Pedro J. Aphalo

Learn R

As a Language

Contents

Preface

"Suppose that you want to teach the 'cat' concept to a very young child. Do you explain that a cat is a relatively small, primarily carnivorous mammal with retractible claws, a distinctive sonic output, etc.? I'll bet not. You probably show the kid a lot of different cats, saying 'kitty' each time, until it gets the idea. To put it more generally, generalizations are best made by abstraction from experience."

R. P. Boas
Can we make mathematics intelligible?, 1981

This book covers different aspects of the use of the R language. Chapters **??** to **??** describe the R language itself. Later chapters describe extensions to the R language available through contributed *packages*, the *grammar of data* and the *grammar of graphics*. In this book, explanations are concise but contain pointers to additional sources of information, so as to encourage the development of a routine of independent exploration. This is not an arbitrary decision, this is the normal *modus operandi* of most of us who use R regularly for a variety of different problems. Some have called approaches like the one used here "learning the hard way," but I would call it "learning to be independent."

I do not discuss statistics or data analysis methods in this book; I describe R as a language for data manipulation and display. The idea is for you to learn the R language in a way comparable to how children learn a language: they work out what the rules are, simply by listening to people speak and trying to utter what they want to tell their parents. Of course, small children receive some guidance, but they are not taught a prescriptive set of rules like when learning a second language at school. Instead of listening, you will read code, and instead of speaking, you will try to execute R code statements on a computer—i.e., you will try your hand at using R to tell a computer what you want it to compute. I do provide explanations and guidance, but the idea of this book is for you to use the numerous examples to find out by yourself the overall patterns and coding philosophy behind the R language. Instead of parents being the sound board for your first utterances in R, the computer will play this role. You will *play* by modifying the examples, see how the computer responds: does R understand you or not? Using a language actively is the most efficient way of learning it. By using it, I mean actually reading, writing, and running scripts or programs (copying and pasting, or typing ready-made examples from books or the internet, does not qualify as using a language).

I have been using R since around 1998 or 1999, but I am still constantly learning new things about R itself and R packages. With time, it has replaced in my work as a researcher and teacher several other pieces of software: SPSS, Systat, Origin, MS-Excel, and it has become a central piece of the tool set I use for producing lecture slides, notes, books, and even web pages. This is to say that it is the most useful piece of software and programming language I have ever learned to use. Of course, in time it will be replaced by something better, but at the moment it is a key language to learn for anybody with a need to analyze and display data.

What is a language? A language is a system of communication. R as a language allows us to communicate with other members of the R community, and with computers. As with all languages in active use, R evolves. New "words" and new "constructs" are incorporated into the language, and some earlier frequently used ones are relegated to the fringes of the corpus. I describe current usage and "modisms" of the R language in a way accessible to a readership unfamiliar with computer science but with some background in data analysis as used in biology, engineering, or the humanities.

When teaching, I tend to lean toward challenging students, rather than telling an over-simplified story. There are two reasons for this. First, I prefer as a student, and I learn best myself, if the going is not too easy. Second, if I would hide the tricky bits of the R language, it would make the reader's life much more difficult later on. You will not remember all the details; nobody could. However, you most likely will remember or develop a sense of when you need to be careful or should check the details. So, I will expose you not only to the usual cases, but also to several exceptions and counterintuitive features of the language, which I have highlighted with icons. Reading this book will be about exploring a new world; this book aims to be a travel guide, but neither a traveler's account, nor a cookbook of R recipes.

Keep in mind that it is impossible to remember everything about R! The R language, in a broad sense, is vast because its capabilities can be expanded with independently developed packages. Learning to use R consists of learning the basics plus developing the skill of finding your way in R and its documentation. In early 2020, the number of packages available in the Comprehensive R Archive Network (CRAN) broke the 15,000 barrier. CRAN is the most important, but not only, public repository for R packages. How good a command of the R language and packages a user needs depends on the type of activities to be carried out. This book attempts to train you in the use of the R language itself, and of popular R language extensions for data manipulation and graphical display. Given the availability of numerous books on statistical analysis with R, in the present book I will cover only the bare minimum of this subject. The same is true for package development in R. This book is somewhere in-between, aiming at teaching programming in the small: the use of R to automate the drudgery of data manipulation, including the different steps spanning from data input and exploration to the production of publication-quality illustrations.

As with all "rich" languages, there are many different ways of doing things in R. In almost all cases there is no one-size-fits-all solution to a problem. There is always a compromise involved, usually between time spent by the user and processing time required in the computer. Many of the packages that are most popular nowadays did not exist when I started using R, and many of these packages make new approaches available. One could write many different R books with a given aim using substantially different ways of achieving the same results. In this book, I limit myself to packages that are currently popular and/or that I consider elegantly designed. I have in particular tried to limit myself to packages with similar design philosophies, especially in relation to their interfaces. What is elegant design, and in particular what is a friendly user interface, depends strongly on each user's preferences and previous experience. Consequently, the contents of the book are strongly biased by my own preferences. I have tried to write examples in ways that execute fast without compromising readability. I encourage readers to take this book as a starting point for exploring the very many packages, styles, and approaches which I have not described.

I will appreciate suggestions for further examples, and notification of errors and unclear sections. Because the examples here have been collected from diverse sources over many years, not all sources are acknowledged. If you recognize any example as yours or someone else's, please let me know so that I can add a proper acknowledgement. I warmly thank the students who have asked the questions and posed the problems that have helped me write this text and correct the mistakes and voids of previous versions. I have also received help on online forums and in person from numerous people, learned from archived e-mail list messages, blog posts, books, articles, tutorials, webinars, and by struggling to solve some new problems on my own. In many ways this text owes much more to people who are not authors than to myself. However, as I am the one who has written this version and decided what to include and exclude, as author, I take full responsibility for any errors and inaccuracies.

Why have I chosen the title "*Learn R: As a Language*"? This book is based on exploration and practice that aims at teaching to express various generic operations on data using the R language. It focuses on the language, rather than on specific types of data analysis, and exposes the reader to current usage and does not spare the quirks of the language. When we use our native language in everyday life, we do not think about grammar rules or sentence structure, except for the trickier or unfamiliar situations. My aim is for this book to help you grow to use R in this same way, to become fluent in R. The book is structured around the elements of languages with chapter titles that highlight the parallels between natural languages like English and the R language.

I encourage you to approach R like a child approaches his or her mother tongue when first learning to speak: do not struggle, just play, and fool around with R! If the going gets difficult and frustrating, take a break! If you get a new insight, take a break to enjoy the victory!

Acknowledgements

First I thank Jaakko Heinonen for introducing me to the then new R. Along the way many well known and not so famous experts have answered my questions in usenet and more recently in Stackoverflow. As time went by, answering other people's questions, both in the internet and in person, became the driving force for me to delve into the depths of the R language. Of course, I still get stuck from time to time and ask for help. I wish to warmly thank all the people I have interacted with in relation to R, including members of my own research group, students participating in the courses I have taught, colleagues I have collaborated with, authors of the books I have read and people I have only met online or at conferences. All of them have made it possible for me to write this book. This has been a time consuming endeavour which has kept me too many hours away from my family, so I specially thank Tarja, Rosa and Tomás for their understanding. I am indebted to Tarja Lehto, Titta Kotilainen, Tautvydas Zalnierius, Fang Wang, Yan Yan, Neha Rai, Markus Laurel, other colleagues, students and anonymous reviewers for many very helpful comments on different versions of the book manuscript, Rob Calver, as editor, for his encouragement and patience during the whole duration of this book writing project, Lara Spieker, Vaishali Singh, and Paul Boyd for their help with different aspects of this project.

Icons used to mark different content

Text boxes are used throughout the book to highlight content that plays specific roles in the learning process or that require special attention from the reader. Each box contains one of five different icons that indicate the type of its contents as described below.

Signals *playground* boxes which contain open-ended exercises—ideas and pieces of R code to play with at the R console.

Signals *advanced playground* boxes which will require more time to play with before grasping concepts than regular *playground* boxes.

Signals important bits of information that must be remembered when using R—i.e., explain some unusual feature of the language.

Signals in-depth explanations of specific points that may require you to spend time thinking, which in general can be skipped on first reading, but to which you should return at a later peaceful time, preferably with a cup of coffee or tea.

Signals text boxes providing general information not directly related to the R language.

1

New grammars of data

Essentially everything in S[R], for instance, a call to a function, is an S[R] object. One viewpoint is that S[R] has self-knowledge. This self-awareness makes a lot of things possible in S[R] that are not in other languages.

Patrick J. Burns
S Poetry, 1998

1.1 Aims of this chapter

Base R and the recommended extension packages (installed by default) include many functions for manipulating data. The R distribution supplies a complete set of functions and operators that allow all the usual data manipulation operations. These functions have stable and well-described behavior, so they should be preferred unless some of their limitations justify the use of alternatives defined in contributed packages. In the present chapter we aim at describing the new syntaxes introduced by the most popular of these contributed R extension packages aiming at changing (usually improving one aspect at the expense of another) in various ways how we can manipulate data in R. These independently developed packages extend the R language not only by adding new "words" to it but by supporting new ways of meaningfully connecting "words"—i.e., providing new "grammars" for data manipulation.

1.2 Introduction

By reading previous chapters, you have already become familiar with base R classes, methods, functions and operators for storing and manipulating data. Most of these had been originally designed to perform optimally on rather small data sets (see Matloff 2011). The R implementation has been improved over the years significantly in performance, and random-access memory in computers has become cheaper, making constraints imposed by the original design of R less limiting, but on the other hand, the size of data sets has also increased. Some contributed packages have aimed at improving performance by relying on different compromises between usability, speed and reliability than used for base R.

Package ‘data.table’ is the best example of an alternative implementation of data storage that maximizes the speed of processing for large data sets using a new semantics and requiring a new syntax. We could say that package ‘data.table’ is based on a “grammar of data” that is different from that in the R language. The compromise in this case has been the use of a less intuitive syntax, and by defaulting to call by reference of arguments instead of by copy, increasing the “responsibility” of the author of code defining new functions.

When a computation includes a chain of sequential operations, if using base R, we can either store at each step in the computation the returned value in a variable, or nest multiple function calls. The first approach is verbose, but allows readable scripts, especially if variable names are wisely chosen. The second approach becomes very difficult too read as soon as there is more than one nesting level. Attempts to find an alternative syntax have borrowed the concept of data *pipes* from Unix shells (Kernigham and Plauger 1981). Interestingly, that it has been possible to write packages that define the operators needed to “add” this new syntax to R is a testimony to its flexibility and extensibility. Two packages, ‘magrittr’ and ‘wrapr’, define operators for pipe-based syntax.

A different aspect of the R syntax is extraction of members from lists and data frames by name. Base R provides two different operators for this, `$` and `[[]]`, with different syntax. These two operators also differ in how *incomplete names* are handled. Package ‘tibble’ alters this syntax for an alternative to base R’s data frames. Once again, a new syntax allows new functionality at the expense of partial incompatibility with base R syntax. Objects of class `"tb"` were also an attempt to improve performance compared to objects of class `"data.frame"`. R performance has improved in recent releases and currently, even though performance is not the same, depending on the operations and data, either R’s data frames or tibbles perform better.

Base R function `subset()` has an unusual syntax, as it evaluates the expression passed as the second argument within the namespace of the data frame passed as its first argument (see **??** on page **??**). This saves typing at the expense of increasing the risk of bugs, as by reading the call to subset, it is not obvious which names are resolved in the environment of `subset()` and which ones within its first argument—i.e., as column names in the data frame. In addition, changes elsewhere in a script can change how a call to subset is interpreted. In reality, subset is a wrapper function built on top of the extraction operator `[]`. It is a convenience function, mostly intended to be used at the console, rather than in scripts or package code. To extract rows from a data frame it is always best to use the `[ , ]` operator.

Package ‘dplyr’ provides convenience functions that work in a similar way as base R `subset()`, although in latest versions more safely. This package has suffered quite drastic changes during its development with respect to how to handle the dilemma caused by “guessing” of the environment where names should be looked up. There is no easy answer; a simplified syntax leads to ambiguity, and a fully specified syntax is verbose. Recent versions of the package introduced a terse syntax to achieve a concise way of specifying where to look up names. My opinion is that for code that needs to be highly reliable and produce reproducible results in the future, we should for the time being prefer base R. For code that is to be used once, or for which reproducibility can depend on the use of a specific (old or soon to be old) version of 'dplyr', or which is not a burden to update, the conciseness and power of the new syntax will be an advantage.

In this chapter you will become familiar with alternative "grammars of data" as implemented in some of the packages that enable new approaches to manipulating data in R. As in previous chapters I will focus more on the available tools and how to use them than on their role in the analysis of data. The books *R for Data Science* (Wickham and Grolemund 2017) and *R Programming for Data Science* (Peng 2016) partly cover the same subjects from the perspective of data analysis.

1.3 Packages used in this chapter

```
install.packages(learnrbook::pkgs_ch_data)
```

To run the examples included in this chapter, you need first to load some packages from the library (see section **??** on page **??** for details on the use of packages).

```
library(learnrbook)
library(tibble)
library(magrittr)
library(wrapr)
library(stringr)
library(dplyr)
library(tidyr)
library(lubridate)
```

1.4 Replacements for `data.frame`

1.4.1 'data.table'

The function call semantics of the R language is that arguments are passed to functions by copy. If the arguments are modified within the code of a function, these changes are local to the function. If implemented naively, this semantic would impose a huge toll on performance, however, R in most situations only makes a copy if and when the value changes. Consequently, for modern versions of R which are very good at avoiding unnecessary copying of objects, the normal R semantics has only a moderate negative impact on performance. However, this impact can still be a problem as modification is detected at the object level, and consequently R may make copies of a whole data frame when only values in a single column or even just an attribute have changed.

Functions and methods from package 'data.table' use arguments by reference, avoiding making any copies. However, any assignments within these functions and methods modify the variable passed as an argument. This simplifies the needed tests for delayed copying and also by avoiding the need to make a copy of arguments, achieves the best possible performance. This is a specialized package but extremely useful when dealing with very large data sets. Writing user code, such as scripts, with 'data.table' requires a good understanding of the pass-by-reference semantics. Obviously, package 'data.table' makes no attempt at backwards compatibility with base-R `data.frame`.

1.4.2 'tibble'

The authors of package 'tibble' describe their `tbl` class as backwards compatible with `data.frame` and make it a derived class. This backwards compatibility is only partial so in some situations data frames and tibbles are not equivalent.

The class and methods that package 'tibble' defines lift some of the restrictions imposed by the design of base R data frames at the cost of creating some incompatibilities due to changed (improved) syntax for member extraction and by adding support for "columns" of class `list` and removing support for columns of class `matrix`. Handling of attributes is also different, with no row names added by default. There are also differences in default behavior of both constructors and methods. Although, objects of class `tbl` can be passed as arguments to functions that expect data frames as input, these functions are not guaranteed to work correctly as a result of the differences in syntax.

⚠ It is easy to write code that will work correctly both with data frames and tibbles. However, code that is syntactically correct according to the R language may fail if a tibble is used in place of a data frame.

The `print()` method for tibbles differs from that for data frames in that it outputs a header with the text "A tibble:" followed by the dimensions (number of rows × number of columns), adds under each column name an abbreviation of its class and instead of printing all rows and columns, a limited number of them are displayed. In addition, individual values are formatted differently even adding color highlighting for negative numbers.

```
tibble(A = LETTERS[1:5], B = -2:2, C = seq(from = 1, to = 0, length.out = 5))
## # A tibble: 5 x 3
##   A         B     C
##   <chr> <int> <dbl>
## 1 A        -2  1
## 2 B        -1  0.75
## 3 C         0  0.5
## 4 D         1  0.25
## 5 E         2  0
```

The default number of rows printed can be set with options, that we set here to only three rows for most of this chapter.

```
options(tibble.print_max = 3, tibble.print_min = 3)
```

In their first incarnation, the name for `tibble` was `data_frame` (with a dash instead of a dot). The old name is still recognized, but its use should be avoided and `tibble()` used instead. One should be aware that although the constructor `tibble()` and conversion function `as_tibble()`, as well as the test `is_tibble()` use the name `tibble`, the class attribute is named `tbl`.

```
my.tb <- tibble(numbers = 1:3)
is_tibble(my.tb)
## [1] TRUE

inherits(my.tb, "tibble")
## [1] FALSE

class(my.tb)
## [1] "tbl_df"     "tbl"        "data.frame"
```

Furthermore, by necessity, to support tibbles based on different underlying data sources, a further derived class is needed. In our example, as our tibble has an underlying `data.frame` class, the most derived class of `my.tb` is `tbl_df`.

We start with the constructor and conversion methods. For this we will define our own diagnosis function (*apply* functions are described in section **??** on page **??**).

```
show_classes <- function(x) {
  cat(
    paste(paste(class(x)[1],
    "containing:"),
    paste(names(x),
          sapply(x, class), collapse = ", ", sep = ": "),
    sep = "\n")
    )
}
```

In the next two chunks we can see some of the differences. The `tibble()` constructor does not by default convert character data into factors, while the `data.frame()` constructor does.

```
my.df <- data.frame(codes = c("A", "B", "C"), numbers = 1:3, integers = 1L:3L)
is.data.frame(my.df)
## [1] TRUE

is_tibble(my.df)
## [1] FALSE
```

```
show_classes(my.df)
## data.frame containing:
## codes: character, numbers: integer, integers: integer
```

Tibbles are data frames—or more formally class `tibble` is derived from class `data.frame`. However, data frames are not tibbles.

```
my.tb <- tibble(codes = c("A", "B", "C"), numbers = 1:3, integers = 1L:3L)
is.data.frame(my.tb)
## [1] TRUE

is_tibble(my.tb)
## [1] TRUE

show_classes(my.tb)
## tbl_df containing:
## codes: character, numbers: integer, integers: integer
```

The `print()` method for tibbles, overrides the one defined for data frames.

```
print(my.df)
##   codes numbers integers
## 1     A       1        1
## 2     B       2        2
## 3     C       3        3

print(my.tb)
## # A tibble: 3 x 3
##   codes numbers integers
##   <chr>   <int>    <int>
## 1 A           1        1
## 2 B           2        2
## 3 C           3        3
```

Tibbles and data frames differ in how they are printed when they have many rows or columns. 1) Construct a data frame and an equivalent tibble with at least 50 rows and then test how the output looks when they are printed. 2) Construct a data frame and an equivalent tibble with more columns than will fit in the width of the Rconsole and then test how the output looks when they are printed.

Data frames can be converted into tibbles with `as_tibble()`.

```
my_conv.tb <- as_tibble(my.df)
is.data.frame(my_conv.tb)
## [1] TRUE

is_tibble(my_conv.tb)
## [1] TRUE

show_classes(my_conv.tb)
## tbl_df containing:
## codes: character, numbers: integer, integers: integer
```

```
my_conv.df <- as.data.frame(my.tb)
is.data.frame(my_conv.df)
## [1] TRUE

is_tibble(my_conv.df)
## [1] FALSE

show_classes(my_conv.df)
## data.frame containing:
## codes: character, numbers: integer, integers: integer
```

Look carefully at the result of the conversions. Why do we now have a data frame with `A` as `character` and a tibble with `A` as a `factor`?

Not all conversion functions work consistently when converting from a derived class into its parent. The reason for this is disagreement between authors on what the *correct* behavior is based on logic and theory. You are not likely to be hit by this problem frequently, but it can be difficult to diagnose.

We have already seen that calling `as.data.frame()` on a tibble strips the derived class attributes, returning a data frame. We will look at the whole character vector stored in the `"class"` attribute to demonstrate the difference. We also test the two objects for equality, in two different ways. Using the operator `==` tests for equivalent objects. Objects that contain the same data. Using `identical()` tests that objects are exactly the same, including attributes such as `"class"`, which we retrieve using `class()`.

```
class(my.tb)
## [1] "tbl_df"     "tbl"        "data.frame"

class(my_conv.df)
## [1] "data.frame"

my.tb == my_conv.df
##      codes numbers integers
## [1,]  TRUE    TRUE     TRUE
## [2,]  TRUE    TRUE     TRUE
## [3,]  TRUE    TRUE     TRUE

identical(my.tb, my_conv.df)
## [1] FALSE
```

Now we derive from a tibble, and then attempt a conversion back into a tibble.

```
my.xtb <- my.tb
class(my.xtb) <- c("xtb", class(my.xtb))
class(my.xtb)
## [1] "xtb"        "tbl_df"     "tbl"        "data.frame"

my_conv_x.tb <- as_tibble(my.xtb)
class(my_conv_x.tb)
## [1] "tbl_df"     "tbl"        "data.frame"

my.xtb == my_conv_x.tb
##      codes numbers integers
## [1,]  TRUE    TRUE     TRUE
## [2,]  TRUE    TRUE     TRUE
## [3,]  TRUE    TRUE     TRUE

identical(my.xtb, my_conv_x.tb)
## [1] FALSE
```

The two viewpoints on conversion functions are as follows. 1) The conversion function should return an object of its corresponding class, even if the argument is an object of a derived class, stripping the derived class. 2) If the object is of the class to be converted to, including objects of derived classes, then it should remain untouched. Base R follows, as far as I have been able to work out, approach 1). Packages in the 'tidyverse' follow approach 2). If in doubt about the behavior of some function, then you will need to do a test similar to the one used in this box.

There are additional important differences between the constructors `tibble()` and `data.frame()`. One of them is that in a call to `tibble()`, member variables ("columns") being defined can be used in the definition of subsequent member variables.

```
tibble(a = 1:5, b = 5:1, c = a + b, d = letters[a + 1])
## # A tibble: 5 x 4
##       a     b     c d
##   <int> <int> <int> <chr>
## 1     1     5     6 b
## 2     2     4     6 c
## 3     3     3     6 d
## # ... with 2 more rows
```

What is the behavior if you replace `tibble()` by `data.frame()` in the statement above?

While data frame columns can be factors, vectors or matrices (with the same number of rows as the data frame), columns of tibbles can be factors, vectors or lists (with the same number of members as rows the tibble has).

```
tibble(a = 1:5, b = 5:1, c = list("a", 2, 3, 4, 5))
## # A tibble: 5 x 3
##       a     b c
##   <int> <int> <list>
## 1     1     5 <chr [1]>
## 2     2     4 <dbl [1]>
## 3     3     3 <dbl [1]>
## # ... with 2 more rows
```

Which even allows a list of lists as a variable, or a list of vectors.

```
tibble(a = 1:5, b = 5:1, c = list("a", 1:2, 0:3, letters[1:3], letters[3:1]))
## # A tibble: 5 x 3
##       a     b c
##   <int> <int> <list>
## 1     1     5 <chr [1]>
## 2     2     4 <int [2]>
## 3     3     3 <int [4]>
## # ... with 2 more rows
```

1.5 Data pipes

The first obvious difference between scripts using some of the new grammars is the frequent use of *pipes*. This is, however, mostly a question of preferences, as pipes can be used equally well with base R functions. Pipes have been at the core of shell scripting in Unix since early stages of its design (Kernigham and Plauger 1981). Within an OS, pipes are chains of small programs or "tools" that carry out a single well-defined task (e.g., `ed`, `gsub`, `grep`, `more`, etc.). Data such as text is described as flowing from a source into a sink through a series of steps at which a specific transformation takes place. In Unix, sinks and sources are files, but files as an abstraction include all devices and connections for input or output, including physical ones as terminals and printers. The connection between steps in the pipe is usually implemented by means of temporary files.

```
stdin | grep("abc") | more
```

How can *pipes* exist within a single R script? When chaining functions into a pipe, data is passed between them through temporary R objects stored in memory, which are created and destroyed automatically. Conceptually there is little difference between Unix shell pipes and pipes in R scripts, but the implementations are different.

What do pipes achieve in R scripts? They relieve the user from the responsibility of creating and deleting the temporary objects and of enforcing the sequential execution of the different steps. Pipes usually improve readability of scripts by allowing more concise code.

Currently, two main implementations of pipes are available as R extensions, in packages 'magrittr' and 'wrapr' and since version 4.1.0 R has pipes as part of the language.

1.5.1 Base R

We describe first R's pipe syntax based on R 4.2.0. We start with a toy example first written using separate steps and traditional R syntax

```
data.in <- 1:10
data.tmp <- sqrt(data.in)
data.out <- sum(data.tmp)
rm(data.tmp) # clean up!
```

next using nested function calls still using traditional R syntax

```
data.out <- sum(sqrt(data.in))
```

written as a pipe using `|>`, the chaining operator from current R.

```
data.in |> sqrt() |> sum() -> data.out
```

The `|>` operator from base R takes two operands. The value returned by the *lhs* (left-hand side) operand, which can be any R expression, is passed by default as the first argument to the *rhs* operand, which must be a function accepting at least one argument. Consequently, in using this simple syntax, the function in the *rhs* must have a suitable signature for the pipe to work. However, it is possible to pass piped arguments to a function by name to any parameter, including the first one, using an underscore (`_`) as placeholder.

Some base R functions like `subset()` have a signature that is natural for use in pipes by implicitly passing the piped value as argument to its first formal parameter. Other functions like `assign()` in many uses we would like to pass the piped value as argument to parameters other than the first. In such cases we can use `_` as a placeholder and pass it by name. Alternatively, we can define a wrapper function, with the desired order for the formal parameters.

```
value_assign <- function(value, x, ...) {
  assign(x = x, value = value, ...)
}
```

1.5.2 'magrittr'

Another set of operators for constructing pipes of R functions is implemented in package 'magrittr' and its availability preceded the native R pipe by a few years. This implementation is used in the 'tidyverse'. The pipe operator defined in package 'magrittr' is imported and re-exported by package 'dplyr'.

The same example as in the previous section, now written as a pipe using `%>%`, the pipe operator from package 'magrittr'.

```
data.in %>% sqrt() %>% sum() -> data.out
```

Package ‘magrittr’ provides additional pipe operators, such as “tee” (`%T>%`) to create a branch in the pipe, and `%<>%` to apply the pipe by reference. These operators are much less frequently used than `%>%`.

1.5.3 ‘wrapr’

The `%.>%`, or “dot-pipe”, operator from package ‘wrapr’, allows expressions both on the rhs and lhs, and *enforces the use of the dot* (`.`), as placeholder for the piped object.

Rewritten using the dot-pipe operator, the pipe in the previous chunk becomes

```
data.in %.>% sqrt(.) %.>% sum(.) -> data1.out
```

However, as operator `%>%` from ‘magrittr’ recognizes the `.` placeholder without enforcing its use, the code below where `%.>%` was replaced by `%>%` returns the same value as that above.

```
data.in %>% sqrt(.) %>% sum(.) -> data2.out
all.equal(data1.out, data2.out)
## [1] TRUE
```

To use operator `|>` from R, we need to edit the code using a different placeholder (`_`) and passing it as argument to parameters by name in the function calls on the *rhs*.

```
data.in |> sqrt(x = _) |> sum(x = _) -> data3.out
all.equal(data1.out, data3.out)
## [1] TRUE
```

The design of R’s native pipes has benefited from the experience gathered by earlier implementations and being now part the language, we can expect it to become the reference one once its implementation is stable. The designers of the three implementations have to some extent disagreed in their decisions. Consequently, some differences are more than aesthetic.

The syntax of operators `|>` and `%>%` is not identical. With R’s `|>` the placeholder `_` can be only passed to parameters by name, while with ‘magrittr’’s `%>%` the placeholder `.` can be used to pass arguments both by name and by position (as of R 4.2.0). With operator `%.>%` the use of the placeholder `.` is mandatory, and it can be passed by name of by position to the function call on the *rhs*.

In the case of R, the pipe is conceptually a substitution with no alteration of the syntax or evaluation order. R’s native pipes requires, consistently with R in all other situations, that functions that are to be evaluated use the parenthesis syntax, while ‘magrittr’ allows the parentheses to be missing when the piped argument is the only one passed to the function call on *rhs*.

```
data.in %>% sqrt %>% sum -> data.out
```

⚠ In some situation the semantics of the operator `%>%` from package 'magrittr' can behave unexpectedly. One example is attempting to use `assign()` in a pipe.

```
data.in |> assign(x = "data4.out", value = _)
all.equal(data.in, data4.out)
## [1] TRUE
```

Named arguments are also supported with the dot-pipe operator from 'wrapr' resulting in the expected behavior.

```
data.in %.>% assign(x = "data5.out", value = .)
all.equal(data.in, data5.out)
## [1] TRUE
```

In contrast, the pipe operator (`%>%`) from package 'magrittr' silently and unexpectedly fails to create the variable for the same example. This can be a problem when the name passed as argument to `assign()`'s parameter `x` is a computed value, otherwise it is possible to use the `->` to assign the value.

```
data.in %>% assign(x = "data6.out", value = .)
if (exists("data6.out")) {
  all.equal(data.in, data6.out)
} else {
  print("'data6.out' not found!")
}
## [1] "'data6.out' not found!"
```

The dot-pipe operator `%.>%` from 'wrapr' allows us to use the placeholder `.` in expressions on its *rhs* in addition to in function calls

```
data.in %.>% (.^2) -> data7.out
```

meanwhile operators `|>` and `%>%` do not support expressions, only function call syntax on their *rhs*, forcing us to call operators with parenthesis syntax and named arguments

```
data.in |> `^`(e1 = _, e2 = 2) -> data8.out
all.equal(data7.out, data8.out)
## [1] TRUE
```

or

```
data.in %>% `^`(e1 = ., e2 = 2) -> data9.out
all.equal(data7.out, data9.out)
## [1] TRUE
```

In conclusion, R syntax for expressions is preserved when using the dot-pipe operator, with the only caveat that because of the higher precedence of the `%.>%` operator, we need to "protect" bare expressions containing other operators by enclosing them in parentheses. In the examples above we showed a simple expression so that could be easily converted into a function call. The `%.>%` operator supports also more complex expressions, even with multiple uses of the placeholder.

```
data.in %.>% (.^2 + sqrt(. + 1))
##  [1]   2.414214   5.732051  11.000000  18.236068  27.449490  38.645751
##  [7]  51.828427  67.000000  84.162278 103.316625
```

Under-the-hood, the implementations of operators `|>` and `%>%` and `%.>%` are different, with `|>` expected to have the best performance, followed by `%.>%` and `%>%` being slowest. As implementations evolve performance depends on versions. However, `|>` being part of R is likely to remain the fastest.

Being part of the R language, `|>` will remain available and backwards compatible, while packages could be abandoned or redesigned by their maintainers. For this reason, for scripts or code expected to be reused, and not requiring compatibility with R versions before 4.2.0, new code would benefit from using this new operator.

In the rest of the book when possible we will use *R's pipes* and otherwise *dot pipes* and avoid implicit (”invisible”) passing of arguments in examples to ensure easier understanding. In most cases the examples can be easily rewritten using the older operator `%>%`.

Pipes can make scripts visually more compact than the use of assignments of intermediate results to temporary variables. What makes pipes most convenient is the availability of classes, functions, and methods defined in 'tidyr', 'dplyr', and other packages from the 'tidyverse'. Debugging pipes usually requires dividing them, with one approach being the insertion of calls to `print()`. This is possible, because `print()` returns its input invisibly in addition to displaying it.

```
data.in |> print() |> sqrt() |> print() |> sum() |> print() -> data10.out
##  [1]  1  2  3  4  5  6  7  8  9 10
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
##  [9] 3.000000 3.162278
## [1] 22.46828

all.equal(data.out, data10.out)
## [1] TRUE
```

1.6 Reshaping with 'tidyr'

Data stored in table-like formats can be arranged in different ways. In base R most model fitting functions and the `plot()` method using (model) formulas and accepting data frames, expect data to be arranged in “long form” so that each row in a data frame corresponds to a single observation (or measurement) event on a subject. Each column corresponds to a different measured feature, time of measurement, or a factor describing a classification of subjects according to treatments or features of the experimental design (e.g., blocks). Covariates measured on the same subject at an earlier point in time may also be stored in a column. Data arranged in *long form* has been nicknamed as “tidy” and this is reflected in the name given to the 'tidyverse' suite of packages. Data in which columns correspond to measurement events is described as being in a *wide form*.

Although long-form data is and has been the most commonly used arrangement of data in R, manipulation of such data has not always been possible with concise R statements. The packages in the 'tidyverse' provide convenience functions to simplify coding of data manipulation, which in some cases, have, in addition, improved performance compared to base R—i.e., it is possible to code the same operations using only base R, but may require more and/or more verbose statements.

Real-world data is rather frequently stored in wide format or even ad hoc formats, so in many cases the first task in data analysis is to reshape the data. Package 'tidyr' provides functions for reshaping data from wide to long form and *vice versa* (replacing the older packages 'reshape' and 'reshape2').

We use in examples below the `iris` data set included in base R. Some operations on R `data.frame` objects with 'tidyverse' packages will return `data.frame` objects while others will return tibbles—i.e., `"tb"` objects. Consequently it is safer to first convert into tibbles the data frames we will work with.

```
iris.tb <- as_tibble(iris)
```

Function `pivot_longer()` converts data from wide form into long form (or ”tidy”). We use `pivot_longer()` to obtain a long-form tibble. By comparing `iris.tb` with `long_iris.tb` we can appreciate how `pivot_longer()` reshaped its input.

```
head(iris.tb, 2)
## # A tibble: 2 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl> <fct>
## 1          5.1         3.5          1.4         0.2 setosa
## 2          4.9         3            1.4         0.2 setosa

iris.tb |>
  gather(data = _, key = part, value = dimension, -Species) -> long_iris.tb
long_iris.tb
## # A tibble: 600 x 3
##   Species part         dimension
##   <fct>   <chr>            <dbl>
## 1 setosa  Sepal.Length       5.1
## 2 setosa  Sepal.Length       4.9
## 3 setosa  Sepal.Length       4.7
## # ... with 597 more rows
```

In this statement, we can see the convenience of dispensing with quotation marks for the new (`part` and `dimension`) and existing (`Species`) column names. Use of bare names as above triggers errors when package code is tested, requiring the use of a less convenient but more consistent and reliable syntax instead. As it is also possible to pass column names as strings but not together with the subtraction operator, equivalent code becomes more verbose but with the intention explicit and easier to grasp.

```
long_iris.tb_1  <-  gather(iris.tb,  key  =  "part",  value  =  "dimension",  setd-
iff(colnames(iris.tb), "Species"))
long_iris.tb_1
```

```
## # A tibble: 600 x 3
##   Species part         dimension
##   <fct>   <chr>            <dbl>
## 1 setosa  Sepal.Length       5.1
## 2 setosa  Sepal.Length       4.9
## 3 setosa  Sepal.Length       4.7
## # ... with 597 more rows
```

⚠ Altering R's normal interpretation of the name passed as an argument to `key` and `value` prevents these arguments from being recognized as the name of a variable in the calling environment. We need to use a new operator `!!` to restore the normal R behavior.

```
part <- "not part"
long_iris.tb_2 <- gather(iris.tb, key = !!part, value = dimension, -Species)
long_iris.tb_2
## # A tibble: 600 x 3
##   Species `not part`   dimension
##   <fct>   <chr>            <dbl>
## 1 setosa  Sepal.Length       5.1
## 2 setosa  Sepal.Length       4.9
## 3 setosa  Sepal.Length       4.7
## # ... with 597 more rows
```

This syntax has been recently subject to debate and led to John Mount developing package 'seplyr' which provides wrappers on functions and methods from 'dplyr' that respect standard evaluation (SE). At the time of writing, 'seplyr' can be considered as experimental.

To better understand why I added `-Species` as an argument, edit the code by removing it, and execute the statement to see how the returned tibble is different.

For the reverse operation, converting from long form to wide form, we use `spread()`.

```
spread(long_iris.tb, key = c(!!part, Species), value = dimension) # does not work!!
```

⚠ Starting from version 1.0.0 of 'tidyr', `gather()` and `spread()` are deprecated and replaced by `pivot_longer()` and `pivot_wider()`. These new functions use a different syntax but are not yet fully stable.

1.7 Data manipulation with ‘dplyr’

The first advantage a user of the ‘dplyr’ functions and methods sees is the completeness of the set of operations supported and the symmetry and consistency among the different functions. A second advantage is that almost all the functions are defined not only for objects of class `tibble`, but also for objects of class `data.table` (packages ‘dtplyr’) and for SQL databases (‘dbplyr’), with consistent syntax (see also section ?? on page ??). A further variant exists in package ‘seplyr’, supporting a different syntax stemming from the use of “standard evaluation” (SE) instead of non-standard evaluation (NSE). A downside of ‘dplyr’ and much of the ‘tidyverse’ is that the syntax is not yet fully stable. Additionally, some function and method names either override those in base R or clash with names used in other packages. R itself is extremely stable and expected to remain forward and backward compatible for a long time. For code intended to remain in use for years, the fewer packages it depends on, the less maintenance it will need. When using the ‘tidyverse’ we need to be prepared to revise our own dependent code after any major revision to the ‘tidyverse’ packages we may use.

A new package, ‘poorman’, implements many of the same words and grammar as ‘dplyr’ using pure R in the implementation instead of compiled C++ and C code. This light-weight approach could be useful when dealing with relatively small data sets or when the use of R’s data frames instead of tibbles is preferred.

1.7.1 Row-wise manipulations

Assuming that the data is stored in long form, row-wise operations are operations combining values from the same observation event—i.e., calculations within a single row of a data frame or tibble. Using functions `mutate()` and `transmute()` we can obtain derived quantities by combining different variables, or variables and constants, or applying a mathematical transformation. We add new variables (columns) retaining existing ones using `mutate()` or we assemble a new tibble containing only the columns we explicitly specify using `transmute()`.

Different from usual R syntax, with `tibble()`, `mutate()` and `transmute()` we can use values passed as arguments, in the statements computing the values passed as later arguments. In many cases, this allows more concise and easier to understand code.

```
tibble(a = 1:5, b = 2 * a)
## # A tibble: 5 x 2
##       a     b
##   <int> <dbl>
## 1     1     2
## 2     2     4
## 3     3     6
## # ... with 2 more rows
```

Continuing with the example from the previous section, we most likely would like to split the values in variable `part` into `plant_part` and `part_dim`. We use `mutate()` from ‘dplyr’ and `str_extract()` from ‘stringr’. We use regular expressions as arguments passed to `pattern`. We do not show it here, but `mutate()` can be used with variables of any `mode`, and calculations can involve values from several columns. It is even possible to operate on values applying a lag or, in other words, using rows displaced relative to the current one.

```
long_iris.tb %.>%
  mutate(.,
         plant_part = str_extract(part, "^[:alpha:]*"),
         part_dim = str_extract(part, "[:alpha:]*$")) -> long_iris.tb
long_iris.tb
## # A tibble: 600 x 5
##   Species part         dimension plant_part part_dim
##   <fct>   <chr>            <dbl> <chr>      <chr>
## 1 setosa  Sepal.Length       5.1 Sepal      Length
## 2 setosa  Sepal.Length       4.9 Sepal      Length
## 3 setosa  Sepal.Length       4.7 Sepal      Length
## # ... with 597 more rows
```

In the next few chunks, we print the returned values rather than saving them in variables. In normal use, one would combine these functions into a pipe using operator `%.>%` (see section 1.5 on page 9).

Function `arrange()` is used for sorting the rows—makes sorting a data frame or tibble simpler than by using `sort()` and `order()`. Here we sort the tibble `long_iris.tb` based on the values in three of its columns.

```
arrange(long_iris.tb, Species, plant_part, part_dim)
## # A tibble: 600 x 5
##   Species part         dimension plant_part part_dim
##   <fct>   <chr>            <dbl> <chr>      <chr>
## 1 setosa  Petal.Length       1.4 Petal      Length
## 2 setosa  Petal.Length       1.4 Petal      Length
## 3 setosa  Petal.Length       1.3 Petal      Length
## # ... with 597 more rows
```

Function `filter()` can be used to extract a subset of rows—similar to `subset()` but with a syntax consistent with that of other functions in the ‘tidyverse’. In this case, 300 out of the original 600 rows are retained.

```
filter(long_iris.tb, plant_part == "Petal")
```

```
## # A tibble: 300 x 5
##   Species part         dimension plant_part part_dim
##   <fct>   <chr>            <dbl> <chr>      <chr>
## 1 setosa  Petal.Length       1.4 Petal      Length
## 2 setosa  Petal.Length       1.4 Petal      Length
## 3 setosa  Petal.Length       1.3 Petal      Length
## # ... with 297 more rows
```

Function `slice()` can be used to extract a subset of rows based on their positions—an operation that in base R would use positional (numeric) indexes with the `[ , ]` operator: `long_iris.tb[1:5, ]`.

```
slice(long_iris.tb, 1:5)
## # A tibble: 5 x 5
##   Species part         dimension plant_part part_dim
##   <fct>   <chr>            <dbl> <chr>      <chr>
## 1 setosa  Sepal.Length       5.1 Sepal      Length
## 2 setosa  Sepal.Length       4.9 Sepal      Length
## 3 setosa  Sepal.Length       4.7 Sepal      Length
## # ... with 2 more rows
```

Function `select()` can be used to extract a subset of columns–this would be done with positional (numeric) indexes with `[ , ]` in base R, passing them to the second argument as numeric indexes or column names in a vector. Negative indexes in base R can only be numeric, while `select()` accepts bare column names prepended with a minus for exclusion.

```
select(long_iris.tb, -part)
## # A tibble: 600 x 4
##   Species dimension plant_part part_dim
##   <fct>       <dbl> <chr>      <chr>
## 1 setosa        5.1 Sepal      Length
## 2 setosa        4.9 Sepal      Length
## 3 setosa        4.7 Sepal      Length
## # ... with 597 more rows
```

In addition, `select()` as other functions in 'dplyr' accept "selectors" returned by functions `starts_with()`, `ends_with()`, `contains()`, and `matches()` to extract or retain columns. For this example we use the "wide"-shaped `iris.tb` instead of `long_iris.tb`.

```
select(iris.tb, -starts_with("Sepal"))
## # A tibble: 150 x 3
##   Petal.Length Petal.Width Species
##          <dbl>       <dbl> <fct>
## 1          1.4         0.2 setosa
## 2          1.4         0.2 setosa
## 3          1.3         0.2 setosa
## # ... with 147 more rows
```

```
select(iris.tb, Species, matches("pal"))
## # A tibble: 150 x 3
##   Species Sepal.Length Sepal.Width
##   <fct>          <dbl>       <dbl>
## 1 setosa           5.1         3.5
## 2 setosa           4.9         3
## 3 setosa           4.7         3.2
## # ... with 147 more rows
```

Function `rename()` can be used to rename columns, whereas base R requires the use of both `names()` and `names<-()` and *ad hoc* code to match new and old names. As shown below, the syntax for each column name to be changed is `<new name> = <old name>`. The two names can be given either as bare names as below or as character strings.

```
rename(long_iris.tb, dim = dimension)
## # A tibble: 600 x 5
##   Species part           dim plant_part part_dim
##   <fct>   <chr>        <dbl> <chr>      <chr>
## 1 setosa  Sepal.Length   5.1 Sepal      Length
## 2 setosa  Sepal.Length   4.9 Sepal      Length
## 3 setosa  Sepal.Length   4.7 Sepal      Length
## # ... with 597 more rows
```

1.7.2 Group-wise manipulations

Another important operation is to summarize quantities by groups of rows. Contrary to base R, the grammar of data manipulation, splits this operation in two: the setting of the grouping, and the calculation of summaries. This simplifies the code, making it more easily understandable when using pipes compared to the approach of base R `aggregate()`, and it also makes it easier to summarize several columns in a single operation.

⚠ It is important to be aware that grouping is persistent, and may also affect other operations on the same data frame or tibble if it is saved or piped and reused. Grouping is invisible to users except for its side effects and because of this can lead to erroneous and surprising results from calculations. Do not save grouped tibbles or data frames, and always make sure that inputs and outputs, at the head and tail of a pipe, are not grouped, by using `ungroup()` when needed.

The first step is to use `group_by()` to “tag” a tibble with the grouping. We create a *tibble* and then convert it into a *grouped tibble*. Once we have a grouped tibble, function `summarise()` will recognize the grouping and use it when the summary values are calculated.

```
tibble(numbers = 1:9, letters = rep(letters[1:3], 3)) %.>%
  group_by(., letters) %.>%
  summarise(.,
            mean_numbers = mean(numbers),
            median_numbers = median(numbers),
            n = n())
```

```
## # A tibble: 3 x 4
##   letters mean_numbers median_numbers     n
##   <chr>          <dbl>          <int> <int>
## 1 a                  4              4     3
## 2 b                  5              5     3
## 3 c                  6              6     3
```

How is grouping implemented for data frames and tibbles? In our case as our tibble belongs to class `tibble_df`, grouping adds `grouped_df` as the most derived class. It also adds several attributes with the grouping information in a format suitable for fast selection of group members. To demonstrate this, we need to make an exception to our recommendation above and save a grouped tibble to a variable.

```
my.tb <- tibble(numbers = 1:9, letters = rep(letters[1:3], 3))
is.grouped_df(my.tb)
## [1] FALSE

class(my.tb)
## [1] "tbl_df"     "tbl"        "data.frame"

names(attributes(my.tb))
## [1] "class"     "row.names" "names"
```

```
my_gr.tb <- group_by(.data = my.tb, letters)
is.grouped_df(my_gr.tb)
## [1] TRUE

class(my_gr.tb)
## [1] "grouped_df" "tbl_df"     "tbl"        "data.frame"
```

```
names(attributes(my_gr.tb))
## [1] "class"     "row.names" "names"     "groups"

setdiff(attributes(my_gr.tb), attributes(my.tb))
## $class
## [1] "grouped_df" "tbl_df"     "tbl"        "data.frame"
##
## $groups
## # A tibble: 3 x 2
##   letters       .rows
##   <chr>   <list<int>>
## 1 a               [3]
## 2 b               [3]
## 3 c               [3]
```

```
my_ugr.tb <- ungroup(my_gr.tb)
class(my_ugr.tb)
## [1] "tbl_df"     "tbl"        "data.frame"

names(attributes(my_ugr.tb))
## [1] "class"     "row.names" "names"
```

```
all(my.tb == my_gr.tb)
## [1] TRUE

all(my.tb == my_ugr.tb)
## [1] TRUE

identical(my.tb, my_gr.tb)
## [1] FALSE

identical(my.tb, my_ugr.tb)
## [1] TRUE
```

The tests above show that members are in all cases the same as operator `==` tests for equality at each position in the tibble but not the attributes, while attributes, including `class` differ between normal tibbles and grouped ones and so they are not *identical* objects.

If we replace `tibble` by `data.frame` in the first statement, and rerun the chunk, the result of the last statement in the chunk is `FALSE` instead of `TRUE`. At the time of writing starting with a `data.frame` object, applying grouping with `group_by()` followed by ungrouping with `ungroup()` has the side effect of converting the data frame into a tibble. This is something to be very much aware of, as there are differences in how the extraction operator `[ , ]` behaves in the two cases. The safe way to write code making use of functions from ‘dplyr’ and ‘tidyr’ is to always use tibbles instead of data frames.

1.7.3 Joins

Joins allow us to combine two data sources which share some variables. Variables in common are used to match the corresponding rows before “joining” variables (i.e., columns) from both sources together. There are several *join* functions in ‘dplyr’. They differ mainly in how they handle rows that do not have a match between data sources.

We create here some artificial data to demonstrate the use of these functions. We will create two small tibbles, with one column in common and one mismatched row in each.

```
first.tb <- tibble(idx = c(1:4, 5), values1 = "a")
second.tb <- tibble(idx = c(1:4, 6), values2 = "b")
```

Below we apply the functions exported by ‘dplyr’: `full_join()`, `left_join()`, `right_join()` and `inner_join()`. These functions always retain all columns, and in case of multiple matches, keep a row for each matching combination of rows. We repeat each example with the arguments passed to `x` and `y` swapped to more clearly show their different behavior.

A full join retains all unmatched rows filling missing values with `NA`. By default the match is done on columns with the same name in `x` and `y`, but this can be changed by passing an argument to parameter `by`. Using `by` one can base the match on columns that have different names in `x` and `y`, or prevent matching of columns with the same name in `x` and `y` (example at end of the section).

```
full_join(x = first.tb, y = second.tb)

## Joining, by = "idx"
## # A tibble: 6 x 3
##     idx values1 values2
##   <dbl> <chr>   <chr>
## 1     1 a       b
## 2     2 a       b
## 3     3 a       b
## 4     4 a       b
## 5     5 a       <NA>
## 6     6 <NA>    b
```

```
full_join(x = second.tb, y = first.tb)

## Joining, by = "idx"
## # A tibble: 6 x 3
##     idx values2 values1
##   <dbl> <chr>   <chr>
## 1     1 b       a
## 2     2 b       a
## 3     3 b       a
## 4     4 b       a
## 5     6 b       <NA>
## 6     5 <NA>    a
```

Left and right joins retain rows not matched from only one of the two data sources, `x` and `y`, respectively.

```
left_join(x = first.tb, y = second.tb)

## Joining, by = "idx"
## # A tibble: 5 x 3
##     idx values1 values2
##   <dbl> <chr>   <chr>
## 1     1 a       b
## 2     2 a       b
## 3     3 a       b
## 4     4 a       b
## 5     5 a       <NA>
```

```
left_join(x = second.tb, y = first.tb)

## Joining, by = "idx"
## # A tibble: 5 x 3
##     idx values2 values1
##   <dbl> <chr>   <chr>
## 1     1 b       a
## 2     2 b       a
## 3     3 b       a
## 4     4 b       a
## 5     6 b       <NA>
```

```
right_join(x = first.tb, y = second.tb)

## Joining, by = "idx"
## # A tibble: 5 x 3
##     idx values1 values2
##   <dbl> <chr>   <chr>
## 1     1 a       b
## 2     2 a       b
## 3     3 a       b
## 4     4 a       b
## 5     6 <NA>    b
```

```
right_join(x = second.tb, y = first.tb)

## Joining, by = "idx"
## # A tibble: 5 x 3
##     idx values2 values1
##   <dbl> <chr>   <chr>
## 1     1 b       a
## 2     2 b       a
## 3     3 b       a
## 4     4 b       a
## 5     5 <NA>    a
```

An inner join discards all rows in `x` that do not have a matching row in `y` and *vice versa*.

```
inner_join(x = first.tb, y = second.tb)

## Joining, by = "idx"
## # A tibble: 4 x 3
##     idx values1 values2
##   <dbl> <chr>   <chr>
## 1     1 a       b
## 2     2 a       b
## 3     3 a       b
## 4     4 a       b
```

```
inner_join(x = second.tb, y = first.tb)

## Joining, by = "idx"
## # A tibble: 4 x 3
##     idx values2 values1
##   <dbl> <chr>   <chr>
## 1     1 b       a
## 2     2 b       a
## 3     3 b       a
## 4     4 b       a
```

Next we apply the *filtering join* functions exported by ‘dplyr’: `semi_join()` and `anti_join()`. These functions only return a tibble that always contains only the columns from `x`, but retains rows based on their match to rows in `y`.

A semi join retains rows from `x` that have a match in `y`.

```
semi_join(x = first.tb, y = second.tb)

## Joining, by = "idx"
## # A tibble: 4 x 2
##     idx values1
##   <dbl> <chr>
## 1     1 a
## 2     2 a
## 3     3 a
## 4     4 a
```

```
semi_join(x = second.tb, y = first.tb)

## Joining, by = "idx"
## # A tibble: 4 x 2
##     idx values2
##   <dbl> <chr>
## 1     1 b
## 2     2 b
## 3     3 b
## 4     4 b
```

A anti-join retains rows from `x` that do not have a match in `y`.

```
anti_join(x = first.tb, y = second.tb)

## Joining, by = "idx"
## # A tibble: 1 x 2
##     idx values1
##   <dbl> <chr>
## 1     5 a
```

```
anti_join(x = second.tb, y = first.tb)

## Joining, by = "idx"
## # A tibble: 1 x 2
##     idx values2
##   <dbl> <chr>
## 1     6 b
```

We here rename column `idx` in `first.tb` to demonstrate the use of `by` to specify which columns should be searched for matches.

```
first2.tb <- rename(first.tb, idx2 = idx)
full_join(x = first2.tb, y = second.tb, by = c("idx2" = "idx"))
## # A tibble: 6 x 3
##    idx2 values1 values2
##   <dbl> <chr>   <chr>
## 1     1 a       b
## 2     2 a       b
## 3     3 a       b
## 4     4 a       b
## 5     5 a       <NA>
## 6     6 <NA>    b
```

1.8 Further reading

An in-depth discussion of the ‘tidyverse’ is outside the scope of this book. Several books describe in detail the use of these packages. As several of them are under active development, recent editions of books such as *R for Data Science* (Wickham and Grolemund 2017) are the most useful.

Bibliography

Boas, R. P. (1981). “Can we make mathematics intelligible?” In: *The American Mathematical Monthly* 88.10, pp. 727–731.

Burns, P. (1998). *S Poetry*.

Kernigham, B. W. and P. J. Plauger (1981). *Software Tools in Pascal*. Reading, Massachusetts: Addison-Wesley Publishing Company. 366 pp. (cit. on pp. 2, 9).

Matloff, N. (2011). *The Art of R Programming: A Tour of Statistical Software Design*. No Starch Press, p. 400. ISBN: 1593273843 (cit. on p. 1).

Peng, R. D. (2016). *R Programming for Data Science*. Leanpub. 182 pp. URL: `https://leanpub.com/rprogramming` (visited on 07/31/2019) (cit. on p. 3).

Wickham, H. and G. Grolemund (2017). *R for Data Science*. O’Reilly UK Ltd. ISBN: 1491910399 (cit. on pp. 3, 25).

General index

Index of R names by category

Alphabetic index of R names