Object types

31.03.2020, Data Science (SpSe 2022): T4

Prof. Dr. Claudius Gräbner-Radkowitsch
Europa-University Flensburg, Department of Pluralist Economics
www.claudius-graebner.com | @ClaudiusGraebner | claudius@claudius-graebner.com

Europa-Universität Flensburg
International Institute of Management and Economic Education
Department of Pluralist Economics

Prologue:

Prologue
Feedback and exercises

- XX of you filled out the feedback survey. Main take-aways:
    - TBA
- What were the main problems with the exercises?

Goals for today

I. Understand the main object types in R and their practical relevance

II. Learn how to transform object types into each other

III. Hear about some useful helper functions and the concept of vectorisation

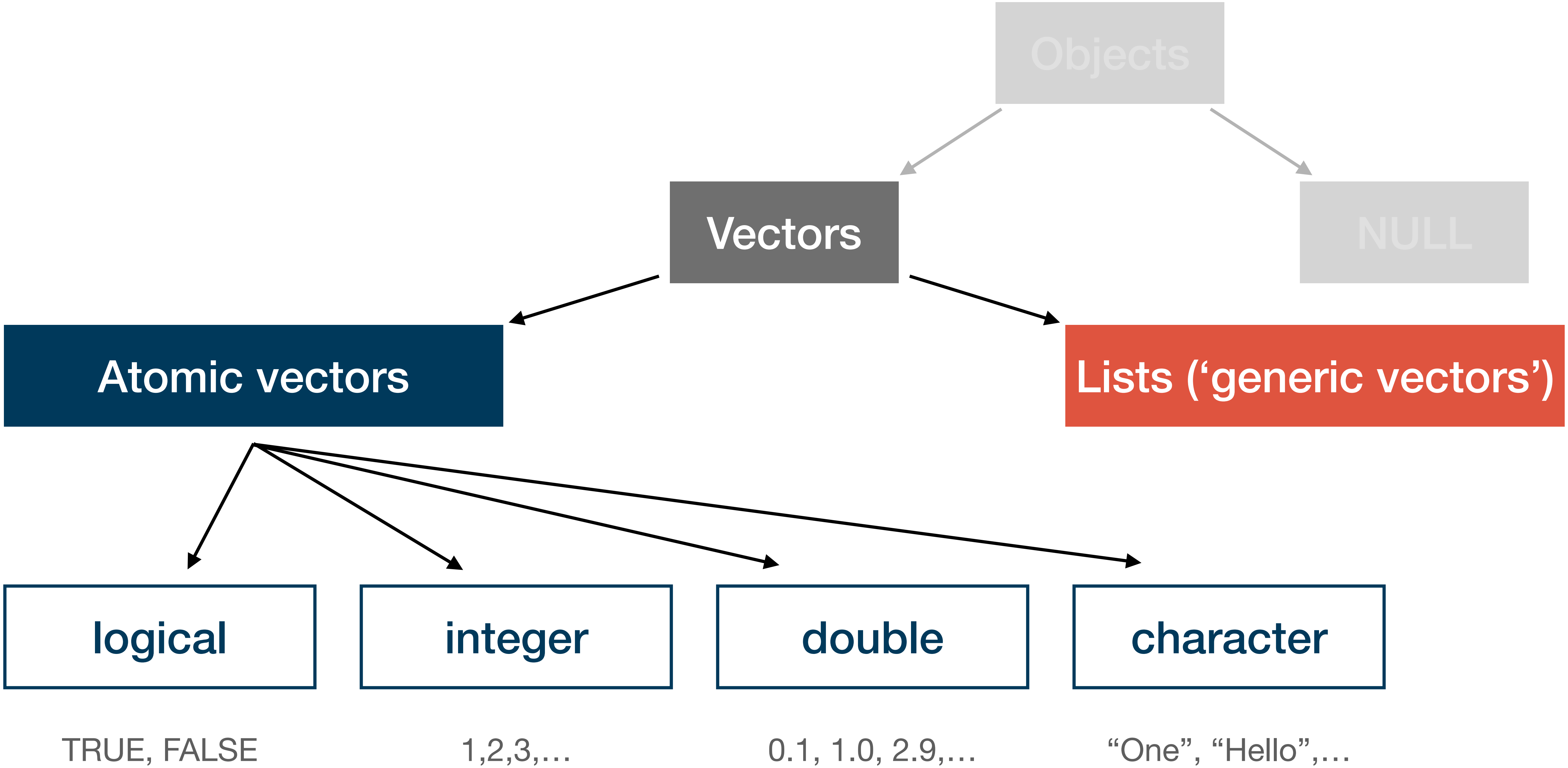
Basic object types in R

Object types in R

> “To understand computations in R, two slogans are helpful:
> Everything that exists is an **object**.
> Everything that happens is a function call.
>
> John Chambers

- We have learned quite a bit about functions, now we turn to objects
- We must distinguish different object types because functions operate differently depending on the type of the object we are processing
    - E.g.: ‘adding up’ numbers is different than ‘adding up’ words
- Fortunately, there are only a few basic types you must know about
    - More complex types are natural modifications of these basic types
- The most general type of object in R is a **vector**

Basic object types in R



- Among the more specific vector types, we will learn about `factors` and `data frames` later

Atomic vectors

- Atomic vectors are composed only of objects of the same type
  - We say that an atomic vector is of the same type as are its elements
  - We can test for this type using the function `typeof()`
- There are four main types of atomic vector that are most important:

Logical values: logical

- Only two* options: `TRUE` or `FALSE`
- Often the result of logical operations (e.g. `4>2`)

Whole numbers: integer

- A whole number, followed by L:
- `1L, 2L, 100L`, etc.
- Often the result of counting

Decimal numbers: double

- A number with the decimal sign `.`
- `2.0, 0.8, -7.5`, etc.
- The 'standard' number you will use

Letters and words: character

- Might contain all kinds of tokens and start and end with `"`
- `"2", "Hello!", “vec_1"`, etc.

*: We will see later that missing values are also considered logical in some instances, but this is basically irrelevant now.

Creating atomic vectors

- The easiest way to create atomic vectors is the function `c()` (‘concatenate’)

```
t_vec <- c(1, 2, 3)
```

- The number of elements that are part of a vector are its length:
  - You can test for the length of a vector using `length()`:

```
length(t_vec)
```

- `c()` can also be used to merge atomic vectors or arbitrary length:

```
t_vec_2 <- c(4, 5, 6)

t_vec_full <- c(t_vec, t_vec_2)
```

Coercion

- Sometimes we might want to change the type of an atomic vector
- In this context, the functions `as.*()` and `is.*()` are useful
  - Substitute the `*` for the type of vector, and you can test and transform them:

```
xx <- “2”

is.double(xx)

yy <- as.double(xx)

is.double(yy)
```

- But be beware of some counter-intuitive transformation behaviour:
  - `as.integer(22.9)`
  - `as.logical(99)`

Intermediate exercises

Do the following tasks with you neighbour(s) and discuss open questions!

1. Create a vector containing the numbers 2, 5, 2.4 and 11.
2. What is the type of this vector?
3. Transform this vector into the type `integer`. What happens?
4. Do you think you can create a vector containing the following elements: `"2"`, `"Hallo"`, `4.0`, and `TRUE`? Why? Why not?

Some useful helper functions

- There are some types of atomic vectors that you create frequently
  - Sequences of numbers, concatenated words, or repetitions
- For case 1 you may use the function `seq()` with the following arguments:
  - `from, to`: starting and end values of the sequence
  - `by`: increment steps of the sequences (must be numeric)
  - `length.out`: desired length of final sequence
  - `along.with`: creates sequence of same length as object
- Only one of the arguments (ii), (iii), and (iv) can be used, e.g.:
  - `seq(-5, 5, by=2.5) ; seq(1, 4, length.out=10)`

Some useful helper functions

- There are some types of atomic vectors that you create frequently
  - Sequences of numbers, concatenated words, or repetitions
- For case 2 you may use the function `paste()` with the argument `sep`:
  - `sep:` How should the input vectors be separated?
- This is useful, for instance, if you want to create file names:

```
paste("file_", seq(1,4), ".pdf", sep = “")
```

- Finally, if you want to repeat something, use `rep()`:

```
rep("Cool!", 5)
```

Indexing

- Indexing means referencing a particular position of a vector
  - You do this by adding the position in square brackets to the end of the vector
  - `v_c[3]`, for instance, returns the third element of the vector `v_c`
  - You can also use this logic to replace these elements:

```
v_c <- c("First", "Second", "Second", "Fourth")
v_c[3] <- "Third!"
```

- But you cannot use this to add new elements to a vector:

```
v_c[5] <- "Fifth…"
```

- Add a fifth element to the vector `v_c`!

Vectorisation

- One reason why atomic vectors are so popular is that they allow for very fast computations
    - For the computer it is much easier to work with sets of objects that all behave the same
- **Vectorisation** means that an operation is applied to each element of a vector:

```
v_2 <- seq(1, 5)

v_2**2
```

- "**To vectorise**" a task means to write it in a way that operations are applied to atomic vectors → in R, you should do that whenever possible
    - A slower alternative are **loops**, which we learn about later and which are unavoidable in certain situations

Intermediate practice

Do the following tasks with you neighbour(s) and discuss open questions!

I. Create a vector with the numbers from -2 to 19 (step size: 0.75)

II. Create an index vector for this first vector (note: an index vector is a vector with all possible indices of the original vector)

III. Compute the log of each element of the first vector using vectorisation. Anything that draws your attention?

IV. What happens if you concatenate vectors of different types using `c()`? Can you derive a systematization?

- Remember that you can check for the type of an atomic vector using `typeof()`

Lists

- The second major type of vectors → sometimes called generic vectors
- Difference to atomic vectors: lists may contain objects of different types
  - Thus, the type of a list is always…

```
l_1 <- list(c(1,2), c("a", "b"), c(TRUE, FALSE, FALSE)); typeof(l_1)
```

- Lists can be complex → get an overview using `str()`:

Types of the elements

Number of list elements

```
> str(l_1)
List of 3
 $ : num [1:2] 1 2
 $ : chr [1:2] "a" "b"
 $ : logi [1:3] TRUE FALSE FALSE
```

Length of the elements

Preview of the elements

Naming and indexing of lists

- The different elements of lists can be named:

```
l_2 <- list("numbers"=c(1,2),
            "letters"=c("a", "b"),
            "logics"=c(TRUE, FALSE, FALSE))
```

- You can retrieve the names using `names()`:

```
names(l_2)
```

- You can subset the list using the names:

```
l_2["letters"]
```

- And access the elements of the sublists with `[[`:

```
l_2[["letters"]]
```

  - Alternatively use the shortcut `$`: `l_2$letters`

Practical differences to atomic vectors

- There are two very important differences to atomic vectors:
  - Vectorisation does not work for lists
  - Indexing works differently for lists
- To illustrate the first issue compare:

```
v_ <- c(1, 2, 3); 2*v_
l_ <- list(1, 2, 3); 2*l_
```

- To illustrate the latter:

```
typeof(l_[1])
typeof(l_[[1]])
```

- Lists are fundamental to more complex data structures we will encounter later

Final remarks on basic object types

- There are two “strange” data types: `NA` and `NULL`
- `NA` is used to represent absent elements of vectors
  - Happens frequently when vectors contain observations
  - Many functions behave differently when NAs are present (remember `na.rm`!):

```
mean(c(1,2,NA)) ; mean(c(1,2,NA), na.rm = TRUE)
```

- You test for `NA` using `is.na()`:

```
is.na(c(1, 2, NA))
```

- To check whether a vector contains missing values, use anyNA():

```
anyNA(c(1,2,NA))
```

Final remarks on basic object types

- There are two “strange” data types: `NA` and `NULL`
- `NULL` is in fact a data type in itself, but in practice its best thought of as a vector of length zero:

```
c()

typeof(NULL)

length(NULL)

is.null(NULL)
```

- You might use `NULL` mainly in two instances:
  - Represent an empty vector of arbitrary type
  - Represent and absent vector ($\neq$ `NA`, which represents absent elements of vectors)

Wrap up basic object types

Objects

Vectors

NULL

Atomic vectors

Lists (‘generic vectors’)

logical

integer

double

character

- The central take-aways concern:
    - How to test for and transform these types: `typeof()`, `is.*()`, `as.*()`
    - How to index them: `[`, `[[`, `$`
    - How to create typical instances: `rep()`, `paste()`, `seq()`
- We learned about vectorisation and its attractiveness in R
- We also encountered “strange” types such as `NA`, `NULL` and `NaN`

Break

Get together in groups of 2 to 5 people and talk about what were your main take aways, and resolve open questions!

Advanced object types in R

On more advanced object types

- While there are many object types in R, understanding the basics is key
  - These are by far the most common ones
  - All other object types are somehow 'built upon' the basic types by adding **attributes**
- Among the special types, two stand out in their prevalence:

Categorical data: factor

- Can also take a pre-specified number of values: levels
- Classical example: Male, Female, Diverse
- Created using the function factor()

Data frames: data.frame & tibble

- A kind of 'table' in which different variables are stored as vectors
- A table-like form of
- Tibbles as a new va
  that "do less and c
- Created using `data`
  `tibble::tibble(`

	gender	height
1	male	189
2	male	175
3	male	180
4	female	166
5	female	150

- Others that we will not cover here are, e.g., matrices, durations, or dates

Digression: some remarks on attributes

- To turn our basic object types into something more fancy we can give them **attributes**, one of which is called **class**
  - This changes their behaviour when functions are applied to them
  - Technically, adding a class attribute changes the class but not the type:

```
ff <- factor(c("F", "M", "M"), levels = c("F", "M", "D"))

typeof(ff)

class(ff)
```

- The class `factor` is an integer with two attributes:

```
attributes(ff)
```

- Not too important for us right now, but good to keep in mind!

Factors

- Factors are used to represent ordinal or categorial data
  - Elements of factors can take one out of several pre-specified values: levels
  - Factors are integers with the attributes `levels` and `class`
- We create factors using the function `factor()`, which takes a vector and an optional argument levels:

```
f_1 <- factor(c(rep("F", 4), rep("D", 5), rep("M", 3)),
              levels = c("D", "F", "M"))
```

- Your turn:
  - What happens if we do not specify `levels` explicitly?
  - What happens if the vector contains elements not pre-specified as levels?

Factors

- Usually levels are not ordered, but for ordinal data you can use the argument ordered:

```
f_2 <- factor(c("high", "high", "low"),
              levels = c("low", "mid", "high"),
              ordered=TRUE)
```

- There are some useful factor-specific functions such as `table()`.
  - What does it do? Try it on `f_1` and `f_2`!
- In general, its usually better to store categorial data as character, and only transform them to factors if necessary

Data frames

- Data frames are special lists of vectors where the length of each vector is equal!
  → Most list operations also work for data.frames
- We usually represent data frames as tables:

```
  gender height   Names of the vectors
1   male    189
2   male    175
3   male    180
4 female    166
5 female    150
```

vector 1 & vector 2

- To create a data frame from scratch use `data.frame()`:

```
df_1 <- data.frame(
  "gender" = c(rep("male", 3), rep("female", 2)),
  "height" = c(189, 175, 180, 166, 150)
)
```

- To create a data frame from a list use `as.data.frame()`

- If you read in data into R, it almost always starts off as a data.frame
- How to transform them is the main subject of the sessions on **data wrangling**

Data frames and tibbles

- A modern version of the data.frame is the `tibble` (from the package *tibble*)
    - We will mostly use `tibbles` in this course, but make sure you familiarise yourself with the differences to the `data.frame`, which continues to be widespread (see the tutorial reading)
- To transform a `data.frame` (or a `list`) into a `tibble`, use `tibble::as_tibble()`:

```
tb_1 <- tibble::as_tibble(df_1)
```

- To extract single columns use the `[` or `[[` operators
    - What's the difference between the two?
    - How do you think you cam test for the type of a column vector?

Data frames and tibbles

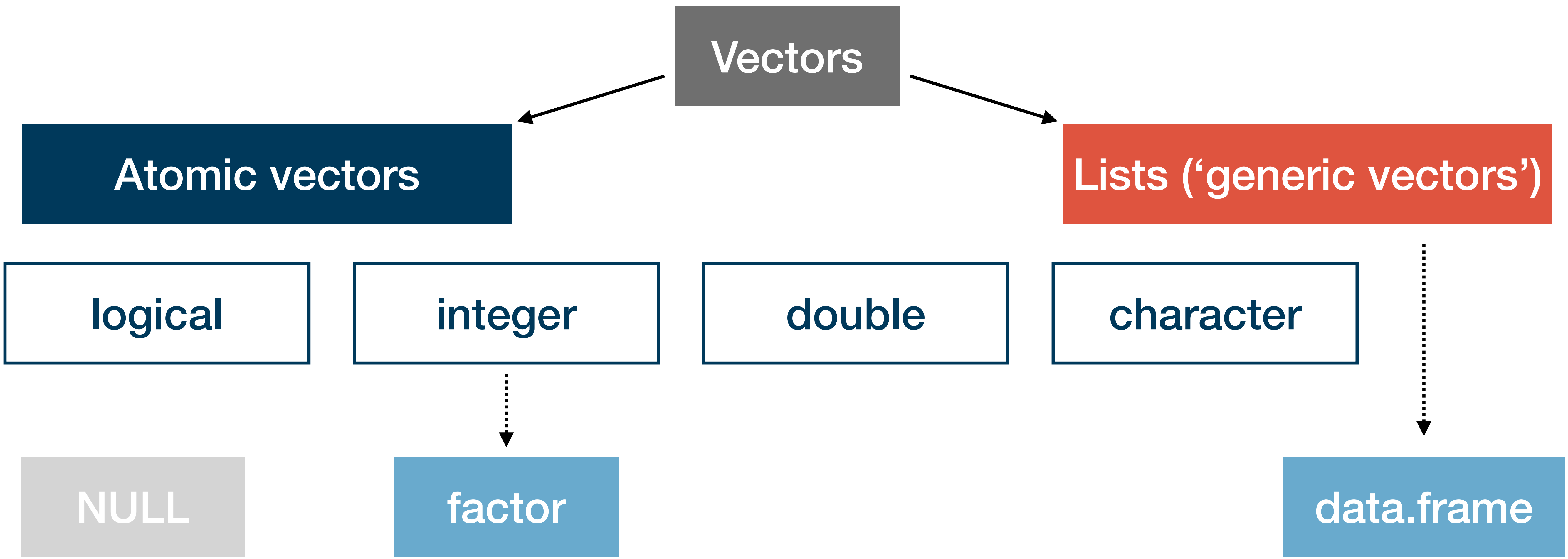
- To get a quick overview about the content, use `dplyr::glimpse()` or `head()`
- A complete overview can be obtained via `View()`
- Data frames are among the most widely used data types
  - There different approaches of how to handle and transform them, each associated with an R **dialect**
  - We mainly rely on the **tidyverse** dialect, which is the easiest to learn and comprehend → built upon **tibbles**
  - Alternatives are the **base** (classical) and **data.table** (fastest) dialect, which mainly use **data.frames** and **data.tables**
- This is useful to keep in mind when searching help in the internet

Quick exercises

- Create a factor with the levels “still”, “medium” and “sparkling”, and arbitrary instances of the three levels
- Transform this factor into a character vector
- Get the relative frequencies for “medium” of this factor
- Create a data frame with two columns, one called “nb” containing the numbers 1 to 5 as double, the other called “char” containing the numbers 6 to 10 as character
- Transform this data.frame into a tibble!
- Extract the second column of this tibble such that you have a vector

Summary and outlook

- This was the last session on the fundamentals of R
- We learned about the most important object types in R
- Functions do different things when applied to different objects → understanding object types is absolutely fundamental



Summary and outlook

- Next week we will learn how to visualise data that is stored in data frames
- This will be the first big intro session into data science fundamentals, succeeded by sessions on data wrangling and project management

Tasks until next week:

1. Fill in the **quick feedback survey** on Moodle
2. Read the **tutorials** posted on the course page
3. Do the **exercises** provided on the course page and **discuss problems** and difficulties via the Moodle forum