Fondamenti di R

Tags

Fondamenti

I commenti in R sono preceduti da un #

```
# questo è un commento
```

Operatori matematici semplici:

```
# An addition
5 + 5
# A subtraction
5 - 5
# A multiplication
3 * 5
# A division
(5 + 5) / 2
# Exponentiation
2 ^ 5
# Modulo
28 %% 6
```

Le variabili in R sembrano essere weakly typed

```
# value 42 assigned to the x variable
x <- 42
```

Possiamo stampare il contenuto di una variabile nella console digitando una espressione senza assegnamento

```
my_apples
my_oranges + my_apples
```

R **non** fa una conversione implicita sommando un numeric e un text

```
# won't work
my_fruit <- 6 + "six"
```

Tipi base in R sono:

- **numeric** ⇒ 4.5
- **integer** ⇒ 4
- **logical** ⇒ TRUE/FALSE
- **characters** ⇒ “any string”

Tramite la funzione `class()` è possibile controllare il tipo di una variabile

```
class(my_numeric)
# > [1] "logical"
```

La funzione `summary()` permette di ottenere un sommario delle informazioni contenute in una qualsiasi variabile

```
survey_vector <- c("M", "F", "F", "M", "M")
factor_survey_vector <- factor(survey_vector)
levels(factor_survey_vector) <- c("Female", "Male")

summary(survey_vector)
#    Length     Class      Mode
#         5 character character

summary(factor_survey_vector)
# Female   Male
#      2      3
```

Gli operatori di logica booleana pre-esistenti in sono:

- `&` : Element-wise AND
- `&&` : Logical AND (only first element).
  Se utilizzato su vettori controlla solo il primo elemento del vettore
- `|` : Element-wise OR
- `||` : Logical OR (only first element)
  Se utilizzato su vettori controlla solo il primo elemento del vettore
- `!` : NOT

La funzione `runif(x, min, max)` permette di generare un vettore casuale di `x` elementi tra `min` e `max` . se `min` e `max` non sono definiti il range di default è [0, 1]

Strutture Dati

Vettori

Creazione

In R è possibile creare un vettore tramite l'utilizzo della funzione combine `c()`

```
my_numeric_vector <- c(3, 4, 5, 6)
```

è possibile creare un vettore come composizione di altri vettori

```
A <- c(1, 2, 3)
B <- c(4, 5, 6)
C <- c(A, B)
C
# > [1] 1  2  3  4  5  6
```

Nomi

è possibile associare delle label(*nomi*) ai campi di un vettore utilizzando la funzione `names()`.

Se i nomi associati al vettore non sono stati inizializzati, la funzione `names()` ritornerà `NULL`

```
some_vector <- c("John Doe", "poker player")
names(some_vector) <- c("Name", "Profession")
```

Operatori

La somma tra due vettori esegue la ***somma "elemento per elemento"***

```
A <- c(1, 1, 1)
B <- c(2, 2, 2)
A + B
#  > [1] 3  3  3
```

La funzione `sum()` permette di ritornare la somma di tutti gli elementi di un vettore

La funzione `mean()` la media ...

```
A <- c(1, 1, 1, 1)
sum(A)
# > [1] 4
mean(A)
# [1] 1
```

Indicizzazione

In R l'indicizzazione dei vettori avviene tramite le quadre `[]` .

L'indicizzazione parte da 1

```
A <- c(1, 2, 3)
names(A) <- c("Primo", "Secondo", "Terzo")
A[1]
# > Primo
#       1
```

è possibile utilizzare un vettore per indicizzare un sottoinsieme di elementi del vettore

```
A <- c("A", "B", "c", "d")
A[c(1, 3, 4)]
# > A  c  d
```

è possibile ottenere lo stesso risultato utilizzando l'operatore di abbreviazione `:`

```
A <- c("A", "B", "c", "d")
A[1:3]
# > A  B  c
```

è possibile indicizzare anche usando i nomi (*se sono inizializzati per il vettore, altrimenti ritorna* `NA` *per ogni elemento*)

```
A <- c(1, 2, 3)
names(A) <- c("Primo", "Secondo", "Terzo")
A[c("Primo", "Secondo")]
# > Primo  Secondo
#       1        2
```

Se indicizziamo un vettore con un vettore di **logical** della stessa dimensione, R ritornerà solo gli elementi per il quale il logical era `TRUE`

```
A <- c(1, 2, 3)
A[c(TRUE, TRUE, FALSE)]
# > [1] 1  2
```

Confronti

è possibile usare gli operatori di confronto anche tra vettori (*eseguono il confronto "elemento per elemento"*)

```
c(4, 5, 6) > 5
# > [1] FALSE FALSE TRUE
```

Sorting

Per ordinare un vettore è possibile utilizzare la funzione `order()` che preso in input un vettore ritorna un nuovo vettore contenente il ranking di ogni elemento nel vettore originale.

Applicando il vettore ottenuto come indice è possibile ritornare il vettore originale ordinato

```
a <- c(100, 10, 1000)
order(a)
# > [1] 2 1 3

a[order(a)]
# > [1]   10  100 1000
```

Alternativamente è possibile utilizzare la funzione `sort()` che ordina direttamente.

```
v <- c(3, 1, 5, 2)
sorted_v <- sort(v)
```

Con il parametro `decreasing` è possibile ordinare il vettore in modo discendente

```
v <- c(3, 1, 5, 2)
sorted_v <- sort(v, decreasing = TRUE)
sorted_v <- v[order(v, decreasing = TRUE)]
```

▼ Matrici

Costruzione

è possibile creare una matrice utilizzando la funzione `matrix()` .

Essa prende 3 parametri

- La collezione di elementi da inserire all'interno della matrice (*un vettore*)
- `byrow`: indica se la matrice sarà riempita per riga (*TRUE*) o per colonna (*FALSE*)
- `nrow`: indica il numero di righe che avrà la matrice

```
matrix(1:9, byrow = TRUE, nrow = 3)
# costruisce una matrice 3x3 riempita per riga
```

Se proviamo ad inizializzare una matrice con un numero di righe e di elementi che non matchano, R riempirà la matrice ciclando il vettore di elementi

```
matrix(1:10, byrow = TRUE, nrow = 3)
#      [,1] [,2] [,3] [,4]
# [1,]    1    2    3    4
# [2,]    5    6    7    8
# [3,]    9   10    1    2
# Warning message:
```

```
# In matrix(1:10, byrow = TRUE, nrow = 3) :
#  data length [10] is not a sub-multiple or multiple of the number of rows [3]
```

Manipolazione matrice

Tramite la funzione `cbind()` è possibile aggiungere 1 o più colonne ad una matrice

```
A <- matrix(c(1,1,1,1,1,1,1,1,1), byrow = TRUE, nrow = 3)
B <- rowSums(A)
C <- cbind(A, B)
C
#      B
# [1,] 1 1 1 3
# [2,] 1 1 1 3
# [3,] 1 1 1 3
```

Allo stesso modo la funzione `rbind()` permette di aggiungere 1 o più righe ad una matrice

```
A <- matrix(c(1,1,1,1,1,1), byrow = TRUE, nrow = 2)
B <- matrix(c(3,3,3,3,3,3), byrow = TRUE, nrow = 2)
C <- rbind(A, B)
C
#      [,1] [,2] [,3]
# [1,]    1    1    1
# [2,]    1    1    1
# [4,]    3    3    3
# [5,]    3    3    3
```

Nomi

Tramite le funzioni `colnames()` e `rownames()` (*che prendono in input dei vettori*) è possibile nominare le righe e colonne di matrici

```
rownames(my_matrix) <- row_names_vector
colnames(my_matrix) <- col_names_vector
```

Operatori

Tramite le funzioni `rowSums()`, `rowMeans()`, ... è possibile effettuare l'operazione per tutte le celle per ogni riga e ritornare un nuovo vettore che contiene il risultato per ogni riga

```
A <- matrix(c(1,1,1,1,1,1,1,1,1), byrow = TRUE, nrow = 3)
rowSums(A)
# > [1] 3  3  3
```

Allo stesso modo è possibile fare la stessa operazione per le colonne con la funzione `colSums()`, `colMeans()`, ...

```
A <- matrix(c(1,1,1,1,1,1), byrow = TRUE, ncol = 2)
colSums(A)
# > [1] 3  3
```

è possibile eseguire tutti gli operatori primari tra matrici, ottenendo l'applicazione dell'operazione "elemento per elemento" tra mat1 e mat2.

```
A <- matrix(1:6, byrow = TRUE, nrow = 2)
B <- matrix(c(3,3,3,3,3,3), byrow = TRUE, nrow = 2)
A * B
#      [,1] [,2] [,3]
# [1,]    3    6    9
# [2,]   12   15   18
```

Tramite l'operatore `%*%` è possibile eseguire la classica moltiplicazione "righe per colonne" tra matrici

```
A <- matrix(1:6, byrow = TRUE, nrow = 2)
B <- matrix(c(3,3,3,3,3,3), byrow = TRUE, nrow = 3)
A %*% B
#      [,1] [,2]
# [1,]   18   18
# [2,]   45   45
```

Indicizzazione

L'indicizzazione per le matrici è identica a quelle per i vettori, con la sola differenza che si applica su 2 dimensioni, usando quindi l'operatore `[,]`.

Per selezionare tutti gli elementi di una riga o una colonna si omette quel parametro.

```
my_matrix[1,2] #Second element of the first row
my_matrix[1:3,2:4] #sub-matrix
my_matrix[,1] #All elements of the first column.
my_matrix[1,] #All elements of the first row
```

▼ Factors

I fattori sono una tipologia speciale di dati usate per contenere variabili categoriche.

Una variabile categorica può appartenere ad un insieme limitato di categorie.

R tratta le variabili continue e quelle categoriche in modo differente.

Un esempio di variabile categorica è il sesso (biologico), che è possibile limitare alle sole categorie "Maschio" e "Femmina"

Creazione

Per creare i fattori in R si fa uso della funzione `factor()` che prende in input un vettore.

Il fattore contiene internamente in campo ***Levels*** che contiene le possibili categorie del fattore

```
sex_vector <- c("Male", "Female", "Female", "Male", "Male")
factor_sex_vector <- factor(sex_vector)
factor_sex_vector
# [1] Male   Female Female Male   Male
# Levels: Female Male
```

Tramite il parametro `levels` è possibile esplicitare direttamente le possibili categorie del fattore, e tramite il parametro `order`/`ordered` è possibile definire se esiste una relazione d'ordine tra gli elementi (*scandita dall'ordine di inserimento nel parametro* `levels`)

```
temperature_vector <- c("High", "Low", "High","Low", "Medium")
factor_temperature_vector <- factor(temperature_vector,
                                                                order = TRU
                                                                levels = c(
```

```
factor_temperature_vector
# [1] High   Low    High   Low    Medium
# Levels: Low < Medium < High
```

Modificare il fattore

tramite la funzione `levels()` è possibile modificare i livelli di un fattore. **l'ordine è importante**

```
survey_vector <- c("M", "F", "F", "M", "M")
factor_survey_vector <- factor(survey_vector) # levels are: "F"  "M"
levels(factor_survey_vector) <- c("Female", "Male")
```

Confronto

Non è possibile confrontare 2 valori di fattori se non è stata definita una relazione d'ordine tra i livelli.

Indicizzazione

Come per vettori e matrici è possibile indicizzare i valori contenuti nel vattore utilizzando l'operatore `[]`

▼ Data Frames

Un data frame è una variabile che ha le variabili di un dataset come colonne e i campioni come righe.

è una struttura dati che permette di manipolare dati che fanno uso di più tipi di dati assieme (*es: un questionario con risposte chiuse e aperte*)

Manipolare un data frame

Tramite le funzioni `head()` e `tail()` è possibile stampare il priomo/ultimo campione di un data frame in aggiunta ad una riga che contiene gli "*headers*"

La funzione `str()` permette di ispezionare la struttura di un data frame, fornendo informazioni del tipo:

- Numero di campioni (e.g. 32 car types)
- Numero totale di variabili (e.g. 11 car features)
- Lista dei nomi delle variabili (e.g. `mpg`, `cyl` ...)
- Il tipo di ogni variabile (e.g. `num`)
- I primi campioni

Creazione

Tramite la funzione `data.frame()` è possibile creare un nuovo data frame che utilizza come colonne (*e relativi valori delle celle*) i vettori passati in input

```
name <- c("Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn",
          "Uranus", "Neptune")
type <- c("Terrestrial planet", "Terrestrial planet", "Terrestrial planet",
          "Terrestrial planet", "Gas giant", "Gas giant", "Gas giant", "Gas giant")
diameter <- c(0.382, 0.949, 1, 0.532, 11.209, 9.449, 4.007, 3.883)
rotation <- c(58.64, -243.02, 1, 1.03, 0.41, 0.43, -0.72, 0.67)
rings <- c(FALSE, FALSE, FALSE, FALSE, TRUE, TRUE, TRUE, TRUE)

planets_df <- data.frame(name, type, diameter, rotation, rings)
```

Indicizzazione

Per indicizzare un data frame utilizziamo le stesse tecniche ed operatore delle matrici `[,]`

é inoltre possibile utilizzare l'operatore `$` per selezionare un'intera colonna per nome in rapidità

```
planets_df$diameter
# è equivalente a planets_df[,3] oppure planets_df[,"diameter"]
```

tramite la funzione `subset()` è possibile mostrare una proiezione del data frame applicando dei filtri (*una espressione o un vettore di logical*) che selezionano una porzione dei campioni

```
subset(planets_df, subset = rings)
subset(planets_df, subset = diameter < 1)
subset(planets_df, subset = (diameter < 1 & rotation > 1))
```

▼ Liste

In R le liste sono delle collezioni multi-tipo che possono contenere qualsiasi elemento (*numeri, vettori, matrici, data frames, ...*)

Creazione

Per creare una lista si fa uso della funzione `list()` che prende come input gli elementi da inserire nella lista

```
my_list <- list(my_vector, my_matrix, my_df, 5, "test")



scores <- c(4.6, 5, 4.8, 5, 4.2)
comments <- c("I would watch it again", "Amazing!", "I liked it", "One of the best

avg_review <- mean(scores)
reviews_df <- data.frame(scores, comments)

departed_list <- list(movie_title = "The Departed",
                                            movie_actors = c("Jack Nicholson", "Leo
                                            reviews_df = reviews_df,
                                            avg_review = avg_review)
```

Nomi

è possibile nominare i componenti di una lista facendo uso della funzione `names()` oppure assegnando una label direttamente durante la creazione

```
my_list <- list(name1 = your_comp1,
                name2 = your_comp2)

my_list <- list(your_comp1, your_comp2)
names(my_list) <- c("name1", "name2")
```

Indicizzazione

Si utilizza l'operatore `[[]]` per recuperare un elemento da una lista, oppure `$` se i campi hanno un nome.

▼ Date

In R le date sono gestite tramite oggetti di tipo `Date` mentre il tempo con oggetti di tipo `POSIXct`

- `Date` gestisce le date come il numero di giorni passati dal 1 gennaio 1970
- `POSIXct` gestisce il tempo come il numero di secondi passati dal 1 gennaio 1970

Per ottenere il giorno corrente facciamo uso di `Sys` e il metodo `Date()`. Allo stesso modo usiamo `Sys.time()` per ottenere il tempo attuale

```
today <- Sys.Date()
now <- Sys.time()
```

Creazione e formato date

Per creare una data da una stringa è possibile usare il cast `as.Date()` passandogli la stringa e il formato tramite il parametro `format`

I parametri utilizzabili per il formato sono:

- `%Y`: 4-digit year (1982)
- `%y`: 2-digit year (82)
- `%m`: 2-digit month (01)
- `%d`: 2-digit day of the month (13)
- `%A`: weekday (Wednesday)
- `%a`: abbreviated weekday (Wed)
- `%B`: month (January)
- `%b`: abbreviated month (Jan)

```
as.Date("1982-01-13") #il formto di default è "%Y-%m-%d" o "%Y/%m/%d"
as.Date("Jan-13-82", format = "%b-%d-%y")
as.Date("13 January, 1982", format = "%d %B, %Y")
```

Invece per ottenere una stringa equivalente da una data facciamo uso del metodo `format()`

```
today <- Sys.Date()
format(Sys.Date(), format = "%d %B, %Y")
format(Sys.Date(), format = "Today is a %A!")
```

Creazione e formato tempi

Per creare un tempo da una stringa è possibile usare il cast `as.POSIXct()` passandogli la stringa e il formato tramite il parametro `format`

I parametri utilizzabili per il formato sono:

- `%H`: hours as a decimal number (00-23)
- `%I`: hours as a decimal number (01-12)
- `%M`: minutes as a decimal number
- `%S`: seconds as a decimal number
- `%T`: shorthand notation for the typical format `%H:%M:%S`
- `%p`: AM/PM indicator

```
time1 <- as.POSIXct(str1, format = "%B %d, '%y hours:%H minutes:%M seconds:%S")
time2 <- as.POSIXct(str2, format = "%Y-%m-%d %H:%M:%S")
```

Invece per ottenere una stringa equivalente da un tempo facciamo uso del metodo `format()`

```
time <- Sys.time()
format(time, "%M")
format(time, "%I:%M %p")
```

Operazioni su date e tempi

Sia per `Date` che per `POSIXct` R esegue le operazioni usando la rappresentazione numerica intrinseca dell'oggetto. Questo rende la loro manipolazione molto semplice e permette l'uso di operatori matematici

```
today <- Sys.Date()
today + 1
today - 1
as.Date("2015-03-12") - as.Date("2015-02-27")

now <- Sys.time()
now + 3600          # add an hour
now - 3600 * 24     # subtract a day
birth <- as.POSIXct("1879-03-14 14:37:23")
death <- as.POSIXct("1955-04-18 03:47:12")
einstein <- death - birth
einstein
```

▼ Objects

Gli oggetti in R possono essere rappresentati da `vectors` o da `NULL`.

> 💡 In realtà tutte le strutture dati in R vengono rappresentate da `vector` o da `NULL`

Gli oggetti sono quindi dei vettori che hanno delle classi aggiuntive (*controllabili utilizzando la funzione* `unclass()` *e* `class()`)

```
tbl <- tibble::tibble()
class(tbl)
# [1] "tbl_df"   "tbl"   "data.frame"
```

Le classi sono in una relazione di ereditarietà da sinistra verso destra. (*"tbl" è una sottoclasse di "data.frame"*)

Creazione

Quelle che generalmente chiamiamo oggetti, in R sono gli oggetti S3 e S4.

S3

Sono semplici strutture dati già descritte in precedenza, ad esempio una lista, al quale andiamo a dare dei nomi ai valori e aggiungiamo una classe "custom".

```
s3 <- list(name = "John", age = 21, GPA = 3.5)
#Classe custom aggiunta
class(s3) <- "student"
```

è possibile definire un **costruttore** creando una funzione con lo stesso nome della classe che abbiamo creato. (*è più che altro una convenzione*)

```
student <- function(n, a, g) {
    if (g > 4 || g < 0) stop("GPA must be between 0 and 4")
    value <- list(name = n, age = g, GPA = g)
    # la classe puù essere assegnata usando class() o attr()
    attr(value, "class") <- "student"
    value
```

```
}

paul <- student("Paul", 26, 3.2)
```

S4

è un modo per creare una struttura dati più robusta e riutilizzabile, un template per creare oggetti di quella classe (*ciò che negli HLPL convenzionali chiamiamo classi*)

Usiamo la funzione `setClass()` per creare la classe. Il parametro `slots` è una lista nel quale possiamo definire quali campi appartengano alla classe

```
setClass("student",
                 slots = list(name = "character",
                                        age = "numeric",
                                        GPA = "numeric"))
```

Tramite la funzione `new()` è possibile creare oggetti di una determinata classe predefinita

```
s4 <-  new("student", name="John", age=21, GPA=3.5)
s4
```

Tramite la funzione `isS4()` è possibile capire con che tipo di classi stiamo lavorando su un determinato oggetto

Indicizzazione

Per accedere al contenuto di un campo in un oggetto di tipo S4 si usa l'operatore `@` alternativamente si può fare uso della funzione `slot()`

```
s4@GPA
s4@GPA <- 3.1
slot(s4, "GPA") <- 2.9
```

Metodi

Tramite la funzione `showMethods()` è possibile identificare quali funzioni sono definite per una determinata classe.

`isS4()` può essere utilizzato anche sulle funzioni per capire se sono state definite per oggetti di tipo S4

▼ Loops

While

Il loop while valuta una condizione booleana e termina quando la condizione diventa `FALSE`

```
speed <- 64
while (speed > 30) {
  print("Slow down!")
  speed <- speed - 7
}
```

Tramite la keyword `break` è possibile terminare il loop in modo anticipato. Tramite `next` è possibile andare direttamente alla prossima iterazione.

For

Ci sono due possibili sintassi per il for loop in R

```
# loop version 1
for (p in primes) {
  print(p)
}

# loop version 2
for (i in 1:length(primes)) {
  print(primes[i])
}
```

la funzione `length()` permette di ottenere la dimensione di una collezione

▼ Funzioni

R ha al suo interno molte funzioni di base, su queste è possibile ottenere un link alla documentazione direttamente all'interno del linguaggio tramite l'utilizzo della funzione `help()` o dell'operatore `?`

```
help(sample)
#oppure
?sample
```

La funzione `args()` invece permette di vedere velocemente gli argomenti di una funzione

```
args(sample)
```

Creazione di una funzione

Creare una funzione in R equivale ad assegnare un oggetto funzione ad una variabile. Dopo l'assegnamento la funzione sarà presente nel workspace e sarà richiamabile come tale.

```
my_fun <- function(arg1, arg2=default) {
  body
}

pow_two <- function(number) {
    number * number
}
pow_two(12)

sum_abs <- function(arg1, arg2) {
    abs(arg1) + abs(arg2)
}
sum_abs(-2, 3)

throw_dice <- function() {
  sample(1:6, size = 1)
}
throw_dice()
```

Le funzioni in R passano i propri **argomenti per valore**

Non è necessario definire un return type (*è weakly typed*). Per ritornare un valore si fa uso della funzione `return()` oppure lasciando come ultima riga la visualizzazione di un valore.

```
interpret <- function(num_views) {
  print("You're popular!")
  return(num_views)
```

```
}

interpret <- function(num_views) {
  print("You're popular!")
  num_views
}
```

▼ Pacchetti R

Essenzialmente ci sono 2 funzioni importanti per utilizzare i pacchetti in R:

- `install.packages()`, installa un determinato pacchetto (*ha bisogno dei permessi da admin*)

  ```
  install.packages("ggplot2")
  ```

- `library()`, carica i pacchetti allegandoli alla "*search list*" del workspace R.

  ```
  library("ggplot2")
  ```

Tramite la funzione `search()` è possibile controllare quali pacchetti R sono installati localmente

`library()` interrompe l'esecuzione se il pacchetto non è installato mentre la funzione `require()` ritorna `TRUE` se è riuscito a caricare il pacchetto, `FALSE` altrimenti

```
if (!require("BiocManager", quietly = TRUE))
    install.packages("BiocManager")
```

▼ Apply functions

Le funzioni `lapply()` , `sapply()` e `vapply()` sono implementazioni di operatori funzionali che permettono di applicare una determinata funzione ad una collezione.

`lapply(x, FUN, ...)`

La funzione `lapply` (*map in java, Select in c#*) prende una collezione `x` e applica la funzione `FUN` a tutti i suoi elementi. Se `FUN` richiede ulteriori argomenti, vengono passati dopo `FUN` (`...`). L'output è una lista della stessa dimensione di `x`, dove ogni elemento è il risultato di `FUN(x[i])`.

```
# The vector pioneers has already been created for you
pioneers <- c("GAUSS:1777", "BAYES:1702", "PASCAL:1623", "PEARSON:1857")
# Split names from birth year
split_math <- strsplit(pioneers, split = ":")
# Convert to lowercase strings: split_low
split_low <- lapply(split_math, tolower)
# Take a look at the structure of split_low
str(split_low)
```

Come in tutti gli altri linguaggi è possibile passare come argomento anche funzioni anonime e non solo funzioni già definite nel workspace

```
names <- lapply(split_low, function(x) { x[1] })
years <- lapply(split_low, function(x) { x[2] })
```

Quando sono presenti ulteriori argomenti vengono assegnati, usando la nomeclatura esplicita, dopo la funzione

```
select_el <- function(x, index) { x[index] }
names <- lapply(split_low, select_el, index = 1)
```

```
years <- lapply(split_low, function(x, index) { x[index] }, index = 2)
```

`sapply(x, FUN, ...)`

Funziona come `lapply()` ma cerca di semplificare la struttura dati ritornata, se possibile la trasforma in un vettore o matrice, altrimenti ritorna una lista.

```
random_temp_vector <- function() { runif(5, min = -20, max = 20) }
temp <- list(random_temp_vector(), random_temp_vector(), random_temp_vector(),
                        random_temp_vector(), random_temp_vector(), random_temp_vecto
                        random_temp_vector())

below_zero <- function(x) { x[x < 0] }
freezing_s <- sapply(temp, below_zero)
# in questo caso i vettori ritornati da below_zero hanno dimensioni diverse e
# sapply() non riesce a costruire una matrice, ritorna invece una lista
```

`vapply(x, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)`

Funziona come `sapply()` ma l'argomento `FUN.VALUE` si aspetta un template per il valore di ritorno di `FUN`. `USE.NAMES` prova a generare un array di nomi se possibile.

Nei casi in cui l'output generato non matchi la struttura passata tramite il parametro `FUN.VALUE` la funzione genera un errore.

```
basics <- function(x) { c(min=min(x), mean=mean(x), median=median(x), max=max(x)) }
vapply(temp, basics, numeric(4))
```

`vapply` risulta una versione più robusta di `sapply` che limita la struttura del risultato, è in generale preferibile (*in quanto maggiormente deterministica*) a `sapply`

▼ Utilities functions

- `abs()`: Calculate the absolute value.
- `sum()`: Calculate the sum of all the values in a data structure.
- `mean()`: Calculate the arithmetic mean.
- `round()`: Round the values to 0 decimal places by default. Try out `?round` in the console for variations of `round()` and ways to change the number of digits to round to.
- `seq()`: Generate sequences, by specifying the `from`, `to`, and `by` arguments.
- `rep()`: Replicate elements of vectors and lists.
- `sort()`: Sort a vector in ascending order. Works on numerics, but also on character strings and logicals.
- `rev()`: Reverse the elements in a data structures for which reversal is defined.
- `str()`: Display the structure of any R object.
- `append()`: Merge vectors or lists.
- `is.*()`: Check for the class of an R object.
- `as.*()`: Convert an R object from one class to another.
- `unlist()`: Flatten (possibly embedded) lists to produce a vector.
- `grepl()`, which returns `TRUE` when a pattern is found in the corresponding character string.
- `grep()`, which returns a vector of indices of the character strings that contains the pattern.
- `sub()`
- `gsub()` come grep ma permettono di sostituire i match con una stringa passata al parametro `replacement`