3

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

3.1 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 R *console*. This teaches not only the syntax and grammar rules, but also gives a glimpse at the advantages and flexibility of this approach to data analysis. In this chapter I focus on the different simple values or items that can be stored and manipulated in R, as well as the role of computer program statements, the equivalent of "sentences" in natural languages.

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 simple "words" used in the R language and you will be able to read and write simple R statements.

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 the mathematics, as long as you understand the diagrams and the R code.

3.2 Natural and computer languages

Computer languages have strict rules and interpreters and compilers that translate these languages into machine code 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 on 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`.

3.3 Numeric values and arithmetic

When working in `R` with arithmetic expressions, the normal mathematical precedence rules are followed and 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 $1 + 2 \times 3$, the product 2×3 has precedence over the addition, and is evaluated first, yielding as the result of the whole expression, 7. Similar rules apply to other operators, even those taking as operands non-numeric values.

The equivalent of the math expression

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

is, in R, written as follows:

```
(3 + exp(2)) / cos(pi)
## [1] -10.38906
```

Where constant `pi` ($\pi = 3.1415\ldots$) and function `cos()` (cosine) are defined in base R. Many trigonometric and mathematical functions are available in addition to operators like `+`, `-`, `*`, `/`, and `^`.

In R angles are expressed in radians, thus $\cos(\pi) = 1$ and $\sin(\pi) = 0$, according to trigonometry. Degrees can be converted into radians taking into account that the circle corresponds to $2 \times \pi$ when expressed in radians and to 360° when expressed in degrees. Thus the cosine of an agle of 45° can be computed as follows.

```
sin(45/180 * pi)
## [1] 0.7071068
```

One thing to remember when translating fractions 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.

3.1 In *playgrounds* the output from running the code in R are not shown, as these are exercises for you to enter at the R console and run. In general you should not skip them as in most cases playgrounds aim to teach or demonstrate concepts or features that I have *not* included in full-detail in the main text. 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
sin(pi)
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, `vct1` and `vct2` are arbitrarily chosen variable names; I should have used names like `height.cm` or `outside.temperature.C` if they had been useful to convey information.

In the book, I use variable names that help recognize the kind of object stored, as this is most relevant when learning R. Here I use `vct1` because in R, as we will see in page 28, numeric objects are always vectors, even when of length one.

```
vct1 <- 1
vct1 + 1
## [1] 2
vct1
## [1] 1
vct2 <- 10
vct2 <- vct1 + vct2
vct2
## [1] 11
```

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.

```
vct1
## [1] 1
print(vct1)
## [1] 1
vct1 + 1
## [1] 2
print(vct1 + 1)
## [1] 2
```

3.2 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 -> vct1` 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 `vct1`, `vct2` and `vct3` are being assigned the same value.

```
vct1 <- vct2 <- vct3 <- 0
vct1
vct2
vct3
1 -> vct1
vct1
vct1 = 3
vct1
remove(vct1, vct2, vct3) # cleanup
```

In R, all numbers belong to mode `numeric` (we will discuss the concepts of *mode* and *class* in section 3.8 on page 58). We can query if the mode of an object is `numeric` with function `is.numeric()`. The returned values are either TRUE or FALSE. These are logical values that will be discussed in section 3.5 on page 48.

```
mode(1)
## [1] "numeric"
vct1 <- 1
is.numeric(vct1)
## [1] TRUE
```

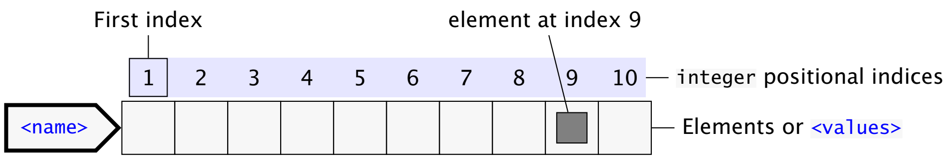
Because numbers can be stored in different formats, most computing languages implement several different types of numbers. In most cases R’s `numeric` values can be used everywhere that a number is expected. However, in some cases it has advantages to explicitly indicate that we will store or operate 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
```

Vectors are one-dimensional in structure, 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 $\vec{\mathbf{F}}$. In R numeric values and other atomic values are always `vector` s 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 3.10 on page 63.



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

$$a_{1\ldots n} = a_1, a_2, \cdots a_i, \cdots, a_n, \qquad (3.1)$$

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

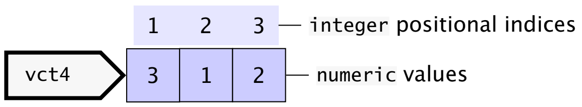
As you have seen above, the results of calculations were printed preceded with `[1]`. This is the index or position in the vector of the first number (or other value) displayed at the head of the current line. As in R single values are vectors of length one, when they are printed, they are also preceded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

One can use function `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.

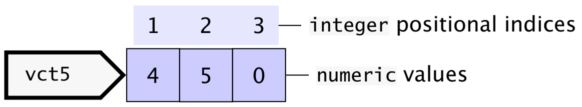
```
c(3, 1, 2)
## [1] 3 1 2
```

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 function `length()`. We show below R code followed by diagrams depicting the structure of the vectors created.

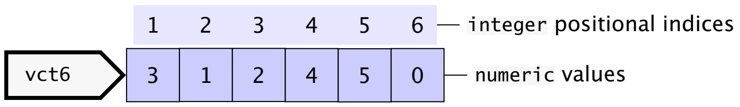
```
vct4 <- c(3, 1, 2)
length(vct4)
## [1] 3
vct4
## [1] 3 1 2
```



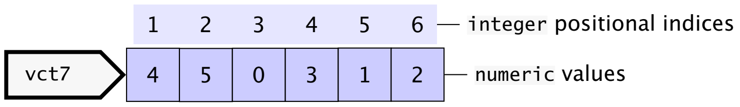
```
vct5 <- c(4, 5, 0)
vct5
## [1] 4 5 0
```



```
vct6 <- c(vct4, vct5)
vct6
## [1] 3 1 2 4 5 0
```



```
vct7 <- c(vct5, vct4)
vct7
## [1] 4 5 0 3 1 2
```



How to create an empty vector?

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

Next I show concatenation with a vector of the same class with length zero.

```
c(vct7, numeric())
## [1] 4 5 0 3 1 2
```

Function `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(vct4, vct5)
## [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(vct4, values = vct5, after = 2)
## [1] 3 1 4 5 0 2
```

3.3 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)`.

```
-1:5
5:-1
seq(from = -1, to = 1, by = 0.1)
rep(-5, times = 4)
rep(1:2, length.out = 4)
```

How to create a vector of zeros?

```
numeric(length = 10)
##  [1] 0 0 0 0 0 0 0 0 0 0
```

or

```
rep(0, times = 10)
##  [1] 0 0 0 0 0 0 0 0 0 0
```

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.

```
vct4 + 1 # we add one to vector a defined above
## [1] 4 2 3
(vct4 + 1) * 2
## [1] 8 4 6
vct4 + vct5
## [1] 7 6 2
vct4 - vct5
## [1] -1 -4  2
```

As it can be seen in the first line of code above, another peculiarity of R, is what is frequently called "recycling" of vector arguments: as vector `vct4` is of length 3, but the `numeric` constant 1 is a vector of length 1, this short constant vector is extended, by recycling (replicating) its value, into a longer vector of 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. Vector recycling is a key feature of the R language. See the box on page 34 for an in-depth explanation of vector recycling.

```
vct8 <- rep(1, 6)
vct8
## [1] 1 1 1 1 1 1
vct8 + 1:2
## [1] 2 3 2 3 2 3
vct8 + 1:3
## [1] 2 3 4 2 3 4
vct8 + 1:4

## Warning in vct8 + 1:4: longer object length is not a multiple of shorter
object length
## [1] 2 3 4 5 2 3
```

3.4 Create further variants of the statements in the code chunk above to work out when a warning is issue and if in any case an error is triggered because of the length of the operands.

Most functions defined in base R apply recycling to vectors passed as argument to at least some of their parameters. When recycling is supported, the conditions triggering warnings or errors are consistent with those you discovered in the playground above. However, if and how recycling is applied depends on how functions have been defined. Thus, there is variation, specially, but not only, in the case of functions and operators defined in contributed extension packages. For example, package 'tibble' and some other packages in the 'tidyverse' support recycling but some boundary cases that trigger a warning in base R functions, trigger an error in functions defined in these packages. See section 8.4.2 on page 241 about package 'tibble'.

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.

```
vct9 <- numeric(length = 0) # named argument
vct9
## numeric(0)
length(vct8)
## [1] 6
```

```
numeric() # default argument
## numeric(0)
```

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

```
length(c(vct4, vct9, vct5))
## [1] 6
length(c(vct4, vct5))
## [1] 6
```

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 (`character` values are described in section 3.4 on page 40).

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

```
numeric(length = 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.

```
vct10 <- 1:5
length(vct10) <- 10
vct10
##  [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.

```
vct11 <- 1:10
vct11
##  [1]  1  2  3  4  5  6  7  8  9 10
length(vct11) <- 5
vct11
## [1] 1 2 3 4 5
```

There are some special values available for numbers. `NA` meaning “not available” is used for missing values. (`NA`) values play a very important role in the analysis of data, as frequently some observations are missing from an otherwise complete data set due to “accidents” during the course of an experiment or survey. It is important to understand how to interpret `NA` values: They are placeholders for something that is unavailable, in other words, whose value is *unknown*. `NA` values propagate when used, so that numerical computations yield `NA` when one or more input of the values is unknown.

```
vct12 <- c(NA, 5)
vct12
## [1] NA  5
vct12 + 1
## [1] NA  6
```

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.

```
vct12 + Inf
## [1]  NA Inf
Inf / vct12
## [1]  NA Inf
-1 / 0
## [1] -Inf
1 / 0
## [1] Inf
Inf / Inf
## [1] NaN
Inf + 4
## [1] Inf
-Inf * -1
## [1] Inf
```

3.5 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` values are 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 elements of the vector individually, returning the result as `TRUE` or `FALSE`.

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 34 for an in-depth explanation.

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

When using `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
```

Using the "usual" division operator yields a floating-point `double` result, while the integer division operator `%/%` yields an `integer` result, and the modulo operator `%%` returns the remainder from the integer division.

```
1L / 3L
## [1] 0.3333333
1L %/% 3L
## [1] 0
1L %% 3L
## [1] 1
```

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 functions in the next chunk return `TRUE` or `FALSE`, i.e., `logical` values (see section 3.5 on page 48).

```
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
```

3.6 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
```

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` amd `double` is explained on page 27. 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 or whole numbers 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 symbolized by “epsilon” (ϵ), commonly abbreviated *eps*, defined as the largest value of ϵ for which $1 + \epsilon = 1$. The finite resolution of floats can lead to unexpected results when testing for equality or inequality. Test for equality is done with operator `==`. Use of this and other comparison operators is explained in section 3.6 on page 51.

```
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 maximun 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 algorithms may be preferable, and others even fail, in specific cases.

As the R program can be used 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 the compiler used to build the R program executable. However, it is possible to obtain these values at run time, i.e., while the R is being used, 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. *Beware that when you run the examples below, the values returned by R in your own computer can differ from those returned in the computer I have used to typeset the book as you are reading it here.*

```
.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
.Machine$double.base
## [1] 2
```

The last two values refer to the exponents of a base number or *radix*, 2, rather than the maximum and minimum size of numbers that can be handled as objects of class `double`. The maximum size of normalized `double` values, given by `.Machine$double.xmax`, is much larger than the maximum value of `integer` values, given by `.Machine$integer.max`.

```
.Machine$double.xmax
## [1] 1.797693e+308
.Machine$integer.max
## [1] 2147483647
```

As `integer` values are stored in machine memory without loss of precision, epsilon is not defined for `integer` values. In R not all out-of-range `numeric` values behave in the same way: while off-range `double` values are stored as `-Inf` or `Inf` and enter arithmetic as infinite values according the mathematical rules, off-range `integer` values become `NA` with a warning.

```
1e1026
## [1] Inf
1e-1026
## [1] 0

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
```

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 `integer` values resulting in overflow.

```
2147483600L * 2147483600L

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

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

Both for display 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, with a positive or negative value for `digits`, respectively. In contrast, function `signif()` rounds to the requested number of significant digits, i.e., ignoring the position of the decimal marker.

```
round(0.0124567, digits = 3)
## [1] 0.012
signif(0.0124567, digits = 3)
## [1] 0.0125
round(1789.1234, digits = -1)
## [1] 1790
round(1789.1234, digits = 3)
## [1] 1789.123
signif(1789.1234, digits = 3)
## [1] 1790
```

```
vct13 <- 0.12345
vct14 <- round(vct13, digits = 2)
vct13 == vct14
## [1] FALSE
vct13 - vct14
## [1] 0.00345
vct14
## [1] 0.12
```

Functions are described in detail in section 6.2 on page 165. Here I describe them briefly in relation to their use. Functions are objects containing R code that can be used to perform an operation on data passed as argument to its parameters. They return the result of the operation as a single R object, or less frequently, as a side effect. Functions have a name like any other R object. If the name of a function followed by parentheses `()` and included in a code statement, it becomes a function *call* or a “request” for the code stored in the function object to be run. Many functions, accept R objects and/or constant values as *arguments* to their *formal parameters*. Formal parameters are placeholder names in the code stored in the function object, or the *definition* of the function. In a function call the code in its definition is evaluated (or run) with formal-parameter names taking the values passed as arguments to them.

In a function definition formal parameters can be assigned default values, which are used if no explicit argument is passed in the call. Arguments can be passed to formal parameters by name or by position. In most cases, passing arguments by name makes the code easier to understand and more robust against coding mistakes. In the examples in the book I most frequently pass arguments by name, except for the first parameter.

Being `digits`, the second parameter, its argument can also be passed by position.

```
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. Hint: you can use `help(trunc)` or `trunc?` at the R console, or the help tab of RStudio to find out the answer.

3.7 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?

R supports complex numbers and arithmetic operations with class `complex`. As complex numbers rarely appear in user-written scripts I give only one example of their use. Complex numbers as defined in mathematics, have two parts, a real component and an imaginary one. Complex numbers can be used, for example, to describe the result of $\sqrt{-1} = 1i$.

```
cmp1 <- complex(real = c(-1, 1), imaginary = c(0, 0))
cmp1
## [1] -1+0i  1+0i
cmp2 <- sqrt(cmp1)
cmp2
## [1] 0+1i 1+0i
cmp2^2
## [1] -1+0i  1+0i
```

Instants in time and periods of time in computers are usually encoded as classes derived from `integer`, and thus considered in R as atomic classes and the objects vectors. Some of these encodings are standardized and supported by R classes `POSIXlt` and `POSIXct`. The computations based on times and dates are difficult because the relationship between local time at a given location and Universal Time Coordinates (UTC) has changed in time, as well as with changes in national borders. Packages 'lubridate' and 'anytime' support operations among time-related data and conversions between character strings and time and date classes easier and less error prone than when using base R functions. Thus I describe classes and operations related to dates and times in chapter 8 on page 237.

It is good to *remove* from the workspace objects that are no longer needed. We use function `remove()` to delete objects stored in the current workspace.

Arguments passed to `remove()` can be bare object names as shown here.

```
an.object <- 1:4
remove(an.object) # using a bare name
```

Function `remove()` also accepts the names of the objects as `character` strings. In spite of the name, the argument passed to parameter `list` must be a `vector` rather than a `list` (see section 3.4 on `character` and section 4.3 on `list` on pages 40 and 86).

```
an.object <- 5:2
remove(list = "an.object") # using a character vector
```

Function `objects()` returns a `character` vector containing the names of all objects visible in the current environment, or by passing an argument to parameter `pattern`, only the objects with names matching it.

```
an.object <- 1:4
another.object <- 2
objects(pattern = "*.object")
## [1] "an.object"      "another.object"
remove(any.object)

## Warning in remove(any.object): object 'any.object' not found

objects(pattern = "*.object")
## [1] "an.object"      "another.object"
```

In RStudio all objects are listed in the **Environment** tab and the search box of this tab can be used to find a given object.

Function `remove()` accepts both bare names of objects as in the chunk above and `character` strings corresponding to object names like in `remove("any.object")`. However, While `objects()` accept patterns to be matched to object names, `remove()` does not. Because of this, these two functions have to be used together for removing all objects with names that match a pattern. The pattern can be given as a regular expression (see section 3.4 on page 45).

Both functions have are available under short names matching those used in Linux and Unix for managing files: `ls()` is a synonym of `objects()` and `rm()` of `remove()`.

Using a simple pattern we obtain the names of all objects with names `"vct1"`, `"vct2"`, and so on. When using a pattern to remove objects, it is good to first use `objects()` on its own to get a list of the objects that would be deleted by calling `remove()` passing the names returned by `objects()` as argument for parameter `list`.

```
objects(pattern = "^vec.*")
## character(0)
```

The code below removes all objects with names `"vct1"`, `"vct2"`, and so on. We do this at the end of the section before reusing the same names in the code examples of the next section.

```
remove(list = objects(pattern = "^vct[[:digit:]]?"))
```

Similar code chunks are included at the end of each section throughout the book to ensure that code examples are self-contained by section. The chunk about is shown above as an example, but kept hidden in later sections.

3.4 Character values

In spite of the the name `character`, values of this mode, are vectors of *character strings*". Character constants are written by enclosing characters strings in quotation marks, i.e., `"this is a character string"`. There are three types of quotation marks 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.

```
vct1 <- "A"
vct1
## [1] "A"
vct2 <- 'A'
vct2
## [1] "A"
vct1 == vct2 # two variables holding character values, or named objects
## [1] TRUE
"A" == 'A' # two constant character values, or anonymous objects
## [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. 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 of character strings.

```
vct3 <- 'ABC'
vct4 <- "bcdefg"
vct5 <- c("123", "xyz")
c(vct3, vct4, vct5)
## [1] "ABC"    "bcdefg" "123"    "xyz"
```

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

```
"He said 'hello' when he came in"
## [1] "He said 'hello' when he came in"
'He said "hello" when he came in'
## [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 codes”. 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.

```
vct6 <- "abc\ndef\tx\"yz\"\\\tm"
print(vct6)
## [1] "abc\ndef\tx\"yz\"\\\tm"
cat(vct6)
## abc
## def x"yz"\ m
```

The *escape codes* work only in some contexts, as when using `cat()` to generate the output.

How to find the length of a character string?
While function `length()` returns the number of member `character` strings in a vector, function `nchar()` returns the number of characters in each string in the vector (see below for examples).

In the example below, function `nchar()` returns the number of characters in each member string.

```
nchar(x = "abracadabra")
## [1] 11
nchar(x = c("abracadabra", "workaholic", ""))
## [1] 11 10  0
```

To convert a string into upper case or lower case we use functions `toupper()` and `tolower()`, respectively.

```
toupper(x = "aBcD")
## [1] "ABCD"
tolower(x = "aBcD")
## [1] "abcd"
```

Function `strtrim()` trims a string to 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"
```

How to Wrap long strings?
Use R function `strwrap()` (see below for examples).

Function `strwrap()` edits a string to a maximum number of characters or width, by inserting new line characters.

```
strwrap(x  =  "This  is  a  long  sentence  used  to  show  how  line  wrap-
ping works.", width = 20)
## [1] "This is a long"   "sentence used to" "show how line"    "wrapping works."
```

3.8 Function `cat()` prints a character vector respecting the embedded special characters such new line (encoded as `\n`) in `character` strings) and without issuing any additional new lines. Study the code below and the output it generates, consult the documentation of the two functions, and modify the example code until you are confident that you understand in detail how these tow functions work.

```
wrapped_sentence <-
strwrap(x = "This is a very long sentence used to show how line wrapping works.",
        width = 10,
        prefix = "\n")
print(wrapped_sentence)
cat(wrapped_sentence, "\n")
```

How to create a single character string from multiple shorter strings?
While function `c()` is used to concatenate `character` vectors into longer vectors, function `paste()` is used to concatenate character strings into a single longer string (see below for examples).

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] "John likes to eat apples."        "Yan likes to eat lichees."
## [3] "Juana likes to eat oranges."      "Mary likes to eat strawberries."
```

3.9 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 = ""` from the statement above and run it to learn the 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. I use here `cat()` so that the newline character is obeyed in the display of the single character string.

```
cat(paste(friends, "likes to eat ", fruits, collapse = ".\n", sep = ""))
## John likes to eat apples.
## Yan likes to eat lichees.
## Juana likes to eat oranges.
## Mary likes to eat strawberries
```

When the vectors are of different length, the shorter one is recycled as many times as needed, which is not always what we want. In this case we need to first collapse the members of the long vector `fruits` to change this vector into a vector of length one. We can achieve this by nesting two calls to `paste()`, and passing an argument to `collapse` in the inner function call.

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

Nesting of function calls is explained in section 5.5 on page 132. However, as the two statements above would in most cases be written as nested function calls, I add this example for reference.

```
paste("My     friends     like     to     eat",     paste(fruits,     col-
lapse = ", "), "and other fruits.")
## [1] "My friends like to eat apples, lichees, oranges, strawberries and other fruits."
```

Function `strrep()` repeats and pastes character strings, while `rep()` repeats character strings into vectors.

```
rep(x = "ABC", times = 3)
## [1] "ABC" "ABC" "ABC"
strrep(x = "ABC", times = 3)
## [1] "ABCABCABC"
strrep(x = "ABC", times = c(2, 4))
## [1] "ABCABC"       "ABCABCABCABC"
strrep(x = c("ABC", "X"), times = 2)
## [1] "ABCABC" "XX"
strrep(x = c("ABC", "X"), times = c(2, 5))
## [1] "ABCABC" "XXXXX"
```

How to trim leading and/or trailing white space in character strings?
Use function `trimws()` (see below for examples).

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

```
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"
```

⌨ **3.10** 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"
```

Replacement is done *in place*, by having function substr() on the left hand side (lhs) of the assignment operator <-. Thus, the argument passed to parameter x of substr() must in this case be a variable rather than a constant. This is a substitution character by character, not insertion, so the number of characters in the string passed as argument to x remains unchanged, i.e., the value returned by nchar() does not change.

```
vct7 <- c("abracadabra", "workaholic")
substr(x = vct7, start = 5, stop = 9) <- "xxx"
vct7
## [1] "abraxxxabra" "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.

```
vct8 <- c("abracadabra", "workaholic")
substr(x = vct8, start = 5, stop = 6) <- "xxx"
vct8
## [1] "abraxxdabra" "workxxolic"
```

⌨ **3.11** 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.

```
VCT1 <- c("abracadabra", "workaholic")
substr(x = VCT1, start = 5, stop = 11) <- "xxx"
VCT1
remove(VCT1) # clean up
```

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 substitute part of a character string *by matching a pattern*, 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"
```

3.12 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 `grepl()` 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 the arguments for `pattern` strings 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 affect 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 including `grep()` and `grepl()` accept regular expressions as patterns.

In 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"
```

3.13 How would you modify the last code example above 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"
```

Several named classes of characters are predefined, for example `[:lower:]` for lower case alphabetic characters according to the current locale (see page 48). In the regular expression in the example below, `[:lower:]` replaces only `a-z`, thus we need to keep the outer square brackets. While `a-z` includes only the unaccented letters, `[:lower:]` does include additional characters such as `ä`, `ö`, or `é` if they are in use in the current locale. In the case of `[:digit:]` and `0-9`, they are equivalent.

```
gsub(pattern = "^.([[:digit:]])[[:lower:]]*$",
     replacement = "gone with \\1",
     x = c("about", "a9out", "a3outx"))
## [1] "about"       "gone with 9" "gone with 3"
```

With parentheses we can isolate part of the matched string and reuse it in the replacement with a numeric back-reference. 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"
```

3.14 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. This list contains as many character vectors as members had the vector passed as argument to `x`, each vector the result of splitting one character string in the input. (Lists are described in section 4.3 on page 86.)

```
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"
```

⚠ The ASCII character set is the oldest and simplest in use. In contains only 128 characters including non-printable characters. These characters support the English language. Several different extended versions with 256 characters provided support for other languages, mostly by adding accented letters and some symbols. The 128 ASCII characters were for a long time the only consistently available across computers set up for different languages and countries (or *locales*). Recently the use of much larger character sets like UTF8 has become common. Since R version 4.2.0 support for UTF8 is available under Windows 10. This makes it possible the processing of text data for many more languages than in the past. Even though now it is possible to use non-ASCII characters as part of object names, it is anyway safer to use only ASCII characters as this support is recent.

The extended character sets include additional characters, that are distinct but may produce glyphs that look very similar to those in the ASCII set. One case are em-dash (—), en-dash (-), minus sign (−) and regular dash (-) which are all different characters, with only the last one recognized by R as the minus operator. For those copying and pasting text from a word-processor into R or RStudio, a frequent difficulty is that even if one types in an ASCII quote character ("), the opening and closing quotes in many languages are automatically replaced with non-ASCII ones (“and”) which R does not accept as character string delimiters. The best solution is to use a plain text editor instead of a word processor when writing scripts or editing text files containing data to be read as code statements or numerical data.

A locale definition determines not only the language, and character set, but also date, time, and currency formats.

3.5 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 rarely used to store data from experiments or surveys. They are used mostly to keep track of binary conditions, like results from comparisons in a script and to operate on them. Most frequent uses of `logical` values do not involve their storage in user-created variables. Most comparisons or tests return a `logical` value and Boolean algebra makes it possible to combine the results from multiple tests or conditions into a single combined outcome or binary decision, i.e., TRUE or False, Yes or No. (See section 3.6 on page 51 for examples.)

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.

```
vct1 <- TRUE
mode(vct1)
## [1] "logical"
vct1
## [1] TRUE
!TRUE # negation
## [1] FALSE
TRUE && FALSE # logical AND
## [1] FALSE
TRUE || FALSE # logical OR
## [1] TRUE
xor(TRUE, FALSE) # exclusive OR
## [1] TRUE
```

The availability of two kinds of logical operators can be troublesome for those new to `R`. Pairs of "equivalent" logical operators behave differently, use similar syntax and use similar symbols! The vectorized operators have single-character names, `&` and `|` (similarly to vectorized arithmetic operators like `+`), 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. In some situations, vectorized `logical` operators can replace non-vectorized ones, but it is important to use the ones that match the intention of the code, as this enables relevant checks for mistakes. Once the distinction is learnt, using the most appropriate operators also contributes to make code easier to read.

```
c(TRUE, FALSE) & c(TRUE,TRUE) # vectorized AND
## [1]  TRUE FALSE
c(TRUE, FALSE) | c(TRUE,TRUE) # 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`.

```
vct2 <- c(TRUE, FALSE, FALSE)
any(vct2)
## [1] TRUE
all(vct2)
```

```
## [1] FALSE
any(c(TRUE, FALSE) & c(TRUE,TRUE))
## [1] TRUE
all(c(TRUE, FALSE) & c(TRUE,TRUE))
## [1] FALSE
any(c(TRUE, FALSE) | c(TRUE,TRUE))
## [1] TRUE
all(c(TRUE, FALSE) | c(TRUE,TRUE))
## [1] TRUE
```

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
```

3.15 Investigate how swapping the order of the operands in the code chunk above affects the values returned, e.g.., the first statement becomes `NA || TRUE`.

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.

```
c(TRUE, FALSE) & c(TRUE,TRUE) & NA
## [1]    NA FALSE
c(TRUE, FALSE) & c(TRUE,TRUE) & c(NA, NA)
## [1]    NA FALSE
c(TRUE, FALSE) | c(TRUE,TRUE) | c(NA, NA)
## [1] TRUE TRUE
```

3.16 Based on the description of "recycling" presented on page 30 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!

```
c(TRUE, FALSE, TRUE, NA) & FALSE
c(TRUE, FALSE, TRUE, NA) | c(TRUE, FALSE)
```

How to test if a vector contains no values other than `NA` (or `NaN`) values?
A call to `is.na()` returns a `logical` vector that we can pass to `all()`. We can save the intermediate vector `temp` and pass it as argument to `is.na()`, or alternatively nest the function calls. The name `tmp`, for *temporary*, is frequently used for variables whose value is retrieved only once.

```
vct2 <- rep(NA, 5) # toy data
tmp <- is.na(vct2) # tmp for temporary
all(tmp)
## [1] TRUE

all(is.na(vct2)) # nested call
## [1] TRUE
```

How to test if a vector contains one or more `NA` (or `NaN`) values?
See previous question. We only need to replace `all()` by `any()` to obtain the answer.

```
vct2 <- rep(NA, 5)
any(is.na(vct2))
## [1] TRUE
```

3.6 Comparison operators and operations

Comparison operators return vectors of `logical` values (see section 3.5 on page 48), 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 operator `=` is an infrequently used synonym of the assignment operator `<-` rather than a comparison operator!

```
# 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
```

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 30: their arguments are recycled when needed. Hint: if you do not know what value is stored in numeric vector `a`, use `print(a)` after the first code statement below to see its contents.

```
vct3 <- 1:10
vct3 > 5
##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
vct3 < 5
##  [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
vct3 == 5
##  [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
all(vct3 > 5)
## [1] FALSE
any(vct3 > 5)
## [1] TRUE
vct4 <- vct3 > 5
vct4
##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
any(vct4)
## [1] TRUE
all(vct4)
## [1] FALSE
```

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 3.5 on page 48.

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 `&`, and use parentheses to override the default order of precedence of the operations.

```
vct5 <- -2:3
vct5 >= -1 & vct5 <= 1
## [1] FALSE  TRUE  TRUE  TRUE FALSE FALSE
```

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

```
!(vct5 >= -1 & vct5 <= 1)
## [1]  TRUE FALSE FALSE FALSE  TRUE  TRUE
```

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

```
vct5 < -1 | vct5 > 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.

3.17 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.

```
vct6 <- 1:10
vct6 > 3 | vct6 + 2 < 3
```

It is important to be aware of the consequences of “short-cut evaluation” (described on page 50). 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.

```
vct7 <- c(1:10, NA)
all(vct7 < 20)
## [1] NA
any(vct7 > 20)
## [1] NA
all(vct7 < 20, na.rm=TRUE)
## [1] TRUE
any(vct7 > 20, na.rm=TRUE)
## [1] FALSE
```

In many situations, when writing programs one should avoid testing for equality of floating point numbers (‘floats’). This is because of how numbers are stored in computers (see the box on page 34 for an in-depth explanation). Here I show how to gracefully handle rounding errors when using comparison operators. As rounding errors may accumulate, in practice `.Machine$double.eps` is frequently too small a value to safely use in tests for “zero.”. 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 equality tests should be done by replacing tests like `x == 1.0` with `abs(x - 1.0) < k`, where `k` is a number larger than `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 $|x| = |-x|$.

```
sin(pi) == 0 # angle in radians, not degrees!
## [1] FALSE
sin(2 * pi) == 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
```

3.7 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. Algebra sets is implemented with functions `union()`, `intersect()`, `setdiff()`, `setequal()`, `is.element()` and operator `%in%` (Figure 3.1). 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.

Set operations applied to vectors with values representing a mundane example, grocery shopping, demonstrate them.

```
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
union(bakery, dairy)
## [1] "bread"   "buns"    "cake"    "cookies" "milk"    "butter"  "cheese"
setdiff(union(bakery, dairy), shopping) # nested call
## [1] "buns"    "cake"    "cookies" "milk"
```

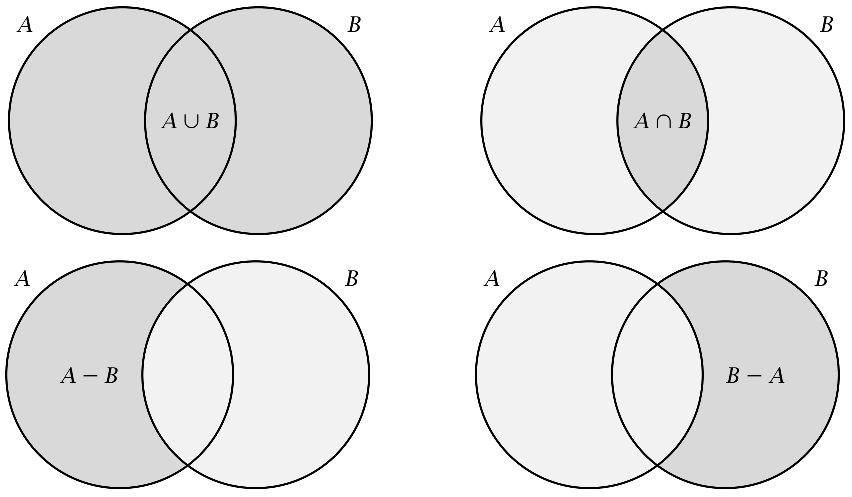


FIGURE 3.1
Boolean algebra. Venn diagrams for algebra of sets operations: *union*, $\cup$, `union()`; *intersection*, $\cap$, `intersect()`; *difference (asymmetrical)*, $-$, `setdiff()`; *equality test* `setequal()`; *membership*, `is.element()` and operator `%in%`

Sets describe membership as a binary property, thus when vectors are interpreted as sets, duplicate members are redundant. Duplicate members although accepted as input are 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")) # vectors compared
## [1] "1 string mismatch"
identical(c("a", "a", "b"), c("b", "a", "b")) # vectors compared
## [1] FALSE
```

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

```
vct1 <- 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", vct1)
## [1] TRUE
"a" %in% vct1
## [1] TRUE
c("a", "a", "z") %in% vct1
## [1]  TRUE  TRUE FALSE
```

Keep in mind that inclusion, implemented in operator `%in%`, is an asymmetrical (not reflective) operation among a vector and a set. The right-hand-side (rhs) argument is interpreted as a set, while the left-hand-side (lhs) argument is interpreted as a vector of values to test for membership in the set. In other words, any duplicate member in the lhs operand is retained and tested while the rhs operand is interpreted as a set of unique values. The returned logical vector has the same length as the lhs operand.

```
vct1 %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", vct1)
## [1] FALSE
!"a" %in% vct1
## [1] FALSE
!c("a", "a", "z") %in% vct1
## [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.

3.18 Use operator `%in%` to write more concisely the following comparisons. Hint: see section 3.5 on page 48 for the difference between `|` and `||` operators.

```
vct2 <- c("a", "a", "z")
vct2 == "a" | vct2 == "b" | vct2 == "c" | xvct2 == "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(vct1)
## [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(vct1))
## [1] 3
```

3.19 Do the values returned by these two statements differ?

```
c("a", "a", "z") %in% vct1
c("a", "a", "z") %in% unique(vct1)
```

Function `duplicated()` 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.

```
duplicated(vct1)
## [1] FALSE FALSE FALSE  TRUE
anyDuplicated(vct1)
## [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.

3.20 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"), vct1)
c(c("a", "a", "z"), vct1)
c("a", "a", "z", vct1)
```

Are set union and concatenation of vectors equivalent operations? why or why not?

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.

```
9 %in% 2:4
## [1] FALSE
9 %in% ((2:4) * (2:4))
## [1] TRUE
c(1, 16) %in% ((2:4) * (2:4))
## [1] FALSE  TRUE
```

Empty sets are an important component of the algebra of sets, in R they are represented as vectors of zero length. These vectors do belong to a class such as `numeric` or `character` and must be compatible with other operands in an expression.

```
c("ab", "xy") %in% character()
## [1] FALSE FALSE
character() %in% c("a", "b", "c")
## logical(0)
union("ab", character())
## [1] "ab"
```

Although set operators are defined for `numeric` vectors, rounding errors in ‘floats’ can result in unexpected results (see section 3.3 on page 34).

```
c(cos(pi), sin(pi)) %in% c(0, -1)
## [1]  TRUE FALSE
c(cos(pi), sin(pi))
## [1] -1.000000e+00  1.224606e-16
```

3.21 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.

3.8 The ‘mode’ and ‘class’ of objects

Classes are abstractions, they determine the “meaning” and behavior of objects belonging to them. New classes can be defined in user code as well as new methods, i.e., functions or operators tailored to fit them. The *class* is like a “tag” that tells how the value in an object should be interpreted and operated upon.

Variables (names given to objects) have a *class* that depends on the object stored in them. In contrast to some other languages in R assignment to a variable already in use to store an object belonging to a different class is allowed. There is a restriction that all elements in a vector, array or matrix, must be of the same mode (these are called atomic, as they contain homogeneous members). Lists and data frames can be heterogenous (to be described in chapter 4). In practice this means that we can assign an object, such as a vector, with a different `class` 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.

Function `class()` is used to query the class of an object, and function `inherits()` is used to test if an object belongs to a specific class or not (including “parent” classes, to be later described).

```
vct1 <- 1:5
class(vct1)
## [1] "integer"
inherits(vct1, "character")
## [1] FALSE
inherits(vct1, "numeric")
## [1] FALSE
```

Functions with names starting with `is.` are tests returning a logical value, `TRUE`, `FALSE` or `NA`.

```
is.numeric(vct1) # no distinction of integer or double
## [1] TRUE
is.double(vct1)
## [1] FALSE
is.integer(vct1)
## [1] TRUE
is.logical(vct1)
## [1] FALSE
is.character(vct1)
## [1] FALSE
```

The *mode* of an object is a fundamental property, and limited to those modes defined as part of the R language. In particular, different R objects of a given mode, such as `numeric`, can belong to different `classes`. Classes and the dispatch of methods are discussed in section 6.3 on page 172, together with object-oriented programming.

```
mode(c(1, 2, 3)) # no distinction of integer or double
## [1] "numeric"
typeof(c(1, 2, 3))
## [1] "double"
class(c(1, 2, 3))
## [1] "numeric"
mode(c(1L, 2L, 3L)) # no distinction of integer or double
## [1] "numeric"
typeof(c(1L, 2L, 3L))
## [1] "integer"
class(c(1L, 2L, 3L))
## [1] "integer"
```

```
mode(factor(c("a", "b", "c"))) # no distinction of integer or double
## [1] "numeric"
typeof(factor(c("a", "b", "c")))
## [1] "integer"
class(factor(c("a", "b", "c")))
## [1] "factor"
```

```
mode(c("a", "b", "c"))
## [1] "character"
typeof(c("a", "b", "c"))
## [1] "character"
class(c("a", "b", "c"))
## [1] "character"
```

```
mode(c(TRUE, FALSE))
## [1] "logical"
typeof(c(TRUE, FALSE))
## [1] "logical"
class(c(TRUE, FALSE))
## [1] "logical"
```

3.9 ‘Type’ conversions

By type conversion we mean converting a value from one class into a value expressed in a different class. usually the meaning can be retained, at least in part. We can for example convert character strings into numeric values, but this conversion is possible only for character strings conformed by digits, like `"100"`. Most conversions, such as the conversion of `character` value `"100"` into `numeric` value `100` are obvious. Type conversions involving logical values are less intuitive. By convention, functions used to convert objects from one mode or class to a different one have names starting with `as.`[1].

```
as.character(102)
## [1] "102"
as.character(TRUE)
## [1] "TRUE"
as.character(3.0e10)
## [1] "3e+10"
as.numeric("203")
## [1] 203
as.logical("TRUE")
## [1] TRUE
as.logical(100)
## [1] TRUE
as.logical(0)
## [1] FALSE
as.logical(-1)
## [1] TRUE
```

Some conversions takes place automatically in expressions involving both `numeric` and `logical` values.

```
TRUE + 10
## [1] 11
1 || 0
## [1] TRUE
FALSE | -2:2
## [1]  TRUE  TRUE FALSE  TRUE  TRUE
```

[1]Except for some packages in the ‘tidyverse’ that use names starting with `as_` instead of `as.`.

3.22 There is 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.

```
as.numeric("5E+5")
as.numeric("50e+4")
as.numeric(".12")
as.numeric("0.12")
as.numeric("A")
as.logical("TRUE")
as.logical("FALSE")
as.logical("T")
as.logical("t")
as.logical("true")
as.logical("NA")
```

3.23 Conversion of fractional numbers into whole numbers can be achieved in different ways, by truncation of the fractional part or rounding it up or down. If we consider both negative and positive numbers, how each of them are handled creates additional possibilities. All these approaches as defined in mathematics, are available through different `R` functions. These functions, are not conversion functions as they return a `numeric` value of class `double`. See page 37. In contrast, `as.integer()` is a conversion function for type `double` into type `integer`, both with mode `numeric`.

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* and *type* 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 becomes clear.

```
vct1 <- c("1", "2", "3")
length(vct1)
## [1] 3

vct2 <- "123.1"
length(vct2)
## [1] 1

as.numeric(vct1)
## [1] 1 2 3
as.numeric(vct2)
## [1] 123.1
as.integer(vct1)
## [1] 1 2 3
as.integer(vct2)
## [1] 123
```

Other functions relevant to the "conversion" of numbers and other values are `format()`, and `sprintf()`. This is sometimes informally called "pretty printing". These two functions return `character` strings, instead of `numeric` or other values, and are useful for printed 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 classes like `character`, `logical`, etc.

When using `format()`, the format used to display numbers is set by passing arguments to several different parameters. As `print()` calls `format()` to convert `numeric` values into `character` strings, it accepts the same options.

```
vct2 = c(123.4567890, 1.0)
format(vct2) # using defaults
## [1] "123.4568" "  1.0000"
format(123.4567890) # using defaults
## [1] "123.4568"
format(1.0) # using defaults
## [1] "1"
format(vct2, digits = 3, nsmall = 1)
## [1] "123.5" "  1.0"
format(vct2, 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: %.4g 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.

⌨ 3.24 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. The examples below use the extraction operator to demonstrate automatic conversion on assignment. This operator is described in section 3.10 below.

```
vct3 <- c(1, NA)
is.numeric(vct3[2])
## [1] TRUE
is.numeric(NA)
## [1] FALSE
```

```
vct4 <- c("abc", NA)
is.character(vct4[2])
## [1] TRUE
```

```
is.character(NA)
## [1] FALSE
class(NA)
## [1] "logical"
class(NA_character_)
## [1] "character"
```

```
vct5 <- NA
c(vct5, 2:3)
## [1] NA  2  3
```

However, even the statement below works transparently.

```
vct3[3] <- vct4[2]
```

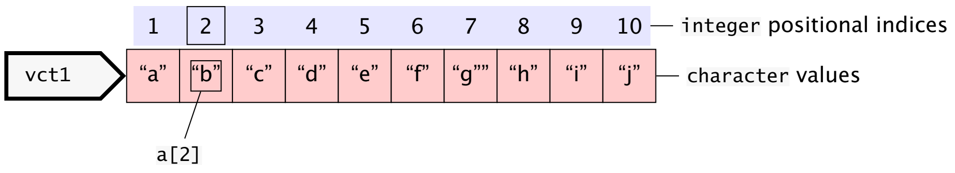
3.10 Vector manipulation

If you have read earlier sections of this chapter, you already know how to create a vector. If not, see pages 28–32 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 $a_{i=1...n}$. 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`.

```
vct1 <- letters[1:10]
vct1
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```


1 2 3 4 5 6 7 8 9 10
integer positional indices
vct1
“a” “b” “c” “d” “e” “f” “g”” “h” “i” “j”
character values
a[2]

```
vct1[2]
## [1] "b"
```

Four constant vectors are available in base R: `letters`, `LETTERS`, `month.name` and `month.abb`, of which I 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.

```
vct1[c(3, 2)]
## [1] "c" "b"
vct1[10:1]
##  [1] "j" "i" "h" "g" "f" "e" "d" "c" "b" "a"
```

3.25 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 NAs. 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)
vct1[c(3, 3, 3, 3)]
vct1[c(10:1, 1:10)]
vct1[c(1, 11)]
vct1[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.

```
vct1[-2]
## [1] "a" "c" "d" "e" "f" "g" "h" "i" "j"
vct1[-c(3,2)]
## [1] "a" "d" "e" "f" "g" "h" "i" "j"
vct1[-3:-2]
## [1] "a" "d" "e" "f" "g" "h" "i" "j"
```

3.26 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.

```
vct1[ ]
vct1[0]
vct1[numeric(0)]
vct1[NA]
vct1[c(1, NA)]
vct1[NULL]
vct1[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 in page 30 in relation to other operators.

```
vct1[TRUE]
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
vct1[FALSE]
## character(0)
vct1[c(TRUE, FALSE)]
## [1] "a" "c" "e" "g" "i"
vct1[c(FALSE, TRUE)]
## [1] "b" "d" "f" "h" "j"
vct1 > "c"
##  [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
vct1[vct1 > "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.

3.27 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.

```
vct1 <- letters[1:10]
vct2 <- 1:10
selector <- vct1 > "c"
selector
vct1[selector]
vct2[selector]
```

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

```
indexes <- which(vct1 > "c")
indexes
vct1[indexes]
vct2[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 one is familiar with the indexing rules. However, if one does not command these rules, many of these terse statements become unintelligible.

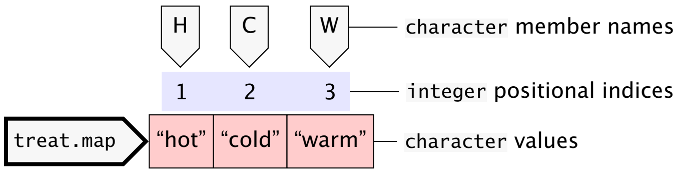
In all earlier examples we have used integer valued indices for extraction of elements. In the vectors used as examples above the elements were anonymous or nameless. In R the elements can be assigned names, and these names used in place of numeric indices to extract the named elements. There is one 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. Above we used function c() to concatenate several character strings, without assigning any names to them, thus they can extracted from the vector using numeric values for indexing by position. Below, we assign a name to each string. Using operator = we assign the name on the left-hand side (*lhs*) to the member of the vector on the right-hand-side (*rhs*).

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



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 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.

```
vct2 <- 1:10
vct2
##  [1]  1  2  3  4  5  6  7  8  9 10
vct2[1] <- 99
vct2
##  [1] 99  2  3  4  5  6  7  8  9 10
vct2 <- 1:10
vct2[c(2,4)] <- -99 # recycling
vct2
##  [1]   1 -99   3 -99   5   6   7   8   9  10
vct2 <- 1:10
vct2[c(2,4)] <- c(-99, 99)
vct2
##  [1]   1 -99   3  99   5   6   7   8   9  10
vct2 <- 1:10
vct2[TRUE] <- 1 # recycling
vct2
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1
vct2 <- 1:10
vct2 <- 1  # no recycling
vct2
## [1] 1
```

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

```
vct3 <- letters[1:10]
vct3[1:2] <- vct3[2:1]
vct3
##  [1] "b" "a" "c" "d" "e" "f" "g" "h" "i" "j"
```

3.28 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.

```
VCT1 <- letters[1:10]
VCT1[5:1] <- VCT1[c(TRUE,FALSE)]
VCT1
```

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.

```
vct4 <- LETTERS[1:10]
vct4
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
vct4[1]
## [1] "A"
vct4[1.1]
## [1] "A"
vct4[1.9999] # surprise!!
## [1] "A"
vct4[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.

```
vct4 <- LETTERS[1:10]
vct4
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
vct4[-1]
## [1] "B" "C" "D" "E" "F" "G" "H" "I" "J"
vct4[-1.1]
## [1] "B" "C" "D" "E" "F" "G" "H" "I" "J"
vct4[-1.9999]
## [1] "B" "C" "D" "E" "F" "G" "H" "I" "J"
vct4[-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 where 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()`.

```
vct5 <- c(10, 4, 22, 1, 4)
sort(vct5)
## [1]  1  4  4 10 22
sort(vct5, 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(vct5)
## [1] 4 2 5 1 3
vct5[order(vct5)]
## [1]  1  4  4 10 22
vct6 <- c("ab", "aa", "c", "zy", "e")
vct6[order(vct5)]
## [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.

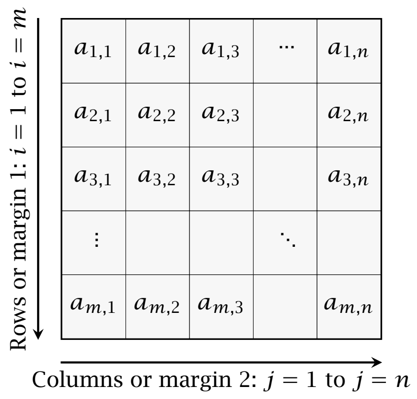
```
vct7 <- letters[c(1, 5, 10, 3, 1, 4, 21, 1, 10)]
vct7
## [1] "a" "e" "j" "c" "a" "d" "u" "a" "j"
sort(vct7)
## [1] "a" "a" "a" "c" "d" "e" "j" "j" "u"
rle(sort(vct7))
## Run Length Encoding
##   lengths: int [1:6] 3 1 1 1 2 1
##   values : chr [1:6] "a" "c" "d" "e" "j" "u"
```

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.

3.11 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 28 above, but the same concepts apply.


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

In R documentation, the individual dimensions of matrices and arrays are frequently called *margins*, numbered in the same order as the indices are given. Thus, in a matrix the first margin corresponds to rows and the second one to columns.

In mathematical notation the same generic matrix is represented as

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

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 28 above, we can query this dimension, their length, with function `length()`. Matrices have two dimensions, which can be queried individually with `ncol()` and `nrow()`, and jointly with `dim()`. As expected `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. Function `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 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.

⌨ 3.29 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.

```
mat1 <- matrix(1:20, ncol = 4)
mat1
##      [,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
mat1[1, 2]
## [1] 6
mat1[2, 1]
## [1] 2
```

Remind yourself of how indexing of vectors works in R (see section 3.10 on page 63). 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.

```
mat1[1, ]
## [1]  1  6 11 16
mat1[ , 1]
## [1] 1 2 3 4 5
mat1[2:3, c(1,3)]
##      [,1] [,2]
## [1,]    2   12
## [2,]    3   13
mat1[3, 4] <- 99
mat1
##      [,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
mat1[4:3, 2:1] <- mat1[3:4, 1:2]
mat1
##      [,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 is simplified into a vector by dropping one dimension. By passing `drop = FALSE`, we can prevent this.

```
is.matrix(mat1[1, ])
## [1] FALSE
is.matrix(mat1[1:2, 1:2])
## [1] TRUE

is.vector(mat1[1, ])
## [1] TRUE
is.vector(mat1[1:2, 1:2])
## [1] FALSE

is.matrix(mat1[1, , drop = FALSE])
## [1] TRUE
is.matrix(mat1[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(mat1)
## NULL
rownames(mat1)
## NULL
colnames(mat1) <- c("a", "b", "c", "d")
mat1
##      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
mat1[ , c("b", "a")]
##       b a
## [1,]  6 1
## [2,]  7 2
## [3,]  4 9
## [4,]  3 8
## [5,] 10 5
colnames(mat1) <- NULL
mat1
##      [,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.

```
mat1 <- matrix(1:20, ncol = 4)
mat1
##      [,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(mat1)
## [1] 5 4
mat1[10]
## [1] 10
mat1[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.

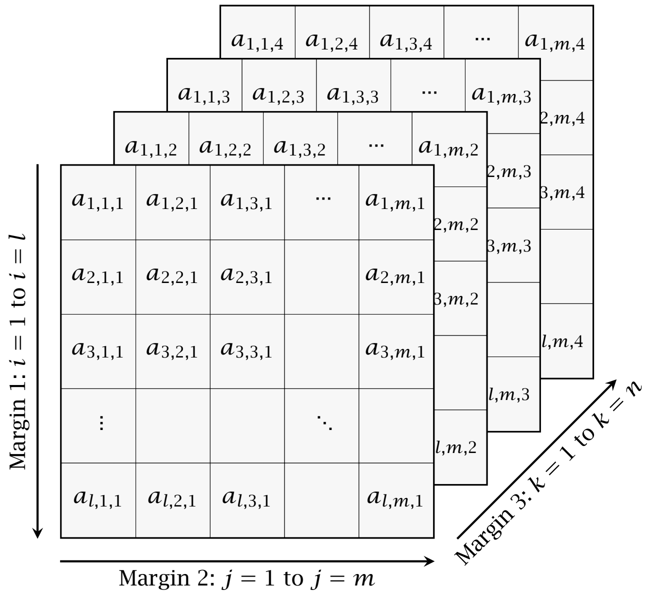
```
mat2 <- matrix(1:20, ncol = 4, byrow = TRUE)
mat2
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]   13   14   15   16
## [5,]   17   18   19   20
dim(mat2)
## [1] 5 4
mat2[10]
## [1] 18
mat2[5, 2]
## [1] 18
```

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 as *dimensions* attribute an `integer` vector of length two.

```
vct1 <- 1:6
dim(vct1)
## 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 `dim()`, similarly as with matrices. Whether an R object is an array can be found out with `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 $a_{i,j,k}$.



Margin 1: $i = 1$ to $i = l$
Margin 2: $j = 1$ to $j = m$
Margin 3: $k = 1$ to $k = n$

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

```
ary1 <- array(1:27, dim = c(3, 3, 3))
ary1
## , , 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
```

```
ary1[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.

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

```
VCT2 <- 1:10
MAT1 <- matrix(VCT2, ncol = 2)
MAT2 <- matrix(VCT2, ncol = 2, byrow = TRUE)
MAT3 <- matrix(VCT2, nrow = 2)
MAT4 <- matrix(VCT2, nrow = 2, byrow = TRUE)

ARY1 <- array(VCT2, dim = c(5, 2))
ARY2 <- array(VCT2, dim = c(5, 2), dimnames = list(NULL, c("c1", "c2")))
ARY3 <- array(VCT2, 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 functions to explore this, and its consequences.
2. Convert a matrix into a vector using `as.vector()` and compare the returned values to those in the matrix. Are values extracted by columns or by rows first.

Operators and functions for matrix algebra are available in R as matrices are used in statistical algorithms. I describe below only some of these matrix-specific functions and operators. I also give examples of the use of some of the usual arithmetic operators together with objects of class `matrix`.

Recycling applies to the usual arithmetic operators when applied to matrices. This is similar to their behavior when all operands are vectors (see page 30).

```
mat3 <- matrix(1:20, ncol = 4)
mat3 + 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
mat3 * 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
mat3 * 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
```

3.31 When a `matrix` and a `vector` are operands in an arithmetic operation, how the positions of the `vector` are mapped to positions in the `matrix` affects the result of the operation. Run the code below to find out. What is the logic behind?

```
matrix(rep(1, 6)) * 1:6
```

Function `t()` transposes a matrix, by swapping columns and rows.

```
mat3
```

```
##      [,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(mat3)
```

```
##      [,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
```

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 distinct. Matrix algebra gives the rules for operations where both operands are matrices. For example, matrix multiplication is indicated by operator `%*%`.

```
mat4 <- matrix(1:16, ncol = 4)
mat4 * mat4
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1   25   81  169
## [2,]    4   36  100  196
## [3,]    9   49  121  225
## [4,]   16   64  144  256
```

```
mat4 %*% mat4
```

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

Function `diag()` makes it possible to easily create a diagonal matrix.

```
mat5 <- diag(4)
mat5
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    1    0
## [4,]    0    0    0    1
mat4 %*% mat5
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
```

The inverse of a matrix can be found by means of function `solve()`.

```
mat6 <- matrix(c(3, 2, 0, 1, 3, 2, 7, 2, 4), ncol = 3)
solve(mat6)
##             [,1]       [,2]       [,3]
## [1,]  0.18181818  0.2272727 -0.4318182
## [2,] -0.18181818  0.2727273  0.1818182
## [3,]  0.09090909 -0.1363636  0.1590909
```

Additional operators and functions for matrix algebra like cross-product (`crossprod()`) and Cholesky root (`chol()`) are available in base R. Packages, including ‘matrixStats’, provide additional functions and operators for matrices.

3.12 Factors

In data analysis and Statistics the distinction between values measured on continuous vs. discrete *scales* is crucial. In a continuous scale, any values are in theory possible. In a discrete scale, the observations are values from a few categories.

In contrast to other statistical software in which a variable is set as continuous or discrete when defining a model to be fitted or when setting up a test, in R this distinction is based on whether the explanatory variable is `numeric` (continuous) or a `factor` (discrete). This approach makes 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 7 and 9, it can affect the contrasts used by some tests of significant, and the arrangement or positions of the levels along axes and keys in plots.

In an R `factor`, values indicate discrete unordered categories, most frequently the treatments in an experiment, or categories in a survey. Factor 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 7.

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.

```
vct1 <- c("treated", "treated", "control", "control", "control", "treated")
factor(vct1)
## [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 = vct1, 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("a", "a", "b", "b", "b", "a"), levels = c("a", "b"), la-
bels = c("treated", "control"))
## [1] treated treated control control control treated
## Levels: treated control
```

The argument passed to labels can be a named vector that *maps* new labels onto the values as stored in the vector passed as argument to parameter `x` (see named vectors and mapping on page 66).

```
factor(x = c("a", "a", "b", "b", "b", "a"), labels = c(a = "treated", b = "con-
trol"))
## [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.

```
fct1 <- factor(x = vct1)
fct1
## [1] treated treated control control control treated
## Levels: control treated
factor(x = fct1, levels = c("treated", "control"))
## [1] treated treated control control control treated
## Levels: treated control
factor(x = fct1, labels = c(control = "cooled", treated = "heated"))
## [1] heated heated cooled cooled cooled heated
## Levels: cooled heated
factor(x = fct1,
       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.

```
fct2 <- gl(4, 3, labels = c("A", "F", "B", "Z"))
fct2
##  [1] A A A F F F B B B Z Z Z
## Levels: A F B Z
factor(fct2,
       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
```

3.32 Edit the code in the chunk above to use only a named vector for `labels` instead of separate vectors passed to `levels` and `labels`.

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, `lengtgh()` to query the number of values stored.

```
fct1
## [1] treated treated control control control treated
## Levels: control treated
levels(fct1)
## [1] "control" "treated"
length(fct1)
```

```
## [1] 6
fct1.control <- fct1[fct1 == "control"]
fct1.control
## [1] control control control
## Levels: control treated
levels(fct1.control) # same as in my.factor
## [1] "control" "treated"
length(fct1.control) # shorter than my.factor
## [1] 3
```

How to drop unused levels in a factor?
It can be seen above that subsetting does not drop unused factor levels. Constructor function `factor()` can be used to explicitly drop the unused factor levels.

```
fct1.control <- factor(fct1.control)
levels(fct1.control) # the unused level was dropped
## [1] "control"
```

How to convert a factor into a vector with matching values?
This operation is not obvious, specially when the factor was created from a `numeric` vector.

```
vct3 <- rep(3:5, 4)
vct3
##  [1] 3 4 5 3 4 5 3 4 5 3 4 5
fct3 <- factor(vct3)
fct3
##  [1] 3 4 5 3 4 5 3 4 5 3 4 5
## Levels: 3 4 5
as.numeric(fct3)
##  [1] 1 2 3 1 2 3 1 2 3 1 2 3
as.numeric(as.character(fct3))
##  [1] 3 4 5 3 4 5 3 4 5 3 4 5
```

Why is a double conversion needed? Internally, factor values are 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. 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 3.8 on page 58, and `str()` on page 91.

```
class(fct3)
## [1] "factor"
mode(fct3)
## [1] "numeric"
str(fct3)
##  Factor w/ 3 levels "3","4","5": 1 2 3 1 2 3 1 2 3 1 ...
```

3.33 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 on page 79. 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()`.

```
fct4 <- factor(c("treated", "treated", "control", "control", "con-
trol", "treated"))
levels(fct4)
## [1] "control" "treated"
fct4 <- factor(fct4, levels = rev(levels(fct4)))
levels(fct4)
## [1] "treated" "control"
```

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

```
fct5 <- factor(fct4,
               levels = sort(levels(fct4), decreasing = TRUE))
levels(fct5)
## [1] "treated" "control"
```

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

```
fct6 <- factor(fct4, levels = levels(fct4)[c(2, 1)])
levels(fct6)
## [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.

```
fct7 <- gl(2, 5, labels = c("A", "B"))
vct4 <- c(5.6, 7.3, 3.1, 8.7, 6.9, 2.4, 4.5, 2.1, 1.4, 2.0)
fct7
##  [1] A A A A A B B B B B
## Levels: A B
fct7ord <- reorder(fct7, vct4)
levels(fct7ord)
## [1] "B" "A"
fct7rev <- reorder(fct7, -vct4) # a simple trick: change sign
levels(fct7rev)
## [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.

3.34 **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 30.)

```r
# gl() keeps order of levels
FCT1 <- gl(4, 3, labels = c("A", "F", "B", "Z"))
FCT1
as.integer(FCT1)

# factor() orders levels alphabetically
FCT2 <- factor(rep(c("A", "F", "B", "Z"), times = rep(3, times = 4))) # nested calls
FCT2
as.integer(FCT2)
levels(FCT2)[as.integer(FCT2)]
```

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.

```r
sort(FCT2)
FCT2[order(FCT2)]
FCT2[order(as.integer(FCT2))]
```

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

Factors encode levels as `integer` values in a vector. In many cases, statistical computations, require the same information to be encoded as binary values using multiple *dummy variables*. Factors are much friendlier for the user to manage. They are converted into the equivalent dummy variables when a model formula is translated into a *model matrix*. This is handled transparently by most functions implementing fitting of statistical models to data (see sections 7.6 and 7.11 on pages 191 and 217).

3.13 Further reading

For further reading on the aspects of R discussed in the current chapter, I suggest the book *The Art of R Programming: A Tour of Statistical Software Design* (Matloff 2011).