

# Data Structures: Factors, Lists, Matrices, Arrays

Enrico Toffalini  
**PSICOSTAT**

# Factors, Lists, Matrices, Arrays

As a data scientist, most of your tasks will probably require working with dataframes and vectors (see Part 1; remember that a dataframe is essentially a collection of vectors of different types)

However, other data structures that you will encounter are:

- **Factors**: store categorical data; factors are both a data type and a data structure
- **Lists**: collections of *objects of different types*, flexible and indexable
- **Matrices**: two-dimensional structures, essentially vectors organized into rows and columns, *all elements must be of the same type*
- **Arrays**: generalization of vectors and matrices to multi-dimensional data (e.g., 3D, 4D arrays), *all elements must be of the same type*

# Factors

**Factors** are a special type of data used to represent **categorical data**. They may look similar to simple *character vectors*. In fact, they function differently:

- Internally, they consist of vectors of integers associated with “levels”
- Levels are unique categories, labelled for readability

```
df$TypeOfCourse
```

```
[1] METHODOLOGY      METHODOLOGY      METHODOLOGY      PROGRAMMING  
[5] METHODOLOGY      METHODOLOGY      METHODOLOGY      METHODOLOGY  
[9] METHODOLOGY      SOFT SKILLS       PROGRAMMING      PROGRAMMING  
[13] SOFT SKILLS      THEMATIC COURSE  METHODOLOGY      METHODOLOGY  
[17] METHODOLOGY      METHODOLOGY      METHODOLOGY      METHODOLOGY  
[21] SOFT SKILLS      SOFT SKILLS       SOFT SKILLS      METHODOLOGY  
[25] METHODOLOGY      SOFT SKILLS       THEMATIC COURSE  PROGRAMMING  
Levels: METHODOLOGY PROGRAMMING SOFT SKILLS THEMATIC COURSE
```

*Note how the bottom row lists all existing levels*

# Factors

At any time, you can convert a vector (or a variable in a dataframe) into a factor using the `as.factor()` function

```
df$TypeOfCourse = as.factor(df$TypeOfCourse)
```

Internally, a factor is stored as **integer**, with associated **labels** for levels:

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

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

```
levels(df$TypeOfCourse)
```

```
[1] "METHODOLOGY"      "PROGRAMMING"       "SOFT SKILLS"        "THEMATIC COURSE"
```

*Warning!* Despite storing integers, factors are **not** numeric:

```
df$TypeOfCourse * 2
```

```
[1] NA  
[26] NA NA NA
```

# Factors

By default, **factors** in R are **non-ordered**, there is **no** hierarchy between their categories.

To create **ordered factors**, you can use the `as.ordered()` function.

```
as.ordered(df$TypeOfCourse)
```

```
[1] METHODOLOGY      METHODOLOGY      METHODOLOGY      PROGRAMMING  
[5] METHODOLOGY      METHODOLOGY      METHODOLOGY      METHODOLOGY  
[9] METHODOLOGY      SOFT SKILLS       PROGRAMMING      PROGRAMMING  
[13] SOFT SKILLS      THEMATIC COURSE  METHODOLOGY      METHODOLOGY  
[17] METHODOLOGY      METHODOLOGY      METHODOLOGY      METHODOLOGY  
[21] SOFT SKILLS      SOFT SKILLS       SOFT SKILLS      METHODOLOGY  
[25] METHODOLOGY      SOFT SKILLS       THEMATIC COURSE  PROGRAMMING  
Levels: METHODOLOGY < PROGRAMMING < SOFT SKILLS < THEMATIC COURSE
```

Ordered factors include a hierarchical relationship between levels (e.g., "low" < "medium" < "high"; or a Likert scale like "Strongly disagree" < "Disagree" < "Neutral" < "Agree" < "Strongly agree"). Using ordered factors may be especially important for certain data analysis, e.g., *Structural Equation Modeling (SEM)* with ordinal data (e.g., using the `lavaan` package)

# Factors

## Why use factors?

In many cases, you might ignore and avoid them. However:

- Help **ensure consistency** when data is actually categorical
- Many *functions for statistical modeling* (e.g., `lm()`) **automatically treat characters as factors**, assigning dummy variables for each level; also tools like `ggplot2` for *visualization* use factors for grouping or labeling axes
- Ensure **efficient storage** of information as compared to characters, thanks to their internal structure
- **Ordered data:** see previous slide

# Lists

Lists are flexible structure that contain **objects of different types and different lengths** (including other lists... potentially creating an infinite *Inception...*)

```
myChaos = list(TRUE, 0:5, df$Hours, letters[8:18], "PSICOSTAT")  
myChaos
```

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

```
[[2]]  
[1] 0 1 2 3 4 5
```

```
[[3]]  
[1] 10 15 20 10 15 5 5 5 5 10 10 5 5 10 10 15 20 5 15 5 10 5 5 10  
[26] 15 5 5
```

```
[[4]]  
[1] "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r"
```

```
[[5]]
```

# Lists

You can access elements of a list with indexing using the double square brackets `[[ ]]`

```
myChaos[[3]]
```

```
[1] 10 15 20 10 15 5 5 5 5 10 10 5 5 10 10 15 20 5 15 5 10 5 5 10  
[26] 15 5 5
```

```
myChaos[[5]]
```

```
[1] "PSICOSTAT"
```

A convenient function for inspecting the structure of a list is `str()`:

```
str(myChaos)
```

```
List of 5  
$ : logi TRUE  
$ : int [1:6] 0 1 2 3 4 5  
$ : num [1:28] 10 15 20 10 15 5 5 5 5 5 ...  
$ : chr [1:11] "h" "i" "j" "k" ...  
$ : chr "PSICOSTAT"
```

# Lists

If you **name** each element in the list, you can also access them using the **\$** operator, just like a dataframe

```
myLittleList = list(name = "Enrico",
                     sector = "m-psi/01",
                     hours = c(42,40,10,10),
                     school = c("psychology", "amv", "psychology", "psychology"))
```

```
myLittleList$sector
```

```
[1] "m-psi/01"
```

```
myLittleList$hours
```

```
[1] 42 40 10 10
```

*That's not surprising... a dataframe is actually a special kind of list! just two key constraints:  
1) all elements are vectors of the same length; 2) vectors are named.*

# Lists

## Why use lists?

- Provide **very flexible storage** (for example, in a complex Monte Carlo simulation you might want to store not just a single result from each iteration, but multiple objects, such as each simulated dataframe, or whole model outputs)
- **Common in R:** many functions (e.g., `lm()`) return their summaries and results as lists (even dataframes themselves are special cases of lists), so get familiar with them!
- Are used in many context for handling **nested data** (e.g., JSON-formatted data)

# Lists

## example with a power simulation

```
N = 30; b0 = 0; b1 = 0.3; sigma = 1

niter = 1000
results = list()

for(i in 1:niter){
  x = rnorm(N, 0, 1)
  y = b0 + b1*x + rnorm(N, 0, sigma)

  results[[i]] = lm(y ~ x)
}
```

This is an example of using a list in a power simulation. Typically, you store only one or a few values (e.g., p-values), but lists allow storing all fitted objects if needed.

# Matrices

In R, a **matrix** is a **2-dimensional** structure that contains only elements of the **same type**. Essentially, it can be thought of as a 2D vector.

You can create a matrix easily using the **matrix()** function:

```
( myMat = matrix(1:28, nrow=4, ncol=7) )
```

```
[,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    1     5     9    13    17    21    25
[2,]    2     6    10    14    18    22    26
[3,]    3     7    11    15    19    23    27
[4,]    4     8    12    16    20    24    28
```

```
( myMat = matrix(1:28, nrow=4, ncol=7, byrow=T) )
```

```
[,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    1     2     3     4     5     6     7
[2,]    8     9    10    11    12    13    14
[3,]   15    16    17    18    19    20    21
[4,]   22    23    24    25    26    27    28
```

# Matrices

Indexing in matrices is similar to dataframes, with indexes for row(s) and column(s), using [ <row(s) index> , <column(s) index> ]

```
myMat[2, 5] # access a single element
```

```
[1] 12
```

```
myMat[2:3, 5:7] # access ranges of elements
```

```
 [,1] [,2] [,3]  
[1,] 12 13 14  
[2,] 19 20 21
```

Like in vectors, you can perform **appropriate operations** on matrix data:

```
myMat^2 # element-wise squaring of all values
```

```
 [,1] [,2] [,3] [,4] [,5] [,6] [,7]  
[1,] 1 4 9 16 25 36 49  
[2,] 64 81 100 121 144 169 196  
[3,] 225 256 289 324 361 400 441  
[4,] 484 529 576 625 676 729 784
```

# Matrices

Operator	What it does	Example
<code>t()</code>	Transposes a matrix	<code>t(matrix(1:6,2))</code>
<code>%*%</code>	Matrix multiplication	<code>matrix(1:8,2) %*% matrix(1:8,4)</code>
<code>*</code>	Element-wise matrix multiplication	<code>matrix(1:8,2) * matrix(1:8,2)</code>
<code>det()</code>	Determinant of a square matrix	<code>det(matrix(rnorm(16),4))</code>
<code>solve(A, b)</code>	Solves $A^*x = b$	<code>solve(matrix(rnorm(16),4), rnorm(4))</code>

# Matrices

## Why use (know) matrices?

- **Mathematical operations:** matrices are fundamental for many tasks of linear algebra
- **Essential in modeling:** many statistical methods for statistical modeling and machine learning actually operate on matrices (even though this may remain hidden to you)
- **Computational efficiency:** much faster than dataframes for numeric computations

# Arrays

Arrays are multi-dimensional structures in R, generalizing *vectors* (*1-dimensional*) and *matrices* (*2-dimensional*) to the *n-dimensional* case

It's easy to create an array using the `array()` function:

```
myArr = array(1:30, dim = c(3,5,2))
myArr
, , 1
```

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

```
, , 2
```

```
[,1] [,2] [,3] [,4] [,5]
[1,]   16   19   22   25   28
[2,]   17   20   23   26   29
[3,]   18   21   24   27   30
```

→ this is kind of a “cubic-structure” (3D structure): 3 rows, 5 columns, 2 slices

In a similar way, you could create hypercubes and so on (4D+)

# Arrays

## indexing

Indexing is exactly the same as with matrices but... with 3 (sets of) indices!

```
myArr[1, 4, 2] # extract a single element
```

```
[1] 25
```

```
myArr[1:2 , 1:2 , ] # extract subsets of elements
```

```
, , 1
```

```
 [,1] [,2]  
[1,] 1 4  
[2,] 2 5
```

```
, , 2
```

```
 [,1] [,2]  
[1,] 16 19  
[2,] 17 20
```

# Arrays

## Why use (know) arrays?

- Might be useful for storing, and manipulate efficiently structure of **multi-dimensional data**
- Generally used in advanced topics and *machine learning* like when working on **image/video processing** and **spatial data**
- Arrays in R are conceptually similar to **tensors** in Python (e.g., [NumPy](#), [TensorFlow](#)), where they play a fundamental role in *machine learning* and *deep learning*, as they allow researchers to manage large amounts of data with complex structures