

# Introduction to R\*

## Part 2: Atomic Data Types - Atomic/homogeneous vectors

Wim R.M. Cardoen

Last updated: 03/21/2024 @ 18:51:37

## Contents

<b>1</b>	<b>R Objects</b>	<b>3</b>
1.1	Examples . . . . .	3
<b>2</b>	<b>Atomic Data Types</b>	<b>5</b>
2.1	The core/atomic data types . . . . .	5
2.1.1	Examples . . . . .	5
2.2	Operations on atomic data types . . . . .	7
2.2.1	Examples . . . . .	7
2.3	Exercises . . . . .	8
<b>3</b>	<b>Atomic vectors</b>	<b>10</b>
3.1	Creation of atomic vectors . . . . .	10
3.1.1	Examples . . . . .	10
3.1.2	Exercises . . . . .	12
3.2	Operations on vectors: element-wise . . . . .	12
3.2.1	Examples . . . . .	12
3.2.2	Exercises . . . . .	13
3.3	Retrieving elements of vectors . . . . .	13
3.3.1	Examples . . . . .	13
3.3.2	Exercises . . . . .	15
3.4	Hash tables . . . . .	16
3.4.1	Examples . . . . .	17
3.5	NA (Not Available values) . . . . .	18
3.5.1	Examples . . . . .	18
3.5.2	Exercises . . . . .	19
3.6	NaN and infinities . . . . .	19
3.6.1	Examples . . . . .	19
3.7	Note on logical operators . . . . .	21
3.7.1	Examples . . . . .	21
3.7.2	Exercises . . . . .	22

---

\*© - Wim R.M. Cardoen, 2022 - The content can neither be copied nor distributed without the **explicit** permission of the author.

R can be summarized in **three** principles (John M. Chambers, 2016)

- Everything that exists in R is an **object**.
- Everything that happens in R is a **function** call.
- **Interfaces** to other languages are a part of R.

# 1 R Objects

- An object in R is (internally) represented as a pair: (**symbol**, **value**).
- A **symbol** is assigned a **value** by the use of an arrow pointing to the left (**<-**).
- There are **less favored** ways:
  - A simple equality sign (**=**).
  - Using the **assign()** function.

## 1.1 Examples

- Clean up the global environment i.e. remove all objects from the current R environment.  
**Recommended!**

```
rm(list=ls())  
ls()
```

```
character(0)
```

- **preferred** way to assign variables

```
x <- 5.0  
x
```

```
[1] 5
```

- alternative 1: mainly used to assign default function arguments

```
y = 5.0  
y
```

```
[1] 5
```

```
mysamplevariance <- function(x, av=0){
```

```
  n <- length(x)  
  if(n>1){  
    return(1.0/(n-1)*sum((x-av)^2))  
  }  
  else{  
    stop("ERROR:: Dividing by zero (n==1) || (n==0) ")  
  }  
}
```

```
x <- rnorm(10)  
mysamplevariance(x)
```

```
[1] 1.441095
```

```
mysamplevariance(x,mean(x))
```

```
[1] 1.204279
```

```
var(x)
```

```
[1] 1.204279
```

- alternative 2: even less used

```
assign("z", 5.0)
```

```
z
```

```
[1] 5
```

- functions are objects

```
f <- mean
```

```
f
```

```
function (x, ...)
```

```
UseMethod("mean")
```

```
<bytecode: 0x5db312594728>
```

```
<environment: namespace:base>
```

```
val <- f(1:10)
```

```
val
```

```
[1] 5.5
```

"Nothing exists except atoms and empty space; everything else is opinion". (Democritos)

## 2 Atomic Data Types

### 2.1 The core/atomic data types

- R has the following 6 **atomic** data types:
  - logical (i.e. boolean)
  - integer
  - double
  - character (i.e. string)
  - complex
  - raw (i.e. byte)

The latter 2 types (i.e. complex and especially raw) are less common.

The **typeof()** function determines the **INTERNAL** storage/type of an R object.

#### 2.1.1 Examples

- boolean/logical values: either **TRUE** or **FALSE**

```
x1 <- TRUE
x1
```

```
[1] TRUE
```

```
typeof(x1)
```

```
[1] "logical"
```

- integer values ( $\in \mathbb{Z}$ ):

```
x2 <- 3L
x2
```

```
[1] 3
```

```
typeof(x2)
```

```
[1] "integer"
```

- double (precision) values:

```
x3 <- 3.14
x3
```

```
[1] 3.14
```

```
typeof(x3)
```

```
[1] "double"
```

- character values/strings

```
x4 <- "Hello world"
x4
```

```
[1] "Hello world"
```

```
typeof(x4)
```

```
[1] "character"
```

- complex values ( $\in \mathbb{C}$ ):

```
x5 <- 2.0 + 3i
x5
```

```
[1] 2+3i
```

```
typeof(x5)
```

```
[1] "complex"
```

## 2.2 Operations on atomic data types

- **logical** operators: `==`, `!=`, `&&`, `||`, `!`
- **numerical** operators: `+`, `-`, `*`, `/`, `^`, `**` (same as the caret), but also:
  - integer division: `%/%`
  - modulo operation: `%%`
  - **Note**: matrix multiplication will be performed using `%*%`
- **character/string** manipulation:
  - `nchar()`:
  - `paste()`:
  - `cat()`:
  - `sprintf()`:
  - `substr()`:
  - `strsplit()`:
  - **Note**: Specialized R libraries were developed to manipulate strings e.g. *stringr*
- explicit **cast**/conversion: <https://data-flair.training/blogs/r-string-manipulation/>
  - `as.{logical, integer, double, complex, character}()`
- explicit **test** of the type of a variable:
  - `is.{logical, integer, double, complex, character}()`

### 2.2.1 Examples

- Logical operators:

```
x <-3
y <-7
(x<=3) && (y==7)
```

```
[1] TRUE
```

```
!(y<7)
```

```
[1] TRUE
```

- Mathematical operations

```
2**4
```

```
[1] 16
```

```
7%%4
```

```
[1] 3
```

```
7/4
```

```
[1] 1.75
```

```
7%/%4
```

```
[1] 1
```

- String operations

```
s <- "Hello"
nchar(s)
```

```
[1] 5
```

```
news <- paste(s, "World")  
news
```

```
[1] "Hello World"
```

```
sprintf("My new string:%20s\n", news)
```

```
[1] "My new string:          Hello World\n"
```

```
city <- "Witwatersrand"  
substr(city, 4, 8)
```

```
[1] "water"
```

- Conversion and testing of types

```
s <- "Hello World"  
is.character(s)
```

```
[1] TRUE
```

```
s1 <- "-500"  
is.character(s1)
```

```
[1] TRUE
```

```
s2 <- as.double(s1)  
is.character(s2)
```

```
[1] FALSE
```

```
is.double(s2)
```

```
[1] TRUE
```

```
s3 <- as.complex(s2)  
s3
```

```
[1] -500+0i
```

```
sqrt(s3)
```

```
[1] 0+22.36068i
```

## 2.3 Exercises

- - Calculate  $\log_2(10)$  using R's `log()` function
  - Perform the inverse operation and check that you get 10 back
- Let  $z = 3 + 4i$



- Use R's **Re()**, **Im()** functions to extract the real and imaginary parts of  $z$ .
- Calculate the modulus of  $z$  using R's **Mod()** function and check whether you the same answer using  $\sqrt{\Re(z)^2 + \Im(z)^2}$ .
- Calculate the argument of  $z$  using R's **Arg()** function and check whether you have the same answer using  $\arctan\left(\frac{\Im(z)}{\Re(z)}\right)$ .

## 3 Atomic vectors

- An **atomic** vector is a data structure containing elements of **only one atomic** data type. Therefore, an atomic vector is **homogeneous**.
- Atomic vectors are stored in a **linear** fashion.
- R does **NOT** have scalars:
  - An atomic vector of **length 1** plays the role of a scalar.
  - Vectors of **length 0** also exist (and they have some use!).
- A **list** is a vector not necessarily of the atomic type.  
A list is also known as a **recursive/generic** vector (*vide infra*).

### 3.1 Creation of atomic vectors

Atomic vectors can be created in a multiple ways:

- Use of the `vector()` function.
- Use of the `c()` function (`c` stands for concatenate).
- Use of the column operator `:`
- Use of the `seq()` and `rep()` functions.

The length of a vector can be retrieved using the `length()` function.

#### 3.1.1 Examples

- use of the `vector()` function:

```
x <- vector() # Empty vector (Default: 'logical')
x
```

```
logical(0)
```

```
length(x)
```

```
[1] 0
```

```
typeof(x)
```

```
[1] "logical"
```

```
x <- vector(mode="complex", length=4)
x
```

```
[1] 0+0i 0+0i 0+0i 0+0i
```

```
length(x)
```

```
[1] 4
```

```
x
```

```
[1] 0+0i 0+0i 0+0i 0+0i
```

```
x[1] <- 4
```

```
x
```

```
[1] 4+0i 0+0i 0+0i 0+0i
```

- use of the `c()` function:

```
x1 <- c(3, 2, 5.2, 7)
x1
```

```
[1] 3.0 2.0 5.2 7.0
```

```
x2 <- c(8, 12, 13)
x2
```

```
[1] 8 12 13
```

```
x3 <- c(x2, x1)
x3
```

```
[1] 8.0 12.0 13.0 3.0 2.0 5.2 7.0
```

```
x4 <- c(FALSE, TRUE, FALSE)
x4
```

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

```
x5 <- c("Hello", "Salt", "Lake", "City")
x5
```

```
[1] "Hello" "Salt" "Lake" "City"
```

- use of the column operator:

```
y1 <- 1:10
y1
```

```
[1] 1 2 3 4 5 6 7 8 9 10
```

```
y2 <- 5:-5
y2
```

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

```
y3 <- 2.3:10
y3
```

```
[1] 2.3 3.3 4.3 5.3 6.3 7.3 8.3 9.3
```

```
y4 <- 2.0*(7:1)
y4
```

```
[1] 14 12 10 8 6 4 2
```

```
y5 <- (1:7) - 1
y5
```

```
[1] 0 1 2 3 4 5 6
```

- `seq()` and `rep()` functions

```
z1 <- seq(from=1, to=15, by=3)
z1
```

```
[1] 1 4 7 10 13
```

```
z2 <- seq(from=-2, to=5, length=4)
z2
```

```
[1] -2.0000000  0.3333333  2.6666667  5.0000000
```

```
z3 <- rep(c(3,2,4), time=2)
z3
```

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

```
z4 <- rep(c(3,2,4), each=3)
z4
```

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

```
z5 <- rep(c(1,7), each=2, time=3)
z5
```

```
[1] 1 1 7 7 1 1 7 7 1 1 7 7
```

```
length(z5)
```

```
[1] 12
```

### 3.1.2 Exercises

- Use the `seq()` function to generate the following sequence:  
6 13 20 27 34 41 48
- Create the following R vector using **only** the `seq()` and `rep()` functions:  
-8 -8 -8 -8 0 8 8 8 16 16 16 16 16

## 3.2 Operations on vectors: element-wise

- All operations on vectors in R happen **element by element** (cfr. *NumPy*).
- **Vector Recycling**:

If 2 vectors of **different** lengths are involved in an operation, the **shortest vector** will be repeated until all elements of the longest vector are matched.

A *warning* message will be sent to the stdout.

### 3.2.1 Examples

```
x <- -3:3
x
```

```
[1] -3 -2 -1  0  1  2  3
```

```
y <- 1:7
y
```

```
[1] 1 2 3 4 5 6 7
```

```
xy <- x*y
xy
```

```
[1] -3 -4 -3  0  5 12 21
```

```
xpy <- x^y
xpy
```

```
[1] -3  4 -1  0  1 64 2187
```

```
x <- 0:10
y <- 1:2
length(x)
```

```
[1] 11
```

```
length(y)
```

```
[1] 2
```

```
x
```

```
[1] 0 1 2 3 4 5 6 7 8 9 10
```

```
y
```

```
[1] 1 2
```

```
x+y
```

```
Warning in x + y: longer object length is not a multiple of shorter object
length
```

```
[1] 1 3 3 5 5 7 7 9 9 11 11
```

### 3.2.2 Exercises

- Create the following vector (do **not** use `c()`):  
-512 -216 -64 -8 0 8 64 216 512 1000

## 3.3 Retrieving elements of vectors

- Indexing: starts at **1** (**not 0** like C/C++, Python, Java, ...) see also:  
[Edsger Dijkstra: Why numbering should start at zero](#)
- Use of vector with indices to extract values.
- Advanced features:
  - use of boolean values to extract values.
  - the membership operator: `%in%`.
  - the deselect/omit operator: `-`
  - `which()`: returns the indices for which the condition is true.
  - `any()/all()` functions.
    - \* `any()` : **TRUE** if at least 1 value is true
    - \* `all()` : **TRUE** if all values are true

### 3.3.1 Examples

- Use of a simple index:

```
x <- seq(2,100,by=15)
x
```

```
[1] 2 17 32 47 62 77 92
```

```
x[4]
```

```
[1] 47
```

```
x[1]
```

```
[1] 2
```

- Select several indices at once using vectors:

```
x
```

```
[1] 2 17 32 47 62 77 92
```

```
x[3:5]
```

```
[1] 32 47 62
```

```
x[c(1,3,5,7)]
```

```
[1] 2 32 62 92
```

```
x[seq(1,7,by=2)]
```

```
[1] 2 32 62 92
```

- Extraction via booleans (i.e. retain only those values that are equal to **TRUE**):

```
x
```

```
[1] 2 17 32 47 62 77 92
```

```
x>45
```

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

```
x[x>45]
```

```
[1] 47 62 77 92
```

- Use of the **%in%** operator:

```
x
```

```
[1] 2 17 32 47 62 77 92
```

```
10 %in% x
```

```
[1] FALSE
```

```
62 %in% x
```

```
[1] TRUE
```

```
c(32,33,43) %in% x
```

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

```
!(c(32,33,43) %in% x)
```

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

- Negate/filter out the elements with **negative** indices:

```
x <- c(1,13,17,27,49,91)
x
```

```
[1]  1 13 17 27 49 91
```

```
x[-c(2,4,6)]
```

```
[1]  1 17 49
```

```
z <- x[-1] - x[-length(x)]
z
```

```
[1] 12  4 10 22 42
```

- The `which()` function returns **only those indices** of which the condition/expression is **true**.

```
# Sample 10 numbers from N(0,1)
vecnum <- rnorm(n=10)
vecnum
```

```
[1]  1.0202975 -0.8833187  0.9165650 -0.3300365  1.2174873  0.2146407
[7]  0.7091241  0.6092013  0.2009706  0.1623719
```

```
which(vecnum>1.0)
```

```
[1] 1 5
```

- Use of the `any()/all()` functions.

```
y <- seq(0,100,by=10)
x
```

```
[1]  1 13 17 27 49 91
```

```
y
```

```
[1]  0 10 20 30 40 50 60 70 80 90 100
```

```
any(x<y)
```

```
Warning in x < y: longer object length is not a multiple of shorter object
length
```

```
[1] TRUE
```

```
all(x[6:7]>y[2:3])
```

```
[1] NA
```

### 3.3.2 Exercises

- R has the its own inversion function, `rev()`, e.g.:

```
x <- seq(from=2,to=33,by=3)
x
```

```
[1]  2  5  8 11 14 17 20 23 26 29 32
```

```
y <- rev(x)
y
```

```
[1] 32 29 26 23 20 17 14 11 8 5 2
```

Invert the vector `x` without invoking the `rev()` function.

- The Taylor series for  $\ln(1+x)$  is converging when  $|x| < 1$  and is given by:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \frac{x^6}{6} + \dots$$

Calculate the sum of the first 5, 10, 15 terms in the above expression to approximate  $\ln(1.2)$ . Compare with R's value i.e.: `log(1.2)`.

- The logarithmic return in finance is defined as:

$$R_t = \ln\left(\frac{P_t}{P_{t-1}}\right)$$

- Generate a financial time series using the following R code:

```
price <- abs(rcauchy(1000))+1.E-6
```

- Calculate the logarithmic return for the financial time series `price`.  
The newly created time series will be 1 element shorter in length than the original one.  
Compare your result with `diff(log(price))`.

- Monte-Carlo approximation of  $\pi$

Let `S1` be the square spanned by the following 4 vertices:  $\{(0,0), (0,1), (1,0), (1,1)\}$ .

Let `S2` be the first quadrant of the unit-circle  $\mathcal{C} : x^2 + y^2 = 1$ .

The ratio  $\rho$  defined as:

$$\rho := \frac{\text{Area S2}}{\text{Area S1}} = \frac{\text{\#Points in S2}}{\text{\#Points in S1}}$$

allows us to estimate  $\frac{\pi}{4}$  numerically.

Therefore:

- Sample 100000 independent  $x$ -coordinates from `Unif`.
- Sample 100000 independent  $y$ -coordinates from `Unif`.
- Calculate an approximate value for  $\pi$  using the Monte-Carlo approach.

Note: The uniform distribution  $[0,1)$  (`Unif`) can be sampled using `runif()`.

### 3.4 Hash tables

A **hash table** is a data structure which implements an associative array or dictionary. It is an abstract data which maps data to keys.

- There are several ways to create one:
  - Map names to an existing vector
  - Add names when creating the vector
- To remove the map, map the names to NULL



### 3.4.1 Examples

- Creation of 2 independent vectors

```
capitals <- c("Albany", "Providence", "Hartford", "Boston", "Montpelier", "Concord", "Augusta")
states <- c("NY", "RI", "CT", "MA", "VT", "NH", "ME")
capitals
```

```
[1] "Albany"      "Providence" "Hartford"    "Boston"      "Montpelier"
[6] "Concord"     "Augusta"
```

```
states
```

```
[1] "NY" "RI" "CT" "MA" "VT" "NH" "ME"
```

```
capitals[3]
```

```
[1] "Hartford"
```

- Create the hashtable/dictionary

```
# Method 1
```

```
names(capitals) <- states
```

```
capitals
```

```
      NY      RI      CT      MA      VT      NH
"Albany" "Providence" "Hartford" "Boston" "Montpelier" "Concord"
      ME
"Augusta"
```

```
capitals["MA"]
```

```
      MA
"Boston"
```

```
names(capitals)
```

```
[1] "NY" "RI" "CT" "MA" "VT" "NH" "ME"
```

```
# Method 2
```

```
phonecode <- c("801"="SLC", "206"="Seattle", "307"="Wyoming")
```

```
phonecode
```

```
      801      206      307
"SLC" "Seattle" "Wyoming"
```

```
phonecode["801"]
```

```
      801
"SLC"
```

- Dissociate the 2 vectors

```
names(capitals) <- NULL
```

```
capitals
```

```
[1] "Albany"      "Providence" "Hartford"    "Boston"      "Montpelier"
[6] "Concord"     "Augusta"
```

### 3.5 NA (Not Available values)

- **NA**: stands for 'Not Available'/Missing values and has a length of 1.  
There are in essence 4 versions depending on the type:
  - **NA** (logical - **default**)
  - **NA\_integer** (integer)
  - **NA\_real** (double precision)
  - **NA\_character** (string)

Under the hood, the version of NA is subjected to **coercion**:  
*logical* → *integer* → *double* → *character*

- some functions e.g. **mean()** return (by default) NA if 1 or more instances NA are present in a vector.
- **is.na()**: test a vector (element-wise) for NA values.

**Do NOT use:**

```
x == NA
```

**but use INSTEAD:**

```
is.na(x)
```

#### 3.5.1 Examples

- Types of NA

```
x <- NA
typeof(x)
```

```
[1] "logical"
```

```
# logical NA coerced to double precision NA
x <- c(3.0, 5.0, NA)
typeof(x[3])
```

```
[1] "double"
```

\* Functions on a vector containing NA

```
mean(x)
```

```
[1] NA
```

```
mean(x, na.rm=TRUE)
```

```
[1] 4
```

\* Check of the NA availability

```
x <- c(NA, 1, 2, NA)
is.na(x)
```

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

\* Functions on a vector containing NA

```
mean(x)
```

```
[1] NA
```

```
mean(x, na.rm=TRUE)
```

```
[1] 1.5
```

### 3.5.2 Exercises

- A family has installed a device to monitor their daily energy consumption (in kWh). When a measurement fails or is unavailable NA is recorded.

You can invoke the following code to generate the measurements generated by the device.

```
dailyusage <- 30.0 + runif(365, min=0, max=5.0)
dailyusage[sample(1:365, sample(1:50,1), replace=FALSE)] <- NA
```

- How many measurements failed?
- What is the average daily energy consumption (based on the non-failed) measurements?

## 3.6 NaN and infinities

- **NaN** (only for numeric types!), and the infinities **Inf** and **-Inf** are part of the **IEEE 754 floating-point standard**.
- To test whether you have:
  - finite numbers: use **is.finite()**
  - infinite numbers: use **is.infinite()**
  - NaNs: use **is.nan()**
- Further:
  - a **NaN** will return **TRUE** only when tested by **is.nan()**
  - a **NA** will return **TRUE** when tested by either **is.nan()** or **is.na()**

### 3.6.1 Examples

- Infinities:

```
x <- 5.0/0.0
x
```

```
[1] Inf
```

```
is.finite(x)
```

```
[1] FALSE
```

```
is.infinite(x)
```

```
[1] TRUE
```

```
is.nan(x)
```

```
[1] FALSE
```

```
y <- -5.0/0.0
y
```

```
[1] -Inf
```

```
is.finite(y)
```

```
[1] FALSE
```

```
is.infinite(y)
```

```
[1] TRUE
```

```
is.nan(y)
```

```
[1] FALSE
```

```
z <- x + y
z
```

```
[1] NaN
```

```
typeof(z)
```

```
[1] "double"
```

```
is.finite(z)
```

```
[1] FALSE
```

```
is.infinite(z)
```

```
[1] FALSE
```

```
is.nan(z)
```

```
[1] TRUE
```

- `is.na()` vs. `is.nan()`:

```
# is.nan
v <- c(NA, z, 5.0, log(-1.0))
```

```
Warning in log(-1): NaNs produced
```

```
is.nan(v)
```

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

```
# is.na(): also includes NaN!
v <- c(NA, z, 5.0, log(-1.0))
```

```
Warning in log(-1): NaNs produced
```

```
is.na(v)
```

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

### 3.7 Note on logical operators

- `&`, `|`, `!`, `xor()`: **element-wise** operators on vectors (cfr. arithmetic operators)
- `&&`, `||`: evaluated from **left** to **right** until result is determined.

#### 3.7.1 Examples

- Vector operators (`&`, `|`, `!` and `xor()`)

```
x <- sample(x=1:10, size=10, replace=TRUE)
```

```
x
```

```
[1] 2 3 5 7 9 9 4 3 2 7
```

```
y <- sample(x=1:10, size=10, replace=TRUE)
```

```
y
```

```
[1] 10 2 1 3 5 1 4 2 7 2
```

```
v1 <- (x<=3)
```

```
v1
```

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

```
v2 <- (y>=7)
```

```
v2
```

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

```
v1 & v2
```

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

```
v1 | v2
```

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

```
xor(v1, v2)
```

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

```
!v1
```

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

### 3.7.2 Exercises

- Generate a random vector of integers using the following code:

```
x <- sample(x=0:1000,size=100, replace=TRUE)
```

- Invoke the above code to generate the vector **x**
- Find if there are any integers in the vector **x** which can be divided by 4 and 6
- Find those numbers and their corresponding indices in the vector **x**.