R 4 Data Science Quick Reference

A Pocket Guide to APIs, Libraries, and Packages

—

Second Edition

—

Thomas Mailund

Apress®

R 4 Data Science Quick Reference

A Pocket Guide to APIs, Libraries, and Packages

Second Edition

Thomas Mailund

Apress®

R 4 Data Science Quick Reference: A Pocket Guide to APIs, Libraries, and Packages

Thomas Mailund
Aarhus, Denmark

ISBN-13 (pbk): 978-1-4842-8779-8 ISBN-13 (electronic): 978-1-4842-8780-4
https://doi.org/10.1007/978-1-4842-8780-4

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: James Markham
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image by natanaelginting on Freepik (www.freepik.com)

Printed on acid-free paper

Table of Contents

About the Author

Thomas Mailund is an associate professor in bioinformatics at Aarhus University, Denmark. His background is in math and computer science, but for the last decade, his main focus has been on genetics and evolutionary studies, particularly comparative genomics, speciation, and gene flow between emerging species.

About the Technical Reviewer

Matt Wiley leads institutional effectiveness, research, and assessment at Victoria College, facilitating strategic and unit planning, data-informed decision making, and state/regional/federal accountability. As a tenured, associate professor of mathematics, he won awards in both mathematics education (California) and student engagement (Texas). Matt earned degrees in computer science, business, and pure mathematics from the University of California and Texas A&M systems.

Outside academia, he coauthors books about the popular R programming language and was managing partner of a statistical consultancy for almost a decade. He has programming experience with R, SQL, C++, Ruby, Fortran, and JavaScript.

A programmer, a published author, a mathematician, and a transformational leader, Matt has always melded his passion for writing with his joy of logical problem solving and data science. From the boardroom to the classroom, he enjoys finding dynamic ways to partner with interdisciplinary and diverse teams to make complex ideas and projects understandable and solvable.

CHAPTER 1

Introduction

R is a functional programming language with a focus on statistical analysis. It has built-in support for model specifications that can be manipulated as first-class objects, and an extensive collection of functions for dealing with probability distributions and model fitting, both built-in and through extension packages.

The language has a long history. It was created in 1992 and is based on an even older language, S, from 1976. Many quirks and inconsistencies have entered the language over the years. There are, for example, at least three partly incompatible ways of implementing object orientation, and one of these is based on a naming convention that clashes with some built-in functions. It can be challenging to navigate through the many quirks of R, but this is alleviated by a suite of extensions, collectively known as the "Tidyverse."

While there are many data science applications that involve more complex data structures, such as graphs and trees, most bread-and-butter analyses involve rectangular data. That is, the analysis is of data that can be structured as a table of rows and columns where, usually, the rows correspond to observations and the columns correspond to explanatory variables and observations. The usual data sets are also of a size that can be loaded into memory and analyzed on a single desktop or laptop. I will assume that both are the case here. If this is not the case, then you need different, big data techniques that go beyond the scope of this book.

The Tidyverse is a collection of extensions to R: packages that are primarily aimed at analyzing tabular data that fits into your computer's memory. Some of the packages go beyond this, but since data science is predominately manipulation of tabular data, this is the focus of this book.

The Tidyverse packages provide consistent naming conventions, consistent programming interfaces, and more importantly a consistent notation that captures how data analysis consists of different steps of data manipulation and model fitting.

© Thomas Mailund 2022
T. Mailund, *R 4 Data Science Quick Reference*, https://doi.org/10.1007/978-1-4842-8780-4_1

The packages do not merely provide collections of functions for manipulating data but rather small domain-specific languages for different aspects of your analysis. Almost all of these small languages are based on the same overall "pipeline" syntax introduced with the `magrittr` package. The package introduces a syntax for sending the output of one function call into another function call. It provides various operators for this, but the most frequently used is `%>%` that gives you an alternative syntax for writing function calls:

```
x%>%f()%>%g()
```

is equivalent to

```
g(f(x))
```

This syntax was deemed so useful that a similar operator was introduced into the main R language in version 4.1, `|>`, so you will now also see syntax such as

```
x |> f() |> g()
```

I will, when it is convenient, use the built-in pipe operator `|>` in code examples, but I will leave `magrittr` syntax until after Chapter 6 where I describe the `magrittr` package.

The two operators are not identical, and I will highlight a few differences in Chapter 6. They differ in a few ways, but you can use either with the Tidyverse packages, and they are usually designed with the assumption that you use the packages that way.

A noticeable exception is the plotting library `ggplot2`. It is slightly older than the other extensions in the Tidyverse and because of this has a different syntax. The main difference is the operator used for combining different operations. The data pipeline notation uses the `%>%` or `|>` operator, while `ggplot2` combines plotting instructions using `+`. If you are like me, then you will often try to combine `ggplot2` instructions using `%>%`—just out of habit—but once you get an error from R, you will recognize your mistake and can quickly fix it.

This book is a syntax reference for modern data science in R, which means that it is a guide for using Tidyverse packages and it is a guide for programmers who want to use R's Tidyverse packages instead of basic R programming. I will assume that you are familiar with base R and the functions there, or at least that you can read the documentation for these, for example, when you want to know how the function `read.table()` behaves, you type `?read.table` into your R console. I will mention base R functions in the text when they differ from similar functionality in the Tidyverse functions. This will warn you if you are familiar with the base R functions and might have code that uses them that you want to port to Tidyverse code. If you are not familiar with base R and want a reference for the core R language, you might wish to read the book *R Quick Syntax Reference* by Margot Tollefson instead.

This guide does not explain each Tidyverse package exhaustively. The development of Tidyverse packages progresses rapidly, and the book would not contain a complete guide shortly after it is printed anyway. The structure of the extensions and the domain-specific languages they provide are stable, however, and from examples with a subset of the functionality in them, you should not have any difficulties with reading the package documentation for each of them and find the features you need that are not covered in the book.

To get started with the Tidyverse, install and load it:

```
install.packages("tidyverse")
library(tidyverse)
```

The Tidyverse consists of many packages that you can install and load independently, but loading all through the `tidyverse` package is the easiest, so unless you have good reasons to, for example, that you need a package that isn't automatically loaded, just load `tidyverse` when you start an analysis. In this book, I describe three packages that are not loaded from `tidyverse` but are generally considered part of the Tidyverse.[1]

[1] The Tidyverse I refer to here is the ecosystem of Tidyverse packages but not the *package* `tidyverse`, which only loads the key packages.

CHAPTER 2

Importing Data: `readr`

Before we can analyze data, we need to load it into R. The main Tidyverse package for this is called `readr`, and it is loaded when you load the `tidyverse` package:

```
library(tidyverse)
```

but you can also load it explicitly using

```
library(readr)
```

Tabular data is usually stored in text files or compressed text files with rows and columns matching the table's structure. Each line in the file is a row in the table, and columns are separated by a known delimiter character. The `readr` package is made for such data representation and contains functions for reading and writing variations of files formatted in this way. It also provides functionality for determining the types of data in each column, either by inferring types or through user specifications.

Functions for Reading Data

The `readr` package provides the following functions for reading tabular data:

Function	File format
`read_csv()`	Comma-separated values
`read_csv2()`	Semicolon-separated values
`read_tsv()`	Tab-separated values

T. Mailund, *R 4 Data Science Quick Reference*, https://doi.org/10.1007/978-1-4842-8780-4_2

Function	File format
`read_delim()`	General column delimiters[1]
`read_table()`	Space-separated values (fixed-length columns)
`read_table2()`	Space-separated values (variable-length columns)

The interface to these functions differs little. In the following text, I describe `read_csv`, but I highlight when the other functions differ. The `read_csv` function reads data from a file with comma-separated values. Such a file could look like this:

```
A,B,C,D
1,a,a,1.2
2,b,b,2.1
3,c,c,13.0
```

Unlike the base R `read.csv` function, `read_csv` will also handle files with spaces between the columns, so it will interpret the following data the same as the preceding file:

```
A,  B,  C,    D
1,  a,  a,   1.2
2,  b,  b,   2.1
3,  c,  c,  13.0
```

If you use R's `read.csv` function instead, the spaces before columns B and C will be included as part of the data and the text columns will be interpreted as factors.

The first line in the file will be interpreted as a header, naming the columns, and the remaining three lines as data rows.

Assuming the file is named `data/data.csv`, you read its data like this:

```
my_data<-read_csv(file ="data/data.csv")
```

[1] The `read_delim()` can handle any file format that has a special character that delimits columns. The `read_csv()`, `read_csv2()`, and `read_tsv()` functions are specializations of it. The first of these uses commas for the delimiter, the second semicolons, and the third tabs.

```
## Rows: 3 Columns: 4
##---- Column specification -------------------------
## Delimiter: ","
## chr (2): B, C
## dbl (2): A, D
##
## ℹ Use 'spec()' to retrieve the full column specification for this data.
## ℹ Specify the column types or set 'show_col_types = FALSE' to quiet this
message.
```

The message you get from `read_csv()` tells you that you can get information about the type it has inferred for each column if you use the `spec()` function:

```
spec(my_data)
```

```
## cols(
##   A = col_double(),
##   B = col_character(),
##   C = col_character(),
##   D = col_double()
## )
```

When reading the file, `read_csv` will infer that columns A and D are numbers and columns B and C are strings.

If you are happy with that, and don't want to be told about it in the future, you can use the option

```
show_col_types = FALSE:
```

```
my_data<-read_csv(file ="data/data.csv",
                  show_col_types =FALSE)
```

If the file contains tab-separated values

```
A   B   C   D
1   a   a   1.2
2   b   b   2.1
3   c   c   13.0
```

you should use `read_tsv()` instead.

```
my_data<-read_tsv(file ="data/data.tsv",
                  show_col_types =FALSE)
```

The file you read with `read_csv` can be compressed. If the suffix of the file name is .gz, `.bz2`, `.xz`, or `.zip`, it will be uncompressed before `read_csv` loads the data.

```
my_data<-read_csv(file ="data/data.csv.gz",
                  show_col_types =FALSE)
```

If the file name is a URL (i.e., has prefix `http://`, `https://`, `ftp://`, or `ftps://`, the file will automatically be downloaded.

You can also provide a string as the file object:

```
read_csv(
    "A, B, C,    D
     1, a, a,  1.2
     2, b, b,  2.1
     3, c, c, 13.0
", show_col_types = FALSE)
```

```
## # A tibble: 3 × 4
##       A B     C         D
##   <dbl> <chr> <chr> <dbl>
## 1     1 a     a       1.2
## 2     2 b     b       2.1
## 3     3 c     c      13
```

This is rarely useful in a data analysis project, but you can use it to create examples or for debugging.

File Headers

The first line in a comma-separated file is not always the column names; that information might be available from elsewhere outside the file. If you do not want to interpret the first line as column names, you can use the option `col_names = FALSE`.

```
read_csv(
    file ="data/data.csv",
    col_names =FALSE,
    show_col_types =FALSE
)
```

```
## # A tibble: 4 × 4
##   X1    X2    X3    X4
##   <chr> <chr> <chr> <chr>
## 1 A     B     C     D
## 2 1     a     a     1.2
## 3 2     b     b     2.1
## 4 3     c     c     13.0
```

Since the `data/data.csv` file *has* a header, that is interpreted as part of the data, and because the header consists of strings, `read_csv` infers that all the column types are strings. If we did not have the header, for example, if we had the file `data/data-no-header.csv`:

```
1, a, a, 1.2
2, b, b, 2.1
3, c, c, 13.0
```

then we would get the same data frame as before, except that the names would be autogenerated:

```
read_csv(
     file ="data/data-no-header.csv",
     col_names =FALSE,
     show_col_types =FALSE
)
```

```
## # A tibble: 3 × 4
##      X1 X2    X3       X4
##   <dbl> <chr> <chr> <dbl>
## 1     1 a     a       1.2
## 2     2 b     b       2.1
## 3     3 c     c      13
```

The autogenerated column names all start with X and are followed by the number the columns have from left to right in the file.

If you have data in a file without a header, but you do not want the autogenerated names, you can provide column names to the col_names option:

```
read_csv(
    file ="data/data-no-header.csv",
    col_names =c("X","Y","Z","W"),
    show_col_types =FALSE
)
```

```
## # A tibble: 3 × 4
##       X Y     Z         W
##   <dbl> <chr> <chr> <dbl>
## 1     1 a     a       1.2
## 2     2 b     b       2.1
## 3     3 c     c        13
```

If there is a header line, but you want to rename the columns, you cannot just provide the names to read_csv using col_names. The first row will still be interpreted as data. This gives you data you do not want in the first row, and it also affects the inferred types of the columns.

You can, however, skip lines before read_csv parse rows as data. Since we have a header line in data/data.csv, we can skip one line and set the column names.

```
read_csv(
    file ="data/data.csv",
    col_names =c("X","Y","Z","W"),
    skip =1,
    show_col_types =FALSE
)
```

```
## # A tibble: 3 × 4
##       X Y     Z         W
##   <dbl> <chr> <chr> <dbl>
## 1     1 a     a       1.2
## 2     2 b     b       2.1
## 3     3 c     c      13
```

You can also put a limit on how many data rows you want to load using the `n_max` option.

```
read_csv(
    file ="data/data.csv",
    col_names =c("X","Y","Z","W"),
    skip =1,
    n_max =2,
    show_col_types =FALSE
)
```

If your input file has comment lines, identifiable by a character where the rest of the line should be considered a comment, you can skip them if you provide the `comment` option:

```
read_csv(
    "A, B, C,      D   # this is a comment
     1, a, a,   1.2     # another comment
     2, b, b,   2.1
     3, c, c, 13.0",
     comment ="#",
     show_col_types =FALSE)
```

```
## # A tibble: 3 × 4
##       A B     C         D
##   <dbl> <chr> <chr> <dbl>
## 1     1 a     a       1.2
## 2     2 b     b       2.1
## 3     3 c     c      13
```

You can leave a whole line as a comment, but then you want the comment character to start to the left of that line:

```
read_csv(
    "A, B, C,   D   # this is a comment
# whole line comment
     1, a, a,   1.2   # another comment
     2, b, b,   2.1
     3, c, c, 13.0",
     comment ="#",
     show_col_types =FALSE)
```

```
## # A tibble: 3 × 4
##       A B     C         D
##   <dbl> <chr> <chr> <dbl>
## 1     1 a     a       1.2
## 2     2 b     b       2.1
## 3     3 c     c      13
```

If you have space before the comment, the function can't tell if there is an error—it looks like a line with missing columns rather than a blank line—so you will get a warning and a row with NA where the comment line was.

```
read_csv(
    "A, B, C,   D   # this is a comment
    # the indentation is a potential problem; missing columns?
    1, a, a,   1.2   # another comment
    2, b, b,   2.1
    3, c, c, 13.0",
    comment ="#",
    show_col_types =FALSE
)
```

```
## Warning: One or more parsing issues, see
## 'problems()' for details
```

```
## # A tibble: 4 × 4
##       A B     C         D
##   <dbl> <chr> <chr> <dbl>
## 1    NA <NA>  <NA>     NA
## 2     1 a     a       1.2
## 3     2 b     b       2.1
## 4     3 c     c      13
```

For more options affecting how input files are interpreted, read the function documentation: ?read_csv.

Column Types

When `read_csv` parses a file, it infers the type of each column. This inference can be slow, or worse the inference can be incorrect. If you know a priori what the types should be, you can specify this using the `col_types` option. If you do this, then `read_csv` will not make a guess at the types. It will, however, replace values that it cannot parse as of the right type into NA.[2]

String-Based Column Type Specification

In the simplest string specification format, you must provide a string with the same length as you have columns and where each character in the string specifies the type of one column. The characters specifying different types are this:

Character	Type
`c`	Character
`i`	Integer
`n`	Number
`d`	Double
`l`	Logical
`f`	Factor
`D`	Date
`T`	Datetime
`t`	Time
`?`	Guess (default)
`_/-`	Skip the column

[2] There is a gotcha here. The types are guessed at after a fixed number of lines are read (by default 1000). If you have 1000 lines of numbers in a column and line 1001 has a string, then the type will be inferred as numeric and you lose the string. If you know the types, it is always better to tell the functions what they are.

By default, `read_csv` guesses, so we could make this explicit using the type specification `"????"`:

```
read_csv(
    file ="data/data.csv",
    col_types ="????"
)
```

```
## # A tibble: 3 × 4
##       A B     C         D
##   <dbl> <chr> <chr> <dbl>
## 1     1 a     a       1.2
## 2     2 b     b       2.1
## 3     3 c     c      13
```

The results of the guesses are double for columns `A` and `D` and character for columns `B` and `C`. If we wanted to make *this* explicit, we could use `"dccd"`.

```
read_csv(
    file ="data/data.csv",
    col_types ="dccd"
)
```

```
## # A tibble: 3 × 4
##       A B     C         D
##   <dbl> <chr> <chr> <dbl>
## 1     1 a     a       1.2
## 2     2 b     b       2.1
## 3     3 c     c      13
```

If you want an integer type for column `A`, you can use `"iccd"`:

```
read_csv(
    file ="data/data.csv",
    col_types ="iccd"
)
```

```
## # A tibble: 3 × 4
##       A B     C         D
##   <int> <chr> <chr> <dbl>
## 1     1 a     a       1.2
## 2     2 b     b       2.1
## 3     3 c     c      13
```

If you try to interpret column D as integers as well, you will get a list of warning messages, and the values in column D will all be NA; the numbers in column D cannot be interpreted as integers, and read_csv will not round them to integers.

```
read_csv(
    file ="data/data.csv",
    col_types ="icci"
)
```

```
## Warning: One or more parsing issues, see
## 'problems()' for details
```

```
## # A tibble: 3 × 4
##       A B     C         D
##   <int> <chr> <chr> <int>
## 1     1 a     a        NA
## 2     2 b     b        NA
## 3     3 c     c        NA
```

If you specify that a column should have type d, the numbers in the column must be integers or decimal numbers. If you use the type n (the default that read_csv will guess), you will also get doubles, but the latter type can handle strings that can be interpreted as numbers such as dollar amounts, percentages, and group separators in numbers. The column type n will ignore leading and trailing text and handle number separators:

With this function call

```
read_csv(
    'A, B, C,   D,   E
    $1,a,a,1.2%,"1,100,200"
    $2,b,b,2.1%,"140,000"
    $3,c,c,13.0%,"2,005,000"',
    col_types ="nccnn"
    )
```