# Vectors (part 1)

## R Data Objects

Gaston Sanchez



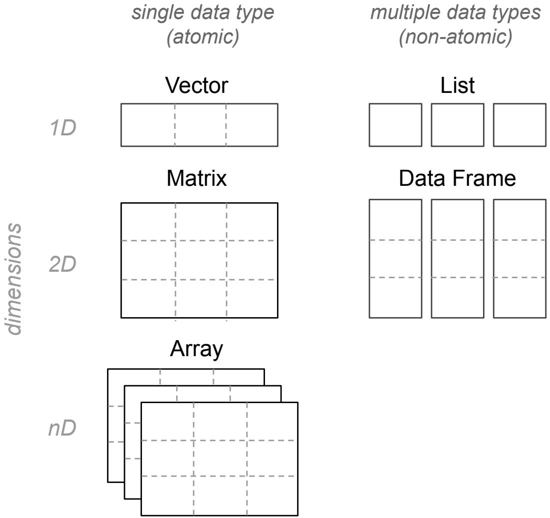
R Coding Compendium

Donate

# About

To make the best of the R language, you'll need a strong understanding of the basic **data types** and **data structures** and how to operate on them.

# Basic Data Objects in R



# Basic Data Objects in R

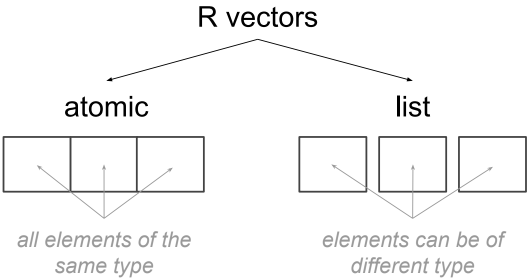
There are various data structures in R (we'll describe them in detail later):

- vectors
- matrices (2-dimensional arrays)
- arrays (n-dim in general)
- lists
- data frames

# Vectors

# R Vectors

- A vector is the most basic data structure in R
- This is why I like to think of R as a *vector-based* language
- There are two flavors of vectors: **atomic** and **list**



# R Vectors

```r
# atomic vector
a = c(1, 2, 3)
a
```

```
## [1] 1 2 3
```

```r
# list vector
b = list(1, "two", TRUE)
b
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "two"
##
## [[3]]
## [1] TRUE
```

# R Vectors

An atomic vector means that **all** its elements are of the same type.

In the R community, the term **vector** is typically associated with the atomic flavor of vectors.

Technically speaking, an R **list** is also a vector, although it's non-atomic in the sense that it can contain elements of different types.

# Atomic Vectors

# Atomic Vectors

The most simple type of atomic vectors are single values (i.e. vectors with just one element):

```
# logical
a <- TRUE

# integer
x <- 1L

# double (real)
y <- 5

# character
b <- "yosemite"
```

Notice the appended `L` when specifying an `integer` type.

# Basic Atomic Vectors

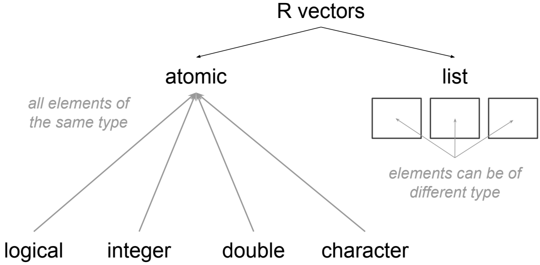
One way to create vectors with more than one element is with the function `c()` , short for **catenate** or **combine**:

```
# some vectors
x <- c(1, 2, 3, 4, 5)

y <- c("one", "two", "three")

z <- c(TRUE, FALSE, FALSE)
```

Separate each element by a comma

# Atomic Vectors

R has 4 (+ 2)* basic types of atomic vectors: `logical`, `integer`, `double` and `character`




*There are two other (less used) data types: `complex` and `raw`.

# Data types

- A **logical** vector hold `TRUE`, `FALSE` values
- An **integer** vector hold integers (no decimal component)
- A **double** vector hold double precision ("real") numbers
- A **character** vector hold character strings
- *A **complex** vector hold complex numbers
- *A **raw** vector hold raw bytes

# R Vectors

- Atomic vectors are contiguous cells containing data
- Can be of any length (including zero): `length()`
- They have a specific data type: `typeof()`
- You can give them additional attributes with `attributes()`

# Atomic Vectors

Use typeof() to find the data-type of an atomic vector:

```
a <- TRUE
typeof(a) # logical

x <- 1L
typeof(x)  # integer

y <- 5
typeof(y)  # double (real)

b <- "yosemite"
typeof(b)  # character
```

Notice the appended L when specifying an integer type.

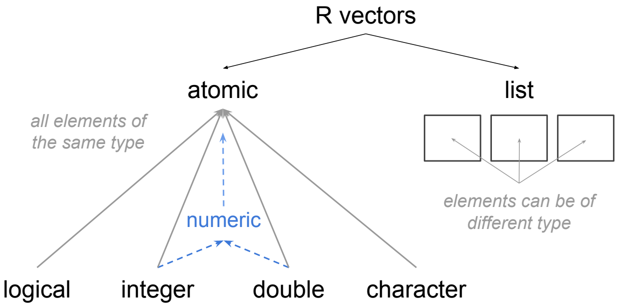
# Special Values

There are some special values

- `NULL` is the null object (it has length zero); it is not a vector but it often plays the role of a zero-length vector.
- Missing values are referred to by the symbol `NA`; there various flavors:
  - `NA` ("plain vanilla" logical missing value)
  - `NA_integer_` (integer missing value)
  - `NA_real_` (double missing value)
  - `NA_character_` (character missing value)
- `Inf` indicates positive infinite
- `-Inf` indicates negative infinite
- `NaN` indicates Not a Number

# Atomic Vectors

For historical reasons (i.e. compatibility with S) R considers `integer` and `double` vectors as `numeric` types:



# Atomic Vectors

Both `integer` and `double` are grouped together under the `numeric` umbrella

```
# integer
x <- 1L
is.numeric(x)
```

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

```
# double (real)
y <- 5
is.numeric(y)
```

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

# Atomic Vectors

Among useRs, it is common to refer to the **mode** of a vector, and to use the `mode()` function as a pseudo-equivalent of data-type:

| value | example | mode | storage |
|---|---|---|---|
| logical | `TRUE, FALSE` | `logical` | `logical` |
| integer | `1L, 2L` | `numeric` | `integer` |
| double | `1, -0.5` | `numeric` | `double` |
| complex | `3 + 5i` | `complex` | `complex` |
| character | `"hello"` | `character` | `character` |

This notion of `mode` in R also has its historical roots to have a compatible syntax with the S language.

# Atomic Vectors

To summarize:

- vectors are **atomic** structures
- the values in a vector must be **ALL** of the same type
- atomic vectors are contiguous cells containing data
- can be of any length (including zero)
- either all integers, or reals, or complex, or characters, or logicals
- you **cannot** have a vector of different data types

# Coercion

# Coercion

What happens if you mix different data types in a vector?

```r
x <- c(1, 2, 3, "four", "five")

y <- c(TRUE, FALSE, 3, 4)

z <- c(TRUE, 1L, 2 + 3i, pi)
```

What will the data-types of `x`, `y` and `z` be?

# Atomic Vectors: Implicit Coercion

If you mix different data values, R will **implicitly coerce** them so they are all of the same type

```
# mixing numbers and characters
x <- c(1, 2, 3, "four", "five")
x
```

```
## [1] "1"    "2"    "3"    "four" "five"
```

```
# mixing numbers and logical values
y <- c(TRUE, FALSE, 3, 4)
y
```

```
## [1] 1 0 3 4
```

# Atomic Vectors

```r
# mixing numbers and logical values
z <- c(TRUE, FALSE, "TRUE", "FALSE")
z
```

```
## [1] "TRUE"  "FALSE" "TRUE"  "FALSE"
```

```r
# mixing integer, real, and complex numbers
w <- c(1L, -0.5, 3 + 5i)
w
```

```
## [1]  1.0+0i -0.5+0i  3.0+5i
```

# How does R coerce data types?

There is a hierarchy of data-types used by to apply its implicit coercion rules:

`logical` < `integer` < `double` < `character`

`character` type is the dominant one in the sense that if a character is present, R will coerce everything else to characters.

Then we have `double` (or `real`): assuming a non-character vector, if it has at least one `double` element, then all other elements will be coerced as doubles. And so on with `integer` and `logical`.

# Coercion

The following table shows the resulting data-types from mixing two or more different flavors of atomic vectors:

| | logical | integer | double | character |
|---|---|---|---|---|
| logical | logical | integer | double | character |
| integer | integer | integer | double | character |
| double | double | double | double | character |
| character | character | character | character | character |

# Explicit Coercion functions

R provides a set of **explicit** coercion functions that allow us to "convert" one type of data into another

- `as.character()`
- `as.integer()`
- `as.double()`
- `as.numeric()`
- `as.logical()`

# Properties of Vectors

- all vectors have a length
- vector elements can have associated names
- vectors are objects of class `"vector"`
- vectors have a mode (storage mode)

# Properties of Vectors

```r
# vector with named elements
x <- c(a = 1, b = 2.5, c = 3.7, d = 10)
x
```

```
##    a    b    c    d
##  1.0  2.5  3.7 10.0
```

```r
length(x)
```

```
## [1] 4
```

```r
mode(x)
```

```
## [1] "numeric"
```

# Vectorization and Recycling

# Vectorized Operations

You've probably operated with vectors like so:

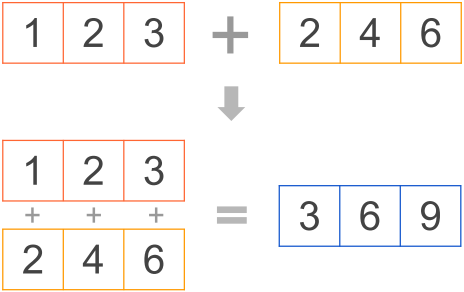
```
vec1 <- c(1, 2, 3)

vec2 <- c(2, 4, 6)

vec1 + vec2
```

```
## [1] 3 6 9
```

This is an example of **vectorized code**

# Vectorization

Vectorized code in pictures:




Vectorized code refers to operations that are performed on the contents of `vec1` and `vec2`, **element-by-element** at the same time.

# Vectorized Operations

A vectorized computation is any computation that when applied to a vector operates on all of its elements

```
c(1, 2, 3) + c(3, 2, 1)
```

```
## [1] 4 4 4
```

```
c(1, 2, 3) * c(3, 2, 1)
```

```
## [1] 3 4 3
```

```
c(1, 2, 3) ^ c(3, 2, 1)
```

```
## [1] 1 4 3
```

# Vectorization

All arithmetic, trigonometric, math and other vector functions are vectorized:

```r
log(c(1, 2, 3))
```

```
## [1] 0.0000000 0.6931472 1.0986123
```

```r
cos(seq(1, 3))
```

```
## [1]  0.5403023 -0.4161468 -0.9899925
```

```r
sqrt(1:3)
```

```
## [1] 1.000000 1.414214 1.732051
```

# Recycling and Vectorization

What happens if you operate on two vectors of different length, in which one of them is a one element ("scalar") vector?

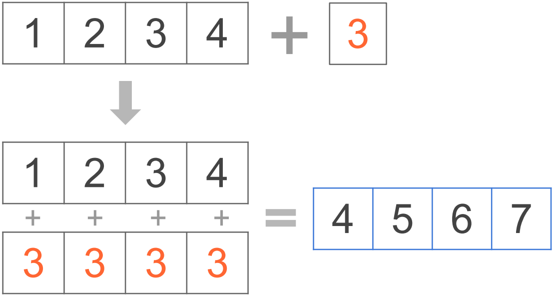
```
vec <- c(1, 2, 3, 4)

vec + 3
```

```
## [1] 4 5 6 7
```

This is an example of recycling and vectorization. The value 3 gets recycled as many times as the length of longer vector `vec`, and then vectorization applies.

# Recycling and Vectorization

Recycling and vectorization in pictures



# Recycling Rule

The recycling rule can be very useful, like when operating between a vector and a "scalar" (ie. one-element vector)

```r
x <- c(2, 4, 6, 8)
x + 3  # add 3 to all elements in x
```

```
## [1]  5  7  9 11
```

```r
x / 3  # divide all elemnts by 3
```

```
## [1] 0.6666667 1.3333333 2.0000000 2.6666667
```

```r
x ^ 3  # all elements to the power of 3
```

```
## [1]   8  64 216 512
```

# Recycling

What happens if you operate on two vectors (of length $> 1$) that have different lengths?

```
c(1, 3, 5, 7) + c(2, 4)
```

```
## [1]  3  7  7 11
```

The same **recycling rule** applies: the shorter vector is replicated enough times so that the result has the length of the longer vector

# Recycling

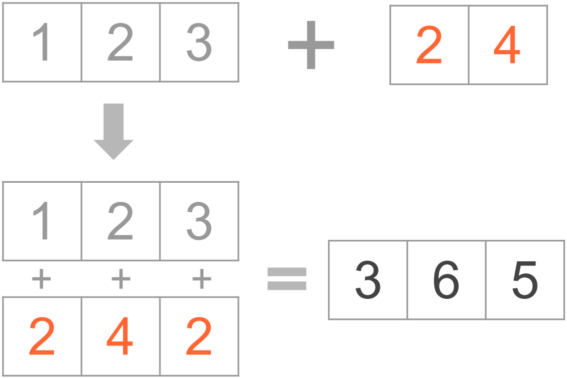
When vectorized computations are applied, some “issues” may occur when the length of the shorter vector is not a multiple of the length of the longer vector:

```
c(1, 2, 3) + c(2, 4)
```

```
## Warning in c(1, 2, 3) + c(2, 4): longer object length is
## shorter object length
```

```
## [1] 3 6 5
```

# Recycling



# Recycling Rule

The recycling rule states that the shorter vector is replicated enough times so that the result has the length of the longer vector

```
c(1, 2, 3, 4) + c(2, 1)
```

```
## [1] 3 3 5 5
```

```
1:10 * 1:5
```

```
##  [1]  1  4  9 16 25  6 14 24 36 50
```

*Donation*

*If you find any value and usefulness in this set of slides, please consider making a one-time donation in any amount (via paypal). Your support really matters.*

Donate

https://www.paypal.com/donate?business=ZF6U7K5MW25W2&currency_code=USD