A tutorial on common data structures in R

Version 1.0.0

Konstantinos I. Bougioukas

07/10/2021

Contents

Learning Objectives

- Get familiar with the different data structures (vectors, matrices, data frames)
- Learn how to use R as a calculator and how to assign variables
- Understand the concepts of coercion and vector recycling
- Understand how to extract elements from a vector, a matrix, or a list (subsetting)
- Understand how to access variables in a data frame

Packages installation

We should first install the following packages to execute the full tutorial, if we don't have them installed on our computer:

- `{tidyverse}`,
- `{lubridate}`,
- `{rstatix}`,
- `{matlib}`,
- `{lobstr}`

1 Introduction

The R language supports many types of data structures that we can use to organize and store values in our code.

The most fundamental concept in base R are the **vectors**. Vectors come in two flavours: **atomic vectors** and **lists (generic vectors)**. The atomic vectors must have all elements of the same basic type (e.g., integers, character…). On the contrary, in the lists different elements can have different types (e.g., some elements may be integers and some characters).

We will see that more complex structures such as matrices, arrays, and data frames can be created. Each data structure type serves a specific purpose and can contain specific kinds of data. They differ in terms of the type of data they can hold, how they're created, their structural complexity, the RAM that they occupy, and the notation used to identify and access individual elements. So, it's important to understand the differences between them so we can make the right choice based on our scenario. Figure 1 shows a diagram of these data structures:

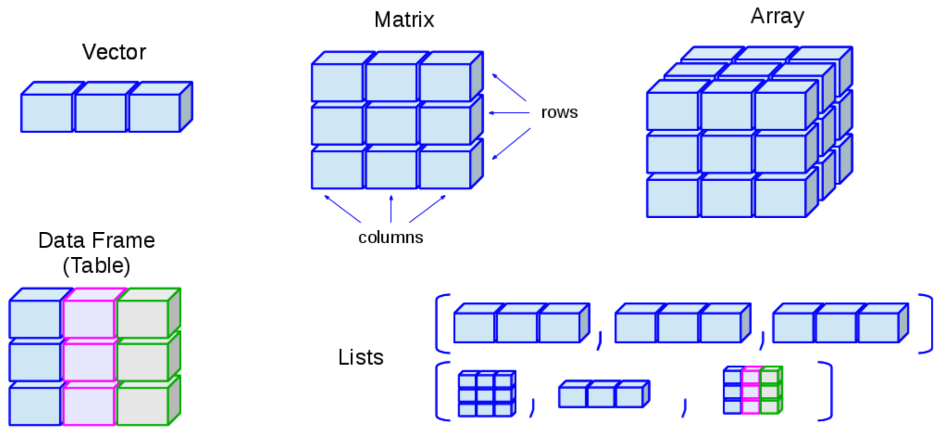

Figure 1: Data structures in R

2 Atomic vectors: the basics

There are four primary types of atomic vectors (also known as “atomic” classes):

- logical
- integer
- double
- character (which may contain strings)

Collectively integer and double vectors are known as numeric vectors.

There are also two rare types: complex and raw. We won’t discuss them further because complex numbers are rarely needed in statistics, and raw vectors are a special type that’s only needed when handling binary data (raw bytes).

2.1 One-element atomic vectors: Scalars

Each of the four primary types has a special syntax to create an individual value, AKA a scalar. A scalar object is just a single value like a number (one-element vector) and they can be used to construct more complex objects (longer vectors). We present some examples of scalars for each of the four primary types (in order from least to most general):

1. **Logical scalars:** Logical values are boolean values of `TRUE` or `FALSE` which can be abbreviated, when we type them as `T` or `F` (we do not suggest this).

```
# Examples of logical scalars
sclr_a <- TRUE  # assign the TRUE value to an object named sclr_a
sclr_a
```

```
## [1] TRUE
```

```r
sclr_b <- FALSE
sclr_b
```

```
## [1] FALSE
```

```r
sclr_c <- T
sclr_c
```

```
## [1] TRUE
```

```r
sclr_d <- F
sclr_d
```

```
## [1] FALSE
```

2. **Numeric (integer or double) scalars:** Even if we see a number like `1` or `2` in R, which we might think of as integers, they are likely represented behind the scenes as `1.00` or `2.00`. We need to place an "L" suffix for integer numbers. Doubles can be specified in decimal (e.g., 0.03) or scientific (e.g, 3e-2) format.

```r
# Examples of integer and double scalars
sclr_e <- 3L          # integer
sclr_e
```

```
## [1] 3
```

```r
sclr_f <- 100L        # integer
sclr_f
```

```
## [1] 100
```

```r
sclr_g <- sclr_e / sclr_f
sclr_g                # double
```

```
## [1] 0.03
```

```r
sclr_scientific <- 3e-2
sclr_scientific     # double
```

```
## [1] 0.03
```

Note Double format is a computer number format, usually occupying 64 bits in computer memory.

3. **Character scalars:** Scalars can also be characters (also known as `strings`). In R, we denote characters using quotation marks " or '. Here are examples of `character` scalars:

```r
# Examples of character scalars
sclr_h <- "hello"      # double quotation marks
sclr_h
```

```
## [1] "hello"
```

```r
sclr_i <- 'covid-19'   # single quotation marks
sclr_i
```

```
## [1] "covid-19"
```

```r
sclr_j <- "I love data analysis"
sclr_j
```

```
## [1] "I love data analysis"
```

R treats numeric and character scalars differently. For example, while we can do basic arithmetic operations on numeric scalars – they won't work on character scalars. If we try to perform numeric operations (like addition) on character scalars, we'll get an error like the following:

```r
h <- "1"
k <- "2"
h + k
```

Error in h + k : non-numeric argument to binary operator

If we see an error like this one, it means that we're trying to apply numeric operations to character objects that's wrong.

It's very rare that single values (scalars) will be the center of an R session, so one of the first questions encountered when working with data in R is what sort of object should be used to hold collections of data. Next, we are going to talk about "longer" atomic vectors.

2.2 Making longer atomic vectors

Atomic vectors can consisted of more than one element. In this case, the vector elements are ordered, and they must all be of the same type of data. Common example types of "long" atomic vectors are numeric (whole numbers and fractions), logical (e.g., TRUE or FALSE), and character (e.g., letters or words).

Let’s see how we can create “long” atomic vectors and some usefull vector properties through examples.

2.2.1 The colon operator `(:)`

The **colon** operator `:` will generate sequences of consecutive values. For example:

```
1:5
```

```
## [1] 1 2 3 4 5
```

In this example, the colon operator `:` takes two integers 1 and 5 as arguments, and returns an atomic vector of integer numbers from the starting point 1 to the ending point 5 in steps of 1.

We can assign (or name) the atomic vector to an object named `x_seq`:

```
x_seq <- 1:5
```

and call it with its name:

```
x_seq
```

```
## [1] 1 2 3 4 5
```

We can determine the type of a vector with `typeof()`.

```
typeof(x_seq)
```

```
## [1] "integer"
```

The elements of the `x_seq` vector are integers.

We can also find how many elements a vector contains applying the `length()` function:

```
length(x_seq)
```

```
## [1] 5
```

Note Another way to create vectors of consequtive values is the **seq()** function which stands for sequence.

```
seq(1, 5)     # increment by 1
```

```
## [1] 1 2 3 4 5
```

Other examples:

```
5:1
```

```
## [1] 5 4 3 2 1
```

```
2.5:8.5
```

```
## [1] 2.5 3.5 4.5 5.5 6.5 7.5 8.5
```

```
-3:4
```

```
## [1] -3 -2 -1  0  1  2  3  4
```

2.2.2 The `c()` function

We can also create atomic vectors "by hand" using the `c()` function (or concatenate command) which *combines* values into a vector. Let's create a vector of values `2`, `4.5`, and `1`:

```
c(2, 4.5, -1)
```

```
## [1]  2.0  4.5 -1.0
```

Note that an atomic vector can be element of another vector:

```
x_seq
```

```
## [1] 1 2 3 4 5
```

```
c(x_seq, 2, 4.5, -1)
```

```
## [1]  1.0  2.0  3.0  4.0  5.0  2.0  4.5 -1.0
```

Of course, we can have an atomic vector with logical elements as the following example:

```
c(TRUE, FALSE, TRUE, FALSE)
```

```
## [1]  TRUE FALSE  TRUE FALSE
```

or equivalently

```
c(T, F, T, F)
```

```
## [1]  TRUE FALSE  TRUE FALSE
```

and an atomic vector with character elements:

```
c("male", "female", "female", "male")
```

```
## [1] "male"   "female" "female" "male"
```

2.2.3 Repeating vectors

The `rep()` function allows us to conveniently repeat complete a vector or the elements of a vector. Let's see some examples:

1. Repeating the **complete** vector.

```
rep(1:4, times = 3)  # 3 times to repeat the complete vector
```

```
##  [1] 1 2 3 4 1 2 3 4 1 2 3 4
```

```
rep(c(0, 4, 7), times = 3)  # 3 times to repeat the complete vector
```

```
## [1] 0 4 7 0 4 7 0 4 7
```

```
rep(c("a", "b", "c"), times = 2) # 2 times to repeat the complete vector
```

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

2. Repeating **each element** of the vector. Examples:

```
rep(1:4, each = 3) # each element is repeated 3 times
```

```
##  [1] 1 1 1 2 2 2 3 3 3 4 4 4
```

```
rep(c(0, 4, 7), each = 3) # each element is repeated 3 times
```

```
## [1] 0 0 0 4 4 4 7 7 7
```

```
rep(c("a", "b", "c"), each = 2) # each element is repeated 2 times
```

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

2.2.4 Default vectors

R comes with a few built-in default vectors, containing useful values:

```
LETTERS
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P"
## [17] "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

```
letters
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p"
## [17] "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
month.abb
```

```
##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov"
## [12] "Dec"
```

```
month.name
```

```
##  [1] "January"   "February"  "March"     "April"     "May"
##  [6] "June"      "July"      "August"    "September" "October"
## [11] "November"  "December"
```

We will use some of these built-in vectors in the examples that follow.

2.3 Mixing things in a vector-Coercion

2.3.1 Implicit coercion

In general, **implicit coercion** is an attempt by R to be flexible with data types. When an entry does not match the expected value, R tries to guess what we meant before throwing in an error.

For example, R assumes that everything in our atomic vector is of the same data type – that is, all numbers or all characters or all logicals. If we create a "mixed" vector such as:

```
my_vector <- c(1, 4, "hello", TRUE)
```

we will not have a vector with two numeric objects, one character object and one logical object. Instead, R will do what it can to convert them all into all the same object type, in this case all character objects. So `my_vector` will contain `1`, `4`, `hello` and `TRUE` as characters.

The hierarchy for coercion is:

`logical < integer < numeric < character`

1. **Example:** `numeric` Vs `character`

```
a <- c(10.5 , 3.2, "I am a character")
a
```

```
## [1] "10.5"             "3.2"              "I am a character"
```

```
typeof(a)
```

```
## [1] "character"
```

Adding a character string to a numeric vector converts all the elements in the vector to character values.

2. **Example:** `logical` Vs `character`

```
b <- c(TRUE, FALSE, "Hello")
b
```

```
## [1] "TRUE"  "FALSE" "Hello"
```

```
typeof(b)
```

```
## [1] "character"
```

Adding a character string to a logical vector converts all the elements in the vector to character values.

3. **Example:** `logical` Vs `numeric`

```
d <- c(FALSE, TRUE, 2)
d
```

```
## [1] 0 1 2
```

```
typeof(d)
```

```
## [1] "double"
```

Adding a numeric value to a logical vector converts all the elements in the vector to double (numeric) values. Logical values are converted to numbers as foloowing: **TRUE is converted to 1** and **FALSE to 0**.

2.3.2 Explicit coercion

R also offers functions to force a specific coercion (**explicit coercion**). For example, we can turn numbers into characters with the `as.character()` function. Let's create a numeric vector `f`, with numbers 1 through 5, and convert it to a character vector `g`:

```
f <- 1:5


g <- as.character(f)
g
```

```
## [1] "1" "2" "3" "4" "5"
```

We can turn the characters back to numbers using the **`as.numeric()`** function which converts characters or other data types into numeric:

```
as.numeric(g)
```

```
## [1] 1 2 3 4 5
```

This function is actually quite useful in practice, because many public datasets that include numbers, include them in a form that makes them appear to be character strings.

Now, suppose we define an object `q` of character strings “1”, “b”, “3” and we want to convert them to numbers using the `as.numeric()` function:

```
q <- c("1", "b", "3")


as.numeric(q)
```

```
## Warning: NAs introduced by coercion
```

```
## [1]  1 NA  3
```

We can see that R is able to convert the strings “1” and “3” to the numeric values 1 and 3, but it does not know what to do with “b”. As a result, if we call `as.numeric()` on this vector, we get a warning that `NAs` introduced by coercion (the element “b” was converted to a missing value `NA`).

Moreover, when nonsensical coercion takes place, we will usually get a warning from R. For example:

```r
x_abc <- c("a", "b", "c")
as.numeric(x_abc)
```

```
## Warning: NAs introduced by coercion
```

```
## [1] NA NA NA
```

2.4 Sort, rank, and order numeric vectors

We have learned how to create atomic vectors of different types including numeric, character and logical. In addition, we know how to create atomic vectors with patterns and how coercion works. An atomic vector usually contains more than one elements. Sometimes, we want to order the elements in various ways. In this section, we will introduce important functions that relate to ordering elements in an atomic vector.

Firstly, let's create a numeric vector which will be used throughout this part.

```r
num_vect <- c(2, 3, 2, 0, 4, 7)
num_vect
```

```
## [1] 2 3 2 0 4 7
```

2.4.1 Sort vectors

The first function we will introduce is `sort()`. By default, the `sort()` function sorts elements in vector in the ascending order, namely from the smallest to largest.

```r
sort(num_vect)
```

```
## [1] 0 2 2 3 4 7
```

If we want to sort the vector in the descending order, namely from the largest to smallest, we can set a second argument `decreasing = TRUE`.

```
sort(num_vect, decreasing = TRUE)
```

```
## [1] 7 4 3 2 2 0
```

2.4.2 Ranks of vectors

Next, let's talk about ranks. The `rank()` function references the position of the value in the sorted vector and is in the same order as the original sequence.

```
num_vect
```

```
## [1] 2 3 2 0 4 7
```

```
rank(num_vect)
```

```
## [1] 2.5 4.0 2.5 1.0 5.0 6.0
```

If we check the values of `num_vect`, we can see that the smallest value of `num_vect` is `0`, which corresponds to the fourth element. Thus, the fourth element has rank 1. The second smallest value of `num_vect` is 2, which is shared at the first and the third elements, resulting a tie (elements with the same value will result in a tie). Normally, these two elements would have ranks 2 and 3. To break the tie, the `rank()` function assigns all the elements involving in the tie (the first and third elements in this example) the same rank, which is average of all their ranks (the average of 2 and 3), by default. In addition to this default behavior for handling ties, `rank()` also provides other options by setting the `ties.method` argument.

If we want to break the ties by the order element appears in the vector, we can set `ties.method = "first"`. Then the earlier appearing element will have smaller ranks than the later one. In this example, the first element will have rank 2 and the third element has rank 3, since the first element appears earlier than the third element.

```
rank(num_vect, ties.method = "first")
```

```
## [1] 2 4 3 1 5 6
```

Note that unlike `sort()`, we can't get positions in the descending order from the `rank()` function, which means we can't add `decreasing = TRUE` in `rank()`.

2.4.3 Order of vectors

The next function we want to introduce is the `order()` function. Note that the function name order could be a bit misleading since ordering elements also has the same meaning of sorting. However, although it is related to sorting, `order()` is a very different function from `sort()`.

Let's recall the values of `num_vect` and apply `order()` on `num_vect`:

```
num_vect
```

```
## [1] 2 3 2 0 4 7
```

```
order(num_vect)
```

```
## [1] 4 1 3 2 5 6
```

From the result, we can see that the `order()` function returns the position of the original value and is in the order of sorted sequence, that is smallest value to largest value.

For example, the first output is 4, indicating the 4th element in `num_vect` is the smallest. The second output is 1, showing the 1st element in `num_vect` is the second smallest.

2.5 Mathematical operations and functions applied to numeric vectors

Mathematical operations applied to all the elements of a **numeric** vector (that is called vectorization):

```
(1:5) * 2
```

```
## [1]  2  4  6  8 10
```

```
2^(1:5)
```

```
## [1]  2  4  8 16 32
```

The same rule is applied to the elements of the vectors using mathematical functions:

```
x_seq  # we recall x_seq that we have defined previously
```

```
## [1] 1 2 3 4 5
```

```
sqrt(x_seq)  # calculate the square root of all the elements of x_seq
```

```
## [1] 1.000000 1.414214 1.732051 2.000000 2.236068
```

We can also round the results using the `round()` function and set the argument `digits = 2`, as following:

```
round(sqrt(x_seq), digits = 2)
```

```
## [1] 1.00 1.41 1.73 2.00 2.24
```

2.6 Relational operators applied between a vector and a scalar

For relational operators (>, <, ==, <=, >=, !=), each element of the vector is compared with a defined value (scalar). The result of comparison is a Boolean value (TRUE or FALSE).

Examples:

```
m <- c(4, 2, 3, 8)
```

```
m > 3
```

```
## [1]  TRUE FALSE FALSE  TRUE
```

```
m >= 3
```

```
## [1]  TRUE FALSE  TRUE  TRUE
```

```
m == 3
```

```
## [1] FALSE FALSE  TRUE FALSE
```

```
m != 3
```

```
## [1]  TRUE  TRUE FALSE  TRUE
```

2.7 Operators applied between two vectors

2.7.1 Arithmetic Operators

The arithmetic operators `(+, -, *, /, ^)` act on each element of the vector.

Examples:

```
v <- c(1, 2, 3)
t <- c(8, 3, 2)

t + v
```

```
## [1] 9 5 5
```

```
t^v
```

```
## [1] 8 9 8
```

```
t + 3 * v / 2
```

```
## [1] 9.5 6.0 6.5
```

Note that R will follow the BODMAS (Brackets, Orders (powers/roots), Division, Multiplication, Addition, Subtraction) rule for the order in which it will carry out calculations.

2.7.2 Relational Operators

For relational operators (>, <, ==, <=, >=, !=), each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a Boolean value (TRUE or FALSE).

Examples:

```
w <- c(2, 5.5, 6, 9)
z <- c(8, 2.5, 14, 9)

z > w
```

```
## [1]  TRUE FALSE  TRUE FALSE
```

```
z == w
```

```
## [1] FALSE FALSE FALSE  TRUE
```

```
z >= w
```

```
## [1]  TRUE FALSE  TRUE  TRUE
```

```
z != w
```

```
## [1]  TRUE  TRUE  TRUE FALSE
```

2.7.3 Logical Operators applied to vectors

The logical (Boolean) operators are:

- `&` (AND)
- `|` (OR)
- `!` (NOT)

Logical operators are applicable to vectors of type logical or numeric. All non-zero values are considered as logical value `TRUE` and all zeros are considered as `FALSE`. The result of comparison is a logical (Boolean) value.

- The `&` operator combines each element of the first vector with the corresponding element of the second vector and gives an output TRUE if both the elements are TRUE.

```
s <- c(1, 0, - 1, 0, TRUE, TRUE, FALSE)
u <- c(2, 0, - 2, 2, TRUE, FALSE, FALSE)
```

```
s
```

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

```
as.logical(s)
```

```
## [1]  TRUE FALSE  TRUE FALSE  TRUE  TRUE FALSE
```

```
u
```

```
## [1]  2  0 -2  2  1  0  0
```

```
as.logical(u)
```

```
## [1]  TRUE FALSE  TRUE  TRUE  TRUE FALSE FALSE
```

Therefore:

```
s & u
```

```
## [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE
```

Additionally, the `&&` operator takes the first element of both vectors and gives TRUE only if both are TRUE.

```
s && u
```

```
## [1] TRUE
```

- The `|` operator combines each element of the first vector with the corresponding element of the second vector and gives an output TRUE if one of the elements is TRUE.

```
s | u
```

```
## [1]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE FALSE
```

Additionally, the `||` operator takes the first element of both vectors and gives TRUE if one of them is TRUE.

```
s || u
```

```
## [1] TRUE
```

- The `!` operator takes each element of the vector and gives the opposite logical value.

```
! s
```

```
## [1] FALSE  TRUE FALSE  TRUE FALSE FALSE  TRUE
```

```
! u
```

```
## [1] FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE
```

2.8 Statistical functions applied to vectors

Statistical functions in R such as `sum()` and arithmetic `mean()` take in the numeric values of a vector and return a single numeric value:

```
x_seq  # we recall x_seq that we have defined previously
```

```
## [1] 1 2 3 4 5
```

```
sum(x_seq) # adds all the elements of a vector
```

```
## [1] 15
```

```r
mean(x_seq) # calculate the arithmetic mean
```

```
## [1] 3
```

```r
sd(x_seq) # calculate the standard deviation
```

```
## [1] 1.581139
```

Next, we add a missing value `NA` in the `x_seq` vector:

```r
x_seq2 <- c(x_seq, NA)
typeof(x_seq2)
```

```
## [1] "integer"
```

We can see that the `x_seq2` vector is of integer type.

However, if we try to calculate the mean of the `x_seq2`, R returns a `NA` value:

```r
mean(x_seq2)
```

```
## [1] NA
```

Therefore, if some of the values in a vector are missing, then the `mean` of the vector is unknown (NA). In this case, it makes sense to remove the `NA` and compute the mean of the other values in the vector setting the `na.rm` argument equals to TRUE:

```r
mean(x_seq2, na.rm = TRUE)
```

```
## [1] 3
```

3 Subsetting vectors

In R, the first element of a vector has an index of 1.

(In many other programming languages [e.g., C and Python], the first element of a vector has an index of 0)

3.1 Subsetting (indexing) a vector using `[ ]`

3.1.1 Extract specific elements of a vector

Having defined a vector, it's often useful to *extract* parts of a vector. We do this with the `[ ]` operator. For example, using the built in `month.name` vector:

```r
month.name[2]    # we extract only the second month of the year
```

```
## [1] "February"
```

```r
month.name[2:4]    # we extract the second, third, and forth month of the year
```

```
## [1] "February" "March"    "April"
```

Let's see the second example analytically. The vector `2:4` generates the sequence 2, 3, 4. This gets passed to the extract operator `[ ]`.

We can also generate this sequence using the vector `c(2, 3, 4)`:

```r
month.name[c(2, 3, 4)]
```

```
## [1] "February" "March"    "April"
```

Values are returned in the order that we specify with the indices:

```
month.name[4:2]  # extraction of 4, 3, 2 elements of the vector
```

```
## [1] "April"    "March"    "February"
```

We can also extract the same element more than once:

```
month.name[c(1, 1, 2, 3, 4)]
```

```
## [1] "January"  "January"  "February" "March"    "April"
```

3.1.2 Missing data (NA) in vectors

Note that if we try and extract an element that doesn't exist in the vector, the missing values are `NA`:

```
month.name[10:13]
```

```
## [1] "October"  "November" "December" NA
```

The `NA` is a special value, that is used to represent "Not Available", or "missing". If we perform computations which include `NA`, the result is usually `NA`:

```
1 + NA
```

```
## [1] NA
```

3.1.3 Skipping and removing elements from vectors

If we use a negative number as the index of a vector, R will return every element *except* for the one specified:

```r
month.name[-2]   # remove the second month February from the vector
```

```
##  [1] "January"   "March"     "April"     "May"       "June"
##  [6] "July"      "August"    "September" "October"   "November"
## [11] "December"
```

We can also skip multiple elements:

```r
month.name[c(-1, -5)]  # remove the first and fifth elements of the vector
```

```
##  [1] "February"  "March"     "April"     "June"      "July"
##  [6] "August"    "September" "October"   "November"  "December"
```

which is equivalent to:

```r
month.name[-c(1, 5)]  # remove the first and fifth elements of the vector
```

```
##  [1] "February"  "March"     "April"     "June"      "July"
##  [6] "August"    "September" "October"   "November"  "December"
```

A common error occurs when trying to skip slices of a vector. Suppose we want to skip the first five elements form the `month.name` vector. Most people first try to negate a sequence like so:

```r
month.name[-1:5]
```

This gives an error:

Error in month.name [-1:5]: only 0's may be mixed with negative subscripts

Remember that the colon operator `:` is a function and in this example takes its first argument as -1, and second as 5, so generates the sequence of numbers: `-1, 0, 1, 2, 3, 4, 5`.

The correct solution is to wrap that function call in brackets, so that the `-` operator is applied to the sequence:

```r
-(1:5)
```

```
## [1] -1 -2 -3 -4 -5
```

```r
month.name[-(1:5)]  # remove the 1st to fifth element of the vector
```

```
## [1] "June"      "July"      "August"    "September" "October"  
## [6] "November"  "December"
```

3.2 Subsetting with logical vectors (indexing by conditon)

As well as providing a list of indices we want to keep (or delete, if we prefix them with -), we can pass a *logical vector* to R indicating the indices we wish to select.

For example, let's say that we want to select only the first four months of the year:

```r
fourmonths <- month.name[1:4]
fourmonths
```

```
## [1] "January"  "February" "March"    "April"
```

which is equivalent to:

```r
month.name[c(TRUE, TRUE, TRUE, TRUE, FALSE, FALSE,
             FALSE, FALSE, FALSE, FALSE, FALSE, FALSE)]
```

```
## [1] "January"  "February" "March"    "April"
```

Furthermore, if we want to exlude “February” from the `fourmonths` vector we should code:

```r
fourmonths[c(TRUE, FALSE, TRUE, TRUE)]
```

```
## [1] "January" "March"   "April"
```

4 Vector recycling

What happens if we supply a logical vector that is shorter than the vector we're extracting the elements from?

For example:

```
fourmonths[c(TRUE, FALSE)]
```

```
## [1] "January" "March"
```

This illustrates the idea of *vector recycling*. The `[ ]` extract operator silently "recycled" the values of the shorter vector `c(TRUE, FALSE)` in order to make the length compatible to the `fourmonths` vector:

```
fourmonths[c(TRUE,FALSE,TRUE,FALSE)]
```

```
## [1] "January" "March"
```

For a further example, suppose we have two vectors `c(1,2,4)` , `c(6,0,9,10,13)`, where the first one is shorter with only 3 elements. Now if we sum these two, we will get a warning message as follows.

```
c(1,2,4) + c(6,0,9,10,13)
```

```
## Warning in c(1, 2, 4) + c(6, 0, 9, 10, 13): longer object length is
## not a multiple of shorter object length
```

```
## [1]  7  2 13 11 15
```

Here R , sums those vectors by recycling or repeating the elements in shorter one, until it is long enough to match the longer one as follows:

```r
c(1, 2, 4, 1, 2) + c(6, 0, 9, 10, 13)
```

```
## [1]  7  2 13 11 15
```

5 Matrices

Every data object in R contains a number of attributes to describe the nature of the information in that object. For example, we can generate objects such as matrices and arrays using the `dim` (dimension) attribute.

A matrix is an atomic vector with two dimensions and it is used to represent two-dimensional data (they have rows and columns) of the same type (numeric, character, or logical).

Using matrices we can perform a matrix algebra operations, a powerful type of mathematical technique. We do not describe analytically these operations in this tutorial, but much of what happens in the background when we perform a data analysis involves matrices.

5.1 Creating a matrix

Adding a dimension attribute to a vector allows it to behave like a 2-dimensional matrix. For example:

```
x_20 <- 1:20

dim(x_20) <- c(5, 4)

x_20
```

```
##      [,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
```

Most often we define a matrix using the `matrix()` function. We need to specify the number of rows and columns.

Example 1: numeric matrix

```
X1 <- matrix(1:20, nrow=5, ncol=4)
X1
```

```
##      [,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
```

The matrix is filled by columns (default column-wise), so entries can be thought of starting in the "upper left" corner and running down the columns. If we want the matrix to be filled by rows we must add an extra argument (`byrow=TRUE`) in the `matrix()` function, as follows:

```
X2 <- matrix(1:20, nrow=5, ncol=4, byrow=TRUE)
X2
```

```
##      [,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
```

The `type` of data, the `class` and the `dimensions` of the `X2` object are:

```
typeof(X2)
```

```
## [1] "integer"
```

```
class(X2)
```

```
## [1] "matrix" "array"
```

```
dim(X2)
```

```
## [1] 5 4
```

Of note, the `typeof()` function gives the type of data that the object includes (*integer*), while the `class` is the type of structure (*matrix*) of the object.

The `dim()` is an inbuilt R function that either sets or returns the dimension of the matrix, array, or data frame. The `dim()` function takes the R object as an argument and returns its dimension (as in our example), or if you assign the value to the `dim()` function, then it sets the dimension for that R Object.

Example 2: logical matrix

```
x_logical <- c(TRUE, FALSE, FALSE, TRUE, FALSE, FALSE)
X3 <- matrix(x_logical, nrow=2, ncol=3)
X3
```

```
##       [,1]  [,2]  [,3]
## [1,]  TRUE FALSE FALSE
## [2,] FALSE  TRUE FALSE
```

The `type` of data, the `class` and the `dimensions` of the `X3` object are:

```
typeof(X3)
```

```
## [1] "logical"
```

```
class(X3)
```

```
## [1] "matrix" "array"
```

```
dim(X3)
```

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

Example 3: character matrix

```
x_char <- c("a", "b", "c", "d", "e", "f")
X4 <- matrix(x_char, nrow=2, ncol=3)
X4
```

```
##      [,1] [,2] [,3]
## [1,] "a"  "c"  "e"
## [2,] "b"  "d"  "f"
```

The `type` of data, the `class` and the `dimensions` of the `X4` object are:

```
typeof(X4)
```

```
## [1] "character"
```

```
class(X4)
```

```
## [1] "matrix" "array"
```

```
dim(X4)
```

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

5.2 Using matrix subscripts

We can identify rows, columns, or elements of a matrix by using subscripts and brackets. Particularly, *X[i,]* refers to the *ith* row of matrix X, *X[, j]* refers to *jth* column, and *X[i, j]* refers to the *ijth* element, respectively.

The subscripts *i* and *j* can be numeric vectors in order to select multiple rows or columns, as shown in the following examples.

```r
X <- matrix(1:10, nrow=2)  # create a 2x5 numeric matrix filled by column
X
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

```r
X[2, ]   # select the 2nd row
```

```
## [1]  2  4  6  8 10
```

```r
X[, 2]  # select the 2nd column
```

```
## [1] 3 4
```

```r
X[1, 4]  # select the element in the 1st row, 4th column
```

```
## [1] 7
```

```r
X[1, c(4, 5)]  # select the elements in the 1st row, 4th and 5th column
```

```
## [1] 7 9
```

5.3 Basic matrix algebra

5.3.1 The identity matrix

A square matrix with ones on the main diagonal and zeros elsewhere:

```
diag(5)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    1    0    0
## [4,]    0    0    0    1    0
## [5,]    0    0    0    0    1
```

5.3.2 The transpose of a matrix

The transpose operation simply changes columns to rows. For example, for a matrix `A`:

```
A <-matrix(c(4, -1, -5, 0, 1, -2), 2, 3, byrow = TRUE)
A
```

```
##      [,1] [,2] [,3]
## [1,]    4   -1   -5
## [2,]    0    1   -2
```

the transpose matrix is:

```
t(A)
```

```
##      [,1] [,2]
## [1,]    4    0
## [2,]   -1    1
## [3,]   -5   -2
```

5.3.3 Multiplying a scalar with a matrix

In scalar multiplication, each element in the matrix is multiplied by the given scalar. For example:

```
A
```

```
##      [,1] [,2] [,3]
## [1,]    4   -1   -5
## [2,]    0    1   -2
```

```
-3 * A
```

```
##      [,1] [,2] [,3]
## [1,]  -12    3   15
## [2,]    0   -3    6
```

5.3.4 Element-wise multiplication of two matrices of the same dimensions

The element-wise multiplication of two matrices, **A** and **B**, of the same dimensions can also be computed with the ∗ operator.

```
A
```

```
##      [,1] [,2] [,3]
## [1,]    4   -1   -5
## [2,]    0    1   -2
```

```
B <-matrix(c(3, 1, -5, 0, 2, -2), 2, 3, byrow = TRUE)
B
```

```
##      [,1] [,2] [,3]
## [1,]    3    1   -5
## [2,]    0    2   -2
```

The output will be a matrix of the same dimensions of the original matrices:

```
A * B
```

```
##      [,1] [,2] [,3]
## [1,]   12   -1   25
## [2,]    0    2    4
```

5.3.5 The dot product (inner product) of two matrices

In R, an inner product of two matrices can be performed with the %*% operator.

```
P <- matrix(c(3, 0, -5, -1, -3, 4), nrow = 2, ncol = 3, byrow = TRUE)
Q <- matrix(c(-5, 5, 2, 1, -2, 0), nrow = 3, ncol = 2, byrow = TRUE)
P
```

```
##      [,1] [,2] [,3]
## [1,]    3    0   -5
## [2,]   -1   -3    4
```

```
Q
```

```
##      [,1] [,2]
## [1,]   -5    5
## [2,]    2    1
## [3,]   -2    0
```

```
P %*% Q
```

```
##      [,1] [,2]
## [1,]   -5   15
## [2,]   -9   -8
```

Before inner multiplying two matrices check that the dimensions are compatible. The number of columns of the first matrix must be equal to the number of rows of the second.

5.3.6 Matrix crossproduct

If we need to calculate the inner product of a matrix and the transpose we can type `t(A) %*% B`, being A and B the names of the matrices.

```
A
```

```
##      [,1] [,2] [,3]
## [1,]    4   -1   -5
## [2,]    0    1   -2
```

```
B
```

```
##      [,1] [,2] [,3]
## [1,]    3    1   -5
## [2,]    0    2   -2
```

```
t(A) %*% B
```

```
##      [,1] [,2] [,3]
## [1,]   12    4  -20
## [2,]   -3    1    3
## [3,]  -15   -9   29
```

However, in R it is more efficient and faster using the `crossprod`:

```
crossprod(A, B)
```

```
##      [,1] [,2] [,3]
## [1,]   12    4  -20
## [2,]   -3    1    3
## [3,]  -15   -9   29
```

5.3.7 The determinant of a matrix

The determinant is a scalar value that is a function of the entries of a square matrix:

```
M <- matrix( c(5, 1, 0, 3,-1, 2, 4, 0,-1), nrow = 3, byrow = TRUE)
M
```

```
##      [,1] [,2] [,3]
## [1,]    5    1    0
## [2,]    3   -1    2
## [3,]    4    0   -1
```

```
det(M)
```

```
## [1] 16
```

5.3.8 The inverse of a matrix

The `det(M)` is not zero, so inverse exists:

```
M_inv  <- inv(M)
M_inv
```

```
##         [,1]    [,2]   [,3]
## [1,] 0.0625  0.0625  0.125
## [2,] 0.6875 -0.3125 -0.625
## [3,] 0.2500  0.2500 -0.500
```

The inverse of a matrix `M` is defined as the `M_inv` matrix which multiplies M to give the identity matrix:

```
M_inv %*% M
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

5.3.9 Symmetric matrix

In linear algebra, a **symmetric** matrix is a square matrix that is equal to its transpose. For example:

```
S <- matrix(c(13, -4, 2, -4, 11, -2, 2, -2, 8), 3, 3, byrow = TRUE)
S
```

```
##      [,1] [,2] [,3]
## [1,]   13   -4    2
## [2,]   -4   11   -2
## [3,]    2   -2    8
```

```
t(S)
```

```
##      [,1] [,2] [,3]
## [1,]   13   -4    2
## [2,]   -4   11   -2
## [3,]    2   -2    8
```

A symmetric matrix guarantees that its eigenvalues are real numbers. Eigenvalues and eigenvectors are highly used by the data scientists as they are the core of the data science field. For example, eigenvalues and eigenvectors are very much useful in the principal component analysis which is a dimensionality reduction technique in machine learning and is highly used in the field of data science.

The `eigen()` built-in function in R calculates the eigenvalues and eigenvectors of a symmetric matrix. It returns a named list, with eigenvalues named values and eigenvectors named vectors:

```
ev <- eigen(S)
ev
```

```
## eigen() decomposition
## $values
## [1] 17  8  7
##
## $vectors
##            [,1]       [,2]      [,3]
## [1,]  0.7453560  0.6666667 0.0000000
## [2,] -0.5962848  0.6666667 0.4472136
## [3,]  0.2981424 -0.3333333 0.8944272
```

The eigenvalues are always returned in decreasing order, and each column of vectors corresponds to the elements in values.

5.3.10 Application: calculation of the average using matrices

```r
my_values <- c(2, 5, 7, -4, 8, 6, 3)
mean(my_values)
```

```
## [1] 3.857143
```

```r
n <- length(my_values)  # get the length (number of elements) of vector
U <- matrix(1, n, 1)
U
```

```
##      [,1]
## [1,]    1
## [2,]    1
## [3,]    1
## [4,]    1
## [5,]    1
## [6,]    1
## [7,]    1
```

```r
V <- matrix(my_values, n, 1)
V
```

```
##      [,1]
## [1,]    2
## [2,]    5
## [3,]    7
## [4,]   -4
## [5,]    8
## [6,]    6
## [7,]    3
```

```r
average_my_values <- t(U) %*% V/n
average_my_values
```

```
##          [,1]
## [1,] 3.857143
```

6 Arrays

Arrays are similar to matrices but can have more than two dimensions. They're created with an `array()` function from base R:

```r
# build the 2x3x4 array
my_array <- array(1:24, dim = c(2,3,4))
my_array
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   13   15   17
## [2,]   14   16   18
##
## , , 4
##
```

```
##      [,1] [,2] [,3]
## [1,]   19   21   23
## [2,]   20   22   24
```

As you can see, arrays are a natural extension of matrices. They can be useful in programming new statistical methods. Like matrices, they contain a single type of data (e.g., numeric).

We can find the `type`, `class` and the `dimensions` of the array:

```
typeof(my_array)
```

```
## [1] "integer"
```

```
class(my_array)
```

```
## [1] "array"
```

```
dim(my_array)
```

```
## [1] 2 3 4
```

7 Lists

A list in R allows us to gather a variety of objects under one name (that is, the name of the list) in an ordered way. These objects can be matrices, vectors, data frames, even other lists, etc. It is not even required that these objects are related to each other in any way. We could say that a list is some kind super data type: we can store practically any piece of information in it!

7.1 Creating a list

We construct a list using the `list()` function. The list items (or components of a list) can be matrices, vectors, other lists. For example:

```r
my_list <- list(1:5, c("apple", "orange"), TRUE)
my_list
```

```
## [[1]]
## [1] 1 2 3 4 5
##
## [[2]]
## [1] "apple"  "orange"
##
## [[3]]
## [1] TRUE
```

This list consists of three components that are atomic vectors of different types of data (numeric, characters, and logical).

We can assign names to the list items:

```r
my_list <- list(
              numbers = 1:5,
              strings = c("apple", "orange"),
              logicals = TRUE)
my_list
```

```
## $numbers
## [1] 1 2 3 4 5
##
## $strings
## [1] "apple"  "orange"
##
## $logicals
## [1] TRUE
```

We can also confirm that the class of the object is `list`:

```r
class(my_list)
```

```
## [1] "list"
```

7.2 Subsetting a list

7.2.1 Subset list and preserve output as a list

We can use the `[ ]` operator to extract one or more list items while preserving the output in list format:

```
my_list[1]     # extract the first list item
```

```
## $numbers
## [1] 1 2 3 4 5
```

```
class(my_list[1])
```

```
## [1] "list"
```

7.2.2 Subset list and simplify the output

We can use the `[[ ]]` to extract one or more list items while simplifying the output:

```
my_list[[1]]    # extract the first list item and simplify it to a vector
```

```
## [1] 1 2 3 4 5
```

```
class(my_list[[1]])
```

```
## [1] "integer"
```

```
my_list[["numbers"]]    # same as above but using the item's name
```

```
## [1] 1 2 3 4 5
```

We can also access the content of the list by typing the name of the list followed by a dollar sign `$` folowed by the name of the list item:

```
my_list$numbers  # extract the numbers and simplify to a vector
```

```
## [1] 1 2 3 4 5
```

One thing that differentiates the `[[` operator from the `$` is that the `[[` operator can be used with computed indices and names. The `$` operator can only be used with names.

Note It's important to understand the difference between simplifying and preserving subsetting. Simplifying subsets returns the simplest possible data structure that can represent the output. Preserving subsets keeps the structure of the output the same as the input.

7.3 Subset list to get individual elements out of a list item

To extract individual elements out of a specific list item combine the `[[` (or $) operator with the `[` operator:

```
my_list[[2]][2]  # using the index
```

```
## [1] "orange"
```

```
my_list[["strings"]][2]  # using the name of the list item
```

```
## [1] "orange"
```

```
my_list$strings[2]
```

```
## [1] "orange"
```

7.4 Unlist a list

We can turn a list into an atomic vector with `unlist()`:

```
my_unlist <- unlist(my_list)
my_unlist
```

```
## numbers1 numbers2 numbers3 numbers4 numbers5 strings1 strings2
##      "1"      "2"      "3"      "4"      "5"  "apple" "orange"
## logicals
##   "TRUE"
```

```
class(my_unlist)
```

```
## [1] "character"
```

7.5 Nested List

A list can contain sublists, which in turn can contain sublists themselves, and so on. This is known as **nested** list or recursive vectors.

```
lst <- list(item1 = 3.14,
            item2 = list(seq_num = 5:10,
                         seq_char = c("a","b","c")))


lst
```

```
## $item1
## [1] 3.14
```

```
##
## $item2
## $item2$seq_num
## [1]  5  6  7  8  9 10
##
## $item2$seq_char
## [1] "a" "b" "c"
```

7.6 Subsetting Nested List

We can access individual elements in a nested list by using the combination of [[ ]] or $ operator and the [ ] operator.

```
# preserve the output as a list
lst[[2]][1]
```

```
## $seq_num
## [1]  5  6  7  8  9 10
```

```
class(lst[[2]][1])
```

```
## [1] "list"
```

```
# same as above but simplify the output
lst[[2]][[1]]
```

```
## [1]  5  6  7  8  9 10
```

```
class(lst[[2]][[1]])
```

```
## [1] "integer"
```

```
# same as above with names
lst[["item2"]][["seq_num"]]
```

```
## [1]  5  6  7  8  9 10
```

```
# same as above with $ operator
lst$item2$seq_num
```

```
## [1]  5  6  7  8  9 10
```

```
# extract individual element
lst[[2]][[2]][3]
```

```
## [1] "c"
```

```
class(lst[[2]][[2]][3])
```

```
## [1] "character"
```

8 Data frames

A data frame is the most common way of storing data in R and, generally, is the data structure most often used for data analyses.

A **data frame** is a special type of list with equal-length atomic vectors. Each component of the list can be thought of as a column and the length of each component of the list is the number of rows.

8.1 Basic characteristics

- Each **column** of a data frame is an **atomic vector** and remember that the data in an atomic vector must only be of one type (numeric, character, or logical).

- Data frames are similar to the **datasets** we'd typically see in spreadsheets (e.g., Excell files, Google sheets, LibreOffice Calc).

- Data frames are the most common data structure we'll deal with in R. A data frame can be created with the `data.frame()` function in base R, the `tibble()` function in the tidyverse package (an improvement over data.frame), or the "fast" `data.table()` in the {data.table} package.

Different traditions have different names for the **rows** and **columns** of a dataset. Statisticians refer to them as observations and variables, database analysts call them records and fields, and those from the data mining/machine learning disciplines call them examples and attributes. We'll use the terms **observations** and **variables** throughout this textbook.

The two basic variable categories are the numerical variables and the categorical variables. On the image below, we can view briefly all categories and variable types.

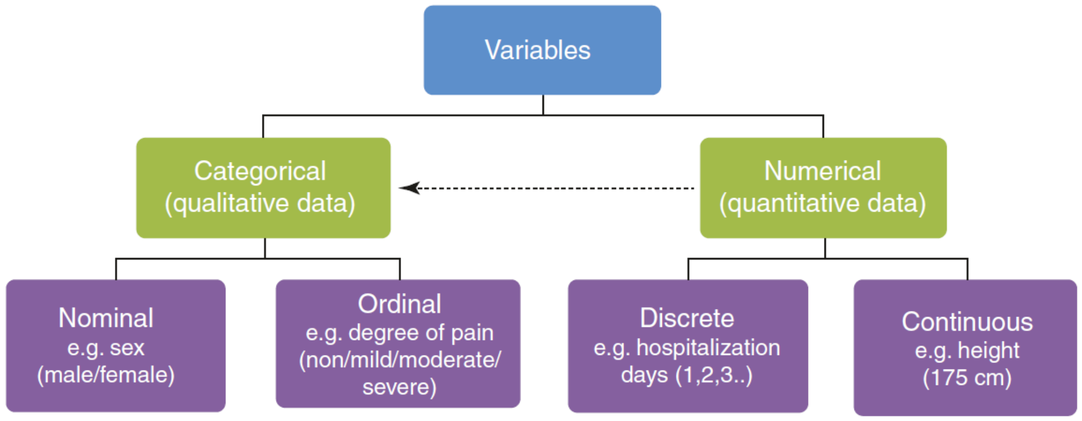
Variables
Categorical (qualitative data)
Numerical (quantitative data)
Nominal e.g. sex (male/female)
Ordinal e.g. degree of pain (non/mild/moderate/ severe)
Discrete e.g. hospitalization days (1,2,3..)
Continuous e.g. height (175 cm)

Figure 2: Different types of variables

Numerical variables

`Numerical` variables take arithmetic values. They can be subdivided in discrete or continuous variables and are expressed in a unit of measure.

- `Discrete` variables can only take values of a countable set of numbers (which are usually the whole numbers 0, 1, 2, 3, etc.). Examples of discrete variables are the heart rate (beats/min), the number of visits to a GP in a year and the hospitalization days.
- `Continuous` variables have no limitation on the values that they can take. The baseline characteristics such as weight, height or blood pressure of the participants are examples of common continuous variables in a study. However, the actual measurements are restricted by the accuracy of the method used for measuring the value.

Categorical variables

Variables are categorical when their data are placed into distinct groups with appropriate labels according to some qualitative characteristic or attribute, for instance place of birth, ethnic group, or type of drug. Categorical variables can be further divided into either nominal or ordinal variables.

- `Nominal` variables have two or more categories, such as sex (male or female) or blood group (A, B, AB, or O), without natural ordering. A nominal variable like sex (male or female) or survival status (alive or dead) is also called binary or dichotomous variable.

- `Ordinal` variables have an intrinsic order such as degree of pain (none, mild, moderate, or severe). Note that ordinal variables can be created from numeric variables. For example, the continuous variable of BMI is usually categorized into four categories: `underweight`, `obese`, `normal`, and `overweight`. However, there is a cost of loosing information with this categorization.

Note The names of the categories may be numerical, but the numbers are simply codes to identify the groups into which the individuals are divided (e.g., "0 = alive" and "1 = dead").

8.2 Creating a data frame with `tibble()`

We can create a data frame from column vectors with the `tibble()` function from `tibble` package which belongs to the "tidyverse" family of packages. A tibble is a specific kind of data frame and is short for "tidy table".

In order to use the tidyverse packages, our input data frames must be in "tidy" format (long/narrow format). In tidy data:

1. Each variable forms a column.

2. Each observation forms a row.

3. Each type of observational unit forms a table.

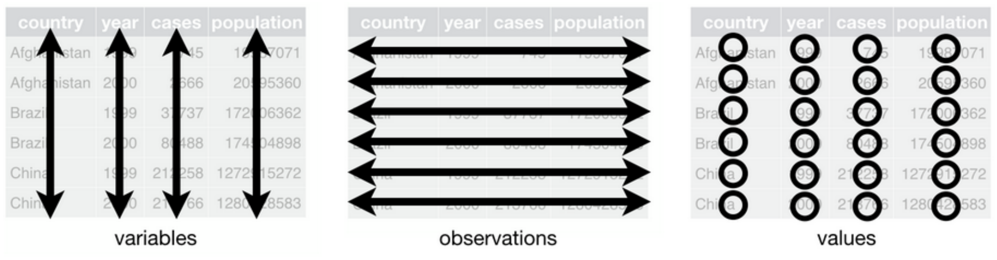

Figure 3: Tidy data graphic from R for Data Science

Here, we provide a simple example of patient dataset:

```r
patientID <- c(1, 2, 3, 4, 5, 6, 7, 8)
age <- c(25, 30, 28, 22, 31, 45, 37, 43)
weight <- c(94, 83, 71, 87, 94, 73, 89, 74)
diabetes <- c("Type 1", "Type 2", "Type 1", "Type 1",
              "Type 2", "Type 1", "Type 1", "Type 2")
status <- c("Poor", "Improved", "Excellent", "Poor",
            "Poor","Excellent", "Improved", "Improved")
dates <- ymd("2020-10-09", "2020-10-12", "2020-10-18", "2020-10-27",
           "2020-11-04", "2020-11-09", "2020-11-22", "2020-12-02")

patient_data <- tibble(patientID, age, weight, diabetes, status, dates)
patient_data
```

```
## # A tibble: 8 x 6
##   patientID   age weight diabetes status    dates
##       <dbl> <dbl>  <dbl> <chr>    <chr>     <date>
## 1         1    25     94 Type 1   Poor      2020-10-09
## 2         2    30     83 Type 2   Improved  2020-10-12
## 3         3    28     71 Type 1   Excellent 2020-10-18
## 4         4    22     87 Type 1   Poor      2020-10-27
## 5         5    31     94 Type 2   Poor      2020-11-04
## 6         6    45     73 Type 1   Excellent 2020-11-09
## 7         7    37     89 Type 1   Improved  2020-11-22
## 8         8    43     74 Type 2   Improved  2020-12-02
```

In this data frame, `patientID` is a row or case identifier, `age` (in years) and `weight` (in kg) are continuous variables, `diabetes` (Type 1, Type 2) is a nominal (dichotomus) variable, `status` is an ordinal variable (Poor/Improved/Excellent), and `dates` variable with dates (note that we used the `ymd()` function to create this variable).

First, let's take a look at the structure of the object *patient_data* with the `glimpse()`:

```
glimpse(patient_data)
```

```
## Rows: 8
## Columns: 6
## $ patientID <dbl> 1, 2, 3, 4, 5, 6, 7, 8
## $ age       <dbl> 25, 30, 28, 22, 31, 45, 37, 43
## $ weight    <dbl> 94, 83, 71, 87, 94, 73, 89, 74
## $ diabetes  <chr> "Type 1", "Type 2", "Type 1", "Type 1", "Type 2", ~
## $ status    <chr> "Poor", "Improved", "Excellent", "Poor", "Poor", "~
## $ dates     <date> 2020-10-09, 2020-10-12, 2020-10-18, 2020-10-27, 20~
```

We can also find the **type**, **class** and **dim** for the data table:

```
typeof(patient_data)
```

```
## [1] "list"
```

```
class(patient_data)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
dim(patient_data)
```

```
## [1] 8 6
```

The type is a *list* but the class is a `tbl` *(tibble)* object which is a “tidy” data frame. The dimensions are 8x6.

The `attribute()` function help us to explore the characteristics/atributes of the tibbles :

```
attributes(patient_data)
```

```
## $class
## [1] "tbl_df"     "tbl"        "data.frame"
##
## $row.names
## [1] 1 2 3 4 5 6 7 8
##
## $names
## [1] "patientID" "age"       "weight"    "diabetes"  "status"
## [6] "dates"
```

8.3 Access only one variable at a time from a tibble

To access the variable *age*, we can use the **dollar sign ($)** like this:

```
patient_data$age
```

```
## [1] 25 30 28 22 31 45 37 43
```

For example, if we want to **cross tabulate** diabetes type by status, we could use the following code:

```
table(patient_data$diabetes, patient_data$status)
```

```
##
##          Excellent Improved Poor
##   Type 1         2        1    2
##   Type 2         0        2    1
```

Note The **“table()"** function builds a contingency table of the counts at each combination of levels of the variables.

8.4 Access variables using the with() function

It can get tiresome typing patient_data$ at the beginning of every variable name. An altrernative approach is to use `with()` function. For example:

```
with(patient_data, table(diabetes, status))
```

```
##         status
## diabetes Excellent Improved Poor
##   Type 1         2        1    2
##   Type 2         0        2    1
```

8.5 Factors

Categorical (nominal) and ordered categorical (ordinal) variables in R are usually transformed to **factors**. Factors can contain only predefined values and are crucial in R because they determine how data will be analyzed in statistical models and presented visually.

In our example, we should convert the `diabetes` and `status` variables from character to factor variables. This can be done by applying the `convert_as_factor()` function from the `rstatix` package:

```
# convert from character to factor
patient_data <- convert_as_factor(patient_data, diabetes, status)
patient_data
```

```
## # A tibble: 8 x 6
##   patientID   age weight diabetes status    dates
##       <dbl> <dbl>  <dbl> <fct>    <fct>     <date>
## 1         1    25     94 Type 1   Poor      2020-10-09
## 2         2    30     83 Type 2   Improved  2020-10-12
## 3         3    28     71 Type 1   Excellent 2020-10-18
## 4         4    22     87 Type 1   Poor      2020-10-27
## 5         5    31     94 Type 2   Poor      2020-11-04
## 6         6    45     73 Type 1   Excellent 2020-11-09
## 7         7    37     89 Type 1   Improved  2020-11-22
## 8         8    43     74 Type 2   Improved  2020-12-02
```

Now, we can inspect the order of the levels for both factor variables using the `levels()` function:

```
levels(patient_data$diabetes) # show the levels of diabetes variable
```

```
## [1] "Type 1" "Type 2"
```

```
levels(patient_data$status) # show the levels of status variable
```

```
## [1] "Excellent" "Improved"  "Poor"
```

Additionally, we may want to reorder the levels in the `status` variable using the `fct_relevel()` function from the `{forcats}` package. In this case, the argument `rev` in the function will reverse the order:

```
# reverse the order of the levels
patient_data$status <-  fct_relevel(patient_data$status, rev)


levels(patient_data$status)  # show the order of the levels
```

```
## [1] "Poor"      "Improved"  "Excellent"
```

or using the `fct_rev()` from `{forcats}` package:

```
patient_data$status <- fct_rev(patient_data$status)
```

The `status` variable has a natural ordering between its categories. If we want to compare the values, we have to pass this information to R.

```
patient_data$status <-  factor(patient_data$status, ordered = TRUE)


patient_data$status
```

```
## [1] Poor      Improved  Excellent Poor      Poor      Excellent
## [7] Improved  Improved
## Levels: Poor < Improved < Excellent
```

Now we can use, for example, the comparison operators > to check whether one element of the ordered vector is larger than the other.

```
patient_data$status[2] > patient_data$status[6]
```

```
## [1] FALSE
```

Factors are also useful when we know the set of possible values (based on theory or experimental design, not of the data) but **they're not all present** in a given dataset. In contrast to a character vector, when we tabulate a factor (e.g. using the `table()` function) we will get counts of all categories, even unobserved ones:

```
sex_char <- c("m", "m", "m")
sex_factor <- factor(sex_char, levels = c("m", "f"))

table(sex_char)
```

```
## sex_char
## m
## 3
```

```
table(sex_factor)
```

```
## sex_factor
## m f
## 3 0
```

8.5.1 Dates in R

There are three types of date/time data that refer to an instant in time:

- A **date**. Tibbles print this as `<date>`.
- A **time** within a day. Tibbles print this as `<time>`.
- A **date-time** is a date plus a time: it uniquely identifies an instant in time (typically to the nearest second). Tibbles print this as `<dttm>`. Elsewhere in R these are called POSIXct.

To get the current date or date-time you can use `Sys.Date()` or `now()`:

```
Sys.Date()
now()
```

Date vectors are built on top of double vectors but they have the special class “Date”:

```
today <- Sys.Date()
typeof(today)
```

```
## [1] "double"
```

```
class(today)
```

```
## [1] "Date"
```

The value of the double (which can be seen by stripping the class), represents the number of days since `1970-01-01`:

```r
unclass(today)
```

```
## [1] 18907
```

We've seen one approach to parsing strings into `dates` variable in our data frame using the `ymd()` function from `lubridate` package. This is the most concise way to create a single date/time object, as we might need when filtering date/time data. The class of this object is:

```r
class(patient_data$dates)
```

```
## [1] "Date"
```

9 Advanced topics

9.1 Copy-on-modify (vectors)

Consider the following code. It binds `x` and `y` to the same underlying value, then modifies `y`:

```
x <- c(1, 2, 3)
y <- x

y[[3]] <- 4
x
```

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

```
y
```

```
## [1] 1 2 4
```

```
obj_addr(x) # get the x current address
```

```
## [1] "0x25ea4050"
```

```
obj_addr(y) # get the y current address
```

```
## [1] "0x25eab1e0"
```

Modifying `y` clearly didn't modify `x`. So what happened to the shared binding? While the value associated with y changed, the original object did not. Instead, R created a new object in the memory location, 0x25eab1e0, a copy of the object with memory location

0x25ea4050 with one value changed, then rebound y to that object. This behaviour is called **copy-on-modify**. Understanding it will radically improve our intuition about the performance of R code.

If we modify `y` again, it won't get copied. That's because the new object now only has a single name bound to it, so R applies **modify-in-place** optimisation. However, when exploring copy-on-modify behaviour interactively, we should be aware that we'll get different results inside of RStudio. That's because the environment pane must make a reference to each object in order to display information about it. This distorts the interactive exploration but it doesn't affect code inside of functions, and so doesn't affect performance during data analysis. For experimentation, we recommend running R directly from the terminal.

9.2 Copy-on-modify (data frames)

Data frames are lists of vectors, so copy-on-modify has important consequences when we modify a data frame. Take this data frame as an example:

```
d1 <- data.frame(x = c(1, 5, 6), y = c(2, 4, 3))
```

If we modify a column, only that column needs to be modified; the others will still point to their original references:

```
d2 <- d1
d2[, 2] <- d2[, 2] * 2


ref(d1, d2)
```

```
## o [1:0x26797488] <df[,2]>
## +-x = [2:0x3bdbce18] <dbl>
## \-y = [3:0x3bdbcdc8] <dbl>
##
## o [4:0x26ad02b0] <df[,2]>
## +-x = [2:0x3bdbce18]
## \-y = [5:0x27051728] <dbl>
```

However, if we modify a row, every column is modified, which means every column must be copied:

```
d3 <- d1
d3[1, ] <- d3[1, ] * 3

ref(d1, d3)
```

```
## o [1:0x26797488] <df[,2]>
## +-x = [2:0x3bdbce18] <dbl>
## \-y = [3:0x3bdbcdc8] <dbl>
##
## o [4:0x27192cd0] <df[,2]>
## +-x = [5:0x1d92e1a0] <dbl>
## \-y = [6:0x1d92e150] <dbl>
```

9.3 Object sizes

We can find out how much memory an object takes with `obj_size()` from package `lobstr`:

```
x_unif <- runif(1e6)
obj_size(x_unif)
```

```
## 8,000,048 B
```

```
y_list <- list(x_unif, x_unif, x_unif)
obj_size(y_list)
```

```
## 8,000,128 B
```

we observe that `y_list` is only 80 bytes bigger than `x_unif`. That's the size of an empty list with three elements:

```
obj_size(list(NULL, NULL, NULL))
```

```
## 80 B
```

10 Activities

Activity 1

Try the following:

```
help(median)
?sd
?max
```

Which of the following are valid R object names?

```
min_age
_age
max.temperature
nik(2020)
BmiCategorical
min-BMI
4height
number_of_deaths
.2way
```

What will be the value of each object after each statement in the following program?

```
mass <- 47.5
age <- 72
mass <- mass * 2.3
age <- age - 20
```

What are the class of the following vectors:

```r
x <- c(T, 14, pi)
y <- c(T, "red", 45)
```

Find the elements of the following vector:

```r
c(c(1, 2, 3, 4, 5) * 1, c(1, 2, 3, 4, 5) * 2, c(1, 2, 3, 4, 5) * 3)
```

Consider the following vector:

```r
w <- c(4,6,5,7,10,9,4,15)
```

What is the value of:

```r
w < 7
```

What do the following pairs of examples do?

```r
ceiling(18.33); signif(9488, 3)
exp(1); log10(1000)
sign(-2.9); sign(32)
abs(-27.9); abs(11.9)
```

Given the following code:

```r
x <- c(5.4, 6.2, 7.1, 7.5, 4.8)
x
```

Come up with at least 3 different commands that will produce the following output:

```
x[2:4]
```

If the following vectors are given:

```
x <- c(2, 4, 6, 8)
y <- c(TRUE, TRUE, FALSE, TRUE)
```

calculate:

```
x + y
2*x + y^2
sum(x[y])
```

Calculate the following:

```
ages <- c(20, 21, 19, 22, 19, 20, 22, 21, 20, 19, 21)
sum(ages >= 21)
```

compare the result to:

```
sum(ages[ages >= 21])
```

Activity 2

We collect a survey about ten stocks position referred to last day, getting the sequence `U D D D U U U U D U`, where `U` stands for "up" and `D` for "down." Create a factor vector with this information and specify that the value "up" is greater than "down".

Activity 3

Create a matrix 3×5 containing all consecutive integer numbers between 16 and 30 by columns.

Activity 4

Given the `y <- c(1, 5, 7, 4, 3, 8, 2, 1, 4, 7)`, find the sample variance using the matrix notation.

Activity 5

Can you explain the output of the following code?

```r
a <- runif(1e6)
obj_size(a)

b <- list(a, a)
obj_size(b)
obj_size(a, b)



b[[1]][[1]] <- 10
obj_size(b)
obj_size(a, b)

b[[2]][[1]] <- 10
```

```
obj_size(b)

obj_size(a, b)
```