

Packages & object types

Theoretical and Empirical Research Methodology,
Implementation Lab 4 - Part I

Prof. Dr. Claudius Gräbner-Radkowitz

Europa-University Flensburg, Department of Pluralist Economics

www.claudius-graebner.com | [@ClaudiusGraebner](https://twitter.com/ClaudiusGraebner) | claudius@claudius-graebner.com

Goals for today

- I. Learn about the use of R packages
- II. Understand the main object types in R and their practical relevance
- III. Learn how to transform object types into each other
- IV. Hear about some useful helper functions and the concept of vectorisation

Packages

R packages

- One cool thing about R is that there is a great community of R users that write objects and functions that perform useful purposes and makes them available to all
 - This process of ‘making available objects to others’ is done via the use of **R packages**
- You can think of an R package as a collection of assignments and documentations that people pass around
- If you install R, you can use all objects that...
 - ...you defined for yourself
 - ...are pre-defined in R
- If you want to use objects defined by someone else in her package you need to install this package

Installing packages

- The official way to distribute packages is via CRAN, the *The Comprehensive R Archive Network*
- To install a package that was deployed on CRAN you must execute the following command:

```
install.packages("NAME OF PACKAGE")
```

- To install the package `ineq`, for instance, do:

```
install.packages("ineq")
```

- To install packages that were not yet released on CRAN, other functions are available
- After having installed the `ineq` package, you can use all objects defined by it

Calling objects defined in packages

- One object defined in `ineq` is the function `Gini()`
 - Simply calling `Gini()` does, however, not work
- You need to tell R that `Gini()` is defined by the package `ineq`
- To do use, use `::`

```
ineq::Gini(c(1,2,3,4))
```

 - You may think of `::` as building a bridge between your R session and all objects defined in a package
- A sometimes more convenient way is to use the function `library()` at the beginning of your script:

```
library(ineq)
```
- This makes available all objects of `ineq` in your current R session

Packages and masking

- Packages are written by many different people
- It is not unlikely that two packages assign the same name to different objects
- If you then attach both packages, the assignment of the earlier package will be masked
 - Try this by attaching the two packages `dp1yr` and `p1m`
 - In these cases, you must use `::` to access the masked object of the first package
- As a general rule: always use `::` whenever masking is a potential problem → makes your code much easier to understand for you and others
 - Use the function `conflicts()` to see all names for which conflicts exists

Recap questions

- What is the main rationale for the use of R packages?
- What is an R package in the most basic sense?
- How can you install R packages from CRAN? Illustrate this using the package “dplyr”
- How can you access objects from a package that you have installed previously? What are the advantages and disadvantages of the different ways you learned about?
- What does it mean to ‘attach a package’?
- What do we mean with ‘masking’ in the context of using R packages?

Basic object types in R

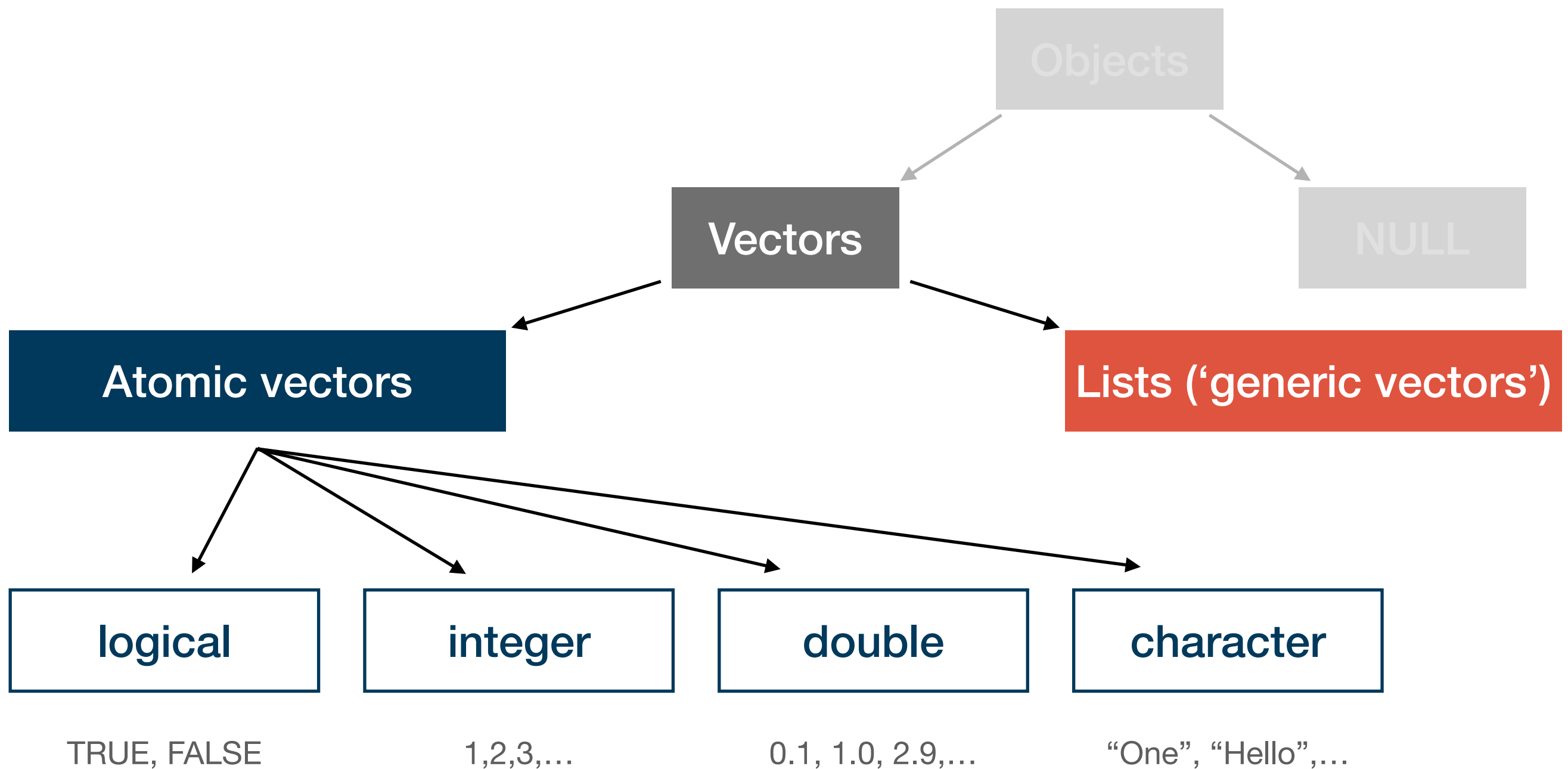
Object types in R

“ 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

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?

Helper functions, indexing, and vectorisation

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)
```

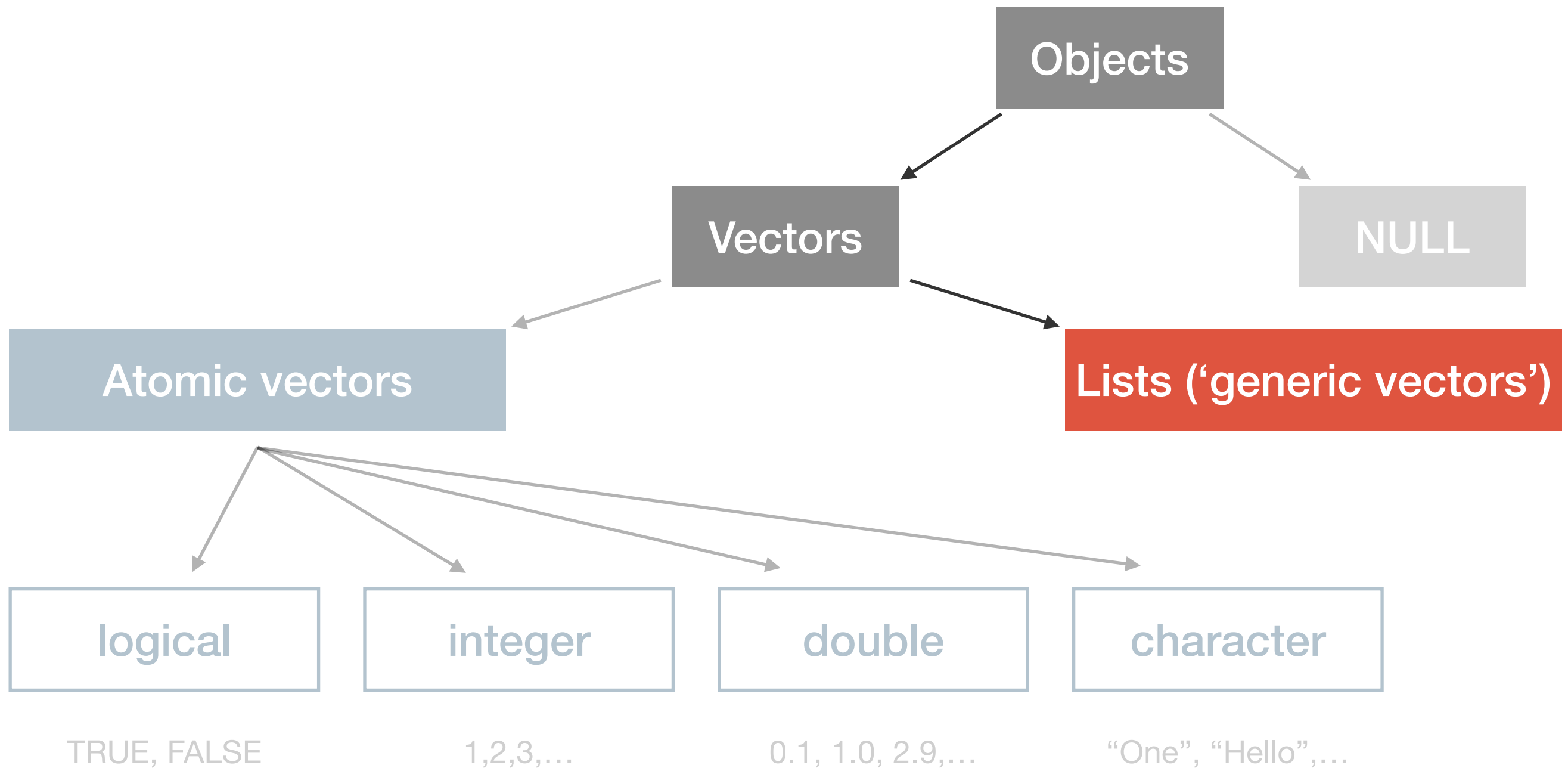
```
v_2**2
```
- “**To vectorise**” a task means to write it in a way that operations are applied to atomic vectors → in R, you should do that whenever possible
 - A slower alternative are **loops**, which we learn about later and which are unavoidable in certain situations

Intermediate exercises

- 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

Basic object types in R



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

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

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

Number of list elements

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))`

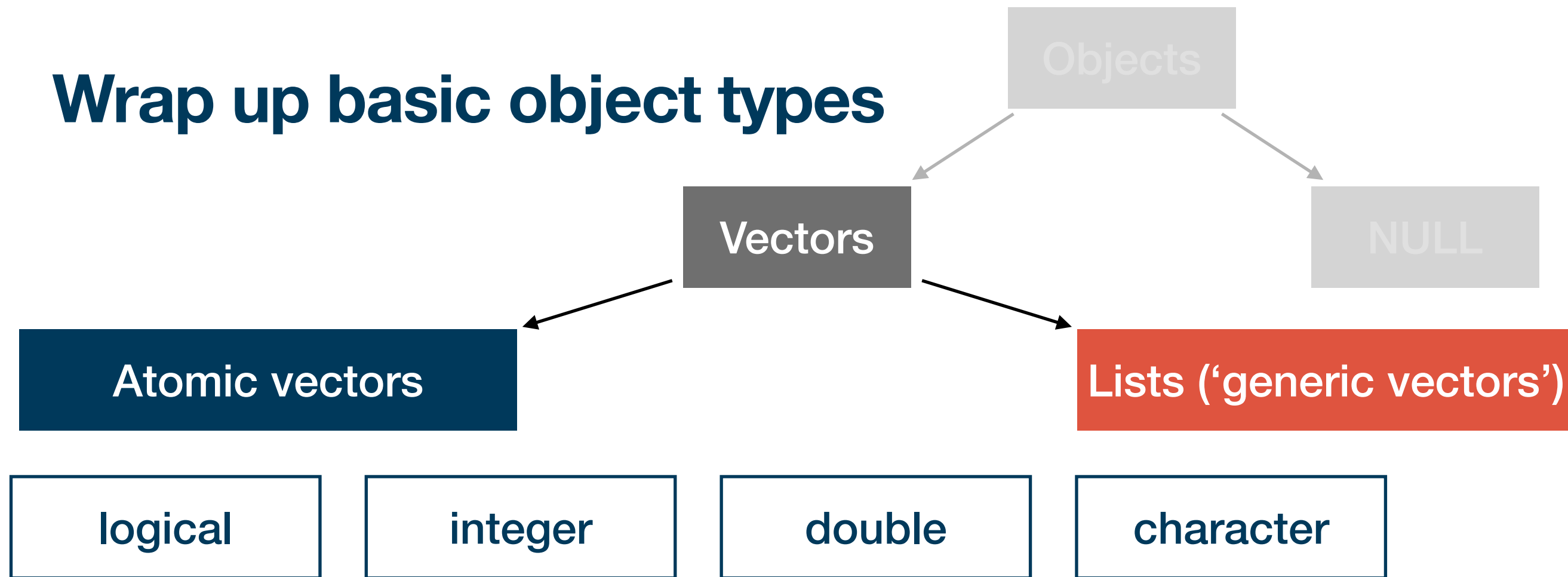
Final remarks on basic object types

- 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 an absent vector (\neq **NA**, which represents absent elements of vectors)

Intermediate exercises

- Create a list that has three named elements: "A", "B", and "C"
- The element "A" should contain the square root of the numbers from -2 to 8 (step size: 1)
- The element "B" should contain the log of numbers between 2 and 4 (step size: 0.5)
- The element "C" should contain letters from a1 to g7 (hint: use the pre-defined vector letters and the function `paste()`)

Wrap up basic object types



- 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`

Summary and outlook

- Next time we will learn about two more advanced object types: **factors** and **data.frames**
- We will learn how our knowledge about the basic object types helps us to deal with more advanced types, and how they relate to each other

Tasks until next session:

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