# Data Science and Data Analytics

## The Essentials of R Programming

Julian Amon, PhD
Charlotte Fresenius Privatuniversität

March 21, 2025

# R and RStudio

# Programming languages for Data Science

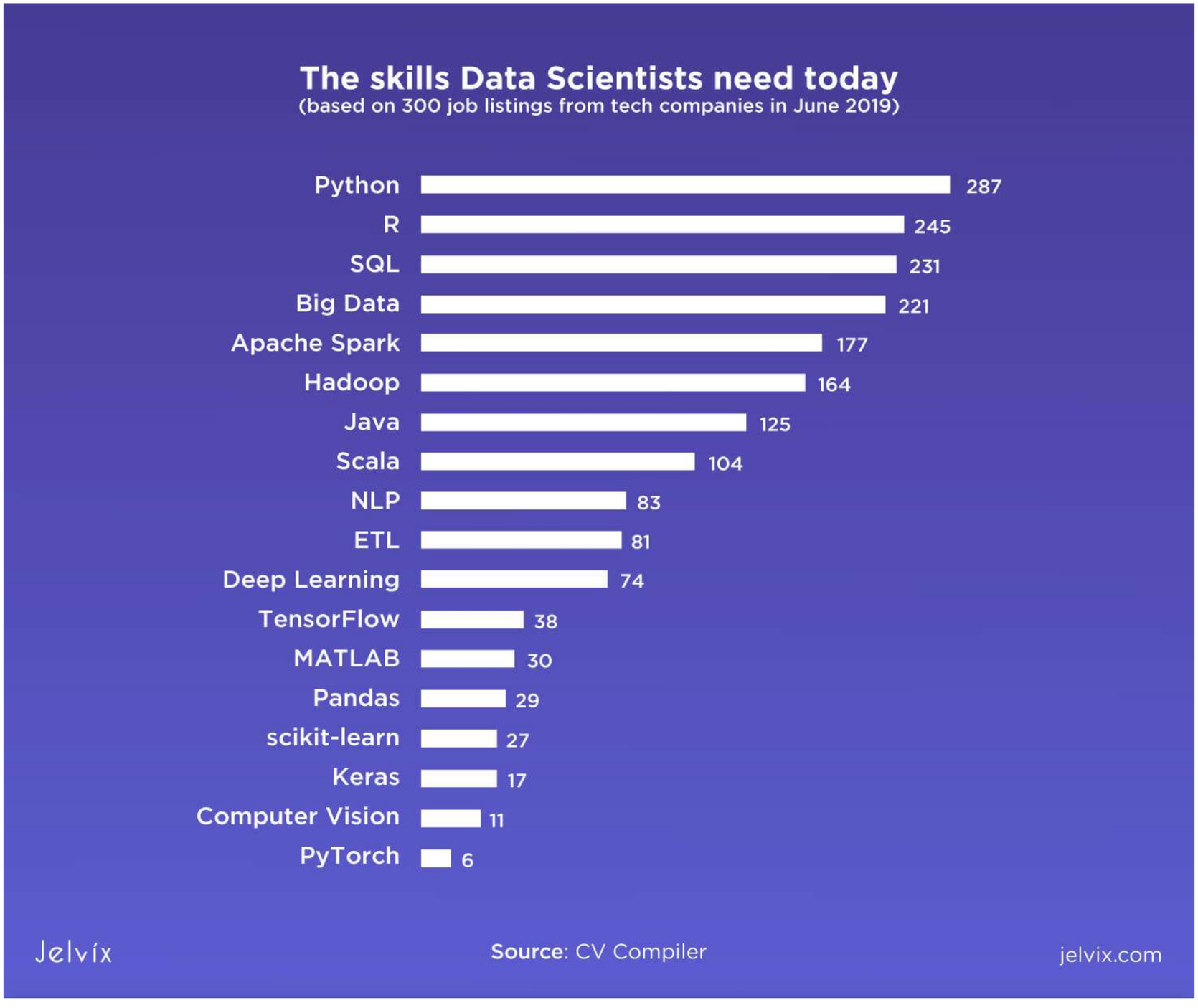

# What is R?

- **R** is an open-source statistical **programming language**.
- R is *not* a programming language like C or Java. It was not created by software engineers for software development. Instead, it was developed by statisticians as an environment for statistical computing and graphics.
- Some of its attractive features are:
  - R is free and open source.
  - R runs on all major platforms: Windows, macOS, UNIX/Linux.
  - There is a large, growing, and active community of R users and, as a result, there are numerous resources for learning and asking questions.
  - It is easily extensible through **packages**. In this way, it is easy to share software implementations of new data science methodologies.

# A (very) brief history of R

- R was first developed by Robert Gentleman and Ross Ihaka (University of Auckland, New Zealand) in the 1990s.
- Very early in the development of R, two statisticians at **TU Vienna** (Kurt Hornik and Fritz Leisch) got word about the initiative in New Zealand and started contributing to R.
- Soon, the growing interest in this project from researchers all over the world led to the establishment of the **R Core team**, which maintains R to this day.
- The strong involvement of the two Viennese researchers also led to the establishment of the **R Foundation for Statistical Computing** in Vienna (now at WU). Its main mission is the support of the continued development of R.
- Today, R is the lingua franca of statistics and an essential tool for data science.

# Importance of R in Data Science



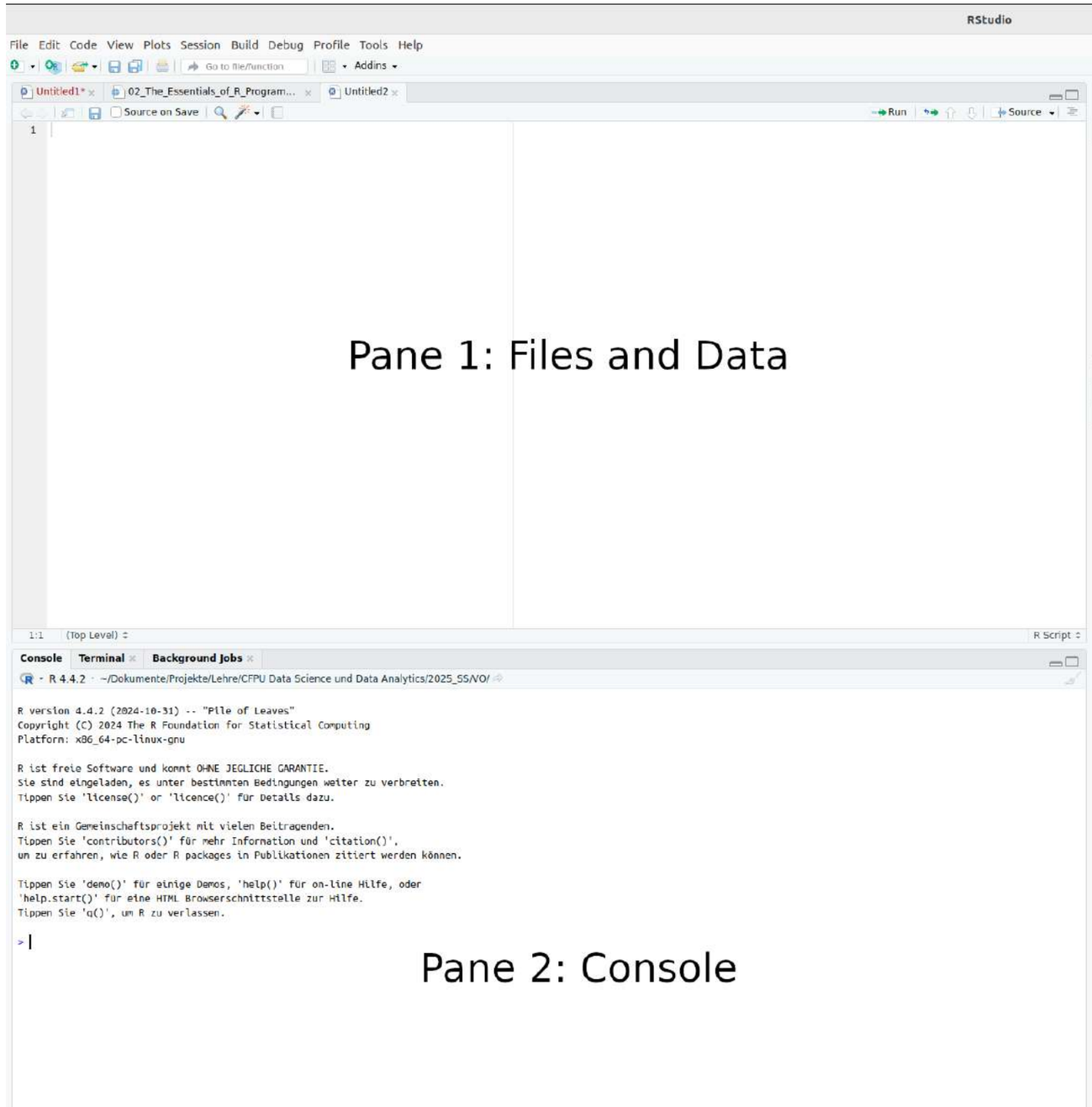
# What is RStudio?

- **RStudio** is a convenient interface for R called an integrated development environment (IDE). An IDE is the program on a computer that a developer uses to write code.
- RStudio is an open-source IDE by Posit specifically designed to support developers in writing **R code**. It is not a requirement for programming with R, but it is very commonly used by R programmers and data scientists.

| R: Engine | RStudio: Dashboard |
| --- | --- |
|  |  |

# What is RStudio?

# A Tour of RStudio



# A Tour of RStudio

- While we can write and execute code directly in the **R console** in the bottom left pane in RStudio, it is better to write all code in an **R script**.
- An R script is simply a text file that contains R code.
- In RStudio, R scripts can be opened, viewed and edited in the top left pane. To open a new R script, go to `File > New File > R script`.
- A single line of code in an R script can be sent to the console for execution via `Ctrl + Enter` (or `Cmd + Enter` on Mac).
- To write a comment in an R script, simply put a hashtag (#) in front of the comment:

```
1  # This is our first R script
2  2 + 2 # We calculate 2 + 2
```

```
[1] 4
```

# First steps

# First steps in R – Mathematical operators

First, let's use R as a **calculator**. We can write a calculation into our R script and R will give us the result, when the code is executed:

```
2 + 3 # Addition
```

```
[1] 5
```

```
2 - 4 # Subtraction
```

```
[1] -2
```

```
3 * 4 # Multiplication
```

```
[1] 12
```

```
4 / 2 # Division
```

```
[1] 2
```

```
3^3 # Exponentiation
```

```
[1] 27
```

```
(2 + 3)*(3 + 4) # Bracketing
```

```
[1] 35
```

```
2^(3 + 4/2)*4 # All together
```

```
[1] 128
```

# First steps in R – Relational and logical operators

We can also perform comparisons using R:

```
1 3 < 5
```

```
[1] TRUE
```

```
1 3 > 5
```

```
[1] FALSE
```

```
1 3 <= 3
```

```
[1] TRUE
```

```
1 3 >= 5
```

```
[1] FALSE
```

```
1 3 == 5 # Attention for the double equal sign!
```

```
[1] FALSE
```

```
1 3 != 5
```

```
[1] TRUE
```

With the help of & and |, we can also carry out AND and OR operations:

```
1 (3 == 5) | (5 == 5) # OR
```

```
[1] TRUE
```

```
1 (3 == 5) & (5 == 5) # AND
```

```
[1] FALSE
```

# First steps in R – Functions

R comes with many functions that you can use to perform tasks from simple to sophisticated. Functions have **inputs** (aka **arguments**) that you pass into them and **outputs** (aka **return values**) that they give back. Functions are fundamental to how R works.

Let's see an example. Say, we want to round the number 5.293586 to two decimal digits. Fortunately, there is a function in R called `round`. But how do we use it?

To find out, we can bring up the R help, which provides documentation for every function in R, by typing `?` and the name of the function into the console:

```
1  ?round
```

# First steps in R – Functions

Round {base} R Documentation

## Rounding of Numbers

### Description

`ceiling` takes a single numeric argument `x` and returns a numeric vector containing the smallest integers not less than the corresponding elements of `x`.

`floor` takes a single numeric argument `x` and returns a numeric vector containing the largest integers not greater than the corresponding elements of `x`.

`trunc` takes a single numeric argument `x` and returns a numeric vector containing the integers formed by truncating the values in `x` toward `0`.

`round` rounds the values in its first argument to the specified number of decimal places (default 0). See 'Details' about "round to even" when rounding off a 5.

`signif` rounds the values in its first argument to the specified number of *significant* digits. Hence, for `numeric x`, `signif(x, dig)` is the same as `round(x, dig - ceiling(log10(abs(x))))`.

### Usage

```
ceiling(x)
floor(x)
trunc(x, ...)

round(x, digits = 0, ...)
signif(x, digits = 6)
```

### Arguments

| | |
|---|---|
| `x` | a numeric vector. Or, for `round` and `signif`, a complex vector. |
| `digits` | integer indicating the number of decimal places (`round`) or significant digits (`signif`) to be used. For `round`, negative values are allowed (see 'Details'). |
| `...` | arguments to be passed to methods. |

# First steps in R – Functions

```
1  round(5.293586, 2)
```

```
[1] 5.29
```

Generally, we can use the `args` function to see the arguments that a function takes. For example, let's say, we wanted to simulate someone throwing a regular die, i.e. randomly **sample** numbers from 1 to 6. Luckily, there is an R function called `sample`. Let's inspect it.

```
1  args(sample)
```

```
function (x, size, replace = FALSE, prob = NULL)
NULL
```

To find out more information on what these arguments actually mean, let's consult the documentation again:

```
1  ?sample
```

# First steps in R – Functions

sample {base} R Documentation

## Random Samples and Permutations

### Description

`sample` takes a sample of the specified size from the elements of x using either with or without replacement.

### Usage

```
sample(x, size, replace = FALSE, prob = NULL)

sample.int(n, size = n, replace = FALSE, prob = NULL,
           useHash = (n > 1e+07 && !replace && is.null(prob) && size <= n/2))
```

### Arguments

| | |
|---|---|
| `x` | either a vector of one or more elements from which to choose, or a positive integer. See 'Details.' |
| `n` | a positive number, the number of items to choose from. See 'Details.' |
| `size` | a non-negative integer giving the number of items to choose. |
| `replace` | should sampling be with replacement? |
| `prob` | a vector of probability weights for obtaining the elements of the vector being sampled. |
| `useHash` | `logical` indicating if the hash-version of the algorithm should be used. Can only be used for `replace = FALSE`, `prob = NULL`, and `size <= n/2`, and really should be used for large `n`, as `useHash=FALSE` will use memory proportional to `n`. |

# First steps in R – Functions

So, to simulate 20 throws of a regular die, we have to use the function like this:

```
 1  sample(6, 20, replace = TRUE)
```

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

Note that we can give the arguments to the function

- in the order that they are given in the documentation (then they do not need to be named).
- by explicitly naming them when calling the function (then the order does not matter).
- by mixing unnamed and named arguments (like we have done here).

Let’s see another example of this.

# First steps in R – Functions

Let's say, we wanted to compute the base 2 logarithm of 10. For this, we need the `log` function in R:

```
1 args(log)
```

```
function (x, base = exp(1))
NULL
```

So, based on the three options given on the previous slide, all of the following three ways of calling `log` lead to the correct result:

```
1 log(10, 2)
```

```
[1] 3.321928
```

```
1 log(base = 2, x = 10)
```

```
[1] 3.321928
```

```
1 log(10, base = 2)
```

```
[1] 3.321928
```

However, options 1 and 3 are preferable, as switching the order of arguments tends to make code less easily readable for others (which could be yourself in the future…)

# First steps in R – Variables and assignment

Let's say, you were given the task to solve the quadratic equation

$$3x^2 - 5x - 1 = 0$$

You will remember from back in your high school days that we can use the "midnight formula" for this:

$$x_{1/2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

With R, we can easily compute this of course:

```
1  (-(-5) + sqrt((-5)^2 - 4*3*(-1)))/(2*3)
```

```
[1] 1.847127
```

```
1  (-(-5) - sqrt((-5)^2 - 4*3*(-1)))/(2*3)
```

```
[1] -0.1804604
```

# First steps in R – Variables and assignment

Next, we are given another quadratic equation

$$4x^2 - 8x + 2 = 0$$

To compute the solution with R, we would have to replace every occurrence of $a$, $b$ and $c$, so in total, we would have to make 10 replacements to get both solutions. That's too cumbersome and error-prone…

Instead, we can define **variables** $a$, $b$ and $c$ and simply **assign** different values to them every time we have to solve a quadratic equation. Such assignments happen in R with the help of the assignment operator `<-` (read it as "gets"):

```
1 a <- 3
2 b <- -5
3 c <- -1
4 # Note that nothing is printed to the console when assigning variables.
```

Now, we can write expressions using these variables like we would in maths.

# First steps in R – Variables and assignment

```
(-b + sqrt(b^2 - 4*a*c))/(2*a)
```

```
[1] 1.847127
```

```
(-b - sqrt(b^2 - 4*a*c))/(2*a)
```

```
[1] -0.1804604
```

Now, to compute the solution to the second quadratic equation, we simply re-define the variables a, b and c and then evaluate the same expression again:

```
a <- 4
b <- -8
c <- 2

(-b + sqrt(b^2 - 4*a*c))/(2*a)
```

```
[1] 1.707107
```

```
(-b - sqrt(b^2 - 4*a*c))/(2*a)
```

```
[1] 0.2928932
```

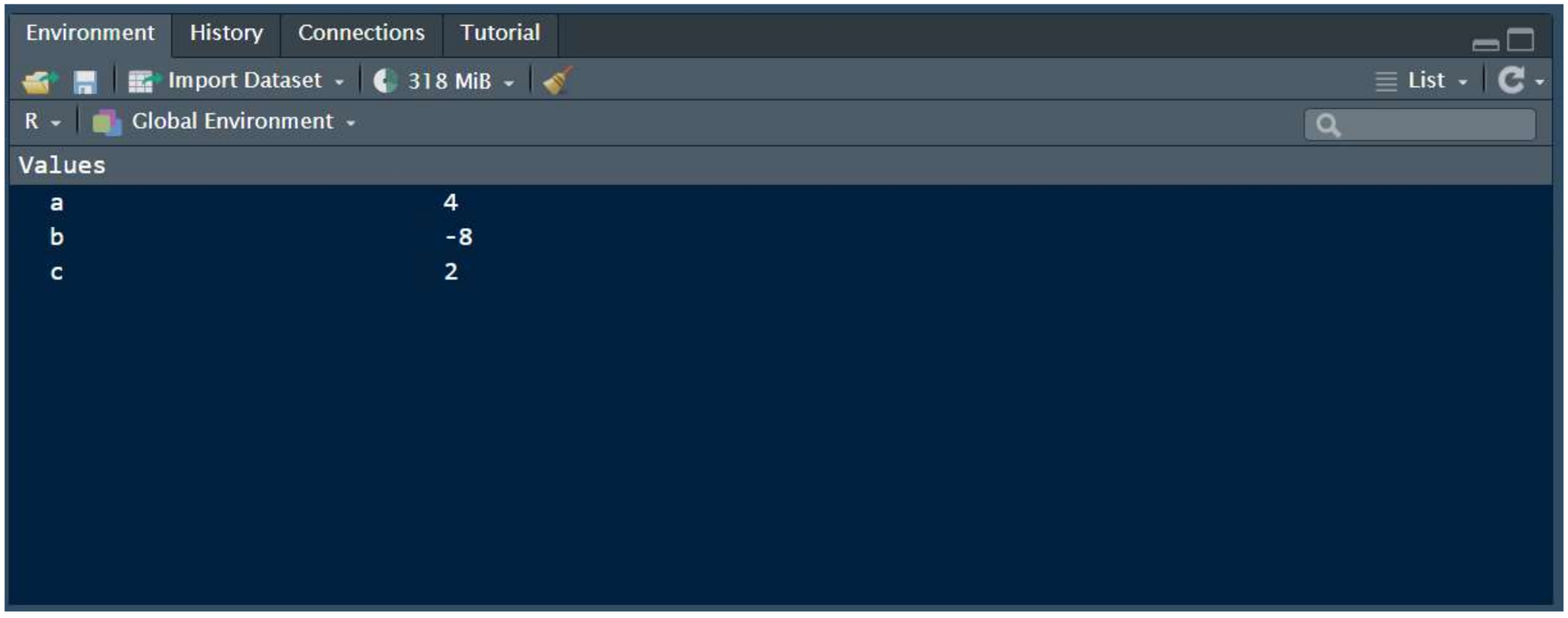
Note that R is **case-sensitive**, so it now knows variables with names a, b and c, but it does not know A, B or C:

```
A
```

```
Error: Objekt 'A' nicht gefunden
```

# First steps in R – Variables and assignment

With assignments, we are creating **objects** in R that are saved and can be referenced by the names that we give them (e.g. a, b and c). Creating **objects** like this will make them appear in the work space in pane 4 of the RStudio window:



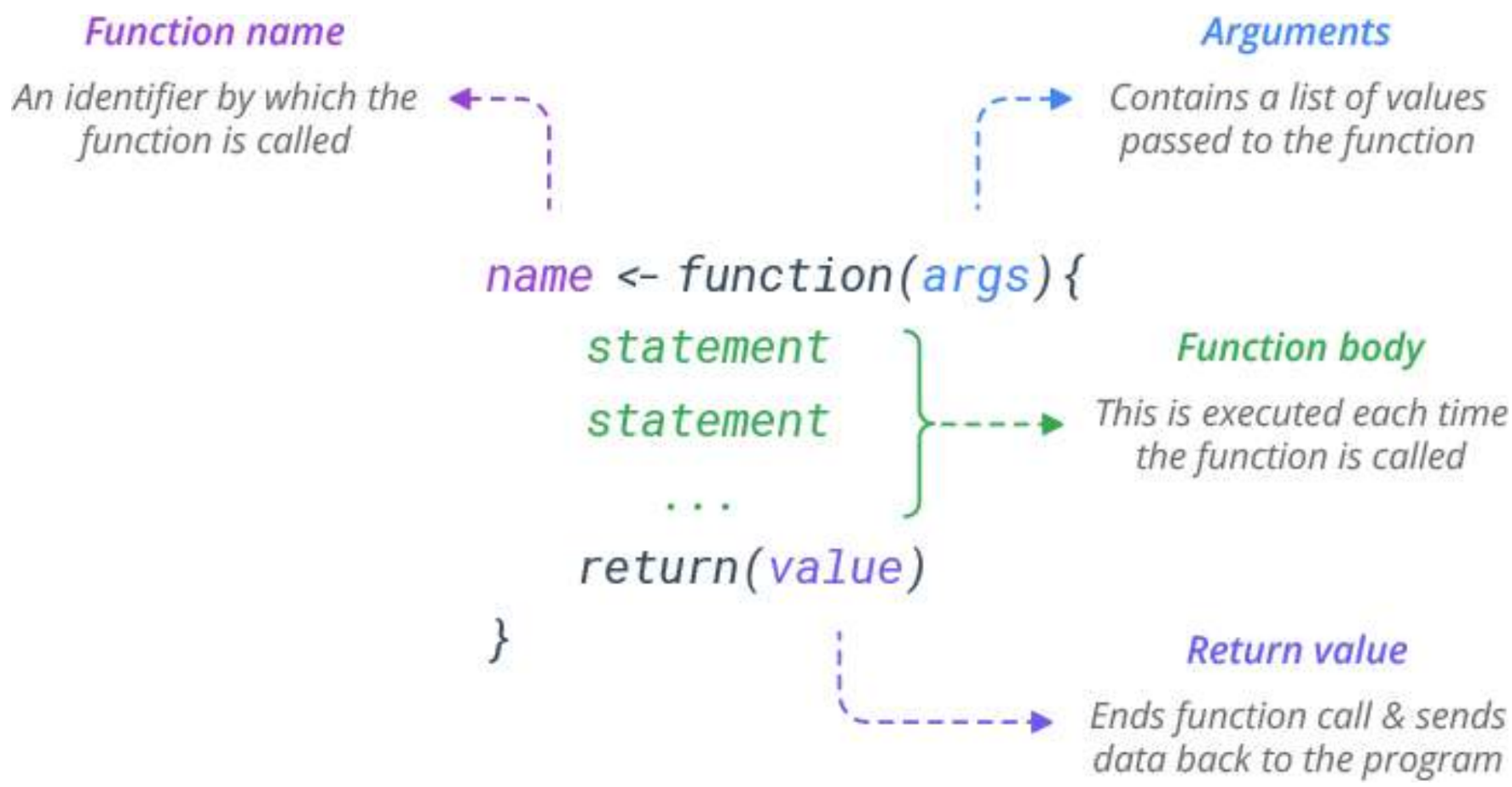

We can also see all variables currently defined in the work space by typing `ls()`:

```
1 ls()
```

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

# First steps in R – User-defined functions

With the assignment operator, we can also define our own functions of course! To define a function, we need a **function name**, **arguments**, a **function body** and a **return value**:



# First steps in R – User-defined functions

Consider the following example: say we want to write a function called `quadratic_solver` (function **name**) that gives us the solution to any quadratic equation. It needs input **arguments** `a`, `b` and `c` and should **return** the two solutions. In the function **body**, the two solutions should be computed for the three inputs. So, we could create the function as follows:

```
1 quadratic_solver <- function(a, b, c){
2   sol_1 <- (-b + sqrt(b^2 - 4*a*c))/(2*a)
3   sol_2 <- (-b - sqrt(b^2 - 4*a*c))/(2*a)
4   return(c(sol_1, sol_2))
5 }
```

## Now, we can use this function like any other:

```
1 quadratic_solver(3, -5, -1)
```

```
[1]  1.8471271 -0.1804604
```

```
1 quadratic_solver(4, -8, 2)
```

```
[1] 1.7071068 0.2928932
```

# Data types and Data structures

# Data types in R

In R, the following six **data types** are available:

- double (i.e., double precision floating-point number – R lingo for real number),
- integer,
- character (sometimes referred to as string),
- logical/boolean (can only take values `TRUE` or `FALSE`),
- complex (as in complex numbers, not relevant for us),
- raw (not relevant for us),

We can find the type of an object using the function `typeof`. We can verify, whether an object is of a certain `<type>` by using the function `is.<type>`. Let's see some examples.

# Double

By default, R will save any number that you type in as a `double`.

```
1 a <- 2
2 b <- 3.141593
3
4 typeof(a)
```

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

```
1 typeof(b)
```

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

```
1 is.double(a)
```

```
[1] TRUE
```

```
1 is.double(b)
```

```
[1] TRUE
```

Together with `integer`, the data type `double` is one of the two `numeric` types, i.e. representing numbers:

```
1 is.numeric(a)
```

```
[1] TRUE
```

```
1 is.numeric(b)
```

```
[1] TRUE
```

# Integer

**Integers** (whole numbers) are (positive or negative) numbers that can be written without a decimal component. This data type is more important for developers as it saves memory (compared to doubles). To specify an integer over a double, the number has to be followed by an uppercase L:

```
1 i <- 10L
2
3 typeof(i)
```

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

```
1 is.integer(i)
```

```
[1] TRUE
```

```
1 is.double(i)
```

```
[1] FALSE
```

## (Pure) integers are also `numeric` of course:

```
1 is.numeric(i)
```

```
[1] TRUE
```

# Character

Text data is represented in R with the help of the `character` data type. To demarcate a string of characters, you can use double or single quotes (`""` or `''`).

```
letter <- "a"
w <- 'Hello, world.'

typeof(letter)
```

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

```
typeof(w)
```

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

```
is.character(letter)
```

```
[1] TRUE
```

```
is.character(w)
```

```
[1] TRUE
```

Of course, `character` objects are *not* numeric.

```
is.numeric(letter)
```

```
[1] FALSE
```

```
is.numeric(w)
```

```
[1] FALSE
```

# Logical

When working with relational operators, we already saw a couple of instances of the `logical` data type. It can only take the values TRUE and FALSE. Negation of a `logical` object (i.e. saying **NOT**) can be achieved with the help of the `!` operator:

```
bigger <- 5 > 3

typeof(bigger)
```

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

```
is.logical(bigger)
```

```
[1] TRUE
```

```
!bigger
```

```
[1] FALSE
```

`Logical` objects are also *not* numeric.

```
is.numeric(bigger)
```

```
[1] FALSE
```

```
is.numeric(!bigger)
```

```
[1] FALSE
```

# Type coercion

We can **coerce** an object to be of a certain `<type>` by using the function `as.<type>`. This process is called **type coercion**.

In some cases, this is very intuitive…

```
as.double("2.71828")
```

```
[1] 2.71828
```

```
as.integer("10")
```

```
[1] 10
```

```
as.numeric("10") # creates a double
```

```
[1] 10
```

```
as.character(42)
```

```
[1] "42"
```

## … in others, maybe not so much…

```
as.integer(2.9) # returns the closest smaller integer
```

```
[1] 2
```

```
as.logical(2.9) # returns TRUE for any number other than 0
```

```
[1] TRUE
```

```
as.logical(0)
```

```
[1] FALSE
```

# Special values

Non-sensical attempts at coercion are translated as `NA` (not available).

```
1 as.numeric("Hello, world.")
```

```
Warning: NAs durch Umwandlung erzeugt
```

```
[1] NA
```

```
1 as.logical("5")
```

```
[1] NA
```

`NA` is one of four **special values**. While `NA` can occur for any data type, the other three can only occur for `numeric` data types. These are:

- `Inf`: positive infinity
- `-Inf`: negative infinity
- `NaN`: not a number

While they are not technically numbers, these special values still follow logical rules when applying mathematical operations on them.

# Special values

```
1 2 / 0
```

```
[1] Inf
```

```
1 2 / 0 + 2 / 0
```

```
[1] Inf
```

```
1 -2 / 0
```

```
[1] -Inf
```

```
1 -2 / 0 - 2 / 0
```

```
[1] -Inf
```

```
1 0 / 0
```

```
[1] NaN
```

```
1 2 / 0 - 2 / 0
```

```
[1] NaN
```

```
1 - 1 / 0 + 2 / 0
```

```
[1] NaN
```

Unlike `NaN`, `NA`s are genuinely **unknown** values. Nevertheless, they also have their logical rules. Consider the following example. Let's think about why each of the following four results makes sense:

```
1 TRUE | NA
```

```
[1] TRUE
```

```
1 TRUE & NA
```

```
[1] NA
```

```
1 FALSE | NA
```

```
[1] NA
```

```
1 FALSE & NA
```

```
[1] FALSE
```

Note: all special values have their own `is.<special>` function to check for them, i.e. `is.na`, `is.nan` and `is.infinite`.

# Data structures

- On top of these six basic **data types**, R builds several kinds of more complex **data structures** in different ways to suit different applications that are regularly encountered in the statistics / data science context.
- We will use the following regularly and hence discuss them in more detail:
  - (Atomic) `vector`
  - `list`
  - `matrix`
  - `data.frame`
  - `factor`
  - `Date`

# Atomic vector

An **atomic vector** is a simple vector of values of **one data type**. Values can be **concatenated** together into a vector using the `c` function. For example:

```
1 x <- c(3, 4, 5)
2 x
```

```
[1] 3 4 5
```

Since we have filled the vector `x` with values of type `double`, it will itself also be of type `double`. It has several attributes / characteristics such as `length`:

```
1 typeof(x)
```

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

```
1 is.double(x)
```

```
[1] TRUE
```

```
1 length(x)
```

```
[1] 3
```

We can also assign names to the vector elements:

```
1 names(x) <- c("a", "b", "c")
2 x
```

```
a b c
3 4 5
```

# Atomic vector

What happens if we try to create a vector with data of different types?

```
1 y <- c(1, "a", TRUE)
2 y
```

```
[1] "1"    "a"    "TRUE"
```

When we attempt to combine different types, R will **coerce** the data in a fixed order, namely `character` → `double` → `integer` → `logical`, i.e. if any data of a higher-order type appears in the vector creation, the vector will be of that type:

```
1 typeof(y)
```

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

So, what will be the type of the following vector?

```
1 z <- c(0L, 1L, 2, TRUE)
```

As the `2` is of type `double`, this is the highest-order type in the vector, so:

```
1 typeof(z)
```

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

# Working with vectors – Vectorization

Most operations in R are **vectorized**, which means they are (automatically) performed element by element:

```
x
```

```
a b c
3 4 5
```

```
x^2
```

```
 a  b  c
 9 16 25
```

```
log(x)
```

```
       a        b        c
1.098612 1.386294 1.609438
```

This also applies when we want to add / subtract / multiply / divide two vectors **element-wise**:

```
y <- c(5, 12, 13)
x + y
```

```
 a  b  c
 8 16 18
```

```
x * y
```

```
 a  b  c
15 48 65
```

# Working with vectors – Vectorization

If two vectors are of different lengths, then R recycles the smaller one to allow operations like the following:

```r
x <- c(1, 2, 3)
y <- c(10, 20, 30, 40, 50, 60)
x + 2
```

```
[1] 3 4 5
```

```r
x + y
```

```
[1] 11 22 33 41 52 63
```

However, **beware of recycling**:

```r
x <- c(1, 2, 3)
y <- c(10, 20, 30, 40, 50, 60, 70)
x + y
```

```
Warning in x + y: Länge des längeren Objektes
    ist kein Vielfaches der Länge des kürzeren Objektes
```

```
[1] 11 22 33 41 52 63 71
```

# Working with vectors – Sequences

To create vectors that are **sequences**, there is a very useful R function called `seq`:

```
1 seq(1, 10)
```

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

```
1 seq(2, 20, by = 2)
```

```
 [1]  2  4  6  8 10 12 14 16 18 20
```

```
1 seq(1.2, 0.2, by = -0.1)
```

```
 [1] 1.2 1.1 1.0 0.9 0.8 0.7 0.6 0.5 0.4 0.3 0.2
```

Sequences of **consecutive integers** – like the first one we saw – are particularly frequently needed in (R) programming. For this reason, there is a short-hand notation ("syntactic sugar") to create sequences of that kind, which is `start_value:end_value`:

```
1 s <- 1:10
2 typeof(s)
```

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

```
1 s
```

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

# Working with vectors – Subsetting

If we want to extract or replace elements of a vector, you can use **square brackets** [ ] with logical or numeric input:

```r
alphabet <- c("a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n",
              "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z")
# Numeric subsetting
alphabet[1:3] # first three
```

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

```r
alphabet[3:1] # first three in descending order
```

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

```r
alphabet[c(2, 4)] # second and fourth
```

```
[1] "b" "d"
```

```r
alphabet[-(1:10)] # all except the first 10
```

```
 [1] "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```r
alphabet[5] <- "E" # replace the fifth element by a capital E

# Logical subsetting
alphabet == "E"
```

```
 [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[25] FALSE FALSE
```

```r
alphabet[alphabet == "E"]
```

```
[1] "E"
```

# Working with vectors – Subsetting

We can also use the `which` function to turn a logical vector into an integer vector that gives the indices of all elements in the vector that are `TRUE`:

```
 1 which(alphabet == "E")
```

```
[1] 5
```

```
 1 alphabet[which(alphabet == "E")]
```

```
[1] "E"
```

With the `%in%` operator, we can check whether elements of the vector are contained **in** another vector:

```
 1 alphabet %in% c("x", "y", "z")
```

```
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
[25]  TRUE  TRUE
```

```
 1 which(alphabet %in% c("x", "y", "z"))
```

```
[1] 24 25 26
```

```
 1 alphabet[alphabet %in% c("x", "y", "z")]
```

```
[1] "x" "y" "z"
```

# Working with vectors – Sorting

For sorting vectors, we can use the `sort` function. This works with `character` vectors:

```
# Sorting a character vector
unsorted_letters <- sample(alphabet)
unsorted_letters
```

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

```
sorted_letters <- sort(unsorted_letters)
sorted_letters
```

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

## … as well as with numeric vectors:

```
# Sorting a numeric vector in decreasing order:
unsorted_numbers <- sample(1:20)
unsorted_numbers
```

```
 [1]  3 17 18  6 19  2  4 20 16  5 15  7 12 10  9 11  1 14 13  8
```

```
sorted_numbers <- sort(unsorted_numbers, decreasing = TRUE)
sorted_numbers
```

```
 [1] 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1
```

# List

Lists are a step up in complexity from atomic vectors: each element can be any type, not just vectors. We construct lists with the function `list`:

```
l1 <- list(1:3, "R", c(TRUE, FALSE, TRUE), c(2.3, 5.9))

l1
```

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

[[2]]
[1] "R"

[[3]]
[1]  TRUE FALSE  TRUE

[[4]]
[1] 2.3 5.9
```

```
typeof(l1)
```

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

Lists are sometimes called **recursive** vectors because a list can contain other lists. This makes them fundamentally different from atomic vectors.

# List

The elements of a list can also have names. These can be accessed with the help of the $ operator. Alternatively, to access **a single** element of a list, we can also use **double square brackets** [[]]:

```
names(l1) <- c("A", "B", "C", "D")
l1
```

```
$A
[1] 1 2 3

$B
[1] "R"

$C
[1]  TRUE FALSE  TRUE

$D
[1] 2.3 5.9
```

```
l1$A
```

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

```
l1[[3]]
```

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

```
l1[["D"]]
```

```
[1] 2.3 5.9
```

# Working with lists – Subsetting

For subsetting lists, essentially the same rules apply as for atomic vectors, i.e. we can subset using numeric or logical arguments and single square brackets `[]`:

```
1 # Numeric subsetting
2 l1[1:2] # First two elements
```

```
$A
[1] 1 2 3

$B
[1] "R"
```

```
1 l1[-(1:2)] # All elements except the first
```

```
$C
[1]  TRUE FALSE  TRUE

$D
[1] 2.3 5.9
```

```
1 # Logical subsetting
2 l1[c(FALSE, FALSE, FALSE, TRUE)]
```

```
$D
[1] 2.3 5.9
```

```
1 l1[lengths(l1) == 3] # all list elements whose length is 3
```

```
$A
[1] 1 2 3

$C
[1]  TRUE FALSE  TRUE
```

# Matrix

One way to construct more complex data structures on top of elementary building blocks like vectors or lists is to assign a `class` to them. A **class** is metadata about the object that can determine how common functions operate on that object.

Probably the easiest example of this is a `matrix`. A matrix is just a two-dimensional array of numbers:

```
1  m <- matrix(1:12, nrow = 3, ncol = 4)
2  m
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

The function `matrix` will fill up the matrix column by column by default. There is an option to fill them by row instead, see `?matrix`.

# Matrix

Technically speaking, in R, a matrix is actually just a vector with an **attribute** that specifies the **dimensions** (i.e. number of rows and columns) of the matrix. We can access the dimensions of a matrix with the help of the functions `dim`, `nrow` and `ncol`:

```
typeof(m) # just an integer vector behind the scenes
```

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

```
dim(m) # but dimensions are 3 x 4 (3 rows, 4 columns)
```

```
[1] 3 4
```

```
nrow(m) # number of rows
```

```
[1] 3
```

```
ncol(m) # number of columns
```

```
[1] 4
```

```
class(m) # It is of class matrix and array
```

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

# Matrix

We can add additional rows or columns with the help of `rbind` or `cbind`:

```
m <- cbind(m, c(13, 14, 15))
m
```

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

```
m <- rbind(rep(1, 5), m)
m
```

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

For a matrix, we can set both **row names** and **column names**:

```
rownames(m) <- c("row_1", "row_2", "row_3", "row_4")
colnames(m) <- c("A", "B", "C", "D", "E")
m
```

```
      A B C  D  E
row_1 1 1 1  1  1
row_2 1 4 7 10 13
row_3 2 5 8 11 14
row_4 3 6 9 12 15
```

# Working with matrices – Subsetting

As a matrix is a two-dimensional object, we need **two indices** separated by a comma for subsetting. If we do not specify one dimension, all elements across that dimension are selected:

```
 1 m[2, ] # 2nd row
```

```
 A  B  C  D  E
 1  4  7 10 13
```

```
 1 m[, 3] # 3rd column
```

```
row_1 row_2 row_3 row_4
    1     7     8     9
```

```
 1 m[3, 4] # 3rd element of the 4th column
```

```
[1] 11
```

```
 1 m[c(FALSE, FALSE, FALSE, TRUE), ] # 4th row
```

```
 A  B  C  D  E
 3  6  9 12 15
```

We can also subset by name if column and / or row names are provided:

```
 1 m["row_4", "E"]
```

```
[1] 15
```

# Data Frame

As a matrix is basically just a vector underneath, all values still need to be of the same data type. However, real tabular data often includes **data of different types** (e.g. name, height, education). To represent this, we need a data structure where the columns can be of different types.

In R, such a data structure is provided by the `data.frame`. Underlying it is a named list, whose elements represent columns. Therefore, all elements need to have **the same length**. Consider the following example:

```r
heights <- c(176, 178, 156)
names <- c("Anna", "Jakob", "Lisa")
educ <- c("BSc", "MA", "PhD")
df <- data.frame(heights = heights, names = names, educ = educ)
df
```

```
  heights names educ
1     176  Anna  BSc
2     178 Jakob   MA
3     156  Lisa  PhD
```

# Data Frame

Let's inspect the "list nature" of the `data.frame` object `df` we just created:

```
typeof(df)
```

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

```
class(df)
```

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

Note how, even though `df` is fundamentally a `list`, it behaves differently than a standard list:

```
df
```

```
  heights names educ
1     176  Anna  BSc
2     178 Jakob   MA
3     156  Lisa  PhD
```

```
as.list(df)
```

```
$heights
[1] 176 178 156

$names
[1] "Anna"  "Jakob" "Lisa"

$educ
[1] "BSc" "MA"  "PhD"
```

This is because, on top of being a list, the object `df` is of class `data.frame`, which changes how common functions operate on it.

# Working with Data Frames

Since a `data.frame` also represents two-dimensional data, many of the functions we used for matrices work on it as well:

```
1 dim(df)
```

```
[1] 3 3
```

```
1 nrow(df)
```

```
[1] 3
```

```
1 colnames(df)
```

```
[1] "heights" "names"   "educ"
```

In particular, all the subsetting methods we showed are also applicable:

```
1 df[1, ]
```

```
  heights names educ
1     176  Anna  BSc
```

```
1 df[, 2]
```

```
[1] "Anna"  "Jakob" "Lisa"
```

```
1 df[c(TRUE, TRUE, FALSE), 3]
```

```
[1] "BSc" "MA"
```

# Working with Data Frames

However, as `data.frame`s are lists underneath, we can also use the `$` operator to access the elements of the list as before.

```
df$heights
```

```
[1] 176 178 156
```

```
df$names
```

```
[1] "Anna"  "Jakob" "Lisa"
```

```
df$educ
```

```
[1] "BSc" "MA"  "PhD"
```

This is a particularly common action when working with real data, as it allows us to access the variables in the columns of the `data.frame`. For example, let's say we made a mistake and found out that Jakob was really 187 cm tall. We could change the corresponding data point as follows:

```
df$heights[2] <- 187
df
```

```
  heights names educ
1     176  Anna  BSc
2     187 Jakob   MA
3     156  Lisa  PhD
```

# Factor

A `factor` is R's way to represent **categorical data**. Say, for example, we want to add the sex of the three people in our data set to the `data.frame`. For this, we create a factor and add it as an additional column to `df`:

```
1 sex <- factor(c("f", "m", "f"))
2 df$sex <- sex
3 df
```

```
  heights names educ sex
1     176  Anna  BSc   f
2     187 Jakob   MA   m
3     156  Lisa  PhD   f
```

A `factor` has **levels** that represent the categories that this variable can take (here: "m" for male and "f" for female). In the background, R stores these levels as integers and keeps a map to keep track of the labels. This is more memory efficient than storing all the characters.

```
1 df$sex
```

```
[1] f m f
Levels: f m
```

# Factor

A `factor` is another example of a data structure that is built on top of an atomic vector using a `class` attribute. The data is stored as an integer vector (**data type**), but because the object is of class `factor`, R uses different **methods** to act on the object (compared to a standard integer vector). This type of behaviour is one of the things that makes R very powerful for data science.

```
typeof(df$sex)
```

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

```
as.integer(df$sex)
```

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

```
class(df$sex)
```

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

```
df$sex
```

```
[1] f m f
Levels: f m
```

As we will see, properly encoding categorical data as factors is an essential step in data preparation.

# Date

Finally, a common type of data that needs to be represented in R are **dates**. Dates are typically represented in some sort of format like “DD-MM-YYYY” in the European context, for instance. Let’s say, we are given Anna’s, Jakob’s and Lisa’s birthday in different formats:

```
1 bday_anna <- "1999-12-01" # YYYY-MM-DD
2 bday_jakob <- "16.4.98" # MM.DD.YYYY
3 bday_lisa <- "7/June/1995" # DD/MM/YYYY
```

R has a specific way of recognizing such date formats using so called **format strings**. For each way of representing date information, there is a corresponding format string. Some important ones are:

| Format string | Description | Format string | Description |
|---|---|---|---|
| %Y | Year with century | %y | Year without century |
| %m | Month of year (01-12) | %j | Day of the year (0-366) |
| %d | Day of month (01-31) | %W | Calendar week (0-53) |
| %B | Full month (e.g. June) | %b | Abbreviated month (e.g. Jun) |

# Working with dates

So let's try to use this to convert our dates (which are now just `character` objects) to actual dates that R understands using `as.Date`:

```
bday_anna <- as.Date("1999-12-01", "%Y-%m-%d")
bday_jakob <- as.Date("16.4.98", "%d.%m.%y")
bday_lisa <- as.Date("7/June/1995", "%d/%B/%Y")
bday_anna
```

```
[1] "1999-12-01"
```

```
bday_jakob
```

```
[1] "1998-04-16"
```

```
bday_lisa
```

```
[1] "1995-06-07"
```

## Now, we can add these dates as a column to our `data.frame`

```
df$bday <- c(bday_anna, bday_jakob, bday_lisa)
df
```

```
  heights names educ sex       bday
1     176  Anna  BSc   f 1999-12-01
2     187 Jakob   MA   m 1998-04-16
3     156  Lisa  PhD   f 1995-06-07
```

# Working with dates

Dates are stored in R as the number of days since Unix time, which is the 1st January, 1970. Hence, date vectors are simply `double` vectors of class `Date`.

```
1 typeof(df$bday)
```

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

```
1 as.double(df$bday)
```

```
[1] 10926 10332  9288
```

```
1 class(df$bday)
```

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

With this underlying representation, we can do maths on dates:

```
1 df$bday
```

```
[1] "1999-12-01" "1998-04-16" "1995-06-07"
```

```
1 df$bday + 30
```

```
[1] "1999-12-31" "1998-05-16" "1995-07-07"
```

```
1 df$bday[2] - df$bday[1]
```

```
Time difference of -594 days
```

# Working with dates

Note that these operations cannot be performed on the original character strings representing the birthdays:

```
1 "1999-12-01" + 10
```

```
Error in "1999-12-01" + 10: nicht-numerisches Argument für binären Operator
```

To extract individual parts of a date (like year, month, day, weekday, etc.), we can use the `format` function in conjunction with the format strings we saw before for date creation:

```
1 format(df$bday, "%Y") # extract birth year
```

```
[1] "1999" "1998" "1995"
```

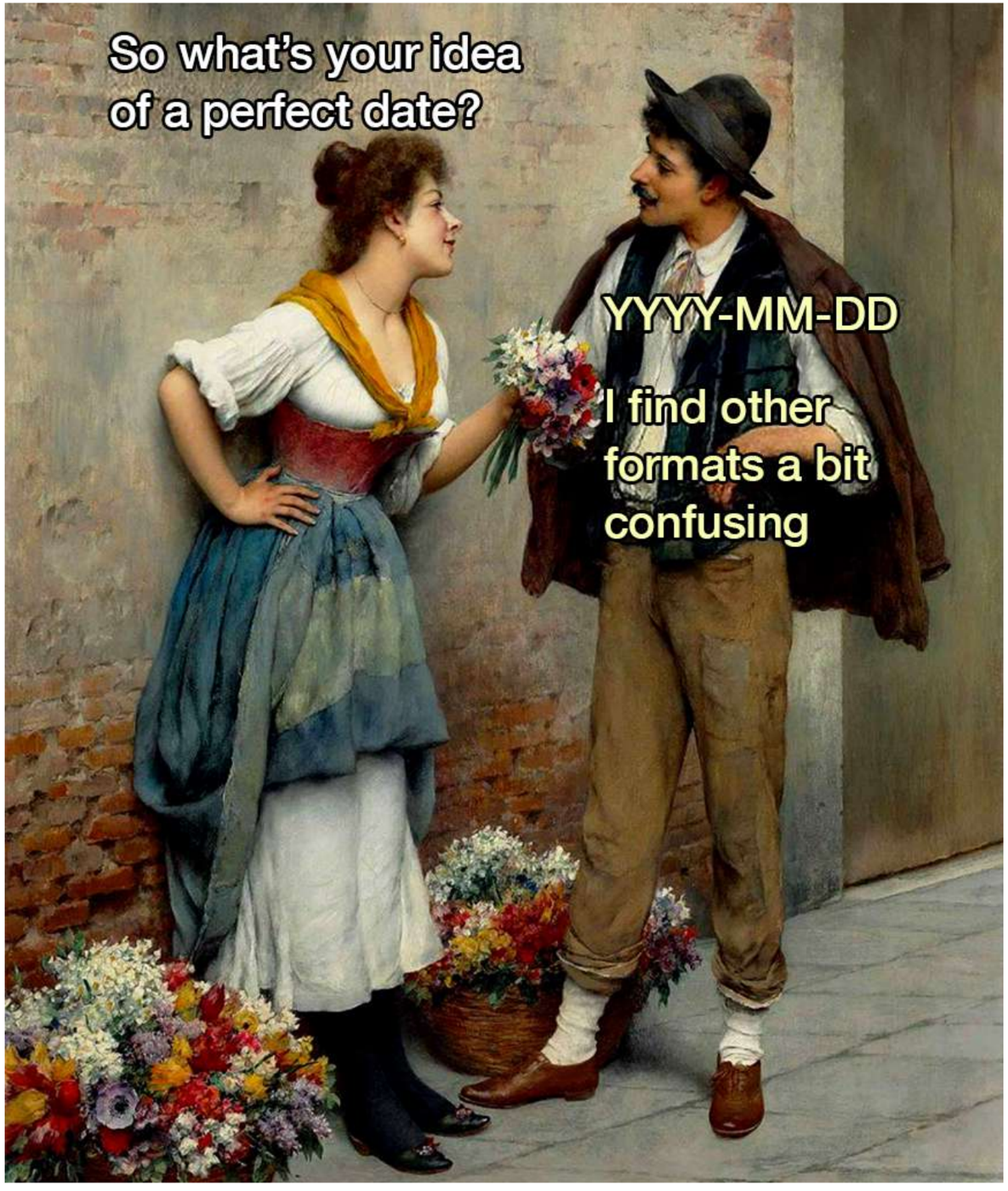
```
1 format(df$bday, "%B") # extract birth month
```

```
[1] "December" "April"    "June"
```

```
1 format(df$bday, "%A") # extract week day of birth date
```

```
[1] "Wednesday" "Thursday"  "Wednesday"
```

# Working with dates



# Programming basics

# R programming for Data Science

- By coding in R, we can efficiently perform exploratory data analysis, build machine learning pipelines, and prepare beautiful plots to communicate results effectively using data visualization tools.
- However, R is not just a data analysis environment but a **programming language**.
- The goal of this is course is not to teach advanced R programming, but in order to facilitate our lives as data scientists, we do need access to certain core programming principles like **conditionals**, **loops** and **functionals**.
- So let’s dive right in!

# Conditionals

**Conditionals** are one of the basic features of programming. They are used for what is called **control flow**. The most common conditional expression is the `if-else` statement. It's best illustrated with an example:

```
a <- 0
if (a != 0) {
  print(1/a)
} else{
  print("Can't divide by zero!")
}
```

```
[1] "Can't divide by zero!"
```

The condition `a != 0` was evaluated to be `FALSE` (since `a` was assigned 0). So, the conditional went to the `else` statement and printed "Can't divide by zero!" to the console as we told it to. By contrast:

```
a <- 4
if (a != 0) {
  print(1/a)
} else{
  print("Can't divide by zero!")
}
```

```
[1] 0.25
```

# Conditionals

Let's incorporate conditionals into our `quadratic_solver` function from before. Remember that a quadratic equation has no real solutions if the **discriminant** $D$ is negative (because of the root in the "midnight" formula):

$$D = b^2 - 4ac$$

So far, when this is case, our function produces `NaN`s and a warning:

```
1 quadratic_solver(2, 2, 2)
```

```
Warning in sqrt(b^2 - 4 * a * c): NaNs wurden erzeugt
Warning in sqrt(b^2 - 4 * a * c): NaNs wurden erzeugt
```

```
[1] NaN NaN
```

Let's fix this by:

- Returning no solutions / NA if $D < 0$.
- Returning one solution if $D = 0$.
- Returning two solutions (as before) if $D > 0$.

# Conditionals

## One solution could be the following:

```r
quadratic_solver_new <- function(a, b, c){
  D <- b^2 - 4*a*c
  if(D < 0){
    print("Discriminant is negative, no real solutions!")
    return(NA)
  } else if(D > 0) {
    print("Discriminant is positive, two real solutions!")
    sol_1 <- (-b + sqrt(D))/(2*a)
    sol_2 <- (-b - sqrt(D))/(2*a)
    return(c(sol_1, sol_2))
  } else {
    print("Discriminant is zero, one real solution!")
    sol <- -b/(2*a)
    return(sol)
  }
}
```

```r
quadratic_solver_new(2, 2, 2)
```

```
[1] "Discriminant is negative, no real solutions!"
[1] NA
```

```r
quadratic_solver_new(3, -5, 1)
```

```
[1] "Discriminant is positive, two real solutions!"
[1] 1.4342585 0.2324081
```

```r
quadratic_solver_new(1, 2, 1)
```

```
[1] "Discriminant is zero, one real solution!"
[1] -1
```

# Conditionals

`if-else` statements like the ones we saw only work on a single logical. For a **vectorized** version, there is the very useful `ifelse` function:

```
1 a <- c(0, 2, 4, 6, 0, 8)
2 ifelse(a != 0, 1/a, NA)
```

```
[1]        NA 0.5000000 0.2500000 0.1666667        NA 0.1250000
```

This function takes three arguments: a logical and two possible answers. If the logical is TRUE, the value in the second argument is returned and if FALSE, the value in the third argument is returned. When operating on vectors, `ifelse` takes the corresponding **elements** of the second or third argument.

Here' another example:

```
1 x <- 1:10
2 ifelse(x %% 2 == 0, "even", "odd")
```

```
 [1] "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even"
```

Note: `x %% 2` gives the remainder when dividing x by 2.

# For-loops

In general, **loops** control flow structures that enable the repeated execution of a code block as long as a specified condition is met. This saves a lot of manual work and code duplication. We will discuss two types of loops: `for` loops and `while` loops.

`for` loops are used to **iterate** over items in a vector. The logic is "for every item in this vector, do the following". This logic is implemented in the following basic form:

```
for(item in vector) {
  perform_action
}
```

Following this notation, we can refer to the element of the `vector` in the current loop cycle with the name of `item`. In the first cycle, it will be the first element of the vector, in the second, it will be the second, and so on…

# For-loops

Let's see an example of a `for` loop that prints both solutions to the quadratic equation from before in a nice format:

```
solutions <- quadratic_solver_new(3, -5, 1)
```

```
[1] "Discriminant is positive, two real solutions!"
```

```
n <- length(solutions)

for(i in 1:n){
  print(paste0("Solution ", i, ": ", round(solutions[i], 3)))
}
```

```
[1] "Solution 1: 1.434"
[1] "Solution 2: 0.232"
```

Note that we do not have to loop through an integer vector, we can loop through any atomic vector or list. For example, we could loop through all the elements of the list `l1` from earlier and print it to the console only if it is a numeric vector:

```
for(el in l1){
  if(is.numeric(el)){
    print(el)
  }
}
```

```
[1] 1 2 3
[1] 2.3 5.9
```

# While-loops

Another type of loop is a `while` loop. It repeats a specified action as long as a certain condition is met. It has the following basic form:

```
while(condition){
  perform_action
}
```

Note that `for` and `while` loops are interchangeable in **every circumstance**, i.e. every `for` loop can be implemented as a `while` loop and vice versa. Usually, the choice between the two is a question of code readability and efficiency considerations. Compare the following two examples:

```
i <- 1L
while(i <= 3L){
  print(i)
  i <- i + 1L
}
```

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

```
for(i in 1:3){
  print(i)
}
```

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

Note that the last step in the loop is crucial. Otherwise we loop infinitely!

# Functionals

In R, a very commonly used alternative to loops is the use of **functionals**. Functionals are functions that take another function as an input and returns a vector as output. Common functionals we will have a look at are `lapply`, and `apply`. Let's start with `lapply`.

`lapply` requires as arguments an atomic vector or a list and a function that it should apply to each element of that atomic vector or list. Let's say we wanted to sort each vector in a list of vectors. We could do:

```
1 l2 <- list(sample(1:10), sample(1:20))
2 l2
```

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

[[2]]
 [1]  5 14 20  2 17  8  1 16 18 15  7 11 12  4 19 10  3 13  9  6
```

```
1 lapply(l2, sort)
```

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

[[2]]
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

# Functionals

In fact, we can pass named arguments that we would usually pass to the function to be applied directly to `lapply` instead:

```
 1  lapply(l2, sort, decreasing = TRUE)
```

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

[[2]]
 [1] 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1
```

Instead of using `lapply` with a pre-existing (or user-defined) function, we can also create an inline function that exists only for the purpose of that `lapply` call. For example, we might want to square each vector in the list after sorting:

```
 1  lapply(l2, function(x) sort(x, decreasing = TRUE)^2)
```

```
[[1]]
 [1] 100  81  64  49  36  25  16   9   4   1

[[2]]
 [1] 400 361 324 289 256 225 196 169 144 121 100  81  64  49  36  25  16   9   4
[20]   1
```

Such functions are called **anonymous functions** as they do not have a name.

# Functionals

The **matrix** equivalent of `lapply` is called `apply`. It can apply a given function to every row and / or every column of a matrix. Besides requiring the matrix and the function to be applied, it requires an indication of whether application should happen over **rows** (1) or **columns** (2). Consider the following example to compute the row and column sums of a matrix:

```
1 m <- matrix(1:9, 3, 3)
2 m
```

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

```
 1 apply(m, 1, sum)
```

```
[1] 12 15 18
```

```
 1 apply(m, 2, sum)
```

```
[1]  6 15 24
```

The use of functionals like `lapply` and `apply` is usually preferable over loops because it pre-specifies what should happen with the result (e.g. `lapply` will always hold the results in a list).

# Packages

# Packages

- With the base installation, R comes with a lot of functionality already. However, statisticians and data scientists all over the world constantly come up with new methods and clever code to solve problems.
- To share this code, they create **R packages**: these include fundamental units of reproducible R code, including reusable R functions, the documentation that describes how to use them, and sample data.
- To make these packages available, the creators typically upload them to the **Comprehensive R Archive Network (CRAN)**.
- From there, any R user can download and install these packages and use them for their own workflows.

# Packages

- As of 22nd February 2025, there are 22,100 R packages available for download on CRAN. Each of them has a unique name that we can use to obtain it.
- We will be working only with a select few of them.
- One of them will be **ggplot2**, which is one of the most popular R packages used for data visualization.
- As an R programmer, one typically finds packages that solve a given problem by googling the problem one is encountering. Chances are that someone else has had this problem before…

To install a package, we call the function `install.packages` on the name of the package:

```
1 install.packages("ggplot2")
```

This downloads and installs the package on your computer.

# Packages

As with any software, an R package only needs to be **installed** once.

After installation, we need to tell R to make the functions provided by this package available to us in our current R session. For this we call the function `library` on the name of the package:

```
1 library(ggplot2)
```

- Now, all data and functions offered by this package can be used.
- A package comes with a reference manual and often additional documentation in the form of so called package **vignettes**.
- This helps the user understand how to use the functions in the package.
- Additionally, documentation on individual functions is always available through the help of the `?` operator.

# Packages

```
1  ggplot(mpg, aes(displ, hwy, colour = class)) +
2    geom_point() +
3    xlab("Engine Displacement") + ylab("MPG") +
4    theme(legend.text = element_text(size=14), axis.text = element_text(size=14), axis.title = eleme
```