Lecture 9: Vectors & Factors in R

Economía Laboral

Junghanss, Juan Cruz

Universidad del CEMA

2nd Semester 2024

Contents

Today's lecture content:

- Vectors Basics
- Factors Basics
- TP Nº3 R exercise

Vectors - The basics

So far we've focused on tibbles and packages that work with them. But as you start to write your own functions, and dig deeper into R, you need to learn about vectors, the objects that underlie tibbles.

Vectors - The basics

There are two types of vectors:

1. *Atomic*: just vectors as you know them. It refers to the vector types of logical, integer, double, character, complex and raw.
2. *Lists*: as lists can contain other lists, they are called recursive vectors.

The difference between the two is that atomic vectors are homogeneous (only one type) and lists can be heterogeneous (multiple types).

Vectors - Hierarchy

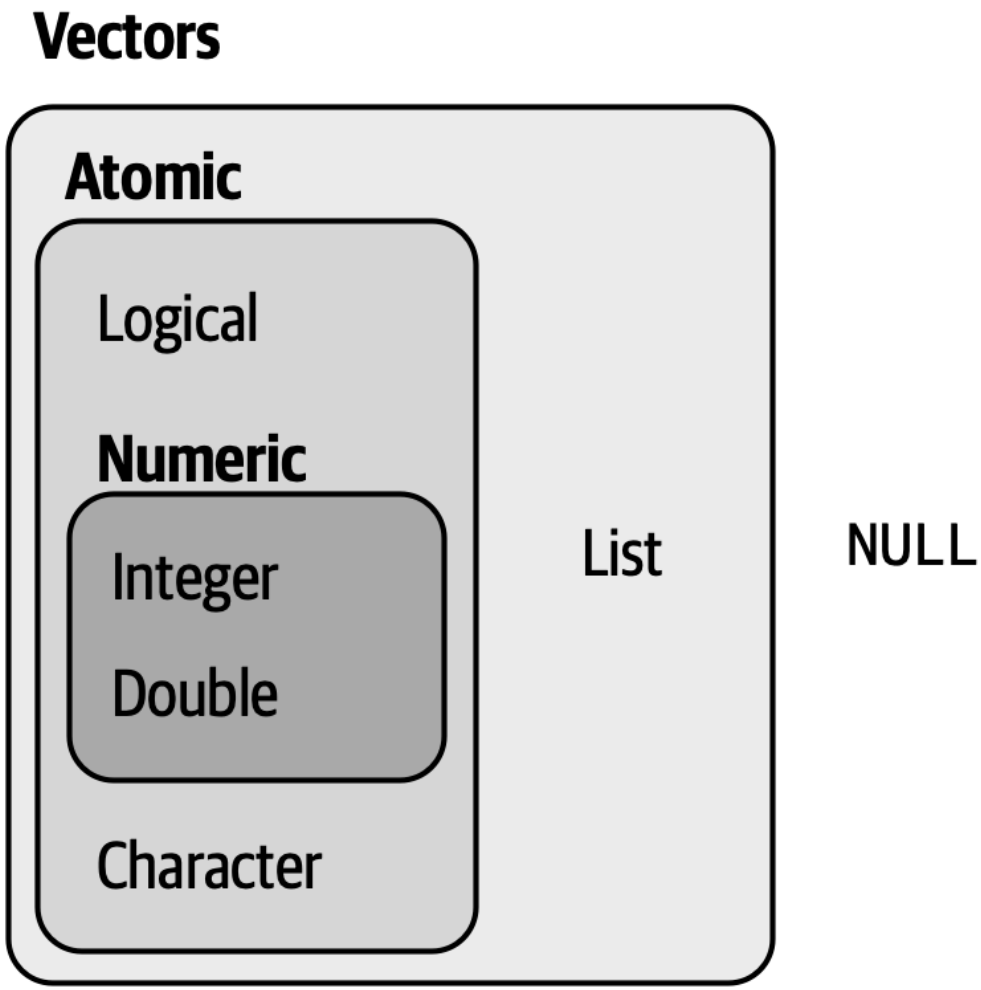


Figure: Hierarchy of R's vector types. Source: R for Data Science

Vectors - The basics

Key properties of vectors:

- Type: you determine it with typeof() function.
- Length: you can determine it with length() function.

Vectors can also contain arbitrary additional metadata in the form of attributes, but we won't explore that.

Logical Vectors

Logical vectors are the simplest type of atomic vector because they can take only three possible values

- FALSE
- TRUE
- NA

Numeric Vectors

Integer and double vectors are known collectively as numeric vectors.

- In R, numbers are doubles by default. They are approximations. Doubles represent floating-point numbers that cannot always be precisely represented with a fixed amount of memory.
- To make an integer, place a L after the number.

Example

```
typeof(1)
#> [1] "double"

typeof(1L)
#> [1] "integer"
```

Numeric Vectors

Integers have one special value, NA, while doubles have four:

- NA
- NaN
- Inf
- -Inf

All three special values can arise during division:

Example

```
c(-1, 0, 1) / 0
#> [1] -Inf NaN Inf
```

Numeric Vectors

Avoid using == to check for these other special values. Instead use the helper functions is.finite(), is.infinite(), and is.nan():

	0	Inf	NA	NaN
is.finite()	x			
is.infinite()		x		
is.na()			x	x
is.nan()				x

Appendix: NA and NaN values

But.. what's the difference between NA and NaN values?

Appendix: NA and NaN values

But.. what's the difference between NA and NaN values?

- A **NAN** value in R represents "NOT A NUMBER". It's basically any numeric calculations with an undefined result, such as '0/0'. This exists only in *numeric vectors*.
- A **NA** value in R represents "NOT AVAILABLE". This can exist in any sort of *numeric or character vectors*.

Character Vectors

Character vectors are the most complex type of atomic vector, because each element of a character vector is a string, and a string can contain an arbitrary amount of data.

Atomic Vectors Methods

Now that you understand the different types of atomic vector, it's useful to review some of the important tools for working with them.
These include:

1. How to convert from one type to another, and when that happens automatically.
2. How to tell if an object is a specific type of vector.
3. What happens when you work with vectors of different lengths.
4. How to name the elements of a vector.
5. How to pull out elements of interest.

Atomic Vectors Methods - Conversion

1. How to convert from one type to another, and when that happens automatically.

The word **conversion** refers to either implicitly or explicitly changing a value from one data type to another, e.g. a string type to an integer.

- **Cast** typically refers to an *explicit* conversion (similar to **parse** functions). It happens when you call a function like as.logical(), as.integer(), as.double(), or as.character().
- **Coercion** is used to denote an *implicit* conversion. It happens when you use a vector in a specific context that expects a certain type of vector.

Atomic Vectors Methods - Test Functions

2. How to tell if an object is a specific type of vector.

Sometimes you want to do different things based on the type of vector. One option is to use typeof(). Another is to use a test function that returns a TRUE or FALSE. Base R provides many functions like is.vector() and is.atomic(), but they often return surprising results. Instead, it's safer to use the is_* functions provided by purrr, which are summarized in the following table:

	lgl	int	dbl	chr	list
is_logical()	x				
is_integer()		x			
is_double()			x		
is_numeric()		x	x		
is_character()				x	
is_atomic()	x	x	x	x	
is_list()					x
is_vector()	x	x	x	x	x

Atomic Vectors Methods - Scalars and Recycling Rules

3. What happens when you work with vectors of different lengths.

As well as implicitly coercing the types of vectors to be compatible, R will also implicitly coerce the length of vectors. This is called vector **recycling**, because the shorter vector is repeated, or recycled, to the same length as the longer vector. This is generally useful when you mix vectors and scalars:

Example

```
sample(10) + 100
#> [1] 109 108 104 102 103 110 106 107 105 101

1:10 + 1:2
#> [1] 2 4 4 6 6 8 8 10 10 12
```

Atomic Vectors Methods - Scalars and Recycling Rules

However, the recycling rule doesn't work in tidyverse functions if you use other than scalars:

Example

```
tibble(x = 1:4, y = 1:2)
#> Error: Variables must be length 1 or 4.
#> Problem variables: 'y'
```

Appendix: Differences with Python

Similarly, in Python you have the "**Broadcasting**" rule, but it's a little bit different in respect to the Recycling rule in R.

- The R code uses the recycling rule, which says that if a vector is too short, it will be repeated as many times as needed to match the other operands.
- The Python code uses the numpy broadcasting rules which describe what happens if an operation involves numpy arrays of different shapes. They add leading dimensions and adjust size 1 dimensions as needed.

These rules are not the same. See
https://numpy.org/doc/stable/user/basics.broadcasting.html

Atomic Vectors Methods - Naming Vectors

4. How to name the elements of a vector.

All types of vectors can be named. You can name them during creation with c():

Example

```
c(x = 1, y = 2, z = 4)
#> x y z
#> 1 2 4
```

Or after the fact with purrr::set_names():

Example

```
set_names(1:3, c("a", "b", "c"))
#> a b c
#> 1 2 3
```

Atomic Vectors Methods - Subsetting

5. How to pull out elements of interest.

So far we've used dplyr::filter() to filter the rows in a tibble. filter() only works with tibble, so we'll need a new tool for vectors: [.
[is the subsetting function, and is called like x[a]. There are four types of things that you can subset a vector with:

- A numeric vector containing only integers. The integers must either be all positive, all negative, or zero.
- Subsetting with a logical vector keeps all values corresponding to a TRUE value
- If you have a named vector, you can subset it with a character vector
- The simplest type of subsetting is nothing, x[], which returns the complete x. This is not useful for subsetting vectors, but it is useful when subsetting matrices. x[1,] selects the first row and all the columns, and x[, -1] selects all rows and all columns except the first.

Recursive Vectors or Lists - The basics

Lists are a step up in complexity from atomic vectors, because lists can contain other lists. This makes them suitable for representing hierarchical or tree-like structures.They can be created with list() function.

Example

```
x <- list(1, 2, 3)
x
#> [[1]]
#> [1] 1
#>
#> [1] 2
#>
#> [[3]]
#> [1] 3
```

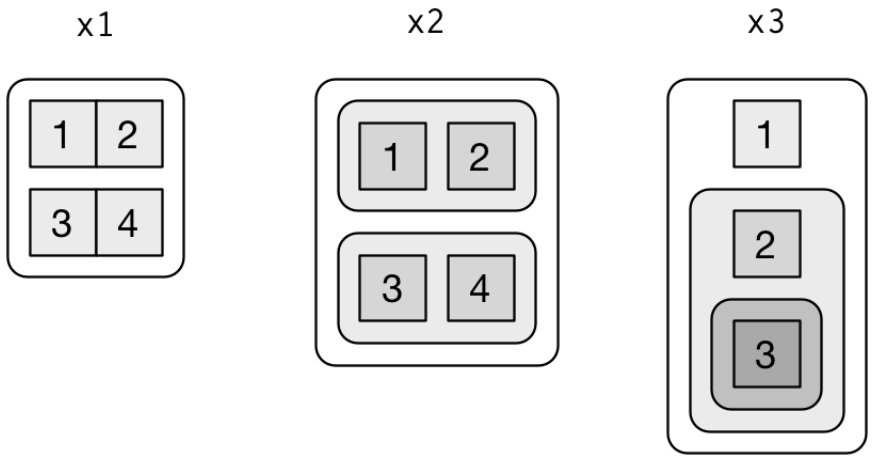
Recursive Vectors or Lists - The basics

A list can contain a list: list(1, list(2,3))
A list can contain a list that contains a list: list(1, list(2, list(3)))

For example, let's visualize the following lists:

Example

```
x1 <- list(c(1, 2), c(3, 4))
x2 <- list(list(1, 2), list(3, 4))
x3 <- list(1, list(2, list(3)))
```



Recursive Vectors or Lists - Subsetting

But... how can we access to the element of the list that's contained by a list that's contained by another list? There's a level of *hierarchy*.

Subsetting Rules in lists:

1. [extracts a sublist. The result will always be a list, even with a single element.
2. [[extracts the single element of the list by removing the level of hierarchy. It's like working with $ just that you don't use quotes.
3. $ is a shorthand for extracting named elements of a list. It works similarly to [[except that you don't need to use quotes

The distinction between [and [[is really important for lists, because [[drills down into the list while [returns a new, smaller list.

Recursive Vectors or Lists - Subsetting

Let's see an interesting example from the book to gain understanding.

Consideren el siguiente pimentero como una lista x:

Recursive Vectors or Lists - Subsetting

Si x es la lista, entonces x[1] debería ser el pimentero conteniendo un solo sobre de pimienta:

Ergo, x[2] se vería idéntico, pero conteniendo el segundo paquete, al igual que x[1:2] conteniendo dos paquetes.

Recursive Vectors or Lists - Subsetting

x[[1]] como elemento individual sería un sobre de pimienta:

Pero, el sobre no es a su vez otra lista? Cómo llegamos al contenido de este?

Recursive Vectors or Lists - Subsetting

Por ende, x[[1]][[1]] sería:

Es el elemento individual del elemento individual de una lista madre.

Factors

Factors are used to work with categorical ("dummy") variables, i.e. that have a fixed and known set of values (strings, integers, etc).

However, consider for example you have a dummy variable for working days: days ¡- c("Lun", "Mar", "Mie", "Jue", "Vie")
Those are just strings, but working with them might be quite complicated.
We can encounter two problems:

- Typos, i.e. errors can ruin your life. (For example: "Martes" vs. "Mrates")
- It doesn't sort in a useful way. (For example: alphabetical sorting would imply "Jue", "Lun", ...)

That's when "factors" come into scene.

Factors with forcats

The package to work with factors is called **forcats** and it's not part of the core tidyverse, so you have to load it explicitly.

Most used functions are:

- factor(array, levels) to create a factor with an array.
- levels() to access and see the set of levels of the factor.
- fct_inorder() to force the levels order of the array as it is.
- fct_reorder(factor, reordering sequence) to reorder all levels.
- fct_relevel(factor, variable) to reorder a specific level.
- fct_infreq() to order levels in increasing frequency.
- fct_recode() to change the value of the level (not the order).

Factors with forcats

Main functions for factors:

1. Creation
2. Factor Order Modification
3. Factor Level Modification

Factors Creation

To create factors we can use the factor() function, which takes a string vector (with the dummy variables) and optionally the levels vector (another string input indicating the levels). If you omit the levels, they'll be taken from the data in alphabetical order and if you want the order of the levels to match the order of first appearance in the data yuu can use:

Example

```
variable <- factor(data, levels=unique(data))



variable <- data %>% factor() %>% fct_inorder()
```

Aditionally you can check your factor variable condition with

- is.factor()
- is.ordered()

Order Modification

It's often useful to change the order of the factor levels in a visualization. For example: ascending or descending order.

fct_reorder() takes three arguments:

- f, the factor whose levels you want to modify.
- x, a numeric vector that you want to use to reorder the levels.
- Optionally, fun, a function that's used if there are multiple values of x for each value of f. The default value is median.

Alternatively, you can use:

- fct_relevel()
- fct_reorder2()

Levels Modification

More powerful than changing the orders of the levels is changing their values. This allows you to clarify labels for publication, and collapse levels for high-level displays. The most general and powerful tool is fct_recode(). It allows you to recode, or change, the value of each level.

It's like a "rename()" function but specifically for factor levels.
See code example in R file.

Factors: A comment

Toda la intuición de los factors es fácilmente vinculable con los datos de la EPH, pues sirven para formalizar la clase de cada variable categórica de la encuesta al formato que R tiene adaptado para este tipo de variables.