

Pedro J. Aphalo

Learn R

As a Language

Contents

List of Figures	vii
List of Tables	ix
1 Base R: “Words” and “Sentences”	1
1.1 Aims of this chapter	1
1.2 Natural and computer languages	2
1.3 Numeric values and arithmetic	2
1.4 Character values	14
1.5 Logical values and Boolean algebra	21
1.6 Comparison operators and operations	24
1.7 Sets and set operations	31
1.8 The ‘mode’ and ‘class’ of objects	36
1.9 ‘Type’ conversions	37
1.10 Vector manipulation	40
1.11 Matrices and multidimensional arrays	47
1.12 Factors	55
1.13 Further reading	61
2 R Extensions: Data Wrangling	63
2.1 Aims of this chapter	63
2.2 Introduction	64
2.3 Packages used in this chapter	66
2.4 Replacements for <code>data.frame</code>	66
2.4.1 Package ‘ <code>data.table</code> ’	66
2.4.2 Package ‘ <code>tibble</code> ’	67
2.5 Data pipes	72
2.5.1 ‘ <code>magrittr</code> ’	73
2.5.2 ‘ <code>wrapr</code> ’	73
2.5.3 Comparing pipes	75
2.6 Reshaping with ‘ <code>tidyr</code> ’	77
2.7 Data manipulation with ‘ <code>dplyr</code> ’	79
2.7.1 Row-wise manipulations	79
2.7.2 Group-wise manipulations	82
2.7.3 Joins	84
2.8 Further reading	88
Bibliography	89
General Index	91

Alphabetic Index of R Names	95
Index of R Names by Category	99
Frequently Asked Questions	103

List of Figures



List of Tables



1

Base R: “Words” and “Sentences”

The desire to economize time and mental effort in arithmetical computations, and to eliminate human liability to error, is probably as old as the science of arithmetic itself.

Howard Aiken

Proposed automatic calculating machine, 1937; reprinted 1964

1.1 Aims of this chapter

In my experience, for those not familiar with computer programming languages, the best first step in learning the R language is to use it interactively by typing textual commands at the *console* or command line. This will teach not only the syntax and grammar rules, but also give you a glimpse at the advantages and flexibility of this approach to data analysis.

In the first part of the chapter we will use R to do everyday calculations that should be so easy and familiar that you will not need to think about the operations themselves. This easy start will give you a chance to focus on learning how to issue textual commands at the command prompt.

Later in the chapter, you will gradually need to focus more on the R language and its grammar and less on how commands are entered. By the end of the chapter you will be familiar with most of the kinds of “words” used in the R language and you will be able to write simple “sentences” in R.

Along the chapter, I will occasionally show the equivalent of the R code in mathematical notation. If you are not familiar with the mathematical notation, you can safely ignore it, as long as you understand the R code.

1.2 Natural and computer languages

Computer languages have strict rules and interpreters and compilers are unforgiving about errors. They will issue error messages, but in contrast to human readers or listeners, will not guess your intentions and continue. However, computer languages have a much smaller set of words than natural languages, such as English. If you are new to computer programming, understanding the parallels between computer and natural languages may be useful.


One can think of constant values and variables (values stored under a name) as nouns and of operators and functions as verbs. A complete command, or statement, is the equivalent of a natural language sentence: “a comprehensible utterance.” The simple statement `a + 1` has three components: `a`, a variable, `+`, an operator and `1` a constant. The statement `sqrt(4)` has two components, a function `sqrt()` and a numerical constant `4`. We say that “to compute $\sqrt{4}$ we *call* `sqrt()` with `4` as its *argument*.”

Although all values manipulated in a digital computer are stored as *bits* in memory, multiple interpretations are possible. Numbers, letters, logical values, etc., can be encoded into bits and decoded as long as their type or *mode* is known. The concept of `class` is not directly related to how values are encoded when stored in computer memory, but instead their interpretation as part of a computer program. We can have, for example, RGB color values, stored as three numbers such as `0, 0, 255`, as hexadecimal numbers stored as characters `#0000FF`, or even use fancy names stored as character strings like `"blue"`. We could create a `class` for colors using any of these representations, based on two different modes: `numeric` and `character`.

In this chapter we will focus on individual program statements, the equivalent of sentences in natural language. In later chapters you will learn how to combine them to create compound statements, the equivalent of natural-language paragraphs, and scripts, the equivalent of essays. You will also learn how to define new verbs, user-defined functions and operators, and new nouns, user-defined classes.

1.3 Numeric values and arithmetic

When working in R with arithmetic expressions, the normal mathematical precedence rules are respected, but parentheses can be used to alter this order. Parentheses can be nested, but in contrast to the usual practice in mathematics, the same parenthesis symbol is used at all nesting levels.

 Both in mathematics and programming languages *operator precedence rules* determine which subexpressions are evaluated first and which later. Contrary to primitive electronic calculators, R evaluates numeric expressions containing operators according to the rules of mathematics. In the expression `3 + 2 × 3`, the product `2 × 3` has precedence over the addition, and is evaluated first, yielding

as the result of the whole expression, 9. In programming languages, similar rules apply to all operators, even those taking as operands non-numeric values.

It is important to keep in mind that in R trigonometric functions interpret numeric values representing angles as being expressed in radians.

The equivalent of the math expression

$$\frac{3 + e^2}{\sin \pi}$$

is, in R, written as follows:

```
(3 + exp(2)) / sin(pi)
## [1] 8.483588e+16
```

It can be seen above that mathematical constants and functions are part of the R language. One thing to remember when translating complex fractions as above into R code, is that in arithmetic expressions the bar of the fraction generates a grouping that alters the normal precedence of operations. In contrast, in an R expression this grouping must be explicitly signaled with additional parentheses.

If you are in doubt about how precedence rules work, you can add parentheses to make sure the order of computations is the one you intend. Redundant parentheses have no effect.

```
1 + 2 * 3
## [1] 7

1 + (2 * 3)
## [1] 7

(1 + 2) * 3
## [1] 9
```

The number of opening (left side) and closing (right side) parentheses must be balanced, and they must be located so that each enclosed term is a valid mathematical expression, i.e., code that can be evaluated to return a value, a value that can be inserted in place of the expression enclosed in parenthesis before evaluating the remaining of the expression. For example, $(1 + 2) * 3$ after evaluating $(1 + 2)$ becomes $3 * 3$ yielding 9. In contrast, $(1 +) 2 * 3$ is a syntax error as $1 +$ is incomplete and does not yield a number.



Here results are not shown. These are examples for you to type at the command prompt. In general you should not skip them, as in many cases, as with the statements highlighted with comments in the code chunk below, they have something to teach or demonstrate. You are strongly encouraged to *play*, in other words, create new variations of the examples and execute them to explore how R works.

```

1 + 1
2 * 2
2 + 10 / 5
(2 + 10) / 5
10^2 + 1
sqrt(9)
pi # whole precision not shown when printing
print(pi, digits = 22)
sin(pi) # oops! Read on for explanation.
log(100)
log10(100)
log2(8)
exp(1)

```

Variables are used to store values. After we *assign* a value to a variable, we can use in our code the name of the variable in place of the stored value. The “usual” assignment operator is `<-`. In R, all names, including variable names, are case sensitive. Variables `a` and `A` are two different variables. Variable names can be long in R although it is not a good idea to use very long names. Here I am using very short names, something that is usually also a very bad idea. However, in the examples in this chapter where the stored values have no connection to the real world, simple names emphasize their abstract nature. In the chunk below, `a` and `b` are arbitrarily chosen variable names; I could have used names like `my.variable.a` or `outside.temperature` if they had been useful to convey information.

```

a <- 1
a + 1
## [1] 2

a
## [1] 1

b <- 10
b <- a + b
b
## [1] 11

3e-2 * 2.0
## [1] 0.06

```

Entering the name of a variable *at the R console* implicitly calls function `print()` displaying the stored value on the console. The same applies to any other statement entered *at the R console*: `print()` is implicitly called with the result of executing the statement as its argument.

```

a
## [1] 1

print(a)
## [1] 1

a + 1
## [1] 2

print(a + 1)
## [1] 2

```



There are some syntactically legal assignment statements that are not very frequently used, but you should be aware that they are valid, as they will not trigger error messages, and may surprise you. The most important thing is to write code consistently. The “backwards” assignment operator `->` and resulting code like `1 -> a` are valid but less frequently used. The use of the equals sign (`=`) for assignment in place of `<=` although valid is discouraged. Chaining assignments as in the first statement below can be used to signal to the human reader that `a`, `b` and `c` are being assigned the same value.

```
a <- b <- c <- 0.0
a
b
c
1 -> a
a
a = 3
a
```



In R, all numbers belong to mode `numeric` (we will discuss the concepts of *mode* and *class* in section 1.8 on page 36). We can query if the mode of an object is `numeric` with function `is.numeric()`.

```
mode(1)
## [1] "numeric"

a <- 1
is.numeric(a)
## [1] TRUE
```

Because numbers can be stored in different formats, most computing languages implement several different types of numbers. In most cases R's `numeric()` values can be used everywhere that a number is expected. However, in some cases it has advantages to explicitly indicate that we will store or operate on whole numbers, in which case we can use class `integer`, with integer constants indicated by a trailing capital “L,” as in `32L`.

```
is.numeric(1L)
## [1] TRUE

is.integer(1L)
## [1] TRUE

is.double(1L)
## [1] FALSE
```

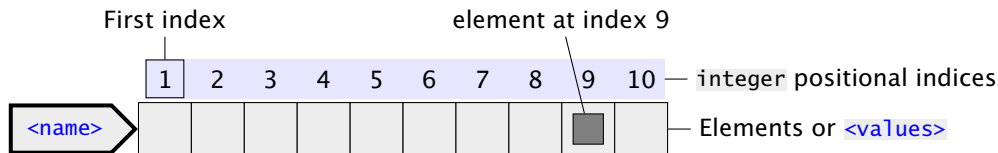
Real numbers are a mathematical abstraction, and do not have an exact equivalent in computers. Instead of Real numbers, computers store and operate on numbers that are restricted to a broad but finite range of values and have a finite resolution. They are called, *floats* (or *floating-point* numbers); in R they go by the name of `double` and can be created with the constructor `double()`.

```
is.numeric(1)
## [1] TRUE

is.integer(1)
## [1] FALSE

is.double(1)
## [1] TRUE
```

R’s vectors are one-dimensional, of varying length and used to store similar values, e.g., numbers. They are different to the vectors, commonly used in Physics when describing directional forces, which are symbolized with an arrow as an “accent,” such as \vec{F} . In R numeric values and other atomic values are always **vectors** that can contain zero, one or more elements. The diagram below exemplifies a vector containing ten elements, also called members. These elements can be extracted using integer numbers as positional indices, and manipulated as described in more detail in section 1.12 on page 55.



Vectors, in mathematical notation, are similarly represented using positional indexes as subscripts,

$$a_{1\dots n} = a_1, a_2, \dots, a_i, \dots, a_n, \quad (1.1)$$

where $a_{1\dots n}$ is the whole vector and a_1 its first member. The length of $a_{1\dots n}$ is n as it contains n members. In the diagram above $n = 10$.

As you have seen above, the results of calculations were printed preceded with `[1]`. This is the index or position in the vector of the first number (or other value) displayed at the head of the current line. As single values are vectors of length one, when they are printed, they are also preceded with `[1]`.

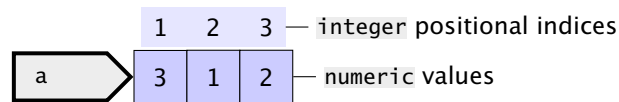
One can use `c()` “concatenate” to create a vector from other vectors, including vectors of length 1, or even vectors of length 0, such as the `numeric` constants in the statements below. The first example shows an anonymous vector created, printed, and then automatically discarded.

```
c(3, 1, 2)
## [1] 3 1 2
```

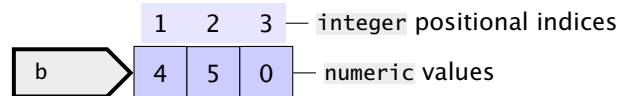
To be able to reuse the vector, we assign it to a variable, giving a name to it. The length of a vector can be queried with method `length()`. We show below R code followed by a diagram depicting the vector created.

```
a <- c(3, 1, 2)
length(a)
## [1] 3

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



```
b <- c(4, 5, 0)
b
## [1] 4 5 0
```



```
c <- c(a, b)
c
## [1] 3 1 2 4 5 0
```



```
d <- c(b, a)
d
## [1] 4 5 0 3 1 2
```



As shown earlier, values can be also printed at the R console. Here we show concatenation with a vector of the same class but with length zero.

```
c(d, numeric())
## [1] 4 5 0 3 1 2
```

Method `c()` accepts as arguments two or more vectors and concatenates them, one after another. Quite frequently we may need to insert one vector in the middle of another. For this operation, `c()` is not useful by itself. One could use indexing combined with `c()`, but this is not needed as R provides a function capable of directly doing this operation. Although it can be used to “insert” values, it is named `append()`, and by default, it indeed appends one vector at the end of another.

```
append(a, b)
## [1] 3 1 2 4 5 0
```

The output above is the same as for `c(a, b)`, however, `append()` accepts as an argument an index position after which to “append” its second argument. This results in an *insert* operation when the index points at any position different from the end of the vector.

```
append(a, values = b, after = 2L)
## [1] 3 1 4 5 0 2
```

Both `c()` and `append()` can also be used with lists (described in section ?? on page ??).



One can create sequences using function `seq()` or the operator `:`, or repeat values using function `rep()`. In this case, I leave to the reader to work out the rules by running these and his/her own examples, with the help of the documentation, available through `help(seq)` and `help(rep)`.

```
a <- -1:5
a
b <- 5:-1
b
c <- seq(from = -1, to = 1, by = 0.1)
c
d <- rep(-5, 4)
d
```

Next, something that makes R different from most other programming languages: vectorized arithmetic. Operators and functions that are vectorized accept, as arguments, vectors of arbitrary length, in which case the result returned is equivalent to having applied the same function or operator individually to each element of the vector.

```
a + 1 # we add one to vector a defined above
## [1] 4 2 3

(a + 1) * 2
## [1] 8 4 6

a + b
## [1] 7 6 2

a - a
## [1] 0 0 0
```

As it can be seen in the first line above, another peculiarity of R, is what is frequently called “recycling” of arguments: as vector `a` is of length 6, but the constant 1 is a vector of length 1, this short constant vector is extended, by recycling its value, into a vector of six ones—i.e., a vector of the same length as the longest vector in the statement, `a`.


Make sure you understand what calculations are taking place in the chunk above, and also the one below.

```
a <- rep(1, 6)
a
## [1] 1 1 1 1 1 1

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

```
a + 1:3
## [1] 2 3 4 2 3 4

a + 1:4
## Warning in a + 1:4: longer object length is not a multiple of shorter object
length
## [1] 2 3 4 5 2 3
```

 As mentioned above, a vector can have a length of zero or more member values. Vectors of length zero may seem at first sight quite useless, but in fact they are very useful. They allow the handling of “no input” or “nothing to do” cases as normal cases, which in the absence of vectors of length zero would require to be treated as special cases. Constructors for R classes like `numeric()` return vectors of a length given by their first argument, which defaults to zero. I describe here a useful function, `length()` which returns the length of a vector or list.

```
z <- numeric(0)
z
## numeric(0)

length(z)
## [1] 0
```

```
z1 <- numeric()
z1
## numeric(0)

z2 <- numeric(length = 0)
z2
## numeric(0)
```

Vectors (and lists) of length zero, behave in most cases, as expected—e.g., they can be concatenated as shown here.

```
length(c(a, numeric(0), b))
## [1] 9

length(c(a, b))
## [1] 9
```

Many functions, such as R’s maths functions and operators, will accept numeric vectors of length zero as valid input, returning also a vector of length zero, issuing neither a warning nor an error message. In other words, *these are valid operations* in R.

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

5 + numeric(0)
## numeric(0)
```


Even when of length zero, vectors do have to belong to a class acceptable for the operation: `5 + character(0)` is an error.

Passing as argument to parameter `length` a value larger than zero creates a longer vector filled with zeros in the case of `numeric()`.

```
numeric(5)
## [1] 0 0 0 0 0
```

The length of a vector can be explicitly increased, with missing values filled automatically with `NA`, the marker for not available.

```
z <- 1:5
z
## [1] 1 2 3 4 5

length(z) <- 10
z
## [1] 1 2 3 4 5 NA NA NA NA NA
```

If the length is decreased, the values in the *tail* of the vector are discarded.

```
z <- 1:10
z
## [1] 1 2 3 4 5 6 7 8 9 10

length(z) <- 5
z
## [1] 1 2 3 4 5
```

It is possible to *remove* variables from the workspace with `rm()`. Function `ls()` returns a *list* of all objects visible in the current environment, or by supplying a `pattern` argument, only the objects with names matching the `pattern`. The pattern is given as a regular expression (see section ?? on page ??), with `[]` enclosing alternative matching characters, `^` and `$`, indicating the extremes of the name (start and end, respectively). For example, `"^z$"` matches only the single character ‘z’ while `"^z"` matches any name starting with ‘z’. In contrast `"^[zy]$"` matches both ‘z’ and ‘y’ but neither ‘zy’ nor ‘yz’, and `"^[a-z]"` matches any name starting with a lowercase ASCII letter. If you are using RStudio, all objects are listed in the Environment pane, and the search box of the panel can be used to find a given object.

```
ls(pattern="^z$")
## [1] "z"

rm(z)
ls(pattern="^z$")
## character(0)
```

There are some special values available for numbers. `NA` meaning “not available” is used for missing values. Calculations can also yield the following values `NaN` “not a number”, `Inf` and `-Inf` for ∞ and $-\infty$. As you will see below, calculations yielding

these values do **not** trigger errors or warnings, as they are arithmetically valid. `Inf` and `-Inf` are also valid numerical values for input and constants.

```
a <- NA
a
## [1] NA

-1 / 0
## [1] -Inf

1 / 0
## [1] Inf

Inf / Inf
## [1] NaN

Inf + 4
## [1] Inf

b <- -Inf
b * -1
## [1] Inf
```

Not available (NA) values are very important in the analysis of experimental data, as frequently some observations are missing from an otherwise complete data set due to “accidents” during the course of an experiment. It is important to understand how to interpret NA's. They are simple placeholders for something that is unavailable, in other words, *unknown*.

```
A <- NA
A
## [1] NA

A + 1
## [1] NA

A + Inf
## [1] NA
```



When to use vectors of length zero, and when NAs? Make sure you understand the logic behind the different behavior of functions and operators with respect to NA and `numeric()` or its equivalent `numeric(0)`. What do they represent? Why NA's are not ignored, while vectors of length zero are?

```
123 + numeric()
123 + NA
```

Model answer: NA is used to signal a value that “was lost” or “was expected” but is unavailable because of some accident. A vector of length zero, represents no values, but within the normal expectations. In particular, if vectors are expected to have a certain length, or if index positions along a vector are meaningful, then using NA is a must.

Any operation, even tests of equality, involving one or more NA's return an NA.

In other words, when one input to a calculation is unknown, the result of the calculation is unknown. This means that a special function is needed for testing for the presence of `NA` values.

```
is.na(c(NA, 1))
## [1] TRUE FALSE
```

In the example above, we can also see that `is.na()` is vectorized, and that it applies the test to each of the two elements of the vector individually, returning the result as a logical vector of length two.

One thing to be aware of are the consequences of the fact that numbers in computers are almost always stored with finite precision and/or range: the expectations derived from the mathematical definition of Real numbers are not always fulfilled. See the box on page 27 for an in-depth explanation.

```
1 - 1e-20
## [1] 1
```

When comparing `integer` values these problems do not exist, as integer arithmetic is not affected by loss of precision in calculations restricted to integers. Because of the way integers are stored in the memory of computers, within the representable range, they are stored exactly. One can think of computer integers as a subset of whole numbers restricted to a certain range of values.

```
1L + 3L
## [1] 4

1L * 3L
## [1] 3

1L %% 3L
## [1] 0

1L %/% 3L
## [1] 1

1L / 3L
## [1] 0.3333333
```

The last statement in the example immediately above, using the “usual” division operator yields a floating-point `double` result, while the integer division operator `%/%` yields an `integer` result, and `%%` returns the remainder from the integer division. If as a result of an operation the result falls outside the range of representable values, the returned value is `NA`.

```
1000000L * 1000000L

## warning in 1000000L * 1000000L: NAs produced by integer overflow
## [1] NA
```

Both doubles and integers are considered numeric. In most situations, conversion is automatic and we do not need to worry about the differences between these two types of numeric values. The next chunk shows returned values that are either `TRUE` or `FALSE`. These are `logical` values that will be discussed in the next section.

```
is.numeric(1L)
## [1] TRUE

is.integer(1L)
## [1] TRUE

is.double(1L)
## [1] FALSE

is.double(1L / 3L)
## [1] TRUE

is.numeric(1L / 3L)
## [1] TRUE
```



Study the variations of the previous example shown below, and explain why the two statements return different values. Hint: 1 is a `double` constant. You can use `is.integer()` and `is.double()` in your explorations.

```
1 * 1000000L * 1000000L
1000000L * 1000000L * 1
```

Both when displaying numbers or as part of computations, we may want to decrease the number of significant digits or the number of digits after the decimal marker. Be aware that in the examples below, even if printing is being done by default, these functions return `numeric` values that are different from their input and can be stored and used in computations. Function `round()` is used to round numbers to a certain number of decimal places after or before the decimal marker, while `signif()` rounds to the requested number of significant digits.

```
round(0.0124567, digits = 3)
## [1] 0.012

signif(0.0124567, digits = 3)
## [1] 0.0125

round(1789.1234, digits = 3)
## [1] 1789.123


signif(1789.1234, digits = 3)
## [1] 1790

round(1789.1234, digits = -1)
## [1] 1790

a <- 0.12345
b <- round(a, digits = 2)
a == b
## [1] FALSE

a - b
## [1] 0.00345


b
## [1] 0.12
```

 Being `digits`, the second parameter of these functions, the argument can also be passed by position. However, code is usually easier to understand for humans when parameter names are made explicit.

```
round(0.0124567, digits = 3)
## [1] 0.012

round(0.0124567, 3)
## [1] 0.012
```

Functions `trunc()` and `ceiling()` return the non-fractional part of a numeric value as a new numeric value. They differ in how they handle negative values, and neither of them rounds the returned value to the nearest whole number.

 What does value truncation mean? Function `trunc()` truncates a numeric value, but it does not return an `integer`.

- Explore how `trunc()` and `ceiling()` differ. Test them both with positive and negative values.
- **Advanced** Use function `abs()` and operators `+` and `-` to reproduce the output of `trunc()` and `ceiling()` for the different inputs.
- Can `trunc()` and `ceiling()` be considered type conversion functions in R?


1.4 Character values

Character variables can be used to store any character. Character constants are written by enclosing characters in quotes. There are three types of quotes in the ASCII character set, double quotes `"`, single quotes `'`, and back ticks ```. The first two types of quotes can be used as delimiters of `character` constants.

```
a <- "A"
a
## [1] "A"

b <- 'A'
b
## [1] "A"

a == b
## [1] TRUE
```

 In many computer languages, vectors of characters are distinct from vectors of character strings. In these languages, character vectors store at each index position a single character, while vectors of character strings store at each index

position strings of characters of various lengths, such as words or sentences. If you are familiar with C or C++, you need to keep in mind that C's `char` and R's `character` are not equivalent and that in R, `character` vectors are vectors of character strings. In contrast to these other languages, in R there is no predefined class for vectors of individual characters and character constants enclosed in double or single quotes are not different.

Concatenating character vectors of length one does not yield a longer character string, it yields instead a longer vector.

```
a <- 'A'
b <- "bcdefg"
c <- "123"
d <- c(a, b, c)
d
## [1] "A"      "bcdefg" "123"
```

Having two different delimiters available makes it possible to choose the type of quotes used as delimiters so that other quotes can be included in a string.

```
a <- "He said 'hello' when he came in"
a
## [1] "He said 'hello' when he came in"

b <- 'He said "hello" when he came in'
b
## [1] "He said \"hello\" when he came in"
```

The outer quotes are not part of the string, they are “delimiters” used to mark the boundaries. As you can see when `b` is printed special characters can be represented using “escape sequences”. There are several of them, and here we will show just four, new line (`\n`) and tab (`\t`), `\` the escape code for a quotation mark within a string and `\\` the escape code for a single backslash `\`. We also show here the different behavior of `print()` and `cat()`, with `cat()` *interpreting* the escape sequences and `print()` displaying them as entered.

```
c <- "abc\\ndef\\tx\\"yz\\"\\\\"tm"
print(c)
## [1] "abc\\ndef\\tx\\"yz\\"\\\\"tm"

cat(c)
## abc
## def x"yz" \ m
```

The *escape codes* work only in some contexts, as when using `cat()` to generate the output. For example, the new-line escape (`\n`) can be embedded in strings used for axis-label, title or label in a plot to split them over two or more lines.

Function `length()` applied to a `character` vector returns the number of member strings, not the number of characters in each member string. To query the “length” of individual character strings expressed as number of characters, we use function `nchar()`.

```
nchar(x = "abracadabra")
## [1] 11

nchar(x = c("abracadabra", "workaholic"))
## [1] 11 10
```

Function `trimstr()` trims a string a maximum number of characters or width.

```
strtrim(x = "abracadabra", width = 6)
## [1] "abraca"

strtrim(x = "abra", width = 6)
## [1] "abra"

strtrim(x = c("abracadabra", "workaholic"), 6)
## [1] "abraca" "workah"

strtrim(x = c("abracadabra", "workaholic"), c(6, 3))
## [1] "abraca" "wor"
```

Pasting together `character` strings has many uses, e.g., assembling informative messages to be printed, programmatically creating file names or file paths, etc. If we pass numbers, they are converted to `character` before pasting. The default separator is a space character, but this can be changed by passing a `character` string as argument for `sep`.

```
paste("n =", 3)
## [1] "n = 3"

paste("n", 3, sep = " = ")
## [1] "n = 3"
```

Pasting constants, as shown above, is of little practical use. In contrast, combining values stored in different variables is a very frequent operation when working with data. A simple use example follows. Assuming vector `friends` contains the names of friends and vector `fruits` the fruits they like to eat we can paste these values together into short sentences.

```
friends <- c("John", "Yan", "Juana", "Mary")
fruits <- c("apples", "lichees", "oranges", "strawberries")
paste(friends, "likes to eat ", fruits, ".", sep = "")
## [1] "Johnlikes to eat apples." "Yanlikes to eat lichees."
## [3] "Juanalikes to eat oranges." "Marylikes to eat strawberries."
```



Why was necessary to pass `sep = ""` in the call to `paste()` in the example above? First try to predict what will happen and then remove `, sep = ""` it from the statement and run it to learn the correct answer. Try your own variations of the code until you understand the role of the separator string.

We can pass an additional argument to tell that the vector resulting from the paste operation is to be collapsed into a single `character` string. The argument passed to collapse is used as the separator.

```
paste(friends, "likes to eat", fruits, collapse = ". ")
## [1] "John likes to eat apples. Yan likes to eat liches. Juana likes to eat oranges. Mary likes to eat strawber
```

When the vectors are of different length, the shorter one is reused as many times as needed, which as in this example, is not be what we want. This can be solved by nesting two calls to `paste`, one of them using `collapse`.

```
paste("My friends like to eat", paste(fruits, collapse = ", "), "and other fruits.")
## [1] "My friends like to eat apples, liches, oranges, strawberries and other fruits."
```

Trimming leading and trailing white space is a frequent operation, and R function `trimws()` implements this operation.

```
trimws(x = " two words ")
## [1] "two words"

trimws(x = c(" eight words and a newline at the end\n", " two words "))
## [1] "eight words and a newline at the end"
## [2] "two words"
```



Function `trimws()` has additional parameters that make it possible to select which end of the string is trimmed and which characters are considered whitespace. Use `help(trimws)` to access the help and study this documentation. Modify the example above so that only trailing white space is removed, and so that the newline character `\n` is not considered whitespace, and thus not trimmed away.

Within character strings, substrings can be extracted and replaced by position using `substring()` or `substr()`.

For extraction we can pass to `x` a constant as shown below or a variable.

```
substr(x = "abracadabra", start = 5, stop = 9)
## [1] "cadab"


substr(x = c("abracadabra", "workaholic"), start = 5, stop = 11)
## [1] "cadabra" "aholic"
```

Substitution is done *in place*, i.e., by reference, and the argument passed to `x` must be always variable. This is a pure substitution, not insertion, so the length of the string passed as argument to `x` remains unchanged.

```
my.text <- c("abracadabra", "workaholic")
substr(x = my.text, start = 5, stop = 9) <- "xxx"
my.text
## [1] "abraxxabra" "workxxxlic"
```

If we pass values to both `start` and `stop` then only part of the value on the *rhs* of the assignment operator `<=` may be used.

```
my.text <- c("abracadabra", "workaholic")
substr(x = my.text, start = 5, stop = 6) <- "xxx"
my.text
## [1] "abraxxdabra" "workxxolic"
```


 Frequently, a very effective way of learning how a function behaves, is to experiment. In the example below, we set `start` and `stop` delimiting more characters than those in `"xxx"`. In this case, is `"xxx"` extended, or `start` or `stop` ignored? Run this "toy example" to find out the answer.

```
my.text <- c("abracadabra", "workaholic")
substr(x = my.text, start = 5, stop = 11) <- "xxx"
my.text
## [1] "abrxxxabra" "workxxxlic"
```

As in R each character value is a string comprised by zero to many characters, in addition to comparisons based on whole strings or values, partial matches among them are of interest.


To replacing or substitute part of a character string, we can use functions `sub()` or `gsub()`. The first example uses three character constants, but values stored in variables can also be passed as arguments.

```
sub(pattern = "ab", replacement = "AB", x = "about")
## [1] "ABout"
```

The difference between `sub()` (substitution) and `gsub()` (global substitution) is that the first replaces only the first match found while the second replaces all matches.

```
sub(pattern = "ab", replacement = "x", x = "abracadabra")
## [1] "xracadabra"

gsub(pattern = "ab", replacement = "x", x = "abracadabra")
## [1] "xracadxra"
```

 Functions `sub()` or `gsub()` accept character vectors as argument for parameter `x`. Run the two statements below and study how the values returned differ.

```
sub(pattern = "ab", replacement = "x", x = c("abra", "cadabra"))
gsub(pattern = "ab", replacement = "x", x = c("abra", "cadabra"))
```

Function `grep()` returns indices to the values in a vector matching a pattern, or alternatively, the matching values themselves.

```
grep(pattern = "C", x = c("R", "C++", "C", "Perl", "Pascal"))
## [1] 2 3

grep(pattern = "C", x = c("R", "C++", "C", "Perl", "Pascal"), value = TRUE)
## [1] "C++" "C"

grep(pattern = "C", x = c("R", "C++", "C", "Perl", "Pascal"), ignore.case = TRUE)
## [1] 2 3 5
```


Function `grep1()` is a variation of `grep()` that returns a vector of logical values instead of numeric indices to the matching values in `x`.

```
grepl(pattern = "C", x = c("R", "C++", "C", "Perl", "Pascal"))
## [1] FALSE TRUE TRUE FALSE FALSE

grepl(pattern = "C", x = c("R", "C++", "C", "Perl", "Pascal"), ignore.case = TRUE)
## [1] FALSE TRUE TRUE FALSE TRUE
```

In the examples above we passed as arguments for `pattern` strings that matched exactly their targets. In R and other languages *regular expressions* are used to concisely describe more elaborate and conditional patterns. Regular expressions themselves are encoded as character strings, where some characters and character sequences have special meaning. This means that when a pattern should be interpreted literally rather than specially, `fixed = TRUE` should be passed in the call. This in addition, ensures faster computation. In the examples above, the patterns used contained no characters with special meaning, thus, the returned value is not affected by passing `fixed = TRUE` as done here.

```
sub(pattern = "ab", replacement = "AB", x = "about", fixed = TRUE)
## [1] "ABout"
```

 Regular expressions are used in Unix and Linux shell scripts and programs, and are part of Perl, C++ and other languages in addition to R. This means that variations exist on the same idea, with R supporting two variations of the syntax. A description of R regular expressions can be accessed with `help(regex)`. We here describe R's default syntax.

Regular expressions are concise, terse and extremely powerful. They are a language in themselves. However, the effort needed to learn their use more than pays back. I will show examples of the use, rather than systematically describe them. I will use `gsub()` for these examples, but several other R functions accept regular expressions as patterns.

Within a regular expression `|` separates alternative matching patterns.

```
gsub(pattern = "ab|t", replacement = "123", x = "about")
## [1] "123ou123"
```

Within a regular expression we can group characters within `[]` as alternative, e.g. `[0123456789]`, or `[0-9]` matches any digit.

```
gsub(pattern = "a[0123456789]",
      replacement = "ab",
      x = c("a1out", "a9out", "a3out"))
## [1] "about" "about" "about"
```

Character `^` indicates that the match must be at the “head” of the string, and `$` that the match should be at its “tail”.

```
gsub(pattern = "^a[0123456789]",
      replacement = "ab",
      x = c("a1out", "a9out", "a3out"))
## [1] "about" "about" "a3out"
```

The replacement can be an empty string.

```
gsub(pattern = "out$",
      replacement = "",
      x = c("about", "a9out", "a3outx"))
## [1] "ab"      "a9"      "a3outx"
```

A dot (.) matches any character. In this example we replace the last character with "".

```
gsub(pattern = ".$",
      replacement = "",
      x = c("about", "a9out", "a3outx"))
## [1] "abou" "a9ou" "a3out"
```



How would you modify the last code example to edit `c("about", "axout", "a3outx")` into `c("about", "axout", "a3out")`? Think of different ways of doing this using regular expressions.

The number of matching characters can be indicated with + (match 1 or more times), ? (match 0 or 1 times), * (match 0 or more times) or even numerically. Matching is in most cases “greedy”.

```
gsub(pattern = "^.[0-9][a-z]*$",
      replacement = "gone",
      x = c("about", "a9out", "a3outx"))
## [1] "about" "gone"  "gone"
```

With parentheses we can isolate part of the matched string and reuse it in the replacement with a numeric backreference. Up to a maximum of nine pairs of parentheses can be used.

```
gsub(pattern = "^.[(0-9)][a-z]*$",
      replacement = "gone with \\1",
      x = c("about", "a9out", "a3outx"))
## [1] "about"      "gone with 9" "gone with 3"
```

Several named classes of characters are predefined, for example `[:lower:]` for lower case alphabetic characters according to the current locale.



Run the two statements below, study the returned values by creating variations of the patterns and explain why the returned values differ.

```
gsub(pattern = "^.+$$",
      replacement = "",
      x = c("about", "a9out", "a3outx"))
gsub(pattern = "^..?$$",
      replacement = "",
      x = c("about", "a9out", "a3outx"))
```

Splitting of character strings based on pattern matching is a frequently used


operation, e.g., treatment labels containing information about two different treatment factors need to be split into their components before data analysis. Function `strsplit()` has an interface consistent with `grep()`. In the examples we will split strings containing date and time of day information in different ways.

```
strsplit(x = "2023-07-29 10:30", split = " ")
## [[1]]
## [1] "2023-07-29" "10:30"
```


Using a simple regular expression we can extract individual strings representing the numbers.

```
strsplit(x = "2023-07-29 10:30", split = "|-|:")
## [[1]]
## [1] "2023" "07" "29" "10" "30"
```

The argument to `split` is by default interpreted as a regular expression, but as discussed above we can pass `fixed = TRUE` to prevent this.

 One needs to be aware that the part of the string matched by the regular expression is not included in the returned vectors. If the regular expression matches more than what we consider a separator, the returned values may be surprising.

```
strsplit(x = "2023-07-29", split = "-[0-9]+$")
## [[1]]
## [1] "2023-07"
```

 When the argument passed to `x` is a vector with multiple member strings, the returned value is a list of character vectors, with one member vector for each member `x`. (Lists are described in section ?? on page ??.)

```
strsplit(x = c("2023-07-29 10:30", "2023-07-29 19:17"), split = " ")
## [[1]]
## [1] "2023-07-29" "10:30"
##
## [[2]]
## [1] "2023-07-29" "19:17"
```

1.5 Logical values and Boolean algebra

What in Mathematics are usually called Boolean values, are called `logical` values in R. They can have only two values `TRUE` and `FALSE`, in addition to `NA` (not available). Logical values `TRUE` and `FALSE` should not be confused with text strings, they are names for the two conditions that can be stored. Logical values are always vectors as all other atomic types in R (by *atomic* we mean that each value is not composed of “parts”).

Logical values are used mostly to keep track of binary conditions, like results from comparisons, and operate on them. In mathematics, Boolean algebra provides the rules of the logic used to combine multiple logical values. Boolean operators like AND and OR take as operands logical values and return a logical value as a result

In R there are two “families” of Boolean operators, vectorized and not vectorized. Vectorized operators accept logical vectors of any length as operands, while non vectorized ones accept only logical vectors of length one as operands. In the chunk below we use non-vectorized operators with two `logical` vectors of length one, `a` and `b`, as operands.

```
a <- TRUE
b <- FALSE
mode(a)
## [1] "logical"

a
## [1] TRUE

!a # negation
## [1] FALSE

a && b # logical AND
## [1] FALSE

a || b # logical OR
## [1] TRUE

xor(a, b) # exclusive OR
## [1] TRUE
```

The availability of two kinds of logical operators is one of the most troublesome aspects of the R language for beginners. Pairs of “equivalent” logical operators behave differently, use similar syntax and use similar symbols! The vectorized operators have single-character names `&` and `|`, while the non-vectorized ones have double-character names `&&` and `||`. There is only one version of the negation operator `!` that is vectorized. In recent versions of R, an error is triggered when a non-vectorized operator is used with a vector with length > 1 , which helps prevent mistakes.

```
a <- c(TRUE, FALSE)
b <- c(TRUE, TRUE)
a
## [1] TRUE FALSE

b
## [1] TRUE TRUE

a & b # vectorized AND
## [1] TRUE FALSE

a | b # vectorized OR
## [1] TRUE TRUE
```

Functions `any()` and `all()` take zero or more logical vectors as their arguments, and return a single logical value “summarizing” the logical values in the vectors. Function `all()` returns `TRUE` only if all values in the vectors passed as arguments are `TRUE`, and `any()` returns `TRUE` unless all values in the vectors are `FALSE`.

```
any(a)
## [1] TRUE

all(a)
## [1] FALSE

any(a & b)
## [1] TRUE

all(a & b)
## [1] FALSE
```

Another important thing to know about logical operators is that they “short-cut” evaluation. If the result is known from the first part of the statement, the rest of the statement is not evaluated. Try to understand what happens when you enter the following commands. Short-cut evaluation is useful, as the first condition can be used as a guard protecting a later condition from being evaluated when it would trigger an error.

```
TRUE || NA
## [1] TRUE

FALSE || NA
## [1] NA

TRUE && NA
## [1] NA

FALSE && NA
## [1] FALSE

TRUE && FALSE && NA
## [1] FALSE


TRUE && TRUE && NA
## [1] NA
```

When using the vectorized operators on vectors of length greater than one, ‘short-cut’ evaluation still applies for the result obtained at each index position.


```
a & b & NA
## [1] NA FALSE

a & b & c(NA, NA)
## [1] NA FALSE

a | b | c(NA, NA)
## [1] TRUE TRUE
```

 Based on the description of “recycling” presented on page 8 for `numeric` operators, explore how “recycling” works with vectorized logical operators. Create logical vectors of different lengths (including length one) and *play* by writing several code statements with operations on them. To get you started, one example is given below. Execute this example, and then create and run your own, making sure that you understand why the values returned are what they are. Sometimes, you will need to devise several examples or test cases to tease out of R an understanding of how a certain feature of the language works, so do not give up early, and make use of your imagination!

```
x <- c(TRUE, FALSE, TRUE, NA)
x & FALSE
x | c(TRUE, FALSE)
```

 **How to check if an entire vector contains no values other than NA (or NaN) values?**

```
na.vec <- rep(NA, 5)
all(is.na(na.vec))
## [1] TRUE
```

 **How to check if a vector contains one or more NA (or NaN) values?**

```
na.vec <- rep(NA, 5)
any(is.na(na.vec))
## [1] TRUE
```

1.6 Comparison operators and operations

Comparison operators return vectors of `logical` values (see section 1.5 on page 21), with values `TRUE` or `FALSE` depending on the outcome.

Equality (`==`) and inequality (`!=`) operators are defined not only for `numeric` values but also for `character` and most other atomic and many other values.

```
# be aware that we use two = symbols
"abc" == "ab"
## [1] FALSE

"ABC" == "abc"
## [1] FALSE

"abc" != "ab"
## [1] TRUE

"ABC" != "abc"
## [1] TRUE
```

In the case of `numeric` values additional comparisons are meaningful and additional operators are defined.

```
1.2 > 1.0
## [1] TRUE

1.2 >= 1.0
## [1] TRUE

1.2 == 1.0
## [1] FALSE

1.2 != 1.0
## [1] TRUE

1.2 <= 1.0
## [1] FALSE

1.2 < 1.0
## [1] FALSE

a <- 20
```

These operators can be used on vectors of any length, returning as a result a logical vector as long as the longest operand. In other words, they behave in the same way as the arithmetic operators described on page 8: their arguments are recycled when needed. Hint: if you do not know what to expect as a value for the vector returned by `1:10`, execute the statement `print(a)` after the first code statement below, or, alternatively, `1:10` without saving the result to a variable.

```
a <- 1:10
a > 5
## [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE

a < 5
## [1] TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE

a == 5
## [1] FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE

all(a > 5)
## [1] FALSE

any(a > 5)
## [1] TRUE

b <- a > 5
b
## [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE

any(b)
## [1] TRUE

all(b)
## [1] FALSE
```

Precedence rules also apply to comparison operators and they can be overridden by means of parentheses.


```
a > 2 + 3
## [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE

(a > 2) + 3
## [1] 3 3 4 4 4 4 4 4 4 4
```

Individual comparisons can be useful, but their full role in data analysis and programming is realized when we combine multiple tests using the operations of the Boolean algebra described in section 1.5 on page 21.

For example to test if members of a numeric vector are within a range, in our example, -1 to +1, we can combine the results from two comparisons using the vectorized logical *AND* operator `&`.

```
c <- -2:3
c >= -1 & c <= 1
## [1] FALSE TRUE TRUE TRUE FALSE FALSE
```

If we want to find those values outside this same range, we can negate the test.

```
c <- -2:3
!(c >= -1 & c <= 1)
## [1] TRUE FALSE FALSE FALSE TRUE TRUE
```

Or we can combine another two comparisons using the vectorized logical *OR* operator `|`.

```
c <- -2:3
c < -1 | c > 1
## [1] TRUE FALSE FALSE FALSE TRUE TRUE
```

In some cases an additional advantage is that `logical` values require less space in memory for their storage than `numeric` values.



Use the statement below as a starting point in exploring how precedence works when logical and arithmetic operators are part of the same statement. *Play* with the example by adding parentheses at different positions and based on the returned values, work out the default order of operator precedence used for the evaluation of the example given below.

```
a <- 1:10
a > 3 | a + 2 < 3
```

It is important to be aware of “short-cut evaluation”. If the result does not depend on the missing value or values, then the result, `TRUE` or `FALSE` is returned. `NA` is returned only if the presence of one or more `NA` values in the input makes the result of the computation unknown.

```
c <- c(a, NA)
c > 5
## [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE NA

all(c > 5)
```

```
## [1] FALSE

any(c > 5)
## [1] TRUE

all(c < 20)
## [1] NA

any(c > 20)
## [1] NA

is.na(a)
## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

is.na(c)
## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE

any(is.na(c))
## [1] TRUE

all(is.na(c))
## [1] FALSE
```

The behavior of many of base-R's functions when `NA`s are present in their input arguments can be modified. `TRUE` passed as an argument to parameter `na.rm`, results in `NA` values being *removed* from the input **before** the function is applied.

```
all(c < 20)
## [1] NA

any(c > 20)
## [1] NA

all(c < 20, na.rm=TRUE)
## [1] TRUE

any(c > 20, na.rm=TRUE)
## [1] FALSE
```



The usual way to store numerical values in computers is to reserve a fixed amount of space in memory for each value, which imposes limits on which numbers can be represented or not, and the maximum precision that can be achieved. The difference between `integer` and `double` is explained on page 5. Integers, or “whole numbers”, like R `integer` values are stored always with the same resolution such that the smallest difference between two integer values is 1. The amount of memory available to store an individual value creates a limit for the size of largest and smallest values that can be represented. Thus integers in R behave like Integers as defined in mathematics, but constrained to a restricted finite range of values. In computing languages like C different types of integer numbers are available `short` and `long`, these differ in the size of the space reserved for them in memory. R `integer` type is equivalent to `long` in C, thus the use of `L` for integer constant values like `5L`.

Floating point numbers like R `double` values are stored in two parts an integer

significand and an integer *exponent*, each part using a fixed amount of space in memory. The relative resolution is constrained by the number of digits that can be stored in the significand while the absolute size of the largest and smallest numbers that can be represented is limited by the largest and smallest values that fit in the memory reserved for the exponent. In computing languages like C different types of floating point numbers are available, these differ in the size of the space reserved for them in memory. The properties of Real numbers as defined in mathematics differ from floating point numbers in assuming unlimited resolution and unlimited range of representable values.

In R, numbers that are not integers are stored as *double-precision floats*. Precision of numerical values in computers is usually described by “epsilon” (ϵ), abbreviated *eps*, defined as the largest value of ϵ for which $1 + \epsilon = 1$. In the second example below, the result of the subtraction is exactly 1 due to insufficient resolution in the returned value, while in the first case there is no loss of information.

```
0 - 1e-20
## [1] -1e-20

1 - 1e-20
## [1] 1
```

The finite resolution of floats can lead to unexpected results when testing for equality or inequality.

```
1e20 == 1 + 1e20
## [1] TRUE

1 == 1 + 1e-20
## [1] TRUE

0 == 1e-20
## [1] FALSE
```

Another way of revealing the limited precision is during conversion to **character**.

```
format(5.123, digits = 16) # near maximum resolution
## [1] "5.123"

format(5.123, digits = 22) # more digits than in resolution
## [1] "5.123000000000000220268"
```

More likely to be a problem in real use of R is the accumulation of successive small losses in precision from multiple operations on R `double` values. Thus when computations involve both very large and very small numbers, the returned value can depend on the order of the operations. In practice ordinary users rarely need to be concerned about losses in precision except when testing for equality and inequality. On the other hand, finite resolution of `double` numerical values can explain why sometimes returned values for equivalent computations differ, and why some computation algorithm may be preferable, or even fail, in specific cases.

As R can run on different types of computer hardware, the actual machine limits for storing numbers in memory may vary depending on the type of processor

and even compiler used to build the R program executable. However, it is possible to obtain these values at run time from the variable `.Machine`, which is part of the R language. Please see the help page for `.Machine` for a detailed and up-to-date description of the available constants.

```
.Machine$double.eps
## [1] 2.220446e-16

.Machine$double.neg.eps
## [1] 1.110223e-16

.Machine$double.max
## [1] 1024

.Machine$double.min
## [1] -1022
```

The last two values refer to the exponents of 10, rather than the maximum and minimum size of numbers that can be handled as objects of class `double`. Values outside these limits are stored as `-Inf` or `Inf` and enter arithmetic as infinite values according the mathematical rules.

```
1e1026
## [1] Inf

1e-1026
## [1] 0

Inf + 1
## [1] Inf

-Inf + 1
## [1] -Inf
```

As `integer` values are stored in machine memory without loss of precision, epsilon is not defined for `integer` values.

```
.Machine$integer.max
## [1] 2147483647

2147483699L
## [1] 2147483699
```

In those statements in the chunk below where at least one operand is `double` the `integer` operands are *promoted* to `double` before computation. A similar promotion does not take place when operations are among `integer` values, resulting in *overflow*, meaning numbers that are too big to be represented as `integer` values.

```

2147483600L + 99L

## Warning in 2147483600L + 99L: NAs produced by integer overflow
## [1] NA

2147483600L + 99
## [1] 2147483699

2147483600L * 2147483600L

## Warning in 2147483600L * 2147483600L: NAs produced by integer overflow
## [1] NA

2147483600L * 2147483600
## [1] 4.611686e+18

```

We see next that the exponentiation operator `^` forces the promotion of its arguments to `double`, resulting in no overflow. In contrast, as seen above, the multiplication operator `*` operates on integers resulting in overflow.


```

2147483600L * 2147483600L

## Warning in 2147483600L * 2147483600L: NAs produced by integer overflow
## [1] NA

2147483600L^2L
## [1] 4.611686e+18

```

 In many situations, when writing programs one should avoid testing for equality of floating point numbers (“floats”). Here we show how to gracefully handle rounding errors. As the example shows, rounding errors may accumulate, and in practice `.Machine$double.eps` is not always a good value to safely use in tests for “zero,” and a larger value may be needed. Whenever possible according to the logic of the calculations, it is best to test for inequalities, for example using `x <= 1.0` instead of `x == 1.0`. If this is not possible, then the tests should be done replacing tests like `x == 1.0` with `abs(x - 1.0) < eps`. Function `abs()` returns the absolute value, in simpler words, makes all values positive or zero, by changing the sign of negative values, or in mathematical notation $|x| = |-x|$.

```

a == 0.0 # may not always work
## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

abs(a) < 1e-15 # is safer
## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

```

```

sin(pi) == 0.0 # angle in radians, not degrees!
## [1] FALSE

sin(2 * pi) == 0.0
## [1] FALSE

abs(sin(pi)) < 1e-15
## [1] TRUE

abs(sin(2 * pi)) < 1e-15
## [1] TRUE

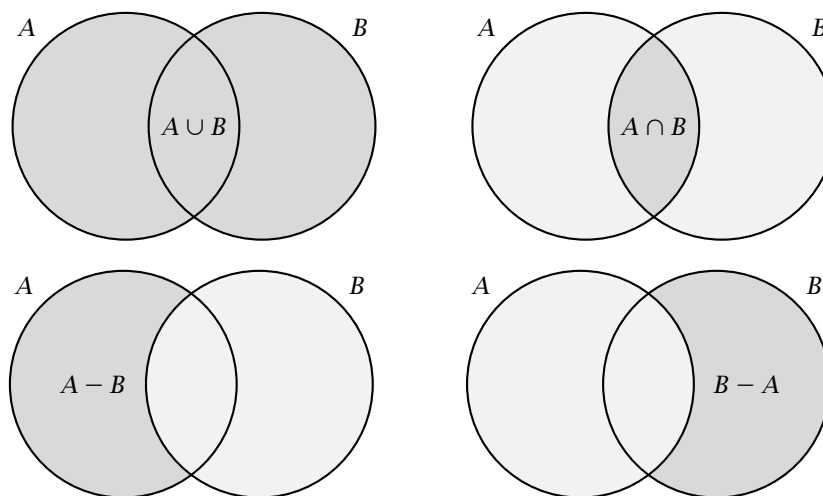
sin(pi)
## [1] 1.224606e-16

sin(2 * pi)
## [1] -2.449213e-16

```

1.7 Sets and set operations

The R language supports set operations on vectors. They can be useful in many different contexts when manipulating and comparing vectors of values. In Bioinformatics it is usual, for example, to make use of character vectors of gene tags. Set algebra operations and their equivalents in mathematical notation and R functions are: *union*, \cup , `union()`; *intersection*, \cap , `intersect()`; *difference (asymmetrical)*, $-$, `setdiff()`; *equality test* `setequal()`; *membership*, `is.element()` and `%in%`. The first three operations return a vector of the same mode as their inputs, and the last three a logical vector. The action of the first three operations is most easily illustrated with Venn diagrams, where the returned value (or result of the operation) is depicted in darker grey.



The remaining operations are easier to exemplify using vectors with values rep-

resenting a mundane example, grocery shopping, only later followed by more abstract examples.

```
fruits <- c("apple", "pear", "orange", "lemon", "tangerine")
bakery <- c("bread", "buns", "cake", "cookies")
dairy <- c("milk", "butter", "cheese")
shopping <- c("bread", "butter", "apple", "cheese", "orange")
intersect(fruits, shopping)
## [1] "apple" "orange"

intersect(bakery, shopping)
## [1] "bread"


intersect(dairy, shopping)
## [1] "butter" "cheese"

"lemon" %in% dairy
## [1] FALSE

"lemon" %in% fruits
## [1] TRUE

dairy %in% shopping
## [1] FALSE TRUE TRUE

setdiff(union(bakery, dairy), shopping)
## [1] "buns" "cake" "cookies" "milk"
```

 Sets describe membership as a binary property, thus when vectors are interpreted as sets, duplicate members are meaningless, although accepted as input and always simplified in the returned values.

```
union(c("a", "a", "b"), c("b", "a", "b")) # set operation
## [1] "a" "b"

setequal(c("a", "a", "b"), c("b", "a", "b")) # sets compared
## [1] TRUE

all.equal(c("a", "a", "b"), c("b", "a", "b")) # objects compared
## [1] "1 string mismatch"

identical(c("a", "a", "b"), c("b", "a", "b")) # objects compared
## [1] FALSE
```

We construct and save a character vector to use in the next examples.


```
my.set <- c("a", "b", "c", "b")
```

To test if a given value belongs to a set, we use operator `%in%` or its function equivalent `is.element()`. In the algebra of sets notation, this is written $a \in A$, where A is a set and a a member. The second statement shows that the `%in%` operator is vectorized on its left-hand-side (lhs) operand, returning a logical vector.

```
is.element("a", my.set)
## [1] TRUE

"a" %in% my.set
## [1] TRUE

c("a", "a", "z") %in% my.set
## [1] TRUE TRUE FALSE
```

 Keep in mind that inclusion is an asymmetrical (not reflective) operation among sets. The rhs argument is interpreted as a set, while the lhs argument is interpreted as a vector of values to test for inclusion. In other words, any duplicate member in the lhs will be retained while the rhs is interpreted as a set of unique values. The returned logical vector has the same length as the lhs.

```
my.set %in% "a"
## [1] TRUE FALSE FALSE FALSE
```


The negation of inclusion is $a \notin A$, and coded in R by applying the negation operator `!` to the result of the test done with `%in%` or function `is.element()`.

```
!is.element("a", my.set)
## [1] FALSE


!"a" %in% my.set
## [1] FALSE

!c("a", "a", "z") %in% my.set
## [1] FALSE FALSE TRUE
```

Although inclusion is a set operation, it is also very useful for the simplification of `if () ... else` statements by replacing multiple tests for alternative constant values of the same mode chained by multiple `|` operators. A useful property of `%in%` and `is.element()` is that they never return `NA`.

 Operator `%in%` is equivalent to function `match()`, although the additional parameters of `match()` provide additional flexibility.

In some cases, such as when accepting partial character strings as input, the aim is not an exact match, but a partial match to target character strings. In this case, either `charmatch()` or `pmatch()` is the correct tool to use depending on the desired handling of partial, ambiguous and exact matches. Use `help()` to find the details if you need to use one of them.

 Use operator `%in%` to write more concisely the following comparisons. Hint: see section 1.5 on page 21 for the difference between `|` and `||` operators.

```
x <- c("a", "a", "z")
x == "a" | x == "b" | x == "c" | x == "d"
```

Convert the logical vectors of length 3 into a vector of length one. Hint: see help for functions `all()` and `any()`.

With `unique()` we convert a vector of possibly repeated values into a set of unique values. In the algebra of sets, a certain object belongs or not to a set. Consequently, in a set, multiple copies of the same object or value are meaningless.

```
unique(my.set)
## [1] "a" "b" "c"
```

Function `unique()` is frequently useful, for example when we want determine the number of distinct values in a vector.

```
length(unique(my.set))
## [1] 3
```



Do the values returned by these two statements differ?

```
c("a", "a", "z") %in% my.set
c("a", "a", "z") %in% unique(my.set)
```



Function `uplicated()` is the counterpart of `unique()`, returning a logical vector indicating which values in a vector are duplicates of values already present at positions with a lower index.

```
uplicated(my.set)
## [1] FALSE FALSE FALSE TRUE

anyDuplicated(my.set)
## [1] 4
```

The R language includes many functions that simplify tasks related to data analysis. Some are well known like `unique()`, but others may need to be searched for in the documentation.

In the notation used in algebra of sets, the set union operator is \cup while the intersection operator is \cap . If we have sets A and B , their union is given by $A \cup B$ —in the next three examples, `c("a", "a", "z")` is a constant, while `my.set` is a variable.

```
union(c("a", "a", "z"), my.set)
## [1] "a" "z" "b" "c"
```

If we have sets A and B , their intersection is given by $A \cap B$.


```
intersect(c("a", "a", "z"), my.set)
## [1] "a"
```



What do you expect to be the difference between the values returned by the three statements in the code chunk below? Before running them, write down your expectations about the value each one will return. Only then run the code. Independently of whether your predictions were correct or not, write down an explanation of what each statement’s operation is.

```
union(c("a", "a", "z"), my.set)
c(c("a", "a", "z"), my.set)
c("a", "a", "z", my.set)
```

In the algebra of sets notation $A \subseteq B$, where A and B are sets, indicates that A is a subset or equal to B . For a true subset, the notation is $A \subset B$. The operators with the reverse direction are \supseteq and \supset . Implement these four operations in four R statements, and test them on sets (represented by R vectors) with different “overlap” among set members.

 All set algebra examples above use character vectors and character constants. This is just the most frequent use case. Sets operations are valid on vectors of any atomic class, including `integer`, and computed values can be part of statements. In the second and third statements in the next chunk, we need to use additional parentheses to alter the default order of precedence between arithmetic and set operators.

```
9L %in% 2L:4L
## [1] FALSE

9L %in% ((2L:4L) * (2L:4L))
## [1] TRUE

c(1L, 16L) %in% ((2L:4L) * (2L:4L))
## [1] FALSE TRUE
```

Empty sets are an important component of the algebra of sets, in R they are represented as vectors of zero length. Vectors and lists of zero length, which the R language fully supports, can be used to “encode” emptiness also in other contexts. These vectors do belong to a class such as `numeric` or `character` and must be compatible with other operands in an expression. By default, constructors for vectors, construct empty vectors.

```
length(integer())
## [1] 0

1L %in% integer()
## [1] FALSE

setdiff(1L:4L, union(1L:4L, integer()))
## integer(0)
```

Although set operators are defined for `numeric` vectors, rounding errors in ‘floats’ can result in unexpected results (see section 1.6 on page 27). The next two examples do, however, return the correct answers.

```
9 %in% (2:4)^2
## [1] TRUE

c(1, 5) %in% (1:10)^2
## [1] TRUE FALSE
```

1.8 The ‘mode’ and ‘class’ of objects

Variables have a *mode* that depends on what is stored in them. But different from other languages, assignment to a variable of a different mode is allowed and in most cases its mode changes together with its contents. However, there is a restriction that all elements in a vector, array or matrix, must be of the same mode. While this is not required for lists, which can be heterogenous. In practice this means that we can assign an object, such as a vector, with a different mode to a name already in use, but we cannot use indexing to assign an object of a different mode to individual members of a vector, matrix or array. Functions with names starting with `is.` are tests returning a logical value, `TRUE`, `FALSE` or `NA`. Function `mode()` returns the mode of an object, as a character string and `typeof()` returns R’s internal type or storage mode.

```
my_var <- 1:5
mode(my_var) # no distinction of integer or double
## [1] "numeric"

typeof(my_var)
## [1] "integer"

is.numeric(my_var) # no distinction of integer or double
## [1] TRUE

is.double(my_var)
## [1] FALSE

is.integer(my_var)
## [1] TRUE

is.logical(my_var)
## [1] FALSE

is.character(my_var)
## [1] FALSE

my_var <- "abc"
mode(my_var)
## [1] "character"
```

While *mode* is a fundamental property, and limited to those modes defined as part of the R language, the concept of *class*, is different in that new classes can be defined in user code. In particular, different R objects of a given mode, such as `numeric`, can belong to different `classes`. The use of classes for dispatching functions is discussed in section ?? on page ??, in relation to object-oriented programming in R. Method `class()` is used to query the class of an object, and method `inherits()` is used to test if an object belongs to a specific class or not (including “parent” classes, to be later described).

```
class(my_var)
## [1] "character"
```

```
inherits(my_var, "character")
## [1] TRUE

inherits(my_var, "numeric")
## [1] FALSE
```

1.9 'Type' conversions

The least-intuitive type conversions are those related to logical values. All others are as one would expect. By convention, functions used to convert objects from one mode to a different one have names starting with `as.`¹.

```
as.character(1)
## [1] "1"

as.numeric("1")
## [1] 1

as.logical("TRUE")
## [1] TRUE

as.logical("NA")
## [1] NA
```

Conversion takes place automatically in arithmetic and logical expressions.

```
TRUE + 10
## [1] 11

1 || 0
## [1] TRUE

FALSE | -2:2
## [1] TRUE TRUE FALSE TRUE TRUE
```



There is some flexibility in the conversion from character strings into `numeric` and `logical` values. Use the examples below plus your own variations to get an idea of what strings are acceptable and correctly converted and which are not. Do also pay attention at the conversion between `numeric` and `logical` values.

¹Except for some packages in the 'tidyverse' that use names starting with `as_` instead of `as.`

```
as.character(3.0e10)
as.numeric("5E+5")
as.numeric("A")
as.numeric(TRUE)
as.numeric(FALSE)
as.logical("T")
as.logical("t")
as.logical("true")
as.logical(100)
as.logical(0)
as.logical(-1)
```



Compare the values returned by `trunc()` and `as.integer()` when applied to a floating point number, such as 12.34. Check for the equality of values, and for the *class* of the returned objects.



Using conversions, the difference between the length of a `character` vector and the number of characters composing each member “string” within a vector is obvious.

```
f <- c("1", "2", "3")
length(f)
## [1] 3

g <- "123"
length(g)
## [1] 1

as.numeric(f)
## [1] 1 2 3

as.numeric(g)
## [1] 123
```

Other functions relevant to the “conversion” of numbers and other values are `format()`, and `sprintf()`. These two functions return `character` strings, instead of `numeric` or other values, and are useful for printing output. One could think of these functions as advanced conversion functions returning formatted, and possibly combined and annotated, character strings. However, they are usually not considered normal conversion functions, as they are very rarely used in a way that preserves the original precision of the input values. We show here the use of `format()` and `sprintf()` with `numeric` values, but they can also be used with values of other modes.

When using `format()`, the format used to display numbers is set by passing arguments to several different parameters. As `print()` calls `format()` to make numbers *pretty* it accepts the same options.

```
x = c(123.4567890, 1.0)
format(x) # using defaults
## [1] "123.4568" " 1.0000"

format(x[1]) # using defaults
```

```
## [1] "123.4568"

format(x[2]) # using defaults
## [1] "1"

format(x, digits = 3, nsmall = 1)
## [1] "123.5" " 1.0"

format(x[1], digits = 3, nsmall = 1)
## [1] "123.5"

format(x[2], digits = 3, nsmall = 1)
## [1] "1.0"

format(x, digits = 3, scientific = TRUE)
## [1] "1.23e+02" "1.00e+00"
```

Function `sprintf()` is similar to C's function of the same name. The user interface is rather unusual, but very powerful, once one learns the syntax. All the formatting is specified using a character string as template. In this template, placeholders for data and the formatting instructions are embedded using special codes. These codes start with a percent character. We show in the example below the use of some of these: `f` is used for numeric values to be formatted according to a "fixed point," while `g` is used when we set the number of significant digits and `e` for exponential or *scientific* notation.

```
x = c(123.4567890, 1.0)
sprintf("The numbers are: %4.2f and %.0f", x[1], x[2])
## [1] "The numbers are: 123.46 and 1"


sprintf("The numbers are: %4.2g and %.2g", x[1], x[2])
## [1] "The numbers are: 123.5 and 1"

sprintf("The numbers are: %4.2e and %.0e", x[1], x[2])
## [1] "The numbers are: 1.23e+02 and 1e+00"
```

In the template "The numbers are: %4.2f and %.0f", there are two placeholders for numeric values, %4.2f and %.0f, so in addition to the template, we pass two values extracted from the first two positions of vector `x`. These could have been two different vectors of length one, or even numeric constants. The template itself does not need to be a character constant as in these examples, as a variable can be also passed as argument.



Function `format()` may be easier to use, in some cases, but `sprintf()` is more flexible and powerful. Those with experience in the use of the C language will already know about `sprintf()` and its use of templates for formatting output. Even if you are familiar with C, look up the help pages for both functions, and practice, by trying to create the same formatted output by means of the two functions. Do also play with these functions with other types of data like `integer` and `character`.

 We have above described `NA` as a single value ignoring modes, but in reality `NA` s come in various flavors. `NA_real_`, `NA_character_`, etc. and `NA` defaults to an `NA` of class `logical`. `NA` is normally converted on the fly to other modes when needed, so in general `NA` is all we need to use.

```
a <- c(1, NA)
is.numeric(a[2])
## [1] TRUE

is.numeric(NA)
## [1] FALSE

b <- c("abc", NA)
is.character(b[2])
## [1] TRUE

is.character(NA)
## [1] FALSE

class(NA)
## [1] "logical"

class(NA_character_)
## [1] "character"

c <- NA
c(c, 2:3)
## [1] NA  2  3
```

However, even the statement below works transparently.

```
a[3] <- b[2]
```

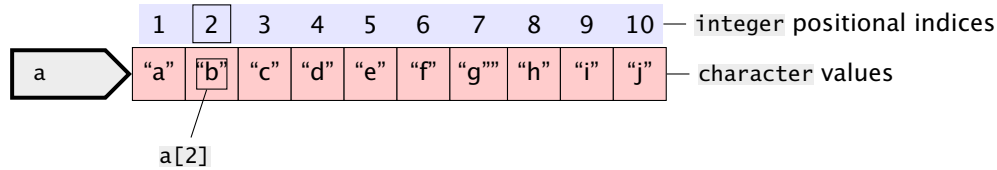
1.10 Vector manipulation

If you have read earlier sections of this chapter, you already know how to create a vector. If not, see pages 6–10 before continuing.


In this section we are going to see how to extract or retrieve, replace, and move elements such as a_2 from a vector $a_{1=1\dots n}$. Elements are extracted using an index enclosed in single square brackets. The index indicates the position in the vector, starting from one, following the usual mathematical tradition. What in maths notation would be a_i , in R is represented as `a[i]` and the whole vector, by excluding the brackets and indexing vector, as `a`.

We extract the first 10 elements of the vector `letters`.

```
a <- letters[1:10]
a
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```




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

 Four constant vectors are available in R: `letters`, `LETTERS`, `month.name` and `month.abb`, of which we used `letters` in the example above. These vectors are always for English, irrespective of the locale.

```
month.name
## [1] "January" "February" "March" "April" "May" "June"
## [7] "July" "August" "September" "October" "November" "December"

month.name[6]
## [1] "June"
```

 In R, indexes always start from one, while in some other programming languages such as C and C++, indexes start from zero. It is important to be aware of this difference, as many computation algorithms are valid only under a given indexing convention.


How to access the last value in a vector?

```
month.name[length(month.name)]
## [1] "December"
```

It is possible to extract a subset of the elements of a vector in a single operation, using a vector of indexes. The positions of the extracted elements in the result (“returned value”) are determined by the ordering of the members of the vector of indexes—easier to demonstrate than to explain.

```
a[c(3, 2)]
## [1] "c" "b"

a[10:1]
## [1] "j" "i" "h" "g" "f" "e" "d" "c" "b" "a"
```


 The length of the indexing vector is not restricted by the length of the indexed vector. However, only numerical indexes that match positions present in the indexed vector can extract values. Those values in the indexing vector pointing to positions that are not present in the indexed vector, result in `NA`s. This is easier to learn by *playing* with R, than from explanations. Play with R, using the following examples as a starting point.

```
length(a)
a[c(3, 3, 3, 3)]
a[c(10:1, 1:10)]
a[c(1, 11)]
a[11]
```


Have you tried some of your own examples? If not yet, do *play* with additional variations of your own before continuing.

Negative indexes have a special meaning; they indicate the positions at which values should be excluded. Be aware that it is *illegal* to mix positive and negative values in the same indexing operation.

```
a[-2]
## [1] "a" "c" "d" "e" "f" "g" "h" "i" "j"

a[-c(3,2)]
## [1] "a" "d" "e" "f" "g" "h" "i" "j"

a[-3:-2]
## [1] "a" "d" "e" "f" "g" "h" "i" "j"
```

 Results from indexing with special values and zero may be surprising. Try to build a rule from the examples below, a rule that will help you remember what to expect next time you are confronted with similar statements using “subscripts” which are special values instead of integers larger or equal to one—this is likely to happen sooner or later as these special values can be returned by different R expressions depending on the value of operands or function arguments, some of them described earlier in this chapter.

```
a[ ]
a[0]
a[numeric(0)]
a[NA]
a[c(1, NA)]
a[NULL]
a[c(1, NULL)]
```

Another way of indexing, which is very handy, but not available in most other programming languages, is indexing with a vector of `logical` values. The `logical` vector used for indexing is usually of the same length as the vector from which elements are going to be selected. However, this is not a requirement, because if the `logical` vector of indexes is shorter than the indexed vector, it is “recycled” as discussed above in relation to other operators.

```

a[TRUE]
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

a[FALSE]
## character(0)

a[c(TRUE, FALSE)]
## [1] "a" "c" "e" "g" "i"

a[c(FALSE, TRUE)]
## [1] "b" "d" "f" "h" "j"

a > "c"
## [1] FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

a[a > "c"]
## [1] "d" "e" "f" "g" "h" "i" "j"

```

Indexing with logical vectors is very frequently used in R because comparison operators are vectorized. Comparison operators, when applied to a vector, return a **logical** vector, a vector that can be used to extract the elements for which the result of the comparison test was **TRUE**.



The examples in this text box demonstrate additional uses of logical vectors: 1) the logical vector returned by a vectorized comparison can be stored in a variable, and the variable used as a “selector” for extracting a subset of values from the same vector, or from a different vector.

```

a <- letters[1:10]
b <- 1:10
selector <- a > "c"
selector
a[selector]
b[selector]

```

Numerical indexes can be obtained from a logical vector by means of function **which()**.

```

indexes <- which(a > "c")
indexes
a[indexes]
b[indexes]

```

Make sure to understand the examples above. These constructs are very widely used in R because they allow for concise code that is easy to understand once you are familiar with the indexing rules. However, if you do not command these rules, many of these terse statements will be unintelligible to you.



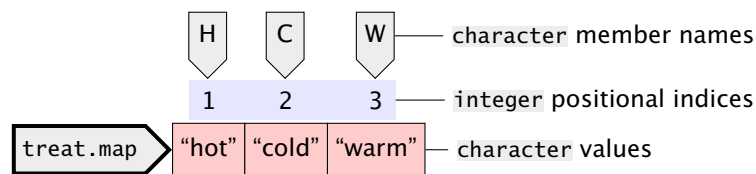
In all earlier examples we have used integer valued indices for extraction of elements. In the vectors used as examples above the elements were anonymous or nameless. In R the elements can be assigned names, and these names used in place of numeric indices to extract the named elements. There is one situation where this is very useful: the mapping of values between two representations.

Let’s assume we have a long vector encoding treatments using single letter codes and we want to replace these codes with clearer names.

```
treat <- c("H", "C", "H", "W", "C", "H", "H", "W", "W")
```

We can create a named vector to *map* the single letter codes into some other codes, in this case full words that are easier to understand.

```
treat.map <- c(H = "hot", C = "cold", W = "warm")
treat.map
##      H      C      W
##  "hot" "cold" "warm"
```



As `treat.map` is a named vector, we can use the element names as indices for element extraction.

```
treat.map["H"]
##      H
##  "hot"
```

The indexing vector can be of a different length than the indexed vector, and that the returned value is a new vector of the same length as the indexing vector.

```
treat.new <- treat.map[treat]
treat.new
##      H      C      H      W      C      H      H      W      W
##  "hot" "cold" "hot" "warm" "cold" "hot" "hot" "warm" "warm"
```

where `treat.new` is a named vector, from which we will frequently want to remove the names.

```
treat.new <- unname(treat.new)
treat.new
## [1] "hot" "cold" "hot" "warm" "cold" "hot" "hot" "warm" "warm"
```

It is more common to use named members with lists than with vectors, but in R, in both cases it is possible to use both numeric positional indices and names.

Indexing can be used on either side of an assignment expression. In the chunk below, we use the extraction operator on the left-hand side of the assignments to replace values only at selected positions in the vector. This may look rather esoteric at first sight, but it is just a simple extension of the logic of indexing described above. It works, because the low precedence of the `<-` operator results in both the left-hand side and the right-hand side being fully evaluated before the

assignment takes place. To make the changes to the vectors easier to follow, we use identical vectors with different names for each of these examples.

```
a <- 1:10
a
## [1] 1 2 3 4 5 6 7 8 9 10

a[1] <- 99
a
## [1] 99 2 3 4 5 6 7 8 9 10

b <- 1:10
b[c(2,4)] <- -99 # recycling
b
## [1] 1 -99 3 -99 5 6 7 8 9 10

c <- 1:10
c[c(2,4)] <- c(-99, 99)
c
## [1] 1 -99 3 99 5 6 7 8 9 10

d <- 1:10
d[TRUE] <- 1 # recycling
d
## [1] 1 1 1 1 1 1 1 1 1 1

e <- 1:10
e <- 1 # no recycling
e
## [1] 1
```

We can also use subscripting on both sides of the assignment operator, for example, to swap two elements.

```
a <- letters[1:10]
a[1:2] <- a[2:1]
a
## [1] "b" "a" "c" "d" "e" "f" "g" "h" "i" "j"
```



Do play with subscripts to your heart's content, really grasping how they work and how they can be used, will be very useful in anything you do in the future with R. Even the contrived example below follows the same simple rules, just study it bit by bit. Hint: the second statement in the chunk below, modifies `a`, so, when studying variations of this example you will need to recreate `a` by executing the first statement, each time you run a variation of the second statement.

```
a <- letters[1:10]
a[5:1] <- a[c(TRUE, FALSE)]
a
```



In R, indexing with positional indexes can be done with `integer` or `numeric` values. Numeric values can be floats, but for indexing, only integer values are meaningful. Consequently, `double` values are converted into `integer` values when

used as indexes. The conversion is done invisibly, but it does slow down computations slightly. When working on big data sets, explicitly using `integer` values can improve performance.

```
b <- LETTERS[1:10]
b
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"

b[1]
## [1] "A"

b[1.1]
## [1] "A"

b[1.9999] # surprise!!
## [1] "A"

b[2]
## [1] "B"
```

From this experiment, we can learn that if positive indexes are not whole numbers, they are truncated to the next smaller integer.

```
b <- LETTERS[1:10]
b
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"

b[-1]
## [1] "B" "C" "D" "E" "F" "G" "H" "I" "J"

b[-1.1]
## [1] "B" "C" "D" "E" "F" "G" "H" "I" "J"

b[-1.9999]
## [1] "B" "C" "D" "E" "F" "G" "H" "I" "J"

b[-2]
## [1] "A" "C" "D" "E" "F" "G" "H" "I" "J"
```

From this experiment, we can learn that if negative indexes are not whole numbers, they are truncated to the next larger (less negative) integer. In conclusion, `double` index values behave as if they were sanitized using function `trunc()`.

This example also shows how one can tease out of R its rules through experimentation.

A frequent operation on vectors is sorting them into an increasing or decreasing order. The most direct approach is to use `sort()`.

```
my.vector <- c(10, 4, 22, 1, 4)
sort(my.vector)
## [1] 1 4 4 10 22


sort(my.vector, decreasing = TRUE)
## [1] 22 10 4 4 1
```

An indirect way of sorting a vector, possibly based on a different vector, is to generate with `order()` a vector of numerical indexes that can be used to achieve the ordering.

```
order(my.vector)
## [1] 4 2 5 1 3

my.vector[order(my.vector)]
## [1] 1 4 4 10 22

another.vector <- c("ab", "aa", "c", "zy", "e")
another.vector[order(my.vector)]
## [1] "zy" "aa" "e" "ab" "c"
```

 A problem linked to sorting that we may face is counting how many copies of each value are present in a vector. We need to use two functions `sort()` and `rle()`. The second of these functions computes *run length* as used in *run length encoding* for which *rle* is an abbreviation. A *run* is a series of consecutive identical values. As the objective is to count the number of copies of each value present, we need first to sort the vector.

```
my.letters <- letters[c(1,5,10,3,1,4,21,1,10)]
my.letters
## [1] "a" "e" "j" "c" "a" "d" "u" "a" "j"

sort(my.letters)
## [1] "a" "a" "a" "c" "d" "e" "j" "j" "u"

rle(sort(my.letters))
## Run Length Encoding
## lengths: int [1:6] 3 1 1 1 2 1
## values : chr [1:6] "a" "c" "d" "e" "j" "u"
```

The second and third statements are only to demonstrate the effect of each step. The last statement uses nested function calls to compute the number of copies of each value in the vector.


1.11 Matrices and multidimensional arrays

Matrices have two dimensions, rows and columns, and like vectors all their members share the same mode, and are atomic, i.e., they are homogeneous. Most commonly, matrices are used to store `numeric`, `integer` or `logical` values. The number of rows and columns can differ, so matrices can be either square or rectangular in shape, but never ragged.

In R, the first index always denotes rows and the second index always denotes columns. The diagram below depicts a matrix, A , with m rows and n columns and size equal to $m \times n$ “cells”, with individual values denoted by $a_{i,j}$. Here we use a

simpler representation than that used for vectors on page 6 above, but the same concepts apply.

Rows or margin 1: $i = 1$ to $i = m$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	\dots	$a_{1,n}$
	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$		$a_{2,n}$
	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$		$a_{3,n}$
	\vdots			\ddots	
	$a_{m,1}$	$a_{m,2}$	$a_{m,3}$		$a_{m,n}$
Columns or margin 2: $j = 1$ to $j = n$					

 In R documentation and in function parameters, the individual dimensions of matrices and arrays are sometimes called *margins*, numbered in the same order as the indices are given.

In mathematical notation the same generic matrix is represented as

$$A_{m \times n} = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,j} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,j} & \dots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots & & \vdots \\ a_{i,1} & a_{i,2} & \dots & a_{i,j} & \dots & a_{i,n} \\ \vdots & \vdots & & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \dots & a_{m,j} & \dots & a_{m,n} \end{bmatrix}$$

where A represents the whole matrix, $m \times n$ its dimensions, and $a_{i,j}$ its elements, with i indexing rows and j indexing columns. The lengths of the two dimensions of the matrix are given by m and n , for rows and columns.

Vectors have a single dimension, and, as described on page 6 above, we can query this dimension, their length, with method `length()`. Matrices have two dimensions, which can be queried individually with `ncol()` and `nrow()`, and jointly with method `dim()`. As expected method `is.matrix()` can be used to query the class.

We can create a matrix using the `matrix()` or `as.matrix()` constructors. The first argument of `matrix()` must be a vector. Method `as.matrix()` is a conversion constructor, with specializations accepting as argument objects belonging to a few other classes.

```
matrix(1:15, ncol = 3)
##      [,1] [,2] [,3]
## [1,]    1    6   11
## [2,]    2    7   12
## [3,]    3    8   13
## [4,]    4    9   14
## [5,]    5   10   15
```

```
matrix(1:15, nrow = 3)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    4    7   10   13
## [2,]    2    5    8   11   14
## [3,]    3    6    9   12   15
```

When a matrix is printed in R the row and column indexes are indicated on the edges left and top margins, in the same way as they would be used to extract whole rows and columns.

When a vector is converted to a matrix, R's default is to allocate the values in the vector to the matrix starting from the leftmost column, and within the column, down from the top. Once the first column is filled, the process continues from the top of the next column, as can be seen above. This order can be changed as you will discover in the playground below.



Check in the help page for the `matrix` constructor how to use the `byrow` parameter to alter the default order in which the elements of the vector are allocated to columns and rows of the new matrix.

```
help(matrix)
```

While you are looking at the help page, also consider the default number of columns and rows.

```
matrix(1:15)
```

And to start getting a sense of how to interpret error and warning messages, run the code below and make sure you understand which problem is being reported. Before executing the statement, analyze it and predict what the returned value will be. Afterwards, compare your prediction, to the value actually returned.

```
matrix(1:15, ncol = 2)
```

Subscripting of matrices and arrays is consistent with that used for vectors; we only need to supply an indexing vector, or leave a blank space, for each dimension. A matrix has two dimensions, so to access an element or group of elements, we use two indices. The first index value selects rows, and the second one, columns.

```
A <- matrix(1:20, ncol = 4)
A
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20

A[1, 1]
## [1] 1
```

Remind yourself of how indexing of vectors works in R (see section 1.10 on

page 40). We will now apply the same rules in two dimensions to extract and replace values. The first or leftmost indexing vector corresponds to rows and the second one to columns, so R uses a rows-first convention for indexing. Missing indexing vectors are interpreted as meaning *extract all rows* and *extract all columns*, respectively.


```
A[1, ]
## [1] 1 6 11 16

A[ , 1]
## [1] 1 2 3 4 5

A[2:3, c(1,3)]
##      [,1] [,2]
## [1,]    2   12
## [2,]    3   13

A[3, 4] <- 99
A
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   99
## [4,]    4    9   14   19
## [5,]    5   10   15   20

A[4:3, 2:1] <- A[3:4, 1:2]
A
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    9    4   13   99
## [4,]    8    3   14   19
## [5,]    5   10   15   20
```

 Vectors are simpler than matrices, and by default when possible the “slice” extracted from a matrix it is simplified into a vector by dropping one dimension. By passing `drop = FALSE`, we can prevent this.

```
is.matrix(A[1, ])
## [1] FALSE

is.matrix(A[1:2, 1:2])
## [1] TRUE

is.vector(A[1, ])
## [1] TRUE

is.vector(A[1:2, 1:2])
## [1] FALSE

is.matrix(A[1, , drop = FALSE])
## [1] TRUE

is.matrix(A[1:2, 1:2, drop = FALSE])
## [1] TRUE
```

Matrices, like vectors, can be assigned names that function as “nicknames” for indices for assignment and extraction. Matrices can have row names and/or column names.

```
colnames(A)
## NULL

rownames(A)
## NULL

colnames(A) <- c("a", "b", "c", "d")
A
##      a  b  c  d
## [1,] 1  6 11 16
## [2,] 2  7 12 17
## [3,] 9  4 13 99
## [4,] 8  3 14 19
## [5,] 5 10 15 20

A[, c("b", "a")]
##      b a
## [1,] 6 1
## [2,] 7 2
## [3,] 4 9
## [4,] 3 8
## [5,] 10 5

colnames(A) <- NULL
A
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    9    4   13   99
## [4,]    8    3   14   19
## [5,]    5   10   15   20
```



Matrices can be indexed as vectors, without triggering an error or warning.

```
A <- matrix(1:20, ncol = 4)
A
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20

dim(A)
## [1] 5 4

A[10]
## [1] 10

A[5, 2]
## [1] 10
```

The next code example demonstrates that indexing as a vector with a single


index, always works column-wise even if matrix **B** was created by assigning vector elements by row.

```
B <- matrix(1:20, ncol = 4, byrow = TRUE)
B
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]   13   14   15   16
## [5,]   17   18   19   20

dim(B)
## [1] 5 4

B[10]
## [1] 18

B[5, 2]
## [1] 18
```

 In R, a `matrix` can have a single row, a single column, a single element or no elements. However, in all cases, a `matrix` will have a *dimensions* attribute of length two defined.

```
my.vector <- 1:6
dim(my.vector)
## NULL

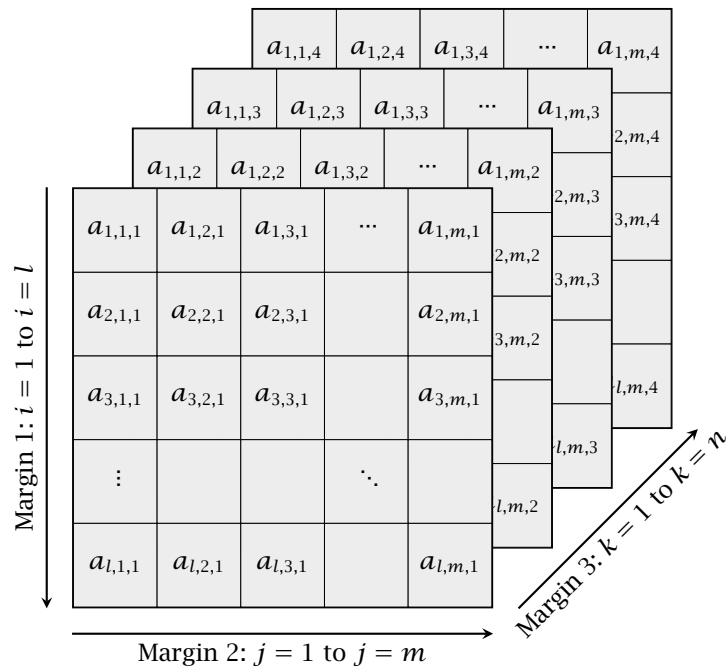
one.col.matrix <- matrix(1:6, ncol = 1)
dim(one.col.matrix)
## [1] 6 1

two.col.matrix <- matrix(1:6, ncol = 2)
dim(two.col.matrix)
## [1] 3 2

one.elem.matrix <- matrix(1, ncol = 1)
dim(one.elem.matrix)
## [1] 1 1

no.elem.matrix <- matrix(numeric(), ncol = 0)
dim(no.elem.matrix)
## [1] 0 0
```

Arrays are similar to matrices, but can have one or more dimensions. The dimensions of an array can be queried with method `dim()`, similarly as with matrices. Whether an R object is an array can be found out with function `is.array()`. The diagram below depicts an array, *A* with three dimensions giving a size equal to $l \times m \times n$, and individual values denoted by $a_{i,j,k}$.



When calling the constructor `array()`, dimensions are specified with the argument passed to parameter `dim`.

```
B <- array(1:27, dim = c(3, 3, 3))
B
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]   10   13   16
## [2,]   11   14   17
## [3,]   12   15   18
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   19   22   25
## [2,]   20   23   26
## [3,]   21   24   27

B[2, 2, 2]
## [1] 14
```

In the chunk above, the length of the supplied vector is the product of the dimensions, $27 = 3 \times 3 \times 3 = 3^3$. Arrays are printed in slices, with slices across 3rd and higher dimensions printed separately, with their corresponding indexes

above each slice and the first two dimensions on the margins of the individual slices, similarly to how matrices are displayed.



How do you use indexes to extract the second element of the original vector, in each of the following matrices and arrays?

```
v <- 1:10
m2c <- matrix(v, ncol = 2)
m2cr <- matrix(v, ncol = 2, byrow = TRUE)
m2r <- matrix(v, nrow = 2)
m2rc <- matrix(v, nrow = 2, byrow = TRUE)
```

```
v <- 1:10
a2c <- array(v, dim = c(5, 2))
a2c <- array(v, dim = c(5, 2), dimnames = list(NULL, c("c1", "c2")))
a2r <- array(v, dim = c(2, 5))
```

Be aware that vectors and one-dimensional arrays are not the same thing, while two-dimensional arrays are matrices.

1. Use the different constructors and query methods to explore this, and its consequences.
2. Convert a matrix into a vector using `unlist()` and `as.vector()` and compare the returned values.

Operators for matrices are available in R, as matrices are used in many statistical algorithms. We will not describe them all here, only `t()` and some specializations of arithmetic operators. Function `t()` transposes a matrix, by swapping columns and rows.

```
A <- matrix(1:20, ncol = 4)
A
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20

t(A)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]   11   12   13   14   15
## [4,]   16   17   18   19   20
```

As with vectors, recycling applies to arithmetic operators when applied to matrices.

```
A + 2
##      [,1] [,2] [,3] [,4]
## [1,]    3    8   13   18
## [2,]    4    9   14   19
```

```
## [3,] 5 10 15 20
## [4,] 6 11 16 21
## [5,] 7 12 17 22

A * 0:1
##      [,1] [,2] [,3] [,4]
## [1,] 0 6 0 16
## [2,] 2 0 12 0
## [3,] 0 8 0 18
## [4,] 4 0 14 0
## [5,] 0 10 0 20

A * 1:0
##      [,1] [,2] [,3] [,4]
## [1,] 1 0 11 0
## [2,] 0 7 0 17
## [3,] 3 0 13 0
## [4,] 0 9 0 19
## [5,] 5 0 15 0
```

In the examples above with the usual multiplication operator `*`, the operation described is not a matrix product, but instead, the products between individual elements of the matrix and vectors. Operators and functions implementing the operations of matrix algebra are available. Matrix algebra gives the rules for operations where operands are whole matrices. For example, matrix multiplication is indicated by operator `%*%`.

```
B <- matrix(1:16, ncol = 4)
B * B
##      [,1] [,2] [,3] [,4]
## [1,] 1 25 81 169
## [2,] 4 36 100 196
## [3,] 9 49 121 225
## [4,] 16 64 144 256

B %*% B
##      [,1] [,2] [,3] [,4]
## [1,] 90 202 314 426
## [2,] 100 228 356 484
## [3,] 110 254 398 542
## [4,] 120 280 440 600
```

Other operators and functions for matrix algebra like cross-product (`crossprod()`), extracting or replacing the diagonal (`diag()`) are available in base R. Packages, including ‘matrixStats’, provide additional functions and operators for matrices.

1.12 Factors

Factors are very important in R. In contrast to other statistical software in which the role of a variable is set when defining a model to be fitted or when setting up a test,

in R, models are specified exactly in the same way for ANOVA and regression analysis, both as *linear models*. The type of model that is fitted is decided by whether the explanatory variable is a factor (giving ANOVA) or a numerical variable (giving regression). This makes a lot of sense, because in most cases, considering an explanatory variable as categorical or not, depends on the quantity stored and/or the design of the experiment or survey. In other words, being categorical is a property of the data. The order of the levels in an unordered `factor` does not affect simple calculations or the values plotted, but as we will see in chapters ?? and ??, it does affect how the output is printed, the order of the levels in the scales and keys of plots, and in some cases how contrasts are applied in significance tests.


In a factor, values indicate discrete unordered categories, most frequently the treatments in an experiment, or categories in a survey. They can be created either from numerical or character vectors. The different possible values are called *levels*. Factors created with `factor()` are always unordered or categorical. R also supports ordered factors, created with function `ordered()` with identical user interface. The distinction, however, only affects how they are interpreted in statistical tests as discussed in chapter ??.

When using `factor()` or `ordered()` we create a factor from a vector, but this vector can be created on-the-fly and anonymous as shown in this example. When the vector is `numeric` and no labels are supplied, level labels are character strings matching the numbers. The default ordering of the levels is alphanumerical.

```
factor(x = c(1, 2, 2, 1, 2, 1, 1))
## [1] 1 2 2 1 2 1 1
## Levels: 1 2

ordered(x = c(1, 2, 2, 1, 2, 1, 1))
## [1] 1 2 2 1 2 1 1
## Levels: 1 < 2

factor(x = c(1, 2, 2, 1, 2, 1, 1), ordered = TRUE)
## [1] 1 2 2 1 2 1 1
## Levels: 1 < 2
```

 When the pattern of levels is regular, it is possible to use function `gl()`, *generate levels*, to construct a factor. Nowadays, it is usual to read data into R from files in which the treatment codes are already available as character strings or numeric values, however, when we need to create a factor within R, `gl()` can save some typing. In this case instead of passing a vector as argument, we pass a *recipe* to create it: `n` is the number of levels, and `k` the number of contiguous repeats (called “replicates” in R documentation) and `length` the length of the factor to be created.

```
gl(n = 2, k = 5, labels = c("A", "B"))
## [1] A A A A A B B B B B
## Levels: A B

gl(n = 2, k = 1, length = 10, labels = c("A", "B"))
## [1] A B A B A B A B A B
## Levels: A B
```

It is always preferable to use meaningful labels for levels, even if R does not

require it. Here the vector is stored in a variable named `my.vector`. In a real data analysis situation in most cases the vector would have been read from a file on disk and would be longer.

```
my.vector <- c("treated", "treated", "control", "control", "control", "treated")
factor(my.vector)
## [1] treated treated control control control treated
## Levels: control treated
```

The ordering of levels is established at the time a factor is created, and by default is alphabetical. This default ordering of levels is frequently not the one needed. We can pass an argument to parameter `levels` of function `factor()` to set a different ordering of the levels.

```
factor(x = my.vector, levels = c("treated", "control"))
## [1] treated treated control control control treated
## Levels: treated control
```

The labels (“names”) of the levels can be set when calling `factor()`. Two vectors are passed as arguments to parameters `levels` and `labels` with levels and matching labels in the same position. The argument passed to `levels` determines the order of the levels based on their old names or values, and the argument passed to `labels` gives new names to the levels.

```
factor(x = c(1, 1, 0, 0, 0, 1), levels = c(1, 0), labels = c("treated", "control"))
## [1] treated treated control control control treated
## Levels: treated control
```


In the examples above we passed a numeric vector or a character vector as an argument for parameter `x` of function `factor()`. It is also possible to pass a `factor` as an argument to parameter `x`. This makes it possible to modify the ordering of levels or replace the labels in a factor.

```
my.factor <- factor(x = my.vector)
my.factor
## [1] treated treated control control control treated
## Levels: control treated

factor(x = my.factor, levels = c("treated", "control"))
## [1] treated treated control control control treated
## Levels: treated control

factor(x = my.factor, labels = c(control = "cooled", treated = "heated"))
## [1] heated heated cooled cooled cooled heated
## Levels: cooled heated

factor(x = my.factor,
      levels = c("treated", "control"),
      labels = c("heated", "cooled"))
## [1] heated heated cooled cooled cooled heated
## Levels: heated cooled
```


 **Merging factor levels.** We use `factor()` as shown below, setting the same label for the levels we want to merge.

```
my.factor1 <- gl(4, 3, labels = c("A", "F", "B", "Z"))
my.factor1
## [1] A A A F F F B B B Z Z Z
## Levels: A F B Z

factor(my.factor1,
       levels = c("A", "B", "F", "Z"),
       labels = c("A", "B", "C", "C"))
## [1] A A A C C C B B B C C C
## Levels: A B C
```

We can use indexing on factors in the same way as with vectors. In the next example, we use a test returning a logical vector to extract all “controls.” We use function `levels()` to look at the levels of the factors, as with vectors, `length()` to query the number of values stored.

```
my.factor
## [1] treated treated control control control treated
## Levels: control treated

levels(my.factor)
## [1] "control" "treated"

length(my.factor)
## [1] 6


control.factor <- my.factor[my.factor == "control"]
control.factor
## [1] control control control
## Levels: control treated

levels(control.factor) # same as in my.factor
## [1] "control" "treated"

length(control.factor) # shorter than my.factor
## [1] 3

control.factor <- factor(control.factor)
levels(control.factor) # the unused level was dropped
## [1] "control"
```

It can be seen above that subsetting does not drop unused factor levels, and that `factor()` can be used to explicitly drop the unused factor levels.


 How to convert factors into numeric vectors is not obvious, even when the factor was created from a `numeric` vector.

```
my.vector2 <- rep(3:5, 4)
my.vector2
## [1] 3 4 5 3 4 5 3 4 5 3 4 5

my.factor2 <- factor(my.vector2)
my.factor2
## [1] 3 4 5 3 4 5 3 4 5 3 4 5
## Levels: 3 4 5

as.numeric(my.factor2)
## [1] 1 2 3 1 2 3 1 2 3 1 2 3


as.numeric(as.character(my.factor2))
## [1] 3 4 5 3 4 5 3 4 5 3 4 5
```


 **Why is a double conversion needed?** Internally, factor values are stored as running integers starting from one, each distinct integer value corresponding to a level. These underlying integer values are returned by `as.numeric()` when applied to a factor instead of the level labels. The labels of the factor levels are always stored as character strings, even when these characters are digits. In contrast to `as.numeric()`, `as.character()` returns the character labels of the levels for each of the values stored in the factor. If these character strings represent numbers, they can be converted, in a second step, using `as.numeric()` into the original numeric values. Use of `class` and `mode` is described on section 1.8 on page 36, and `str()` on page ??.

```
class(my.factor2)
## [1] "factor"

mode(my.factor2)
## [1] "numeric"

str(my.factor2)
## Factor w/ 3 levels "3","4","5": 1 2 3 1 2 3 1 2 3 1 ...
```

 Create a factor with levels labeled with words. Create another factor with the levels labeled with the same words, but ordered differently. After this convert both factors to numeric vectors using `as.numeric()`. Explain why the two numeric vectors differ or not from each other.

 **Safely reordering and renaming factor levels.** The simplest approach is to use `factor()` and its `levels` parameter as shown above. In these more advanced examples we use `levels()` to retrieve the names of the levels from the factor itself to protect from possible bugs due to typing mistakes, or for changes in the naming conventions used.

Reverse previous order using `rev()`.

```
my.factor2 <- factor(c("treated", "treated", "control", "control", "control", "treated"))
levels(my.factor2)
## [1] "control" "treated"

my.factor2 <- factor(my.factor2, levels = rev(levels(my.factor2)))
levels(my.factor2)
## [1] "treated" "control"
```

Sort in decreasing order, i.e., opposite to default.

```
my.factor2 <- factor(my.factor2,
                     levels = sort(levels(my.factor2), decreasing = TRUE))
levels(my.factor2)
## [1] "treated" "control"
```

Alter ordering using subscripting; especially useful with three or more levels.

```
my.factor2 <- factor(my.factor2, levels = levels(my.factor2)[c(2, 1)])
levels(my.factor2)
## [1] "control" "treated"
```

Reordering the levels of a factor based on summary quantities from data stored in a numeric vector is very useful, especially when plotting. Function `reorder()` can be used in this case. It defaults to using `mean()` for summaries, but other suitable summary functions, such as `median()` can be supplied in its place.

```
my.factor3 <- gl(2, 5, labels = c("A", "B"))
my.vector3 <- c(5.6, 7.3, 3.1, 8.7, 6.9, 2.4, 4.5, 2.1, 1.4, 2.0)
my.factor3
## [1] A A A A A B B B B B
## Levels: A B

my.factor3ord <- reorder(my.factor3, my.vector3)
levels(my.factor3ord)
## [1] "B" "A"

my.factor3rev <- reorder(my.factor3, -my.vector3) # a simple trick
levels(my.factor3rev)
## [1] "A" "B"
```

In the last statement, using the unary negation operator, which is vectorized, allows us to easily reverse the ordering of the levels, while still using the default function, `mean()`, to summarize the data.



Reordering factor values. It is possible to arrange the values stored in a factor either alphabetically according to the labels of the levels or according to the order of the levels. (The use of `rep()` is explained on page 8.)

```
# gl() keeps order of levels
my.factor4 <- gl(4, 3, labels = c("A", "F", "B", "Z"))
my.factor4
as.integer(my.factor4)
# factor() orders levels alphabetically
my.factor5 <- factor(rep(c("A", "F", "B", "Z"), rep(3,4)))
my.factor5
as.integer(my.factor5)
levels(my.factor5)[as.integer(my.factor5)]
```

We see above that the integer values by which levels in a factor are stored, are equivalent to indices or “subscripts” referencing the vector of labels. Function `sort()` operates on the values’ underlying integers and sorts according to the order of the levels while `order()` operates on the values’ labels and returns a vector of indices that arrange the values alphabetically.

```
sort(my.factor4)
my.factor4[order(my.factor4)]
my.factor4[order(as.integer(my.factor4))]
```

Run the examples in the chunk above and work out why the results differ.

1.13 Further reading

For further reading on the aspects of R discussed in the current chapter, I suggest the books *R Programming for Data Science* (Peng 2022) and *The Art of R Programming: A Tour of Statistical Software Design* (Matloff 2011).



2

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

2.1 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 I describe the new syntax introduced by the most popular 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. While at the current stage of development of base R not breaking existing code has been the priority, several of the still “young” packages in the ‘tidyverse’ have prioritized experimentation with enhanced features over backwards compatibility. The development of some of these packages seems to have emphasized users’ convenience more than reliability. In contrast, the design of package ‘data.table’ prioritizes performance. Because of this, I do not describe in depth the “new grammars” 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 in the R implementation.

2.2 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 theoretical abstraction, or “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.

Another recent development is the ‘tidyverse’, which is a formidable effort to redefine how data analysis operations are expressed in R code and scripts. In many ways it is a new abstraction, or “grammar of data”. With respect to its implementation, it can also be seen as a new language built on-top of the R language. It is still young and evolving, and the developers from Posit still remain relentless about fixing what they consider earlier misguided decisions in the design of the packages comprising the ‘tidyverse’. This is a wise decision for the future, but can be annoying to occasional users who may not be aware of the changes done. As a user I highly value long-term stability and backwards compatibility of software. Older systems like base R provide this, but their long development history shows up as occasional inconsistencies and quirks. The ‘tidyverse’ as a paradigm is nowadays popular among data analysts while among users for whom data analysis is not the main focus, it is more common to make use of only individual packages as the need arises, e.g., using the new grammar only for some stages of the data analysis work flow.

When a computation included a chain of sequential operations, until R 4.1.0, using base R by itself 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 variables 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 (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. In year 2021 a pipe operator was added to the R language itself and more recently its features enhanced.

In much of my work I emphasize reproducibility and reliability, preferring base R over extension packages, except for plotting, whenever practical. For run once and delete or quick-and-dirty data analyses I tend to use the *tidyverse*. However, with modern computers and some understanding of what are the performance bottlenecks in R code, I have rarely found it worthwhile the effort needed for improved performance by using extension packages. The benefit to effort balance will be different for those readers who analyze huge data sets.

The definition of the *tidyverse* is rather vague, as 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 R scripts.

 In any design, there is a tension between opposing goals. In software for data analysis a key pair of opposed goals are usability, including concise but expressive code, and avoidance of ambiguity. 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, enhancing usability, 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 `[]` (see section 1.10 on page 40). 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’, and much of the ‘tidyverse’, relies on a similar approach as `subset` to enhance convenience at the expense of ambiguity. Package ‘dplyr’ has undergone 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. I do appreciate the advantages of the grammar of data that is implemented in the ‘tidyverse’. However, the actual implementation, can result in ambiguities and subtleties that are even more difficult to deal by inexperienced or occasional users than those caused by inconsistencies in base R. 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 thor-

oughly test and update regularly, the conciseness and power of the new syntax can be an advantage.

i Package ‘poorman’ re-implements many of the functions in ‘dplyr’ and a few from ‘tidyr’ using pure R code instead of compiled C++ code and with no dependencies on other extension packages. This light-weight approach can be useful when R’s data frames rather than tibbles are preferred or when the possible enhanced performance with large data sets is not needed.

2.3 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)
```

2.4 Replacements for `data.frame`

2.4.1 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.

2.4.2 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.

i 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
```

2.5 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.

2.5.1 '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 `%>%`.

2.5.2 '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 ‘wrpr’ 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 ‘wrpr’, 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
```

2.5.3 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.

2.6 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()` from ‘tidyr’ converts data from wide form into long form (or “tidy”). We use it here 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 = -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
```

i Differently to base R, in most functions from the ‘tidyverse’ packages we can use bare column names preceded by a minus sign to signify “all other columns”.


Function `pivot_wider()` does not directly implement the exact inverse operation of `pivot_longer()`. With multiple rows with shared codes, i.e., replication, in our case within each species and flower part, the returned tibble has columns that are lists of vectors. We need to expand these columns with function `unnest()` in a second step.


```
wide_iris.tb <-
  pivot_wider(long_iris.tb,
    names_from = "part",
    values_from = "dimension",
    values_fn = list) |>
  unnest(cols = -Species)
wide_iris.tb
## # A tibble: 150 x 5
##   Species Sepal.Length Sepal.Width Petal.Length Petal.Width
##   <fct>      <dbl>      <dbl>      <dbl>      <dbl>
## 1 setosa      5.1        3.5        1.4        0.2
## 2 setosa      4.9         3         1.4        0.2
## 3 setosa      4.7        3.2        1.3        0.2
## # i 147 more rows
```

i Is `wide_iris.tb` equal to `iris.tb`, the tibble we converted into long shape and back into wide shape? Run the comparisons below, and print the tibbles to find out.

```
identical(iris.tb, wide_iris.tb)
all.equal(iris.tb, wide_iris.tb)
all.equal(iris.tb, wide_iris.tb[, colnames(iris.tb)])
```

What has changed? Would it matter if our code used indexing with a numeric vector to extract columns? or if it used column names as character strings?

 Starting from version 1.0.0 of 'tidyr', functions `gather()` and `spread()` are deprecated and replaced by functions `pivot_longer()` and `pivot_wider()`. These new functions, described above, use a different syntax than the old ones.

 Functions `pivot_longer()` and `pivot_wider()` from package 'poorman' attempt to replicate the behaviour of the same name functions from package 'tidyr'. In some edge cases, the behaviour differs. Test if the two code chunks above return identical or equal values when `poorman::` is prepended to the names of these two functions. First, ensure that package 'poorman' is installed, then run the code below.

```
poor_long_iris.tb <-
  poorman::pivot_longer(
    iris,
    cols = -Species,
    names_to = "part",
    values_to = "dimension")
identical(long_iris.tb, poor_long_iris.tb)
all.equal(long_iris.tb, poor_long_iris.tb)
class(long_iris.tb)
class(poor_long_iris.tb)
```

What is the difference between the values returned by the two functions? Could switching from package 'tidyr' to package 'poorman' affect code downstream of pivoting?


2.7 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` (package 'dtplyr') and for SQL databases (package 'dbplyr'), with consistent syntax (see also section ?? on page ??). 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 use.

2.7.1 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 (see page ??) 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 2.5 on page 72).

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
```


2.7.2 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.

2.7.3 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(id = c(1:4, 5), values1 = "a")
second.tb <- tibble(id = 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
```

2.8 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 et al. 2023) and *R Programming for Data Science* (Peng 2022) are the most useful.

Bibliography

- Aiken, H., A. G. Oettinger, and T. C. Bartee (Aug. 1964). “Proposed automatic calculating machine”. In: *IEEE Spectrum* 1.8, pp. 62–69. DOI: 10.1109/mspec.1964.6500770.
- 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. 64).
- Matloff, N. (2011). *The Art of R Programming: A Tour of Statistical Software Design*. No Starch Press, p. 400. ISBN: 1593273843 (cit. on pp. 61, 64).
- Peng, R. D. (2022). *R Programming for Data Science*. Leanpub. 182 pp. URL: <https://leanpub.com/rprogramming> (visited on 07/27/2023) (cit. on pp. 61, 88).
- Wickham, H., M. Cetinkaya-Rundel, and G. Grolemund (2023). *R for Data Science. Import, Tidy, Transform, Visualize, and Model Data*. O’Reilly Media. ISBN: 9781492097402 (cit. on p. 88).



General Index

- algebra of sets, 31
- arithmetic overflow, 29
 - type promotion, 30
- arrays, 47–54
 - dimensions, 52
- assignment, 4
 - chaining, 5
 - leftwise, 5
- Boolean arithmetic, 21
- C, 15, 27, 28, 39, 41
- C++, 15, 19, 41, 66
- categorical variables, *see* factors
- chaining statements with *pipes*, 72–77
- character escape codes, 15
- character string delimiters, 15
- character strings, 14
 - number of characters, 15
 - partial matching and substitution, 18
 - position-based operations, 17
 - splitting of, 20
 - whitespace trimming, 17
- classes and modes
 - character, 14–21
 - logical, 21–24
 - numeric, integer, double, 2–14
- comparison of floating point numbers, 30–31
- comparison operators, 24–31
- data frame
 - replacements, 66–72
- data manipulation in the tidyverse, 79–87
- ‘data.table’, 63, 64, 66, 67
- ‘dbplyr’, 79
- deleting objects, *see* removing objects
- dot-pipe operator, 73
- double precision numbers
 - arithmetic, 27–30
- ‘dplyr’, 65, 66, 73, 79–81, 84, 86
- ‘dtplyr’, 79
- EPS (ε), *see* machine arithmetic precision
- factors, 55–61
 - arrange values, 60
 - convert to numeric, 59
 - drop unused levels, 58
 - labels, 57
 - levels, 57
 - merge levels, 58
 - ordered, 56
 - reorder levels, 59
 - reorder values, 60
- floating point numbers
 - arithmetic, 27–30
- floats, *see* floating point numbers
- formatted character strings from numbers, 38
- further reading
 - new grammars of data, 88
 - using the R language, 61
- group-wise operations on data, 82–84
- grouping
 - implementation in tidyverse, 83
- inequality and equality tests, 30–31
- integer numbers
 - arithmetic, 27–30
- Integer numbers and computers, 27

- joins between data sources, 84–87
 - filtering, 86
 - mutating, 84
- languages
 - C, 15, 27, 28, 39, 41
 - C++, 15, 19, 41, 66
 - natural and computer, 2
 - Perl, 19
 - S, 77
- logical operators, 21
- logical values and their algebra, 21–24
- long-form- and wide-form tabular data, 77
- loss of numeric precision, 30
- machine arithmetic
 - precision, 27–30
 - rounding errors, 27
- ‘magrittr’, 64, 73–76
- math functions, 2
- math operators, 2
- matrices, 47–55
- matrix
 - dimensions, 52
 - multiplication, 55
 - operations with vectors, 54
 - operators, 54–55
 - transpose, 54
- ‘matrixStats’, 55
- merging data from two tibbles, 84–87
- numbers
 - double, 12
 - floating point, 12
 - integer, 12
 - whole, 12
- numbers and their arithmetic, 2–14
- numeric values, 2
- numeric, integer and double values, 5
- objects
 - mode, 36
- operators
 - comparison, 24–31
 - set, 31–36
- overflow, *see* arithmetic overflow
- packages
 - ‘data.table’, 63, 64, 66, 67
 - ‘dbplyr’, 79
 - ‘dplyr’, 65, 66, 73, 79–81, 84, 86
 - ‘dtplyr’, 79
 - ‘magrittr’, 64, 73–76
 - ‘matrixStats’, 55
 - ‘poorman’, 66, 79
 - ‘reshape’, 77
 - ‘reshape2’, 77
 - ‘stringr’, 80
 - ‘tibble’, 65, 67
 - ‘tidyr’, 65, 66, 77–79, 84
 - ‘tidyverse’, 63–65, 67, 71, 72, 75, 77–79, 81, 88
 - ‘wrappR’, 64, 73, 74, 76
- Perl, 19
- pipe operator, 73
- pipes
 - expressions in rhs, 74
 - tidyverse, 73
 - wrappR, 73–77
- ‘poorman’, 66, 79
- precision
 - math operations, 12
- programmes
 - RStudio, 10
- Real numbers and computers, 27
- recycling of arguments, 8
- regular expressions, 19–21
- removing objects, 10
- ‘reshape’, 77
- ‘reshape2’, 77
- reshaping tibbles, 77–79
- row-wise operations on data, 79–82
- RStudio, 10
- S, 77
- sequence, 8
- sets, 31–36
- special values
 - NA, 10
 - NaN, 10
- ‘stringr’, 80
- tibble

- differences with data frames,
 - 67-72
- 'tibble', 65, 67
- 'tidyr', 65, 66, 77-79, 84
- 'tidyverse', 63-65, 67, 71, 72, 75,
 - 77-79, 81, 88
- type conversion, 37-40
- type promotion, 30
- variables, 4
- vector
 - run length encoding, 47
- vectorized arithmetic, 8
- vectors
 - indexing, 40-47
 - introduction, 6-10
 - member extraction, 40
 - named elements, 43
 - sorting, 46
 - zero length, 11
- 'wrapr', 64, 73, 74, 76
- zero length objects, 11



Alphabetic Index of R Names

`*`, 2, 30
`+`, 2, 14
`-`, 2, 14
`->`, 5
`-Inf`, 10, 11, 29
`.Machine$double.eps`, 29
`.Machine$double.max`, 29
`.Machine$double.min`, 29
`.Machine$double.neg.eps`, 29
`.Machine$integer.max`, 29
`/`, 2
`:`, 8
`<`, 24
`<-`, 4, 5, 44
`<=`, 24
`=`, 5
`==`, 24, 84
`>`, 24
`>=`, 24
`[` , `]`, 65, 84
`[[` `]`, 65
`%*%`, 55
`%.>%`, 73–76, 80
`%/%`, 12
`%<>%`, 73
`%>%`, 73–76
`%T>%`, 73
`%`, 12
`%in%`, 32, 33, 35
`&`, 22, 26
`&&`, 22
`^`, 30
`|`, 22, 26
`|>`, 73–76
`||`, 22

`abs()`, 14, 30
`aggregate()`, 82
`all()`, 23

`anti_join()`, 87
`any()`, 23
`append()`, 7
`arrange()`, 80
`array`, 47
`array()`, 53
`as.character()`, 37, 59
`as.data.frame()`, 70
`as.integer()`, 38
`as.logical()`, 37
`as.matrix()`, 48
`as.numeric()`, 37, 38, 59
`as.vector()`, 54
`as_tibble()`, 68
`assign()`, 76

`c()`, 6
`cat()`, 15
`ceiling()`, 14
`character`, 14, 17, 35, 38
`charmatch()`, 33
`class()`, 36, 70
`contains()`, 81
`crossprod()`, 55

`data.frame`, 67
`data.frame()`, 68, 72
`diag()`, 55
`dim()`, 48, 52
`double`, 5, 27, 29, 30, 45
`double()`, 5
`duplicated()`, 34

`ends_with()`, 81
`exp()`, 3

`factor`, 55
`factor()`, 56–59
`FALSE`, 12
`filter()`, 81

`format()`, 38, 39
`full_join()`, 84

`gather()`, 79
`gl()`, 56
`grep()`, 18
`grep1()`, 18
`group_by()`, 82, 84
`gsub()`, 18, 19

`I()`, 72
`identical()`, 70
`Inf`, 10, 11, 29
`inherits()`, 36
`inner_join()`, 84
`integer`, 5, 12, 27, 29, 45
`iris`, 77
`is.array()`, 52
`is.character()`, 36
`is.element()`, 32, 33
`is.logical()`, 36
`is.matrix()`, 48
`is.na()`, 12
`is.numeric()`, 5, 36
`is_tibble()`, 68

`left_join()`, 84
`length()`, 6, 9, 38, 48
`LETTERS`, 41
`letters`, 41
`levels()`, 58, 59
`list`, 67
`log()`, 3
`log10()`, 3
`log2()`, 3
`logical`, 21, 22, 26, 40
`ls()`, 10

`match()`, 33
`matches()`, 81
`matrix`, 47, 52, 67
`matrix()`, 48, 49
`mode()`, 36
`month.abb`, 41
`month.name`, 41
`mutate()`, 80

`NA`, 10, 11, 40
`NA_character_`, 40

`NA_real_`, 40
`names()`, 82
`names()<-`, 82
`NaN`, 10
`nchar()`, 15
`ncol()`, 48
`nrow()`, 48
`numeric`, 2, 5, 35, 45
`numeric()`, 5, 9, 10

`options()`, 69
`order()`, 47, 61, 80
`ordered()`, 56

`paste()`, 16
`pi`, 3
`pivot_longer()`, 78, 79
`pivot_wider()`, 78, 79
`plot()`, 77
`pmatch()`, 33
`print()`, 15, 38, 69, 75

`read.table()`, 68
`rename()`, 82
`reorder()`, 60
`rep()`, 8
`rev()`, 59
`right_join()`, 84
`rle()`, 47
`rm()`, 10
`round()`, 13

`select()`, 81
`semi_join()`, 86
`seq()`, 8
`signif()`, 13
`sin()`, 3
`slice()`, 81
`sort()`, 46, 47, 61, 80
`spread()`, 79
`sprintf()`, 38, 39
`sqrt()`, 3
`starts_with()`, 81
`str_extract()`, 80
`strsplit()`, 21
`sub()`, 18
`subset()`, 65, 81
`substr()`, 17
`substring()`, 17

summarise(), 82	TRUE, 12
t(), 54	trunc(), 14, 38
tbl, 67	typeof(), 36
tbl_df, 68	ungroup(), 82, 84
tibble, 68, 79	unique(), 34
tibble(), 68, 71, 72, 80	unlist(), 54
transmute(), 80	unnest(), 78
trimstr(), 16	vector, 6
trimws(), 17	



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

- array, 47
- character, 14, 17, 35, 38
- data.frame, 67
- double, 5, 27, 29, 30, 45
- factor, 55
- integer, 5, 12, 27, 29, 45
- list, 67
- logical, 21, 22, 26, 40
- matrix, 47, 52, 67
- numeric, 2, 5, 35, 45
- tbl, 67
- tbl_df, 68
- tibble, 68, 79
- vector, 6

constant and special values

- Inf, 10, 11, 29
- .Machine\$double.eps, 29
- .Machine\$double.max, 29
- .Machine\$double.min, 29
- .Machine\$double.neg.eps, 29
- .Machine\$integer.max, 29
- FALSE, 12
- Inf, 10, 11, 29
- LETTERS, 41
- letters, 41
- month.abb, 41
- month.name, 41
- NA, 10, 11, 40
- NA_character_, 40
- NA_real_, 40
- NaN, 10
- pi, 3
- TRUE, 12

data objects

- iris, 77

functions and methods

- abs(), 14, 30
- aggregate(), 82
- all(), 23
- anti_join(), 87
- any(), 23
- append(), 7
- arrange(), 80
- array(), 53
- as.character(), 37, 59
- as.data.frame(), 70
- as.integer(), 38
- as.logical(), 37
- as.matrix(), 48
- as.numeric(), 37, 38, 59
- as.vector(), 54
- as_tibble(), 68
- assign(), 76
- c(), 6
- cat(), 15
- ceiling(), 14
- charmatch(), 33
- class(), 36, 70
- contains(), 81
- crossprod(), 55
- data.frame(), 68, 72
- diag(), 55
- dim(), 48, 52
- double(), 5
- duplicated(), 34
- ends_with(), 81
- exp(), 3

factor(), 56-59
 filter(), 81
 format(), 38, 39
 full_join(), 84
 gather(), 79
 gl(), 56
 grep(), 18
 grepl(), 18
 group_by(), 82, 84
 gsub(), 18, 19
 I(), 72
 identical(), 70
 inherits(), 36
 inner_join(), 84
 is.array(), 52
 is.character(), 36
 is.element(), 32, 33
 is.logical(), 36
 is.matrix(), 48
 is.na(), 12
 is.numeric(), 5, 36
 is_tibble(), 68
 left_join(), 84
 length(), 6, 9, 38, 48
 levels(), 58, 59
 log(), 3
 log10(), 3
 log2(), 3
 ls(), 10
 match(), 33
 matches(), 81
 matrix(), 48, 49
 mode(), 36
 mutate(), 80
 names(), 82
 names()<-, 82
 nchar(), 15
 ncol(), 48
 nrow(), 48
 numeric(), 5, 9, 10
 options(), 69
 order(), 47, 61, 80
 ordered(), 56
 paste(), 16
 pivot_longer(), 78, 79
 pivot_wider(), 78, 79
 plot(), 77

pmatch(), 33
 print(), 15, 38, 69, 75
 read.table(), 68
 rename(), 82
 reorder(), 60
 rep(), 8
 rev(), 59
 right_join(), 84
 rle(), 47
 rm(), 10
 round(), 13
 select(), 81
 semi_join(), 86
 seq(), 8
 signif(), 13
 sin(), 3
 slice(), 81
 sort(), 46, 47, 61, 80
 spread(), 79
 sprintf(), 38, 39
 sqrt(), 3
 starts_with(), 81
 str_extract(), 80
 strsplit(), 21
 sub(), 18
 subset(), 65, 81
 substr(), 17
 substring(), 17
 summarise(), 82
 t(), 54
 tibble(), 68, 71, 72, 80
 transmute(), 80
 trimstr(), 16
 trimws(), 17
 trunc(), 14, 38
 typeof(), 36
 ungroup(), 82, 84
 unique(), 34
 unlist(), 54
 unnest(), 78


operators

*, 2, 30
 +, 2, 14
 -, 2, 14
 ->, 5
 /, 2
 :, 8

<, 24	%<>%, 73
<-, 4, 5, 44	%>%, 73-76
<=, 24	%T>%, 73
=, 5	%%, 12
==, 24, 84	%in%, 32, 33, 35
>, 24	&, 22, 26
>=, 24	&&, 22
[,], 65, 84	^, 30
[[]], 65	, 22, 26
%*%, 55	>, 73-76
%.>%, 73-76, 80	, 22
%/%, 12	



Frequently Asked Questions

Frequently asked questions and their answers appear in the body of the book preceded by the icon  and highlighted by a marginal bar of the same colour as the icon.

How to access the last value in a vector?, 41

How to check if a vector contains one or more **NA** (or **NaN**) values?, 24

How to check if an entire vector contains no values other than **NA** (or **NaN**) values?,
24