R Training: Session 2

Data Types and Structures

Theresa Wacker

30/09/22

Contents

1 *Previously on R Workshops* : topics covered in the last section

In R Workshop 1, we. . . :

- . . . installed R Studio and got acquainted with it.
- . . . used basic commands and created an object.
- . . . got to know some of R's build in data sets.
- . . . learned how to create and save an R script.
- . . . set a working directory.
- . . . imported our own data set.
- . . . learned about best practices.

If many of these learning outcomes sound unfamiliar to you, especially if you have not installed RStudio yet, you may want to consider doing R Workshop 1 before today's workshop.

2 Learning outcomes

Today you will learn about:

- datatypes

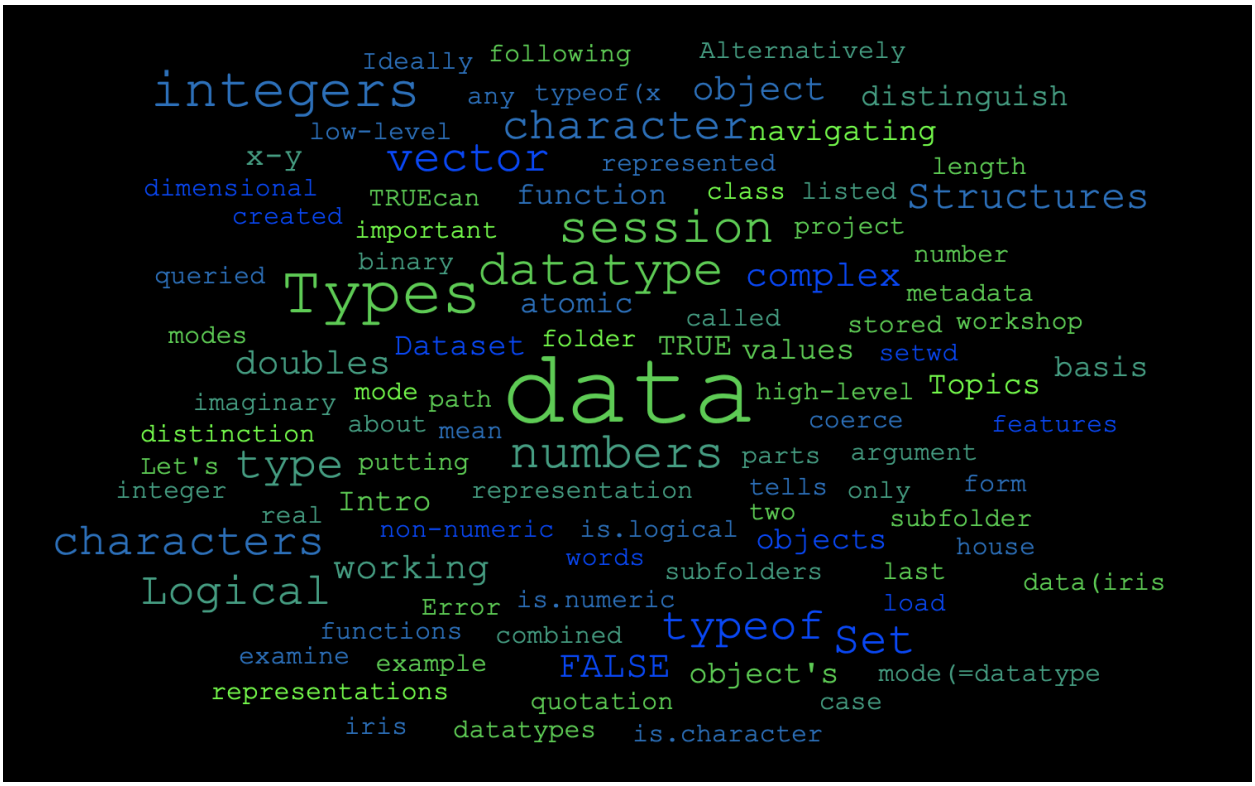


Figure 1: a word cloud image (black background, green/blue words of words related to the Session like data, types, vector, integers, characters and so on.)

  - which data types exist?
  - how do you examine data with respect to data types?
  - how do you convert data types and how to do data type coercion.
- data structures:
  - vectors & lists
  - matrices & data frames
  - indexing of data structures
  - restructuring of data structures
  - data structure exploration
- factors.
- naming conventions.

After today's lesson you should:

- Define the characteristics of the data types and structures that exist and be able to identify the data type of an object.
- Be able to convert/coerce data types and know the hierarchy of coercion.
- Know how to name your R objects.
- Be able to explore R objects/data structures.
- Be able to construct different data structures and retrieve elements from them.
- Know how to restructure your data.

Let the fun begin!

3 Set up

After you **open RStudio**, in the console, **set your working directory**. There, you can either navigate to "Session" → "Set Working Directory" → "Choose Directory..." and navigate to the directory. Alternatively you can set your working directory using the function "setwd" as shown below in your console. As a *reminder*, the console is the lower left pane in RStudio.

> *Ideally you created a workshop folder with subfolders for each session in the last training session. In that case, create a new subfolder for session 2.*

```
setwd("C:/Users/YourUserName/OneDrive -University of Exeter/YourFolder")
```

Your working directory is now the folder you created.

There, navigate to "File" → "New File" → "RScript". You can save all the Tasks in that file you created.

4 Dataset

Read or load in the data for this session. We will be using the built-in R data sets called "iris" & "mtcars". As a *reminder*, with `data()`, we can load data sets that are built-in R (you automatically installed them when you installed R).

```
data(iris)
data(mtcars)
```

Task 1

> 1.) Explore which other data sets are available by typing `data()` and running that line of code in your console.

5 Topics

5.1 Intro to Data Types and Structures

5.1.1 Data Types - *The basis of it all*

Data types, or modes, define how the values are stored in the computer. For your computer (and for R), there are ***3 core data types*:**

- Numeric
- Character
- Logical

Elements of these data types may be combined to form data structures, such as atomic vectors. When we call a vector *atomic*, we mean that the vector only holds data of a single data type. More detail in the next section "Data Structures".

Numeric
The **numeric** data type consists of numbers.

Those can be:

- **integers:** `1,-3,22,0,1890, 2L` (The L explicitely tells R that this is an integer)
- **doubles:** `15.5, 0.01, -0.0004`

Additionally, you can have **complex** numbers:

- **complex:** `1+4i` (complex numbers with real and imaginary parts) [1]

Characters
The **character** data type consists of letters or words such as "a", "f", "datatypes", "Learning R is fun".

- **character**: `Superkalifragilistikexpialegetisch, abc, @%£$`

Importantly, also ***numbers can be characters***. To coerce numbers into characters, you can use `""`. The distinction between a character representation of a number and a numeric one is important. For example:

```
x=3
y=1.5
```

We can subtract them from each other:

```
x-y
```

```
## [1] 1.5
```

However, if we do this with the character representations of the numbers 3 and 1.5, the following happens:

```
x="3"
y="1.5"
x-y
```

```
## Error in x - y: non-numeric argument to binary operator
```

Logical

Logical values can either be `TRUE` or `FALSE`. `TRUE`can also be represented as `1` and `FALSE` as `0`. That means that if you type :

[1] Interestingly, internally, complex numbers are stored as a pair of doubles. That means R knows that this is a complex number, to your computer, however, it is just a set of doubles.

```
x=as.logical(c(1,0,0,1))
```

and use the function `typeof()` which returns the datatype of an object, it will return the type 'logical':

```
typeof(x)
```

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

Task 2 & 3

2.) Find out, using RStudio Help in the menu or `help('as.logical')` in the console, what the command `as.logical()` does.
3.) If you do not use `as.logical()` around `c(1,0,0,1)` (*i.e.* `x=c(1,0,0,1)`), what does `typeof(x)` return and why [make a prediction first, then try out]?

5.1.2 Examining datatypes: *What and how are you?*

R provides many functions to examine features of objects. Some are listed here:

- `typeof()` - what is the object's data type (on the data storage level ("what the computer sees"))?
- `class()` - what is the object's data type (on the abstract type level("what R sees"))?
- `mode()` - what is the object's data type (on the data storage level ("what the computer sees"))?
- `length()`- how long is it?
- `attributes()` - does it have any metadata?
- `str` - display the internal structure of an object.
- `is.numeric()`, `is.character()`, `is.complex()`, `is.logical()` - returns TRUE when an object is the datatype queried, FALSE if not

Some Examples:

```
x <- "dataset"
typeof(x)
```

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

```
attributes(x)
```

```
## NULL
```

```
y <- 1:10
y
```

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

```
typeof(y)
```

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

```
length(y)
```

```
## [1] 10
```

Task 4 & 5

4.) Create the following objects:`x=8y=8.9` Use both `typeof()` and `mode()` to determine the datatypes of `x` and `y`. What differences do you observe?
5.) When you use `class()` on the `iris` dataset (*e.g.* `class(iris)`) and when you use `typeof()` on the iris data set, which results do you get? The resulting data types/structures will be explained later in detail; keep the results in mind for 4.1.4.3 and 4.1.4.4.

Before we look at data structures, we need to look at naming conventions

5.1.3 Naming R objects

You can use any combination of alphanumeric characters, along with dots and underscores, to name an R object. But there are a few ***exceptions:***

- Names cannot start with a number;
- Names cannot have spaces;
- Names cannot be a standalone number such as `12` or `0.34`;
- Names cannot be a reserved word such as `if`, `else`, `function`, `TRUE`, `FALSE` and `NULL` just to name a few (to see the full list of reserved words, type `?Reserved`).

Examples of ***valid names*** include `a`, `dat2`, `cpi_index`, `.tmp`, and even a standalone dot `.` (though a dot can make reading code difficult under certain circumstances).

Examples of ***invalid names*** include `1dat`,`dat 2` (note the space between `dat` and `2`), `df-ver2` (the dash is treated as a mathematical operator), and `Inf` (the latter is a reserved word listed in the `?Reserved` help document).

You can mix cases, but use upper cases with caution since some letters look very much the same in both lower and upper cases (e.g. s and S).

Task 6

6.) Check out `?Reserved` and take a look at the reserved words!

5.1.4 Data Structures

R has many **data structures**. These include:

- vectors:
  - atomic vector
  - list
- matrix
- data frame
- factors

A vector is the most common and basic data structure in R. Technically, **vectors can be one of two types**:

- atomic vectors
- lists

The term "vector" most commonly refers to the atomic types not to lists.

When we call a vector **atomic**, we mean that the vector only holds elements (or atoms) of a single data type and each atom is a scalar, which means it "has length one". It is ***homogenous***. A **list** is still vector, but it can be ***heterogenous***, that means: lists can contain several datatypes and each atom of the list can itself be composed of more than one atom (has a length > one).

Atomic vectors
Let's first take a look at atomic vectors:

```
v_log <- c(TRUE, FALSE, FALSE, TRUE)
v_log
```

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

```
(v_int <- 1:4)
```

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

```
(v_doub <- 1:4 * 1.2)
```

```
## [1] 1.2 2.4 3.6 4.8
```

```
(v_char <- letters[1:4])
```

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

Task 6 & 8

7.)What do `is.numeric()`, `is.integer()`, and `is.double()` return for the vectors that hold floating point number versus integers?
8.)What does `letters()` do?

The concatenate or combine `c()` function will explicitly **construct** a vector. We used it above to construct the logical vector. All the other vectors came about through other means, as in everyday coding, most vectors aren't made explicitly with `c()`. They tend to be created with some generator, like the `1:n` shortcut, or via transformation of an existing object. The function `c()` can also be used to **add elements** to a vector.

```
v_name <- c("Sarah", "Tracy", "Jon")
v_name <- c(v_name, "Annette")
v_name
```

```
## [1] "Sarah"   "Tracy"   "Jon"     "Annette"
```

```
v_name<- c("Greg", v_name)
v_name
```

```
## [1] "Greg"    "Sarah"   "Tracy"   "Jon"     "Annette"
```

To **index a vector** means to address specific atoms of that vector. To index a vector square brackets are used, like so: `x[something]`. There are several ways to express which elements you want, i.e. there are several valid forms for `something`.

5.1.4.0.1 Atomic vector indexing examples:{.unlisted .unnumbered}

Integer vector, all positive: the elements specified in something are kept

```
v_char[2]
```

```
## [1] "b"
```

We get the second element doing that. You can also use index ranges:

```
v_doub[2:3]
```

```
## [1] 2.4 3.6
```

Logical vector: keep elements of x for which something is TRUE and drop those for which it is FALSE

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

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

We can see that only elements remain, that were indexed with TRUE. We can also see that we can index a vector with a vector. That's also the case below:

```
v_char[v_log]
```

```
## [1] "a" "d"
```

Negative integers, all negative: the elements specified in something are dropped

```
v_char[-4]
```

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

Task 9, 10 & 11

> 9.) What happens when you request the zero-th element of one of our vectors?
> 10.) What happens when you ask for an element that is past the end of the vector, i.e. request x[k] when the length of x is less than k?
> 11.)We indexed x with a logical vector of the same length. What happens if the indexing vector is shorter than x?

Complete the Tasks and you'll see it's possible to get an atomic vector of length zero and also to get elements that are NA. In R, missing data is represented as **NA (Not Available)**. NA is a special value and a reserved variable, so you cannot use it in naming objects.

Another neat thing is that if you request an element that is not there the underlying variable type is retained.

Index of 0:

```
v_int[0]
```

```
## integer(0)
```

```
typeof(v_int[0])
```

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

Outside bounds of the vector:

```
v_doub[100]
```

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

```
typeof(v_doub[100])
```

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

So, why does a 0-indexed vector not return NA? The answer is in the manual:

> NA and zero values are allowed: rows of an index matrix containing a zero are ignored, whereas rows containing an NA produce an NA in the result.

So an index of 0 just gets ignored. We can see this in the following

```
x <- 1:10
x[c(1, 3, 0, 5, 0)]
```

```
## [1] 1 3 5
```

So if the only index we give it is 0 then the appropriate response is to return an empty vector.

Data type coercion
Even though R's vectors have a specific data type, it's quite easy to convert them to another type. This is called **coercion**. There's a hierarchy of types: the more primitive ones cheerfully and silently (**implicitly**) convert to those higher up in the food chain. Here's the order:

> **R coercion rules:** logical → integer → numeric → complex → character

where → can be read as "are transformed into". Conversely, character is higher in the hierarchy than complex and that is higher in the hierarchy than numeric *etc.*.

For **explicit coercion**, use the `as.*()` functions.

```
v_log
```

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

```
as.integer(v_log)
```

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

```
v_int
```

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

```
as.numeric(v_int)
```

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

```
v_doub
```

```
## [1] 1.2 2.4 3.6 4.8
```

```
as.character(v_doub)
```

```
## [1] "1.2" "2.4" "3.6" "4.8"
```

```
as.character(as.numeric(as.integer(v_log)))
```

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

But coercion can also be triggered by other actions, such as assigning a scalar of the wrong type into an existing vector.

```
v_doub_copy <- v_doub
str(v_doub_copy)
```

```
##  num [1:4] 1.2 2.4 3.6 4.8
```

```
v_doub_copy[3] <- "uhoh"
str(v_doub_copy)
```

```
##  chr [1:4] "1.2" "2.4" "uhoh" "4.8"
```

Our numeric vector was silently coerced to character. This can be a wonderful source of bugs, so when debugging, always give serious thought to this question: Is this object of the type I think it is? How sure am I about that?

We end the discussion of atomic vectors with two specific examples of "being intentional about type".

- Use of type-specific NAs when doing setup.
- Use of L to explicitly request integer.

```
(big_plans <- rep(NA_integer_, 4))
```

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

```
str(big_plans)
```

```
##  int [1:4] NA NA NA NA
```

```
big_plans[3] <- 5L
## note that big_plans is still integer!
str(big_plans)
```

```
##  int [1:4] NA NA 5 NA
```

```
## note that omitting L results in coercion of big_plans to double
big_plans[1] <- 10
str(big_plans)
```

```
##  num [1:4] 10 NA 5 NA
```

Task 12

> 12.) Recall the hiearchy of the most common atomic vector types: logical < integer < numeric < character. Try to use the `as.*()` functions to go the wrong way. Call `as.logical()`, `as.integer()`, and `as.numeric()` on a character vector, such as letters. What happens?

Lists

Atomic vectors are very constrained: atoms are of a scalar/length 1 and need to be one data type. You might find yourself needing a vector that violates these constraints and for which the following is true:

- Individual atoms might have length greater than 1.
- Individual atoms might not be of the same flavor.

This is a when you need a **list**.

A list is actually still a vector in R, but it's not an atomic vector. Lists are sometimes called **generic vectors**, because the elements of a list can be of any type of R object, even lists containing further lists. They are ultimately nice R containers for data.

We **construct** a list explicitly with `list()` but, like atomic vectors, most lists are created some other way in real life.

```
(x <- list(1:3, c("four", "five")))
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] "four" "five"
```

```
(y <- list(logical = TRUE, integer = 4L, double = 4 * 1.2, character = "character"))
```

```
## $logical
## [1] TRUE
##
## $integer
## [1] 4
##
```

```
## $double
## [1] 4.8
##
## $character
## [1] "character"
```

```
(z <- list(letters[26:22], transcendental = c(pi, exp(1)), f = function(x) x^2))
```

```
## [[1]]
## [1] "z" "y" "x" "w" "v"
##
## $transcendental
## [1] 3.141593 2.718282
##
## $f
## function(x) x^2
```

We have explicit proof above that list components can

- Be heterogeneous, i.e. can be of different "flavors".
- Have different lengths.
- Have names. Or not. Or some of both.

You can also **coerce** other objects using `as.list()`.

To create an empty list of the required length in R, use the `vector()` function. `list()` does not allow you to create an empty list of a *specific length* (it just creates an empty list object). The `vector()` function takes two arguments: mode and length. In our case, the mode is a list, length is the number of elements in the list, and actually the list ends up being empty, filled with NULL.

```
a_list <- vector(mode="list", length = 5) # empty list
length(a_list)
```

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

```
a_list
```

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

The content of elements of a list can be retrieved by using double square brackets (more on that below).

```
a_list[[1]]
```

```
## NULL
```

Vectors can be coerced to lists as follows:

```
other_list <- 1:10
other_list<- as.list(other_list)
length(other_list)
```

```
## [1] 10
```

Task 13 & 14

13.) What is the class of other_list[1]?
14.) What is the class of other_list[[1]]?

List indexing
List indexing is a bit more complicated than that of atomic vectors. There are **3 ways to index a list** and the differences are very important:

1.) **With single square brackets**, i.e. just like we indexed atomic vectors. Note this always returns a list, even if we request a single component.

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

```
## [[1]]
## [1] "four" "five"
```

```
y[2:3]
```

```
## $integer
## [1] 4
##
## $double
## [1] 4.8
```

```
z["transcendental"]
```

```
## $transcendental
## [1] 3.141593 2.718282
```

2.) **With double square brackets**. This can only be used to access a single component and it returns the "naked" component. You can request a component with a positive integer or by name.

```
x[[2]]
```

```
## [1] "four" "five"
```

```
y[["double"]]
```

```
## [1] 4.8
```

3.) **_With the $ addressing named components**, which we will learn a lot more about when we look at data frames. Like `[[`, this can only be used to access a single component, but it is even more limited: You must specify the component by name.

```
z$transcendental
```

```
## [1] 3.141593 2.718282
```

A very good and easy-to-grasp example of the difference between the list-preserving indexing provided by `[` and the behaviour of `[[` is given here: the pepper shaker analogy on R for Data Science. Click the hyperlink to see more!

Task 15

15.) Consider `my_vec <- c(a = 1, b = 2, c = 3)`
`my_list <- list(a = 1, b = 2, c = 3)`
Use [and [[to attempt to retrieve elements 2 and 3 from my_vec and my_list. What succeeds vs. fails? What if you try to retrieve element 2 alone? Does [[even work on atomic vectors? Compare and contrast the results from the various combinations of indexing method and input object.

Overview data frames & matrices
A **data frame** & **matrices** are very important data types in R. They are pretty much the de facto data structures for most tabular data and what we use for statistics.

While the data frame is a special type of list where every element of the list has same length (i.e. data frame is a "rectangular" list), the matrix is a special type of atomic vector.

The table below illustrates how they relate to each other:

Table 1

Dimensions	Homogeneous	Heterogeneous
1-D	atomic vector	list
2-D	matrix	data frame

Matrices
As Table 1 already hints, **matrices are atomic vectors with dimensions; the number of rows and columns**. As with atomic vectors, the elements of a matrix must be of the same data type.

To **create an empty matrix**, we need to define those dimensions:

```
m<-matrix(nrow=2, ncol=2)
m
```

```
##      [,1] [,2]
## [1,]   NA   NA
## [2,]   NA   NA
```

We can **find out how many dimensions a matrix has** by using `dim()`

```
dim(m)
```

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

You can check that matrices are vectors with a class attribute of matrix by using `class()` and `typeof()`.

```
m <- matrix(c(1:3))
class(m)
```

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

```
typeof(m)
```

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

While `class()` shows that `m` is a matrix, `typeof()` shows that in this case ***fundamentally the matrix is an integer vector*** (we will see later that they are not always **integer** vectors, but can be **character** vectors, too.

When creating a **matrix**, it is important to remember that matrices ***are filled column-wise***

```
m<-matrix(1:6, nrow=2, ncol=3)
m
```

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

If that is not what you want, you can use the `byrow` argument (a logical: can be `TRUE` or `FALSE`) to specify how the matrix is filled

```
m<-matrix(1:6, nrow=2, ncol=3, byrow=TRUE)
m
```

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

You can **create a matrix from a vector**:

```
m<-sample(1:100, size=10)
m
```

```
##  [1]  85 100  57  89  98  95  90  56   5   2
```

```
dim(m)<-c(2,5)
m
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   85   57   98   90    5
## [2,]  100   89   95   56    2
```

A lot is going on here. Let's dissect it:

- We generate a random integer vector using `sample()`. `sample()` in this case randomly draws 10 (`size=10`) numbers from 1 to 100 (`1:100`)[2].
- we assign the vector dimensions using `dim()` and `c(2,5)`, with the later being `c(rows, columns)`.

All of the above takes the random integer vector and transforms it into a matrix with 2 rows and 5 columns.

You can also **bind columns and rows** using `cbind()` and `rbind`:

```
x <- 1:3
y <- 10:12
m<-cbind(x, y)
m
```

```
##      x  y
## [1,] 1 10
## [2,] 2 11
## [3,] 3 12
```

```
n<-rbind(x,y)
n
```

```
##   [,1] [,2] [,3]
## x    1    2    3
## y   10   11   12
```

If we want to retrieve an element we can do so by using:

[2] if you want to get the same vector each time with the same parameters, you need to use set.seed() with a defined number first

Matrix indexing
Akin to vectors, we revisit our square-brackets and can retrieve elements of a matrix by specifying the index along each dimension (e.g. "row" and "column") in single square brackets.

```
m[3,2]
```

```
##  y
## 12
```

Note that it is `[row,column]`.

Task 16

> 16.) Transform the built-in dataset iris into a matrix using `as.matrix()` and assign it to a new variable of your choice.
> When you use `class()` and `typeof()`, what results do you get and why? What happened to the doubles in the data frame (hint: remember the coercion rules from earlier)?

Data frames
Usually created by `read.csv()` and `read.table()`, i.e. when importing the data into R. When we imported the dataset "iris" earlier (`data(iris)`), we loaded the dataframe "iris". Have a look by typing `iris` into your console. Assuming all columns in a data frame are of same type, data frame can be converted to a matrix with `data.matrix()` (preferred) or `as.matrix()`. Otherwise type coercion will be enforced and the results may not always be what you expect.

Instead of loading data, you can also **create a new data frame** with `data.frame()` function. Find the number of rows and columns with `nrow(dat)` and `ncol(dat)`, respectively. We will look at that in detail in a moment Rownames are often automatically generated and look like 1, 2, . . . , n.

```
dat <- data.frame(id = letters[1:10], x = 1:10, y = 11:20)
dat
```

First, let's create a data frames by hand:

```
##    id  x  y
## 1   a  1 11
## 2   b  2 12
## 3   c  3 13
## 4   d  4 14
## 5   e  5 15
## 6   f  6 16
## 7   g  7 17
## 8   h  8 18
## 9   i  9 19
## 10  j 10 20
```

You can also convert matrices into data frames using `as.data.frame()`:

```
class(m)
```

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

```
df_m=as.data.frame(m)
class(df_m)
```

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

To explore data frames, there are several interesting functions:

Explorative data frame functions:
* `head()` - shows first 6 rows * `tail()` - shows last 6 rows * `dim()` - returns the dimensions of data frame (i.e. number of rows and number of columns) * `nrow()` - number of rows * `ncol()`- number of columns * `str()` - structure of data frame - name, type and preview of data in each column * `names()` or `colnames()` - both show the names attribute for a data frame * `sapply(dataframe, class)` - shows the class of each column in the data frame

Task 17 & 18

17.) Try out all the functions above on `dat`.
18.) Try `summary(dat)` and `summary(iris)`. What does it do?

Remember Table 1? Below we show that a **data frame is actually a special list**:

```
is.list(dat)
```

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

```
class(dat)
```

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

Lists can contain elements that are themselves multi-dimensional (e.g. a lists can contain data frames or another type of objects). Lists can also contain elements of any length, therefore lists do not necessarily have to be "rectangular". ***However, in order for the list to qualify as a data frame, the length of each element has to be the same.***

Data frame indexing/slicing
As indirectly shown above, there are ways to retrieve specific elements from the data frame, the data frame can be ***sliced*** or ***indexed*** . Because data frames are rectangular, elements of data frame can be referenced by specifying the row and the column index in single square brackets (similar to matrix).

```
dat[1, 3]
```

```
## [1] 11
```

As data frames are also lists, it is possible to refer to columns (which are elements of such list) using the list notation, i.e. either double square brackets or a `$`.

```
dat[["y"]]
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

```
dat$y
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

Restructure your data frame - *pimp up your data frame*
When you look at both the `dat` and `iris` data frames from earlier, they have no rownames:

```
dat
```

```
##    id  x  y
## 1   a  1 11
## 2   b  2 12
```

```
## 3   c  3 13
## 4   d  4 14
## 5   e  5 15
## 6   f  6 16
## 7   g  7 17
## 8   h  8 18
## 9   i  9 19
## 10  j 10 20
```

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

However, when we look at the `mtcars` data set, it does:

```
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

mtcars has rownames! Let's assume we want to make `dat` a bit more fancy and we want to **give each row a name**. What could we do? Albeit a bit non-sensical (but the entire data frame is just a generic example), we could name the rows as follows:

```
names<-c("first","second","third","fourth","fifth","sixth", "seventh","eighth","ninth","tenth")
rownames(dat)=names
dat
```

```
##         id  x  y
## first    a  1 11
## second   b  2 12
## third    c  3 13
## fourth   d  4 14
## fifth    e  5 15
## sixth    f  6 16
## seventh  g  7 17
## eighth   h  8 18
## ninth    i  9 19
## tenth    j 10 20
```

Note that `names` is a character vector that is used to assign the rownames in the data frame `dat`.

We can also **rename columns**. Let's assume we want to change the abbreviation of the first three columns of mtcars to the actual words:

```
colnames(mtcars)[1:3]<-c("miles per gallon","cylinders","displacement")
head(mtcars)
```

```
##                   miles per gallon cylinders displacement  hp drat    wt  qsec
## Mazda RX4                     21.0         6          160 110 3.90 2.620 16.46
## Mazda RX4 Wag                 21.0         6          160 110 3.90 2.875 17.02
## Datsun 710                    22.8         4          108  93 3.85 2.320 18.61
## Hornet 4 Drive                21.4         6          258 110 3.08 3.215 19.44
## Hornet Sportabout             18.7         8          360 175 3.15 3.440 17.02
## Valiant                       18.1         6          225 105 2.76 3.460 20.22
##                   vs am gear carb
## Mazda RX4          0  1    4    4
## Mazda RX4 Wag      0  1    4    4
## Datsun 710         1  1    4    1
## Hornet 4 Drive     1  0    3    1
## Hornet Sportabout  0  0    3    2
## Valiant            1  0    3    1
```

By using `[1:3]` we only changed a subset of the column names. If you want to change them all, the vector with the column names must correspond to the number of columns (be of identical length/ have the same 1D dimension).

Task 19

19.) Try to selectively rename 2 rows of your choice in mtcars.

You can also **append a column of choice** to your data frame. Remember, it needs to have the same length as the other columns:

```
#find out how many rows our mtcars actually has:
nrow(mtcars)
```

```
## [1] 32
```

```
#generate new column
favorites=1:32
#append
mtcars$favorites=favorites
mtcars
```

```
##                     miles per gallon cylinders displacement  hp drat    wt
## Mazda RX4                       21.0         6        160.0 110 3.90 2.620
## Mazda RX4 Wag                   21.0         6        160.0 110 3.90 2.875
## Datsun 710                      22.8         4        108.0  93 3.85 2.320
## Hornet 4 Drive                  21.4         6        258.0 110 3.08 3.215
## Hornet Sportabout               18.7         8        360.0 175 3.15 3.440
## Valiant                         18.1         6        225.0 105 2.76 3.460
## Duster 360                      14.3         8        360.0 245 3.21 3.570
## Merc 240D                       24.4         4        146.7  62 3.69 3.190
## Merc 230                        22.8         4        140.8  95 3.92 3.150
## Merc 280                        19.2         6        167.6 123 3.92 3.440
## Merc 280C                       17.8         6        167.6 123 3.92 3.440
## Merc 450SE                      16.4         8        275.8 180 3.07 4.070
## Merc 450SL                      17.3         8        275.8 180 3.07 3.730
## Merc 450SLC                     15.2         8        275.8 180 3.07 3.780
## Cadillac Fleetwood              10.4         8        472.0 205 2.93 5.250
## Lincoln Continental             10.4         8        460.0 215 3.00 5.424
## Chrysler Imperial               14.7         8        440.0 230 3.23 5.345
## Fiat 128                        32.4         4         78.7  66 4.08 2.200
## Honda Civic                     30.4         4         75.7  52 4.93 1.615
```

```
## Toyota Corolla                 33.9         4         71.1  65 4.22 1.835
## Toyota Corona                  21.5         4        120.1  97 3.70 2.465
## Dodge Challenger               15.5         8        318.0 150 2.76 3.520
## AMC Javelin                    15.2         8        304.0 150 3.15 3.435
## Camaro Z28                     13.3         8        350.0 245 3.73 3.840
## Pontiac Firebird               19.2         8        400.0 175 3.08 3.845
## Fiat X1-9                      27.3         4         79.0  66 4.08 1.935
## Porsche 914-2                  26.0         4        120.3  91 4.43 2.140
## Lotus Europa                   30.4         4         95.1 113 3.77 1.513
## Ford Pantera L                 15.8         8        351.0 264 4.22 3.170
## Ferrari Dino                   19.7         6        145.0 175 3.62 2.770
## Maserati Bora                  15.0         8        301.0 335 3.54 3.570
## Volvo 142E                     21.4         4        121.0 109 4.11 2.780
##                      qsec vs am gear carb favorites
## Mazda RX4           16.46  0  1    4    4         1
## Mazda RX4 Wag       17.02  0  1    4    4         2
## Datsun 710          18.61  1  1    4    1         3
## Hornet 4 Drive      19.44  1  0    3    1         4
## Hornet Sportabout   17.02  0  0    3    2         5
## Valiant             20.22  1  0    3    1         6
## Duster 360          15.84  0  0    3    4         7
## Merc 240D           20.00  1  0    4    2         8
## Merc 230            22.90  1  0    4    2         9
## Merc 280            18.30  1  0    4    4        10
## Merc 280C           18.90  1  0    4    4        11
## Merc 450SE          17.40  0  0    3    3        12
## Merc 450SL          17.60  0  0    3    3        13
## Merc 450SLC         18.00  0  0    3    3        14
## Cadillac Fleetwood  17.98  0  0    3    4        15
## Lincoln Continental 17.82  0  0    3    4        16
## Chrysler Imperial   17.42  0  0    3    4        17
## Fiat 128            19.47  1  1    4    1        18
## Honda Civic         18.52  1  1    4    2        19
## Toyota Corolla      19.90  1  1    4    1        20
## Toyota Corona       20.01  1  0    3    1        21
## Dodge Challenger    16.87  0  0    3    2        22
## AMC Javelin         17.30  0  0    3    2        23
## Camaro Z28          15.41  0  0    3    4        24
## Pontiac Firebird    17.05  0  0    3    2        25
## Fiat X1-9           18.90  1  1    4    1        26
## Porsche 914-2       16.70  0  1    5    2        27
## Lotus Europa        16.90  1  1    5    2        28
## Ford Pantera L      14.50  0  1    5    4        29
## Ferrari Dino        15.50  0  1    5    6        30
## Maserati Bora       14.60  0  1    5    8        31
## Volvo 142E          18.60  1  1    4    2        32
```

Task 20

20.) Try out what happens if you try to add a new column of a length that is less than 32.

We can also **subset** (or filter based on a conditional statement) a data frame using `subset`. The function takes two arguments `subset(x, condition)`. X is the data frame to perform subset on, condition is the conditional statement to subset with:

```
#find out how many rows our mtcars actually has:
subset(mtcars, cylinders>4)
```

```
##                     miles per gallon cylinders displacement  hp drat    wt
## Mazda RX4                        21.0         6        160.0 110 3.90 2.620
## Mazda RX4 Wag                    21.0         6        160.0 110 3.90 2.875
## Hornet 4 Drive                   21.4         6        258.0 110 3.08 3.215
## Hornet Sportabout                18.7         8        360.0 175 3.15 3.440
## Valiant                          18.1         6        225.0 105 2.76 3.460
## Duster 360                       14.3         8        360.0 245 3.21 3.570
## Merc 280                         19.2         6        167.6 123 3.92 3.440
## Merc 280C                        17.8         6        167.6 123 3.92 3.440
## Merc 450SE                       16.4         8        275.8 180 3.07 4.070
## Merc 450SL                       17.3         8        275.8 180 3.07 3.730
## Merc 450SLC                      15.2         8        275.8 180 3.07 3.780
## Cadillac Fleetwood               10.4         8        472.0 205 2.93 5.250
## Lincoln Continental              10.4         8        460.0 215 3.00 5.424
## Chrysler Imperial                14.7         8        440.0 230 3.23 5.345
## Dodge Challenger                 15.5         8        318.0 150 2.76 3.520
## AMC Javelin                      15.2         8        304.0 150 3.15 3.435
## Camaro Z28                       13.3         8        350.0 245 3.73 3.840
## Pontiac Firebird                 19.2         8        400.0 175 3.08 3.845
## Ford Pantera L                   15.8         8        351.0 264 4.22 3.170
## Ferrari Dino                     19.7         6        145.0 175 3.62 2.770
## Maserati Bora                    15.0         8        301.0 335 3.54 3.570
##                      qsec vs am gear carb favorites
## Mazda RX4           16.46  0  1    4    4         1
## Mazda RX4 Wag       17.02  0  1    4    4         2
## Hornet 4 Drive      19.44  1  0    3    1         4
## Hornet Sportabout   17.02  0  0    3    2         5
## Valiant             20.22  1  0    3    1         6
## Duster 360          15.84  0  0    3    4         7
## Merc 280            18.30  1  0    4    4        10
## Merc 280C           18.90  1  0    4    4        11
## Merc 450SE          17.40  0  0    3    3        12
## Merc 450SL          17.60  0  0    3    3        13
## Merc 450SLC         18.00  0  0    3    3        14
## Cadillac Fleetwood  17.98  0  0    3    4        15
## Lincoln Continental 17.82  0  0    3    4        16
## Chrysler Imperial   17.42  0  0    3    4        17
## Dodge Challenger    16.87  0  0    3    2        22
## AMC Javelin         17.30  0  0    3    2        23
## Camaro Z28          15.41  0  0    3    4        24
## Pontiac Firebird    17.05  0  0    3    2        25
## Ford Pantera L      14.50  0  1    5    4        29
## Ferrari Dino        15.50  0  1    5    6        30
## Maserati Bora       14.60  0  1    5    8        31
```

Task 21

21.) Extract (using either `[]` or `$`) the columns Sepal.Length and Sepal.Width from the `iris` dataset and make a new data frame out of them using `data.frame()`. Subset the new data frame for Sepal.Length > 4.6.

5.1.5 Factors

Understanding factors
Factors are so-called ***derived data types***. They are normally **used to group variables into a fixed number of unique categories or levels**. For example, a data set may be grouped by gender or month of the year. Such data are usually loaded into R as a numeric or character data type requiring that they be converted to a factor using the `as.factor()` function.

In the following chunk of code, we create a factor from a character object.

```
a      <- c("March","February","February","November","February","March","March","March","February","Nov
fact <- as.factor(a)
```

Note that `a` is of **character data type and `fact` is the factor representation of `a`**.

```
typeof(a)
```

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

However, the derived object `fact` is now stored as an integer!

```
typeof(fact)
```

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

Yet, when displaying the contents of `fact` we see character values.

```
fact
```

```
##  [1] March    February February November February March    March    March
##  [9] February November
## Levels: February March November
```

How can this be? Well, `fact` is a ***more complicated object*** than the simple objects created thus far in that the factor is storing additional information not seen in its output. This hidden information is stored in attributes. To view these hidden attributes, use the `attributes()` function.

```
attributes(fact)
```

```
## $levels
## [1] "February" "March"    "November"
##
## $class
## [1] "factor"
```

There are two attributes of the factor object `fact` : levels and class. The **levels attribute** lists the three unique values in `fact`. The order in which these levels are listed reflect their ***numeric*** **representation**. So in essence, `fact` is **storing each value as an integer that points to one of the three unique levels**.

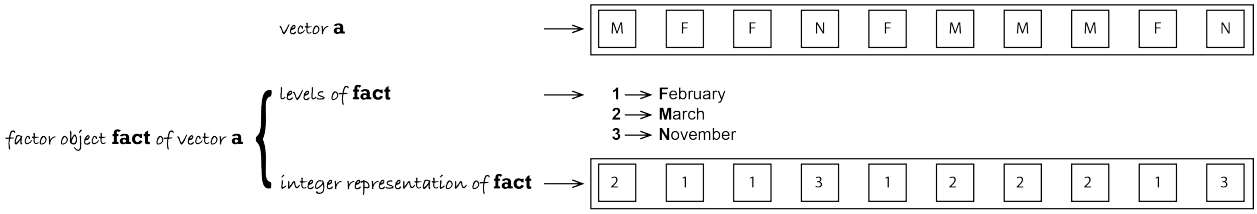


Figure 2: A graphical representation of factors and levels. The vector a consists of the following characters, represented in square boxes: M, F, F, N, F, M, M, M, F, N. These characters have levels, the levels are stored in the factor fact. Level 1 is February (F), level 2 is March (M) and level 2 is November (N). Those are represented in fact as a vector of integers, which is shown below that: 2, 1, 1, 3, 1, 2, 2, 2, 1, 3

So why doesn't R output the integer values when we output `fact`? To understand why, we first need to know that when we call the object name, R is wrapping that object name with the print command, so the following lines of code are identical.

```
fact
```

```
##  [1] March    February February November February March    March    March
##  [9] February November
## Levels: February March November
```

```
print(fact)
```

```
##  [1] March    February February November February March    March    March
##  [9] February November
## Levels: February March November
```

The `print` function then looks for a class attribute in the object. The class type instructs the print function on how to generate the output. Since `fact` has a factor class attribute (`fact` is the factor object of `a`), the print function is instructed to replace the integer values with the level "tags".

Naturally, this all happens behind the scenes without user intervention.

Another way to determine `fact`'s class type is to call the class function.

```
class(fact)
```

```
## [1] "factor"
```

The unique levels of a factor, and the order in which they are stored can be extracted using the levels function.

```
levels(fact)
```

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

Remember, the order in which the levels are displayed match their integer representation.

Note that if `a` class attribute is not present (if it is not the `fact` factor of `a`), the class function will return the object's data type.

```
class(a)
```

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

In such a case, the `a` object is treated as a generic element, it is not its factor representation with levels: `fact`.

To ***appreciate the benefits of a factor we'll first create a data frame***. One column will be assigned the `fact` factor and another will be assigned some random numeric values.

```
x      <- c(166, 47, 61, 148, 62, 123, 232, 98, 93, 110)
dat_fact    <- data.frame(min_sunshine = x, month = fact)
dat_fact
```

```
##    min_sunshine    month
## 1           166    March
## 2            47 February
## 3            61 February
## 4           148 November
## 5            62 February
## 6           123    March
## 7           232    March
## 8            98    March
```

```
## 9             93 February
## 10           110 November
```

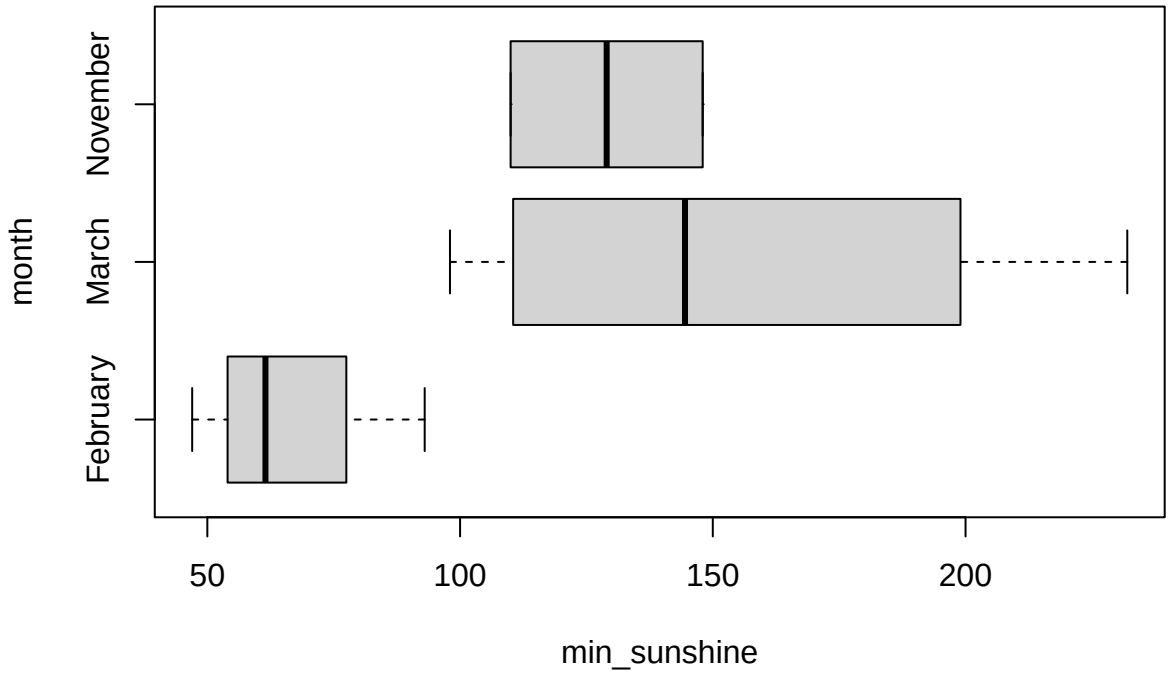
The month column is now a factor with three levels: F, M and N. We can use the '`str()` function to view the dataframe's structure as well as its columns classes.

```
str(dat_fact)
```

```
## 'data.frame':    10 obs. of  2 variables:
##  $ min_sunshine: num  166 47 61 148 62 123 232 98 93 110
##  $ month       : Factor w/ 3 levels "February","March",..: 2 1 1 3 1 2 2 2 1 3
```

Many **functions** other than print will ***recognize factor data types and will allow you to split the output into groups defined by the factor's unique levels***. For example, to create three box plots of the value min_sunshine (we will learn more about box plots later), one for each month group F, M and U, type the following:

```
boxplot(min_sunshine ~ month, dat_fact, horizontal = TRUE)
```



The tilde ~ operator is used in the plot function to split (or condition) the data into separate plots based on the factor month.

Rearranging level order

A factor will define a hierarchy for its levels. When we invoked the levels function in the last example, you may have noted that the levels output were ordered F, M and N–this is the **level hierarchy** defined for months (i.e. F>M>N). This means that regardless of the order in which the factors appear in a table, ***anytime a plot or operation is conditioned by the factor, the grouped elements will appear in the order defined by the levels' hierarchy***. When we created the box plot from our `dat_fact` object, the plotting function ordered the box plot (bottom to top) following months's level hierarchy (i.e. F first, then M, then N).

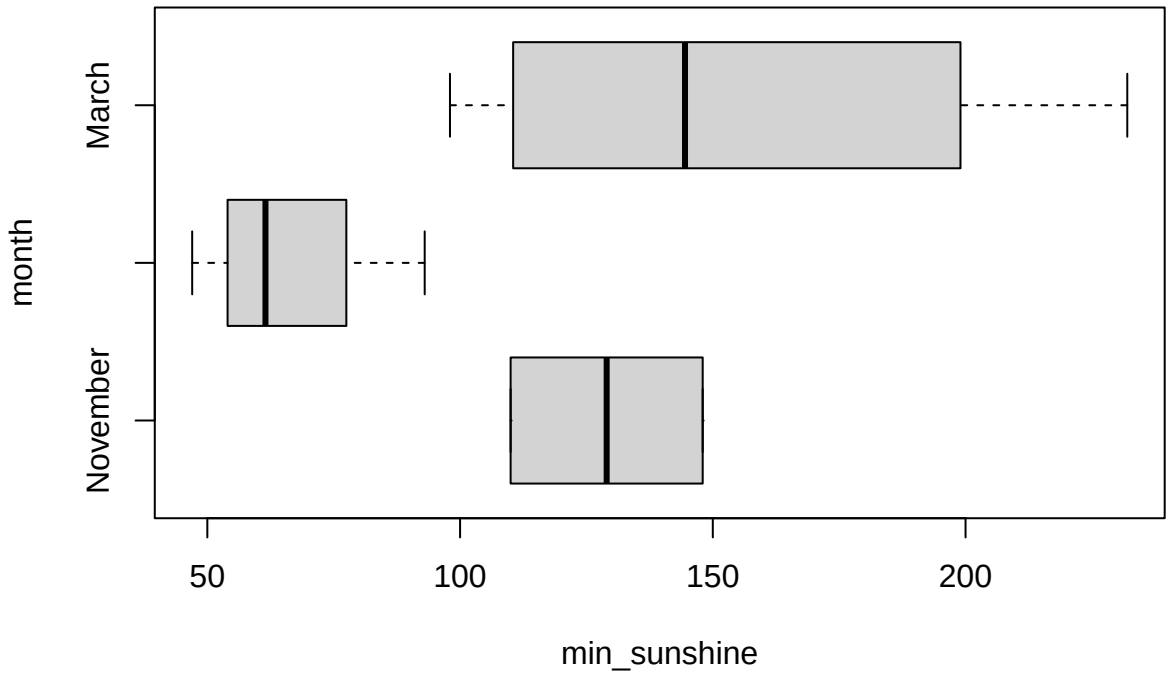
If we wanted the box plots to be plotted in a **different order** (i.e. N first followed by F then M) we would need to **modify the month** column by ***releveling*** the factor object as follows:

```
dat_fact$month <- factor(dat_fact$month, levels=c("November","February","March"))
str(dat_fact)
```

```
## 'data.frame':    10 obs. of  2 variables:
##  $ min_sunshine: num  166 47 61 148 62 123 232 98 93 110
##  $ month       : Factor w/ 3 levels "November","February",..: 3 2 2 1 2 3 3 3 2 1
```

The factor function is giving the original factor values (dat_fact$months) but is also giving the levels in the new order in which they are to appear(`levels=c("November","February","March")`). Now, if we recreate the box plot, the plot order (plotted from bottom to top) will reflect the new level hierarchy.

```
boxplot(min_sunshine ~ month, dat_fact, horizontal = TRUE)
```



5.1.6 Task Challenge

5.1.6.0.1 Task: 22.) Load the dataset esoph. This data frame contains the data from a case-control study of (o)esophageal cancer in Ille-et-Vilaine, France.

- remove the last column of the data frame.
- rename the columns from agegp to Age_Group, from alcgp to Alcohol_consump, from tobgp to Tobacco_consump and leave the column name of ncases the same.
- subset the dataframe to only contain rows that have an Alcohol_consump of 120+.
- convert the agegp into a factor and assign it to a new variable. Assess the attributes of that variable.
- What data type is Alcohol_consump?

5.2 Summary

Data types
Data types are:

- **character**: `Superkalifragilistikexpialegetisch, abc, @%£$`
- **numeric**:
  - **integers:** `1,-3,22,0,1890, 2L` (The L explicitly tells R that this is an integer)
  - **doubles:** `15.5, 0.01, -0.0004`
  - **complex:** `1+4i`
- **logical**: `TRUE FALSE`

Data type coercion
Data types can be **coerced** into different data types following a hierarchy of types: the more primitive ones **implicitly** convert to those higher up in the hierarchy

> **R coercion rules:** logical → integer → numeric → complex → character

where → can be read as "are transformed into". Conversely, character is higher in the hierarchy than complex and that is higher in the hierarchy than numeric *etc.*. For **explicit coercion**, the `as.*()` functions are used.

Data type exploration
To **explore data types** several R functions can be used:

- `typeof()` - what is the object's data type?
- `mode()` - what is the object's data type?
- `length()`- how long is it?
- `attributes()` - does it have any metadata?
- `str` - display the internal structure of an object.
- `is.numeric()`, `is.character()`, `is.complex()`, `is.logical()` - returns TRUE when an object is the datatype queried, FALSE if not

Data structures
R has several **data structures**:

- vectors:
  - atomic vector
  - list
- matrix
- data frame
- factors

While the data frame is a special type of list where every element of the list has same length (i.e. data frame is a "rectangular" list), the matrix is a special type of atomic vector (also rectangular).

Construct data structures
Construct an atomic vector: {-}
Use `c()` for constructing a vector.

```
v_name <- c("Sarah", "Tracy", "Jon")
v_name <- c(v_name, "Annette")
```

Construct a list:
You can construct a list using `list()`:

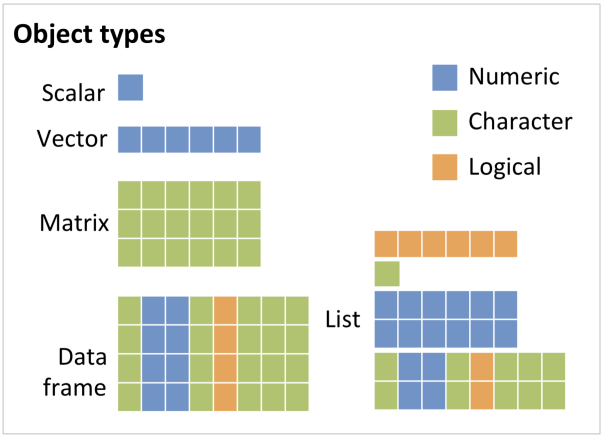


Figure 3: R Object types. Numeric, Character and Logical are shown in different colors. They can be scalar, a vector, a matrix a list or a dataframe.

```
x <- list(1:3, c("four", "five"))

y <- list(logical = TRUE, integer = 4L, double = 4 * 1.2, character = "character")
```

You can construct an empty list of a certain length using `vector()`:

```
a_list <- vector(mode="list", length = 5) # empty list
length(a_list)
```

Construct a matrix:
Use `matrix()`. Remember, matrices are filled column-wise, if you want to change that, set `byrow=TRUE` (by default it is set to `FALSE`.

```
m<-matrix(1:6, nrow=2, ncol=3)
```

You can also construct a data frame by assigning a vector dimensions using `dim()` or by binding rows or columns of atomic vectors of the same length using `rbind()` or `cbind()`.

Construct a data frame:
Use `data.frame()` to create a data frame:

```
df <- data.frame(Name = c("Jon", "Bill", "Maria", "Ben", "Tina"),
                 Age = c(23, 41, 32, 58, 26)
                 )
```

You can also use built-in data frames that ship with R (`data()`).

Index data structures:
Indexing means addressing specific elements of a data structure.

Indexing an atomic vector:
To index a vector square brackets are used, like so: `x[something]`.

Integer vector, all positive: the elements specified in something are kept

```
v_char[2]
```

We get the second element doing that. You can also use index ranges:

```
v_doub[2:3]
```

Negative integers, all negative: the elements specified in something are dropped

```
v_char[-4]
```

Indexing a list:
There are three ways to index a list:

1.) **With single square brackets**, i.e. just like we indexed atomic vectors. Note this always returns a list, even if we request a single component (one specific element of the list).

```
x[c(FALSE, TRUE)]
y[2:3]
z["element"]
```

2.) **With double square brackets.** This can only be used to access a single component and it returns the "naked" component. You can request a component with a positive integer or by name.

```
x[[2]]

y[["double"]]
```

3.) **With the `$` addressing named components.** Like `[[`, this can only be used to access a single component, but it is even more limited: You must specify the component by name.

```
z$element
```

A very good and easy-to-grasp example of the difference between the list-preserving indexing provided by `[` and the behaviour of `[[` is given here: the pepper shaker analogy on R for Data Science. Click the hyperlink to see more!

Indexing a matrix:
Remember that a matrix is a 2D atomic vector. Therefore, you can index a matrix with square brackets like so: `[row,column]`.

```
m[3,2]
```

Indexing a data frame:
As data frames are also lists and are rectangular, it is possible to refer to columns (which are elements of such list) using the list notation, i.e. either double square brackets `[[]]` or a `$`. Due to them being rectangular, you can also address individual components by using `[row, column]`.

```
dat[1, 3]
dat[["y"]]
dat$y
```

Exploring data structures:
There are many exploration functions to assess data structures. Some are:

- `head()` - shows first 6 rows
- `tail()` - shows last 6 rows
- `dim()` - returns the dimensions of data frame (i.e. number of rows and number of columns)
- `nrow()` - number of rows
- `ncol()`- number of columns
- `str()` - structure of data frame - name, type and preview of data in each column

- `names()` or `colnames()` - both show the names attribute for a data frame
- `sapply(dataframe, class)` - shows the class of each column in the data frame

Restructuring data frames:
Use the following functions to change the data frame:

- `colnames()` - assign new column names
- `rownames()` - assign new row names
- `subset()` - filter data frame based on a conditional statement

Add columns of identical length like so:

```
#find out how many rows our mtcars actually has:
nrow(mtcars)
#generate new column
favorites=1:32
#append
mtcars$favorites=favorites
mtcars
```

Remove columns by negative indexing.

Factors
Factors are so-called ***derived data types***. They are normally **used to group variables into a fixed number of unique categories or levels**. Data can be converted to a factor using the `as.factor()` function.

In the example below **there are two attributes of the factor object `fact` derived of the vector `a` : levels and class.** The **levels attribute** lists the three unique values in `fact`. The order in which these levels are listed reflect their ***numeric* representation**.

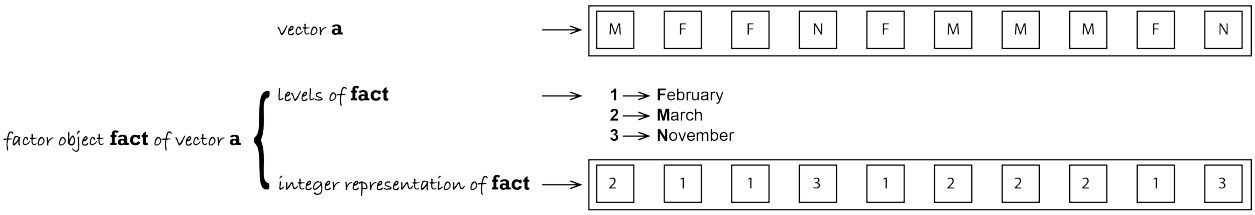


Figure 4: A graphical representation of factors and levels. The vector a consists of the following characters, represented in square boxes: M, F, F, N, F, M, M, M, F, N. These characters have levels, the levels are stored in the factor fact. Level 1 is February (F), level 2 is March (M) and level 2 is November (N). Those are represented in fact as a vector of integers, which is shown below that: 2, 1, 1, 3, 1, 2, 2, 2, 1, 3

---

6 *Next on R Workshops* : topics covered in the next workshop

In Workshop 3, you can look forward to learning about:

- Data exploration and summary information
- Data manipulation
- Basic visualisations incl. different types of plots, plot customization, saving and export.

---

7 Resources

- Basic R cheat sheet
- Data Carpentry Introduction to R
- Software Carpentry: Programming with R.
- R for Data Science

---

8 Acknowledgements

Inspired by and adopted from:

John Blischak, Daniel Chen, Harriet Dashnow, and Denis Haine (eds): "Software Carpentry: Programming with R." Version 2016.06, June 2016, https://github.com/swcarpentry/r-novice-inflammation, 10.5281/zenodo.57541.

Jenny Bryan: https://jennybc.github.io/purrr-tutorial

Manuel Gimond: https://mgimond.github.io/ES218/Week02a.html

Damaris Zurell (Ecology & Macroecology Lab, Univ. Potsdam 2020-2022): https://damariszurell.github.io/EEC-R-prep/index.html

---

9 License