8
## 2 12BHD Computer science        1       8
## 4 01PNN     Free Credits        2       6
```

Filtering a dataframe with logicals

Use a logical *indicator* vector (TRUE for matching rows)

```
sem.2nd.ind <- courses$semester == 2
sem.2nd.ind ## which courses are in 2nd semester
```

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

```
courses.2nd <- courses[sem.2nd.ind,  ]
courses.2nd ##2nd semester courses
```

```
##    code          course semester credits
## 4 01PNN   Free Credits        2       6
## 5 01RKC Linear Algebra        2      10
## 6 17AXO       Physics I        2      10
```


Filtering and summing

```
sum( courses.2nd$credits ) ## 2nd semester credits
```

```
## [1] 26
```

```
sum( sem.2nd.ind ) ## how many courses in 2nd semeste
```

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

Filtering a dataframe with indexes

Use a the function `which()`

```
sem.2nd.ix <- which( courses$semester == 2 )
sem.2nd.ix ## indexes of courses are in 2nd semester
```

```
## [1] 4 5 6
```

```
courses.2nd <- courses[sem.2nd.ix,  ]
courses.2nd ##2nd semester courses
```

```
##     code          course semester credits
## 4 01PNN    Free Credits        2       6
## 5 01RKC Linear Algebra        2      10
## 6 17AXO       Physics I        2      10
```


Reading files

Functions `read.*`

- Read data from a file into dataframe
- Space separated: `read.table( )`
- CSV: `read.csv( )`
- Clipboard: `read.table(pipe( … ))`
  - X11: `"clipboard"`
  - OS X: `"pbpaste"`
- Excel file: `read.xlsx()`
  - require `library(readxl)`

R Advantages

- R is a common tool among data experts, supported wildly by both professional and academic developers
- R can be installed in any environment on any machine and used with no licensing or agreements needed
- R source code is flexible and can be adapted to specific local needs
- R can build routines straight out of a database for common and universal reporting

R Limitations

- R is based on S, which is close to 40 years old
- R only has features that the community contributes
- Not the ideal solution to all problems
- R is a programming language and not a software package – steeper learning curve
- R can be much slower than compiled languages

Software

- R
  - Download at: https://cran.r-project.org
- R-Studio Desktop
  - Download at: https://rstudio.com/products/rstudio/

References

- R. Irizarry. “Introduction to Data Science - Data Analysis and Prediction Algorithms with R”
  - https://rafalab.github.io/dsbook/
- H.Wickham, G.Grolemund. “R for Data Science - Visualize, model, transform, tidy, and import data”, O’Reilly, 2017
  - https://r4ds.had.co.nz/index.html

}

Solutions

Solution to Exercise 1

Define a function `pythagoras()` accepting three values (a,b,c) one of which can be missing and is computed using the Pythagorean theorem.

```
pythagoras <- function(a=NULL, b=NULL, c=NULL){
  if(is.null(a)){
    sqrt(c^2-b^2)
  }else if(is.null(b)){
    sqrt(c^2-a^2)
  }else if(is.null(c)){
    sqrt(a^2+b^2)
  }
}
```

Solution to Exercise 2

Modify the `pythagoras()` function so that it returns a vector with three elements named ‘a’, ‘b’, and ‘c’ according to the Pythagorean theorem.

```
pythagoras <- function(a=NULL, b=NULL, c=NULL){
  nn <- is.null(a) + is.null(b) + is.null(c)
  if(nn!=1) stop("Exactly one among 'a', 'b', 'c' mus
  if(is.null(a)){
    c(a=sqrt(c^2-b^2), b=b, c=c)
  }else if(is.null(b)){
    c(a=a, b=sqrt(c^2-a^2), c=c)
  }else if(is.null(c)){
    c(a=a, b=b, c=sqrt(a^2+b^2))
  }
}
```


Solution to Exercise 3

Modify the `pythagoras()` function so that it accepts a list with two elements named ‘a’, ‘b’, or ‘c’ and computes the missing one, according to the Pythagorean theorem.

```
pythagoras.list <- function(edges){
  edge_names <- c("a","b","c")
  edges_provided <- edge_names %in% names(edges)
  if(sum(edges_provided)!=2) stop("Wrong argument")
  if(! "a" %in% names(edges)){
    edges$a <- sqrt(edges$c^2-edges$b^2)
  }else if(! "b" %in% names(edges)){
    edges$b <- sqrt(edges$c^2-edges$a^2)
  }else if(! "c" %in% names(edges)){
    edges$c <- sqrt(edges$a^2+edges$b^2)
  }
  edges
```