

Pedro J. Aphalo

Learn R

As a Language

Contents

List of Figures	vii
List of Tables	ix
1 R Extensions: Data Wrangling	1
1.1 Aims of this chapter	1
1.2 Introduction	2
1.3 Packages used in this chapter	4
1.4 Replacements for <code>data.frame</code>	4
1.4.1 Package ‘ <code>data.table</code> ’	4
1.4.2 Package ‘ <code>tibble</code> ’	5
1.5 Data pipes	10
1.5.1 ‘ <code>magrittr</code> ’	11
1.5.2 ‘ <code>wrapr</code> ’	11
1.5.3 Comparing pipes	13
1.6 Reshaping with ‘ <code>tidyr</code> ’	15
1.7 Data manipulation with ‘ <code>dplyr</code> ’	16
1.7.1 Row-wise manipulations	17
1.7.2 Group-wise manipulations	19
1.7.3 Joins	21
1.8 Further reading	25
Bibliography	27
General Index	29
Alphabetic Index of R Names	31
Index of R Names by Category	33



List of Figures



List of Tables



1

R Extensions: Data Wrangling

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 in my view they should be preferred unless some of their limitations justify the use of alternatives defined in contributed packages. In the present chapter we describe the new syntax 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. The developers of packages in the ‘tidyverse’ have had a different view than the developers of R about the compromise between innovation and backwards compatibility. While for the development of base R not breaking existing code and avoiding features that can be misinterpreted has been the priority, several of the packages in the ‘tidyverse’ have prioritized experimentation with enhanced features over backwards compatibility, focusing more on users’ convenience than reliability. Because of this, I do not describe in depth the “new grammar” but instead compare the new approach to how the same operations can be achieved within R. It must be pointed out that these and other packages have highlighted weaknesses in R that have subsequently been addressed.

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. 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 and manipulation 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 passing arguments by reference instead of by copy, increasing the “responsibility” of the programmer or data analyst with respect to not overwriting or corrupting data.

When a computation includes a chain of sequential operations, until R 4.1.0 if using base R, we could either store the returned value in a temporary variable at each step in the computation, or nest multiple function calls. The first approach is verbose, but allows readable scripts, especially if the names used for temporary variable are wisely chosen. The second approach becomes very difficult to 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 (Kernighan 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. In year 2021 a pipe operator was added to the R language itself and more recently its features enhanced.

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 by default in how *incomplete names* are handled. Package ‘tibble’ alters details of the default behaviour of an alternative to base R’s data frames. A different default prioritises different behaviour 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. In both cases performance depends on how user code is written and the size and shape of data sets.

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 the call to `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 columns or rows from a data frame it is always safer to use the `[,]` or `[[]]` operators at the expense of some verbosity.

Package ‘`dplyr`’ provides convenience functions that work in a similar way as base R `subset()`, although in recent versions possibly more safely. This package has suffered quite drastic changes during its development history 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 constructs. For code that is to be used once, or for which reproducibility can depend on the use of a specific (old or soon to become old) version of packages like ‘`dplyr`’, or which is not a burden to thoroughly test and update regularly, the conciseness and power of the new syntax can be an advantage.

In much of my work I emphasize reproducibility and reliability, preferring base R over extension packages whenever practical. For run once and delete or quick-and-dirty data analyses I tend to use the *tidyverse*. With modern computers and some understanding of what are the performance bottlenecks in R code, I have rarely found it worthwhile the effort needed to learn a new grammar for improved performance. The case may be different for some readers if they have to analyze huge data sets.

What is usually described as the *tidyverse* is a combination of a code-writing style with some additions to the grammar of R. Package ‘`tidyverse`’ loads and attaches a set of packages of which most but not all follow a consistent design and support this new grammar. In this chapter you will become familiar with packages ‘`tibble`’, ‘`dplyr`’ and ‘`tidyr`’. Package ‘`ggplot2`’ will be described in chapter ?? as it implements the grammar of graphics and has little in common with other members of the ‘`tidyverse`’. As many of the functions in the *tidyverse* can be substituted by existing base R functions, recognizing similarities and differences between them has become important since both approaches are now in common use, and frequently even coexist within the same R scripts.

This chapter gives only a brief overview of some features of the ‘`tidyverse`’ and compares them to their base R equivalents. 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) cover the same subjects in depth 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 and attach 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 Package ‘`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 in memory 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 large objects such as 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`’ pass arguments by reference, avoiding making any copies. However, any assignments within these functions and methods modify the variables passed as arguments. 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`.

In contrast to the design of package ‘`data.table`’, the focus of the ‘`tidyverse`’ is not only performance. The design of this grammar has also considered usability. Design compromises have been resolved differently than in base R or ‘`data.table`’ and in some cases code written using base R can significantly outperform the

‘tidyverse’ and vice versa. There exist packages that implement a translation layer from the syntax of the ‘tidyverse’ into that of ‘data.table’ or relational database queries.

1.4.2 Package ‘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. Tibbles simplify the creation of “columns” of class `list` and remove 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 with tibbles as a result of the differences in syntax of some methods.



It is easy to write code that will work correctly both with data frames and tibbles by avoiding constructs that behave differently. However, code that is syntactically correct according to the R language may fail to work as expected if a tibble is used in place of a data frame. Only functions tested to work correctly with both tibbles and data frames can be relied upon as compatible.

Being newer and not part of the R language, the packages in the ‘tidyverse’ are evolving with rather frequent changes that require edits to the code of scripts and packages that use them. For example, whether attributes set in tibbles by users are copied or not to returned values has changed with updates.



That it has been possible to define tibbles as objects of a class derived from `data.frame` reveals one of the drawbacks of the simple implementation of S3 object classes in R. Allowing this is problematic because the promise of compatibility implicit in a derived class is not always fulfilled. An independently developed method designed for data frames will not necessarily work correctly with tibbles, but in the absence of a specialized method for tibbles it will be used (dispatched) when the generic method is called with a tibble as argument.



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`. This is inconsistent with base R conventions, as it is the use of an underscore instead of a dot in the name of these methods.

```
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, to support tibbles based on different underlying data sources such `data.table` objects or databases, 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 define a function that concisely reports the class of the object passed as argument and of its members (*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")
  )
}
```

The `tibble()` constructor by default does not convert character data into factors, while the `data.frame()` constructor did before R version 4.0.0. The default can be overridden through an argument passed to these constructors, and in the case of `data.frame()` also be setting an R option. This new behaviour extends to function `read.table()` and its wrappers (see section ?? on page ??).

```
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, or pretend to be (see above), 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 differs from that for data frames in that it outputs a header with the text “A tibble:” followed by the dimensions (number of rows \times 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 more compactly and using color to highlight, for example, negative numbers in red.

```
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
```

 The default number of rows printed depends on R option `tibble.print_max` that can be set with a call to `options()`. This option plays for tibbles a similar role as option `max.print` plays for base R `print()` methods.

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

 Print methods for tibbles and data frames also differ in their behaviour when not all columns fit in a printed line. 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 R console 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
```

Tibbles can be converted into “real” data.frames with `as.data.frame()`.

```
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
```

 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). Some packages in the ‘tidyverse’ sometimes follow, or have followed in the past, 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.

As tibbles have been defined as a class derived from *data.frame*, if methods have not been explicitly defined for tibbles, the methods defined for data frames are called, and these are likely to return a data frame rather than a tibble. Even a frequent operation like column binding is affected, at least at the time of writing.

```

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

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

class(cbind(my.df, my.tb))
## [1] "data.frame"

class(cbind(my.tb, my.df))
## [1] "data.frame"

class(cbind(my.df, added = -3:-1))
## [1] "data.frame"

class(cbind(my.tb, added = -3:-1))
## [1] "data.frame"

identical(cbind(my.tb, added = -3:-1), cbind(my.df, added = -3:-1))
## [1] TRUE

```

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
## # i 2 more rows
```



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



While objects passed directly as arguments to the `data.frame()` constructor to be included as “columns” can be factors, vectors or matrices (with the same number of rows as the data frame), arguments passed to the `tibble()` constructor can be factors, vectors or lists (with the same number of members as rows in the tibble). As we saw in section ?? on page ??, base R’s data frames can contain columns of classes `list` and `matrix`. The difference is in the need to use `I()`, the identity function to protect these variables during construction and assignment to true `data.frame` objects as otherwise list members and matrix columns will be assigned to multiple individual columns in the data frame.

```
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]>
## # i 2 more rows
```

A list of lists or a list of vectors can be directly passed to the constructor.

```
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]>
## # i 2 more rows
```

1.5 Data pipes

The first obvious difference between scripts using ‘tidyverse’ packages is the frequent use of *pipes*. This is, however, mostly a question of preferences, as pipes

can be as well used with base R functions. In addition, since version 4.0.0, R has a native pipe operator `|>`, described in section ?? on page ?. Here we describe other earlier implementations of pipes, and the differences among these and R's pipe operator.

1.5.1 'magrittr'

A set of operators for constructing pipes of R functions is implemented in package 'magrittr'. It preceded the native R pipe by several years. The pipe operator defined in package 'magrittr', `%>%`, is imported and re-exported by package 'dplyr', which in turn defines functions that work well in data pipes.

Operator `%>%` plays a similar role as R's `|>`.

```
data.in <- 1:10
```

```
data.in %>% sqrt() %>% sum() -> data0.out
```

The value passed can be made explicit using a dot as placeholder passed as an argument by name and by position to the function on the *rhs* of the `%>%` operator. Thus `.` in 'magrittr' plays a similar but not identical role as `_` in base R pipes.

```
data.in %>% sqrt(x = .) %>% sum(.) -> data1.out
all.equal(data0.out, data1.out)
## [1] TRUE
```

R's native pipe operator 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 -> data5.out
all.equal(data0.out, data5.out)
## [1] TRUE
```

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 `%>%`.

1.5.2 '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. Given the popularity of 'dplyr' the pipe operator from 'magrittr' has been the most used.

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

```
data.in %.>% sqrt(.) %.>% sum(.) -> data2.out
all.equal(data0.out, data2.out)
## [1] TRUE
```

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

```
data.in %>% sqrt(.) %>% sum(.) -> data3.out
all.equal(data0.out, data3.out)
## [1] TRUE
```

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

```
data.in |> sqrt(x = _) |> sum(x = _) -> data4.out
all.equal(data0.out, data4.out)
## [1] TRUE
```

We can, in this case, simply use no placeholder, and pass the arguments by position to the first parameter of the functions.

```
data.in |> sqrt() |> sum() -> data4.out
all.equal(data0.out, data4.out)
## [1] TRUE
```

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

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

In contrast, 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 from ‘wrapr’, 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 it 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
```

1.5.3 Comparing pipes

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 may vary among versions. However, `|>` being part of R is likely to remain the fastest.

Being part of the R language, `|>` will remain available and most likely also backwards compatible, while packages could be abandoned or redesigned by their maintainers. For this reason, it is preferable to use the `|>` in scripts or code expected to be reused, unless compatibility with R versions earlier than 4.2.0 is needed.

In the rest of the book when possible I will use R's pipes and use in examples the `_` placeholder to facilitate understanding. In most cases the examples can be easily rewritten using operator `%>%`.

Pipes can be used with any R function, but how elegant can be their use depends on the order of formal parameters. This is especially the case when passing arguments implicitly to the first parameter of the function on the *rhs*. Several of the functions and methods defined in 'tidyr', 'dplyr', and a few other packages from the 'tidyverse' fit this need.

Writing a series of statements and saving intermediate results in temporary variables makes debugging easiest. Debugging pipes is not as easy, as this usually requires splitting 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

data10.out
## [1] 22.46828
```

Debugging nested function calls is the most difficult as well as code using such calls is difficult to read. So, in general, it is good to use pipes instead of nested function calls. However, it is best to avoid very long pipes. Normally while writing scripts or analysing data it is important to check the correctness of intermediate results, so saving them to variables can save time and effort.

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 design decisions. Consequently, some differences are more than aesthetic. R pipes are simpler, easier to use and expected to be fastest. Those from 'magrittr' are the most feature


rich, but not as safe to use, and purportedly given a more complex implementation, the slowest. Package ‘*wrapr*’ is an attempt to enhance pipes compared to ‘*magrittr*’ focusing in syntactic simplicity and performance. R’s `|>` operator has been enhanced since its addition in R only two years ago. These enhancements have all been backwards compatible.

The syntax of operators `|>` and `%>%` is not identical. With R’s `|>` (as of R 4.3.0) 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. With operator `%.>%` the use of the placeholder `.` is mandatory, and it can be passed by name or by position to the function call on the *rhs*. Other differences are deeper like those related to the use in the *rhs* of the extraction operator or support or not for expressions that are not explicit function calls.

In the case of R, the pipe is conceptually a substitution with no alteration of the syntax or evaluation order. This avoids *surprising* the user and simplifies implementation. In other words, R pipes are an alternative way of writing nested function calls. Quoting R documentation:

Currently, pipe operations are implemented as syntax transformations. So an expression written as `x |> f(y)` is parsed as `f(x, y)`. It is worth emphasizing that while the code in a pipeline is written sequentially, regular R semantics for evaluation apply and so piped expressions will be evaluated only when first used in the *rhs* expression.

While frequently the different pipe operators can substitute for each other by adjusting the syntax, in some cases the differences among them in the order and timing of evaluation of the terms needs to be taken into account.

 In some situations operator `%>%` from package ‘*magrittr*’ can behave unexpectedly. One example is the use of `assign()` in a pipe. With R’s operator `|>` assignment takes place as expected.

```
data.in |> assign(x = "data6.out", value = _)
all.equal(data.in, data6.out)
## [1] TRUE
```

Named arguments are also supported with the dot-pipe operator from ‘*wrapr*’.

```
data.in %.>% assign(x = "data7.out", value = .)
all.equal(data.in, data7.out)
## [1] TRUE
```

However, the pipe operator (`%>%`) from package ‘*magrittr*’ silently and unexpectedly fails to assign the value to the name.

```
data.in %>% assign(x = "data8.out", value = .)
if (exists("data8.out")) {
  all.equal(data.in, data8.out)
} else {
  print("'data8.out' not found!")
}
## [1] "'data8.out' not found!"
```


Although there are usually alternatives to get the computations done correctly, unexpected silent behaviour is not easy to deal with.

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 when 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, or ancillary information like the 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. However, this longitudinal arrangement of data has been the R and S preferred format since their inception. 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 this 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*.

 Package ‘tidyr’ replaced ‘reshape2’ which in turn replaced ‘reshape’, while additionally the functions implemented in ‘tidyr’ have been replaced by new ones with different syntax and name. So, using these functions although convenient, has over a period of several years made necessary to revise or rewrite scripts and relearn how to carry out these operations. If one is a data analyst and uses these functions every day, then the cost involved is frequently tolerable or even desirable given the improvements. However, if as is the case with many users of R in applied fields, to whom this book is targeted, in the long run using stable features from base R is preferable. This does not detract from the advantages of using a clear workflow as emphasized by the proponents of the *tidyverse*.

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.

```
long_iris.tb <- pivot_longer(iris.tb,
                             cols = setdiff(colnames(iris.tb), "Species"),
                             names_to = "part",
                             values_to = "dimension")

long_iris.tb
## # A tibble: 600 x 3
##   Species part      dimension
##   <fct>   <chr>         <dbl>
## 1 setosa Sepal.Length     5.1
## 2 setosa Sepal.Width      3.5
## 3 setosa Petal.Length     1.4
## # i 597 more rows
```



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.

i 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
## # i 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.Width      3.5 Sepal      width
## 3 setosa Petal.Length      1.4 Petal      Length
## # i 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 10).

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
## # i 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.Width       0.2 Petal      width
## 3 setosa Petal.Length      1.4 Petal      Length
## # i 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.Width       3.5 Sepal      width
## 3 setosa Petal.Length      1.4 Petal      Length
## # i 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      3.5 Sepal      width
## 3 setosa      1.4 Petal      Length
## # i 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
## # i 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
## # i 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.Width 3.5 Sepal width
## 3 setosa Petal.Length 1.4 Petal Length
## # i 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” vari-

ables (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 with `by = join_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 with `by = join_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 with `by = join_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 with `by = join_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 with `by = join_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 with `by = join_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 with `by = join_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 with `by = join_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 with `by = join_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 with `by = join_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 with `by = join_by(idx)`
## # A tibble: 1 x 2
##   idx values1
##   <dbl> <chr>
## 1     5 a
```

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

```
## Joining with `by = join_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

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 p. 2).

Matloff, N. (2011). *The Art of R Programming: A Tour of Statistical Software Design*. No Starch Press, p. 400. ISBN: 1593273843 (cit. on p. 2).

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

- C, 17
- C++, 17
- chaining statements with *pipes*, 10–15
- data frame
 - replacements, 4–10
- data manipulation in the tidyverse, 16–25
- ‘data.table’, 2, 4, 5
- ‘dbplyr’, 16
- dot-pipe operator, 11
- ‘dplyr’, 3, 11, 16, 17, 19, 21, 22, 24
- ‘dtplyr’, 16
- further reading
 - new grammars of data, 25
- group-wise operations on data, 19–21
- grouping
 - implementation in tidyverse, 20
- joins between data sources, 21–25
 - filtering, 24
 - mutating, 22
- languages
 - C, 17
 - C++, 17
 - S, 15
- long-form- and wide-form tabular data, 15
- ‘magrittr’, 2, 11–14
- merging data from two tibbles, 21–25
- packages
 - ‘data.table’, 2, 4, 5
 - ‘dbplyr’, 16
 - ‘dplyr’, 3, 11, 16, 17, 19, 21, 22, 24
 - ‘dtplyr’, 16
 - ‘magrittr’, 2, 11–14
 - ‘poorman’, 17
 - ‘reshape’, 15
 - ‘reshape2’, 15
 - ‘seplyr’, 16
 - ‘stringr’, 17
 - ‘tibble’, 2, 3, 5
 - ‘tidyr’, 3, 15, 16, 21
 - ‘tidyverse’, 1, 3–5, 9, 10, 13, 15–18, 25
 - ‘wrapr’, 2, 11, 12, 14
- pipe operator, 11
- pipes
 - expressions in rhs, 12
 - tidyverse, 11
 - wrapr, 11–15
- ‘poorman’, 17
- ‘reshape’, 15
- ‘reshape2’, 15
- reshaping tibbles, 15–16
- row-wise operations on data, 17–19
- S, 15
- ‘seplyr’, 16
- ‘stringr’, 17
- tibble
 - differences with data frames, 5–10
- ‘tibble’, 2, 3, 5
- ‘tidyr’, 3, 15, 16, 21
- ‘tidyverse’, 1, 3–5, 9, 10, 13, 15–18, 25
- ‘wrapr’, 2, 11, 12, 14



Alphabetic Index of R Names

`==`, 21
`[,]`, 3, 21
`[[]]`, 3
`%.>%`, 11-14, 18
`%<>%`, 11
`%>%`, 11-14
`%T>%`, 11
`|>`, 11-14

`aggregate()`, 19
`anti_join()`, 24
`arrange()`, 18
`as.data.frame()`, 8
`as_tibble()`, 5
`assign()`, 14

`class()`, 8
`contains()`, 19

`data.frame`, 5
`data.frame()`, 6, 10

`ends_with()`, 19

`filter()`, 18
`full_join()`, 22

`gather()`, 16
`group_by()`, 20, 21

`I()`, 10
`identical()`, 8
`inner_join()`, 22
`iris`, 15
`is_tibble()`, 5

`left_join()`, 22
`list`, 5

`matches()`, 19
`matrix`, 5
`mutate()`, 17

`names()`, 19
`names()<-`, 19

`options()`, 7
`order()`, 18

`pivot_longer()`, 16
`pivot_wider()`, 16
`plot()`, 15
`print()`, 7, 13

`read.table()`, 6
`rename()`, 19
`right_join()`, 22

`select()`, 18, 19
`semi_join()`, 24
`slice()`, 18
`sort()`, 18
`spread()`, 16
`starts_with()`, 19
`str_extract()`, 17
`subset()`, 2, 18
`summarise()`, 20

`tbl`, 5
`tbl_df`, 6
`tibble`, 5, 6, 16
`tibble()`, 5, 6, 9, 10, 17
`transmute()`, 17

`ungroup()`, 20, 21



Index of R Names by Category

R names and symbols grouped into the categories ‘classes and modes’, ‘constant and special values’, ‘control of execution’, ‘data objects’, ‘functions and methods’, ‘names and their scope’, and ‘operators’.

classes and modes	names(), 19
data.frame, 5	names()<-, 19
list, 5	options(), 7
matrix, 5	order(), 18
tbl, 5	pivot_longer(), 16
tbl_df, 6	pivot_wider(), 16
tibble, 5, 6, 16	plot(), 15
data objects	print(), 7, 13
iris, 15	read.table(), 6
functions and methods	rename(), 19
aggregate(), 19	right_join(), 22
anti_join(), 24	select(), 18, 19
arrange(), 18	semi_join(), 24
as.data.frame(), 8	slice(), 18
as_tibble(), 5	sort(), 18
assign(), 14	spread(), 16
class(), 8	starts_with(), 19
contains(), 19	str_extract(), 17
data.frame(), 6, 10	subset(), 2, 18
ends_with(), 19	summarise(), 20
filter(), 18	tibble(), 5, 6, 9, 10, 17
full_join(), 22	transmute(), 17
gather(), 16	ungroup(), 20, 21
group_by(), 20, 21	operators
I(), 10	==, 21
identical(), 8	[,], 3, 21
inner_join(), 22	[[]], 3
is_tibble(), 5	%.>%, 11-14, 18
left_join(), 22	%<>%, 11
matches(), 19	%>%, 11-14
mutate(), 17	%T>%, 11
	>, 11-14

