Data Structures (part 1)

Vectors

Enrico Toffalini

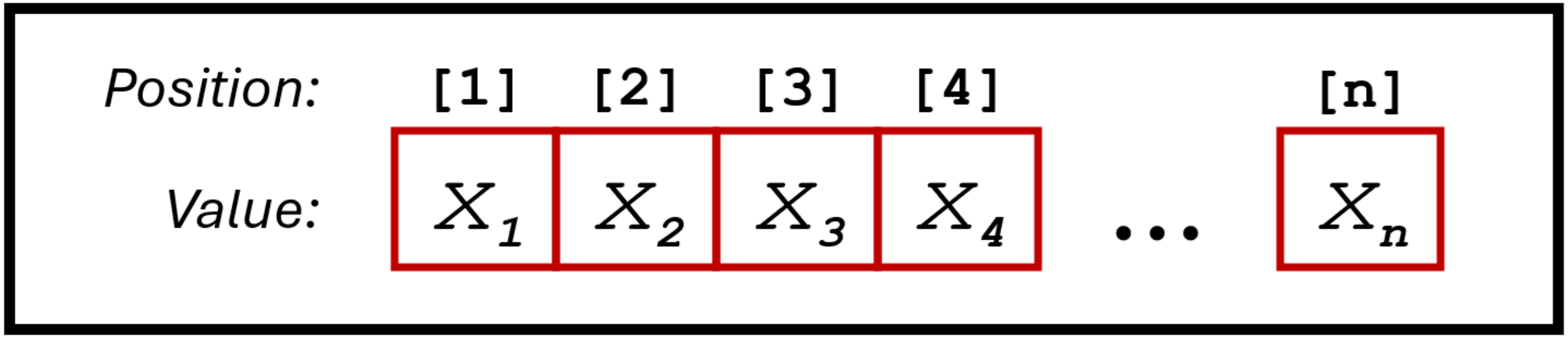
What Are Data Structures

Data structures, like **vectors**, **matrices**, **dataframes**, **lists**, are fundamental tools that allow you to **organize and store complex information**, so that they can be easily **processes by functions** (e.g., `lm()` function may fit a linear model on variables stored in a dataframe)

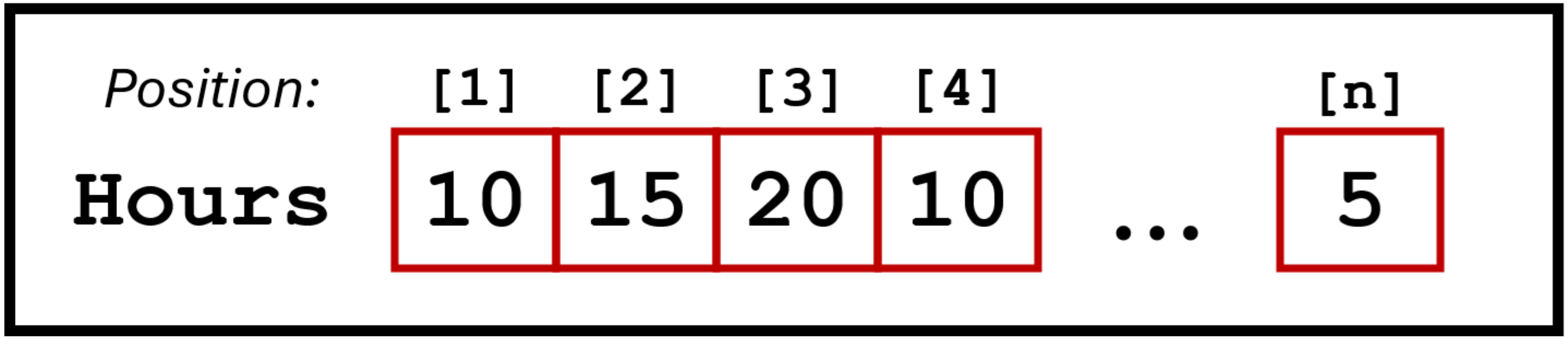
Most operations you will perform in R (e.g., *processing data*, *fitting models*, *plotting outputs*) are performed on these data structures

Vectors

Simple one-dimensional structures that store data of different types



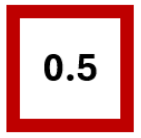
Here is an actual **example** (of a *numerical* vector):



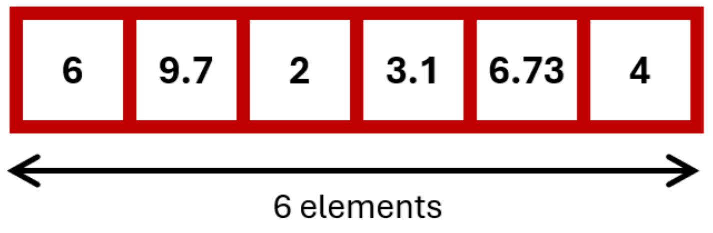
Vectors as 1-D Arrays

Vectors are just special cases of **arrays**

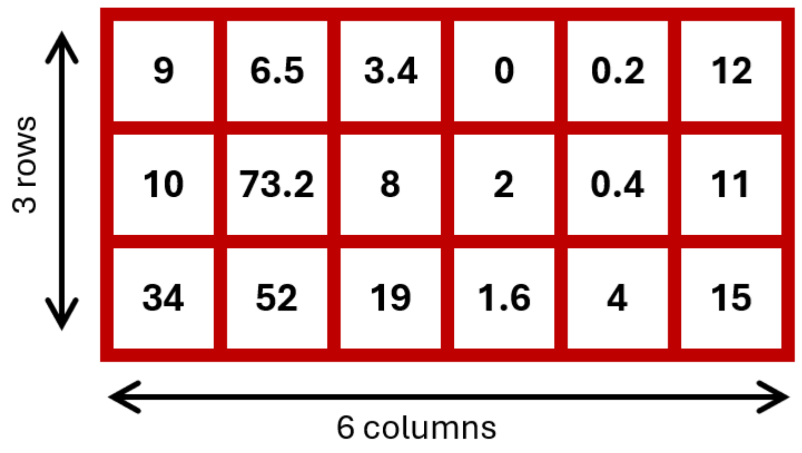
Scalar
0-Dimensional *array*



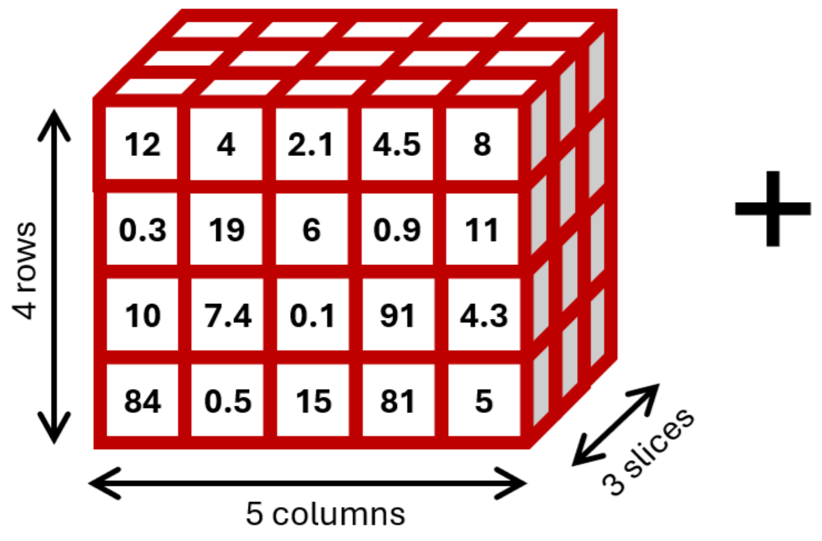
Vectors
1-Dimensional *array*



Matrix
2-Dimensional *array*



3-Dimensional *Array*



Create Vectors with `c()`

Vectors can easily be **created using the** `c()` base function, with a sequence of elements separated by *commas* “`,`”

Vectors can be of different types. The following example shows a *character* vector (note the *quotes* `"  "` around objects):

```r
Teachers = c("Pastore", "Kiesner", "Granziol", "Toffalini"
             "Calignano", "Epifania", "Bastianelli")
```

or numeric:

```r
Hours = c(10, 15, 20, 10, 15, 5, 15, 5)
```

Vectors Must be Homogeneous

Vectors must contain elements of the **same type**. If you mix types, R will automatically **coerce** the elements to a single type, which may lead to undesired results.

Therefore, **avoid mixing data types!** Example:

```
Hours = c(10, 15, 20, 10, 15, "tbd", 15, 5)
Hours
```

```
[1] "10"  "15"  "20"  "10"  "15"  "tbd" "15"  "5"
```

everything was coerced to become a character!

If needed, use **NA** (Not Available):

```
Hours = c(10, 15, 20, 10, 15, NA, 15, 5)
Hours # remains a numerical vector, NA does not affect typ
```

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

Vectors Must be Homogeneous

You may **coerce a vector** to be a particular type if needed

```
Hours = c(10, 15, 20, 10, 15, "tbd", 15, 5)
Hours
```

```
[1] "10"  "15"  "20"  "10"  "15"  "tbd" "15"  "5"
```

```
as.numeric(Hours)
```

```
Warning: NAs introduced by coercion
```

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

But be careful! Elements that cannot be coerced to the target type, will be replace with NA

```
Hours = c("10", "15,", "20", " 10", "15 ", "tbd", "15.", "5_")
as.numeric(Hours)
```

```
Warning: NAs introduced by coercion
```

```
[1] 10 NA 20 10 15 NA 15 NA
```

Indexing Vectors

Select/extract elements with **INDEXING** using square brackets `[]`:

```r
Hours = c(10, 15, 20, 10, 15, 5, 15, 5)
Hours[4] # a single element
```

```
[1] 10
```

```r
Hours[5:7] # a range of elements
```

```
[1] 15  5 15
```

```r
Hours[c(1,3,6)] # specific elements
```

```
[1] 10 20  5
```

Know the **length** of a vector using the `length()` function, and use it:

```r
length(Hours)
```

```
[1] 8
```

```r
Hours[length(Hours)] # use it to extract the last element
```

```
[1] 5
```

Indexing Vectors

Negative indexing

You can use the *minus* sign **-** to select **all elements except some** from a vector. (This method is also applicable to dataframes)

```
Hours = c(10, 15, 20, 10, 15, 5, 15, 5)
Hours[-4] # ALL BUT a single element
```

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

```
Hours[-c(5:7)] # ALL BUT a range of elements
```

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

```
Hours[-c(1,3,6)] # ALL BUT specific elements
```

```
[1] 15 10 15 15  5
```

```
Hours[-length(Hours)] # ALL BUT the last element
```

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

Logical Indexing

Often, you'll need to extract values from a vector based on specific ***logical*** conditions. Here's an example:

```r
Hours = c(10, 15, 20, 10, 15, 5, 15, 5)
Hours[Hours >= 15] # extract only values greater than or equal to 15
```

```
[1] 15 20 15 15
```

This is called ***logical indexing*** because you are selecting elements based on a logical vector (i.e., a sequence of TRUE, FALSE):

```r
Hours >= 15 # the logical vector actually inside the square brackets
```

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

Also, you can use a vector to extract values **from another vector**:

```r
Teachers[Hours >= 15]
```

```
[1] "Kiesner"     "Granziol"    "Calignano"   "Bastianelli"
```

Indexing and Assignment

With indexing, you can not only select, but also **assign or modify** elements in a vector:

```
Hours = c(10, 15, 20, 10, 15, 5, 15, 5)

Hours[1] = 0 # assign a new value
Hours[3] = Hours[3]+50 # modify an existing element
Hours
```

```
[1]  0 15 70 10 15  5 15  5
```

You can even assign values **outside the current range** of the vector. But what happens?

```
Hours[20] = 5
Hours
```

```
 [1]  0 15 70 10 15  5 15  5 NA NA NA NA NA NA NA NA NA NA NA  5
```

Operating on Vectors

you can simultaneously apply an operation to a whole vector, like

```
Hours = c(10, 15, 20, 10, 15, 5, 15, 5)
Hours / 5
```

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

Of course, this is useful when you want to save the result as a new vector:

```
ECTS = Hours / 5
```

Similarly, you can apply functions to all elements of a vector:

```
sqrt(Hours) # computes square root of each element
```

```
[1] 3.162278 3.872983 4.472136 3.162278 3.872983 2.236068 3.872983 2.236068
```

```
log(Hours) # computes the natural logarithm of each element
```

```
[1] 2.302585 2.708050 2.995732 2.302585 2.708050 1.609438 2.708050 1.609438
```

Summary Statistics on Vectors

A whole vector may serve to compute summary statistics, for example using functions such as `mean()`, `sd()`, `median()`, `quantile()`, `max()`, `min()`:

```
mean(Hours) # returns the average value (mean) of the vect
```

```
[1] 11.875
```

```
sd(Hours) # returns the Standard Deviation of the vector
```

```
[1] 5.303301
```

```
median(Hours) # returns the median value of the vector
```

```
[1] 12.5
```

Summary Statistics on Vectors

A whole vector may serve to compute summary statistics, for example using functions such as **`mean()`**, **`sd()`**, **`median()`**, **`quantile()`**, **`max()`**, **`min()`**:

```r
quantile(Hours, probs=c(.25, .50, .75)) # returns desired
```

```
  25%   50%   75% 
 8.75 12.50 15.00 
```

```r
max(Hours) # returns largest value
```

```
[1] 20
```

```r
min(Hours) # returns smallest value
```

```
[1] 5
```

Summary Statistics - Managing Missing (NA) Values

All of the previous summary statistics will **fail** if there is even a single NA value:

```
Hours = c(10, 15, 20, 10, 15, NA, 15, 5)

mean(Hours) # a single NA value implies that the average is impossible
```

```
[1] NA
```

```
quantile(Hours, probs=c(.25, .75)) # quantile() will even return an Er
```

```
Error in quantile.default(Hours, probs = c(0.25, 0.75)): missing values
and NaN's not allowed if 'na.rm' is FALSE
```

You can easily manage missing values by adding the **`na.rm=TRUE`** argument:

```
mean(Hours, na.rm=TRUE) # NA values are ignored
```

```
[1] 12.85714
```

```
quantile(Hours, probs=c(.25, .75), na.rm=TRUE) # NA values are ignored
```

```
25% 75%
 10  15
```

Replacing NA With the Average Value

Replacing a missing value with the average across valid values is risky, as it may alter many other summary statistics, but it is a good example for understanding different concepts seen so far:

```r
Hours = c(10, 15, 20, 10, 15, NA, 15, 5)

# compute the average value ignoring NAs, and put it wherever
# there is a NA value in the vector
Hours[is.na(Hours)] = mean(Hours, na.rm=TRUE)

# now let's inspect the updated content of the vector
Hours
```

```
[1] 10.00000 15.00000 20.00000 10.00000 15.00000 12.85714 15.00000  5.00000
```

```r
# by the way... na.rm=TRUE is no longer needed now, as NA is no longer
mean(Hours)
```

```
[1] 12.85714
```

Frequency Counts

Another useful summary statistic is the **frequency count**, which shows how often each unique value appears in a vector. You can use the `table()` function to calculate frequencies easily:

```
type = c("METHODOLOGY", "METHODOLOGY", "PROGRAMMING", "SOFT SKILLS", "S
         "METHODOLOGY", "SOFT SKILLS", "METHODOLOGY", "PROGRAMMING")
table(type)
```

```
type
METHODOLOGY PROGRAMMING SOFT SKILLS 
          4           2           3 
```

Be careful: R is case sensitive!

```
type = c("METHODOLOGY", "methodology", "PROGRAMMING", "SOFT SKILLS", "S
         "METHODOLOGY", "SOFT SKILLS", "METHODOLOGY", "Programming")
table(type)
```

```
type
methodology METHODOLOGY Programming PROGRAMMING SOFT SKILLS 
          1           3           1           1           3 
```