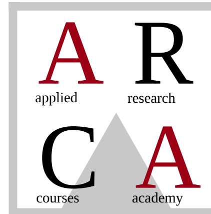
Introduzione a R

Strutture Dati



ARCA - @DPSS

Filippo Gambarota

Recap sulle tipologie di dati

Strutture dati

Strutture dati

- Finora abbiamo visto oggetti semplici, tuttavia poter creare, manipolare e gestire **strutture dati complesse** è fondamentale in R.

- Le strutture dati sono modalità di **immagazzinare diverse informazioni** con una certa logica e utile per eseguire altre operazioni complesse.

Strutture dati

Esiste una struttura dati che abbiamo sempre utilizzato. Quale? 🤔

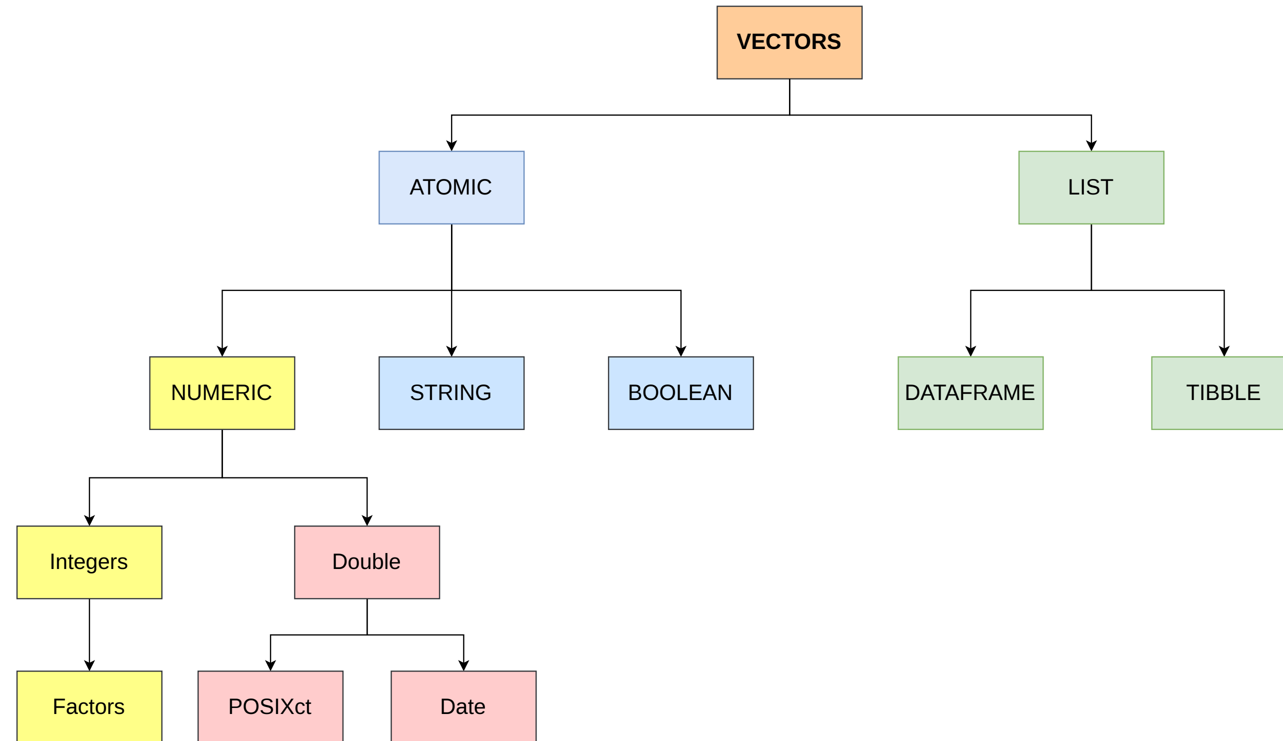
	A	B	C	D	E
1	id	nome	professione		
2	1	Filippo	studente		
3	2	Andrea	lavoratore		
4	3	Francesco	studente		
5	4	Franco	studente		
6	5	Luca	lavoratore		
7	6	Filippo	studente		
8	7	Andrea	studente		
9	8	Francesco	lavoratore		
10	9	Franco	studente		
11	10	Luca	studente		
12	11	Filippo	lavoratore		
13	12	Andrea	studente		
14	13	Francesco	studente		
15	14	Franco	lavoratore		
16	15	Luca	studente		
17					
18					

Strutture dati in R

In R sono presenti diverse strutture dati di diversa complessità:

- Vettori
- Fattori
- Matrici
- Dataframe
- Liste
- ...

The big picture



Attributi

Gli oggetti in R possiedono degli **attributi** che forniscono informazioni aggiuntive:

- *nomi*: possiamo fornire delle etichette ad ogni elemento

- *dimensioni*: fornisce il numero di elementi per ogni dimensione nell'oggetto

Come affrontarle?

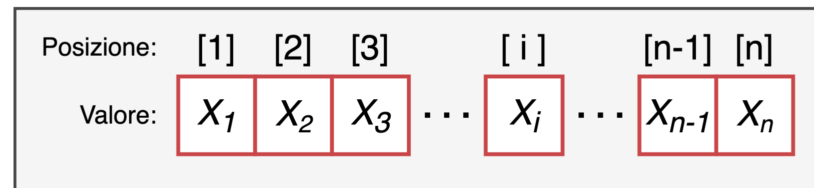
- Creazione
- Attributi
- Indicizzazione
- Manipolazione
- Operazioni (se rilevante)

Vettori (atomic)

Vettori (atomic)

I vettori (**atomic**) sono una struttura dati *unidimensionale* e sono la più semplice presente in R. Ci sono alcune cose importanti:

- Possono contenere informazioni di una sola *tipologia* (come numeri, stringhe, etc.)
- Essendo *unidimensionali* ogni elemento corrisponde ad una posizione
- Sono l'elemento fondante di strutture dati più complesse come matrici e dataframe



Vettori - creazione

Per creare un vettore si usa la funzione `c()` che sta per **concatenazione** e permette appunto di unire una serie di elementi:

```
my_int <- c(1:10) # vettore di interi
my_int
```

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

```
my_dbl <- c(1.2, 3.4, 5.5) # vettore di double
my_dbl
```

```
## [1] 1.2 3.4 5.5
```

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

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

```
my_chr <- c("ciao", "come", "stai")
my_chr
```

```
## [1] "ciao" "come" "stai"
```

```
class(my_int)
```

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

```
class(my_dbl)
```

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

```
class(my_lgl)
```

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

```
class(my_chr)
```

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

`is.*` e `as.*` family

Possiamo *testare* o *convertire* (quando possibile) la tipologia di un vettore usando le funzioni `is.*` e `as.*`:

```
is.integer(my_int)
```

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

```
as.integer(my_dbl)
```

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

```
as.integer(my_lgl) # cosa succede di strano?
```

```
## [1] 1 0 1
```

```
as.numeric(my_chr) # cosa succedere di strano?
```

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

```
## [1] NA NA NA
```

I valori `NA`

C'è una sola tipologia di dato che può coesistere in un vettore (atomic) e sono gli `NA` (**N**ot **A**pplicable) e rappresentano dei valori mancanti per varie ragioni:

```
my_int <- c(1,2,5,"ciao") # convertito a stringa
my_int
```

```
## [1] "1"    "2"    "5"    "ciao"
```

```
my_int <- c(1,2,3, NA) # NA viene convertito in numeric
my_int
```

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

Quello che succede in realtà è che ci sono altrettante tipologie di `NA` come `NA_logical_`, `NA_integer_` etc. e viene usato quello compatibile con il resto del vettore.

Vettori - Attributi

Gli attributi possibili per un vettore (atomic) sono:

- `names()`: ogni elemento può essere associato ad un nome/etichetta
- Dimensione (`dim()`), in realtà con il vettore unidimensionale usiamo `length()`

```r
x <- 1:10
names(x) # di default non ci sono nomi
```

```
## NULL
```

```r
names(x) <- letters[1:10] # assegnamo una lettera ad ogni posizione
setNames(x, letters[1:10]) # modo alternativo
```

```
##  a  b  c  d  e  f  g  h  i  j
##  1  2  3  4  5  6  7  8  9 10
```

```r
dim(x) # NULL
```

```
## NULL
```

```r
length(x) # ci fornisce la lunghezza
```

```
## [1] 10
```

Vettori - Indicizzazione

L'indicizzazione dei vettori è la più semplice essendo *unidimensionali*. Essendo l'unica proprietà la **lunghezza** (`length(vettore)`) possiamo selezionare, eliminare, estrarre elementi semplicemente usando l'**indice di posizione** tramite le parentesi quadre `vettore[pos]`

```
my_vec <- round(runif(20, 1, 100))
my_vec
```

```
##  [1] 45 49 57 16 46 64 96 65 85  4 38 93  8 38 85 72 19 61 13 88
```

```
my_vec[1] # estraggo il primo elemento
```

```
## [1] 45
```

```
my_vec[1:10] # estraggo i primi 10 elementi
```

```
##  [1] 45 49 57 16 46 64 96 65 85  4
```

```
my_vec[c(1,5,10,16)] # estraggo n elementi a varie posizioni
```

```
## [1] 45 46  4 72
```

```
my_vec[length(my_vec)] # estraggo l'ultimo elemento
```

```
## [1] 88
```

Vettori - Indicizzazione

Possiamo anche indicizzare (meno comune) con i nomi (se li abbiamo impostati come nella slide *attributi*):

```
x <- 1:10
names(x) <- letters[1:10]
x["a"]
```

```
## a
## 1
```

```
x[c("a", "b", "c")]
```

```
## a b c
## 1 2 3
```

In generale comunque non è molto comune usare i nomi per i vettori ma in alcuni casi può essere utile.

Vettori - Indicizzazione Logica

Indicizzare con la posizione è l'aspetto più semplice e intuitivo. E' possibile anche selezionare tramite valori `TRUE` e `FALSE`. L'idea è che se abbiamo un vettore di lunghezza n e un'altro vettore logico di lunghezza n, tutti gli elementi `TRUE` saranno selezionati:

```
my_vec <- 1:10
my_selection <- sample(rep(c(TRUE, FALSE), each = 5)) # random TRUE/FALSE
my_selection
```

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

```
my_vec[my_selection]
```

```
## [1] 3 4 5 7 8
```

Vettori - Indicizzazione Logica

Chiaramente non è pratico costruire a mano i vettori logici. Infatti possiamo usare delle *espressioni relazionali* per selezionare elementi:

```
my_vec <- 1:10
my_selection <- my_vec < 6
my_vec[my_selection]
```

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

```
my_vec[my_vec < 6] # in modo più compatto
```

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

Vettori - Indicizzazione Logica

Chiaramente possiamo usare **espressioni di qualsiasi complessità** perchè essenzialmente abbaimo bisogno di un vettore `TRUE/FALSE`:

```
my_vec <- 1:10
my_selection <- my_vec < 2 | my_vec > 8
my_vec[my_selection]
```

```
## [1]  1  9 10
```

```
my_vec[my_vec < 2 | my_vec > 8] # in modo più compatto
```

```
## [1]  1  9 10
```

Vettori - Indicizzazione Intera `which()`

La funzione `which()` è molto utile perchè restituisce la **posizione** associata ad una selezione logica:

```
my_vec <- rnorm(10)
which(my_vec < 0.5)
```

```
## [1]  2  3  5  6  7  8 10
```

```
# Questo

my_vec[which(my_vec < 0.5)]
```

```
## [1]  0.006188074 -1.626343723  0.395309710 -1.971237778 -0.337559317 -1.930926823
## [7] -1.785210705
```

```
# e questo sono equivalenti

my_vec[my_vec < 0.5]
```

```
## [1]  0.006188074 -1.626343723  0.395309710 -1.971237778 -0.337559317 -1.930926823
## [7] -1.785210705
```

Vettori - Indicizzazione Negativa `-`

Allo stesso modo di selezionare elementi con `[]`, indici di posizione e indici logici è possibile "rimuovere" degli elementi da un vettore, o in altri termini **non** selezionare alcuni elementi tramite il segno meno `-`:

```
my_vec <- 1:10

my_vec[-c(1,2)] # rimuovo i primi 2
```

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

```
my_vec[!my_vec > 5] # rimuovo i maggiori di 5, chiaramente uguale a selezionare i minori
```

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

Extra - `NA` values, OMG 😱

Gli `NA` sono dei valori tanto strani quanto utili per certi versi. Sono importanti da capire perchè possono impattare molte delle funzioni comuni in R:

```
myvec <- c(rnorm(100), NA)
mean(myvec)
```

```
## [1] NA
```

```
sd(myvec)
```

```
## [1] NA
```

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

```
## [1] 0.07038227
```

> Per approfondire ho scritto un documento sull'argomento https://arca-dpss.github.io/course-R/extra/dealing_with_NA_NaN_NULL.html

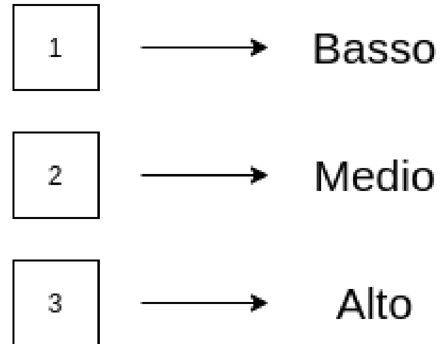
Fattori

Fattori

I fattori sono una tipologia di dato peculiare e per quanto simile a semplici `characters` in realtà sono un tipo di vettore `integer` con delle proprietà aggiuntive.

basso	medio	alto	medio	basso	alto	medio	basso	basso	Stringhe

1	2	3	2	1	3	2	1	1	Numeri interi



Attributi

Fattori - Creazione

I fattori si creano in modi diversi sia convertendo un vettore `character` con `as.factor()` che creando esplicitamente un fattore con `factor()`:

```
my_chr <- rep(c("a", "b", "c"), c(3, 4, 2))
as.factor(my_chr)
```

```
## [1] a a a b b b b c c
## Levels: a b c
```

```
my_fac <- factor(my_chr)
my_fac
```

```
## [1] a a a b b b b c c
## Levels: a b c
```

Fattori - Attributi

- I fattori fanno ampiamente uso degli attributi. In pratica assegnano un etichetta ad un valore numerico intero. Sono usati principalmente perchè sono più efficienti dei caratteri.

- I fattori quindi permettono di avere dei livelli `levels()` come metadati, a prescindere da quali sono effettivamente presenti nel vettore

```
typeof(my_fac)
```

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

```
attributes(my_fac)
```

```
## $levels
## [1] "a" "b" "c"
##
## $class
## [1] "factor"
```

```
# Notate la differenza tra
as.integer(my_fac)
```

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

```
# e
as.integer(my_chr)
```

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

```
## [1] NA NA NA NA NA NA NA NA NA
```

Fattori - Indicizzazione

L'indicizzazione è esattamente la stessa che per i vettori atomic (essendo i fattori degli `integers`) tuttavia anche la selezione preserverà i metadati:

```
my_fac[1]
```

```
## [1] a
## Levels: a b c
```

```
my_fac[1:5]
```

```
## [1] a a a b b
## Levels: a b c
```

Fattori Ordinali (extra)

Un sottotipo dei fattori sono gli `ordered factors` che corrispondo alle variabili ordinali, ovvero dove i livelli sono ordinati in modo crescente:

```
my_ord_fac <- as.ordered(my_fac)
my_ord_fac
```

```
## [1] a a a b b b b c c
## Levels: a < b < c
```

```
attributes(my_ord_fac)
```

```
## $levels
## [1] "a" "b" "c"
##
## $class
## [1] "ordered" "factor"
```

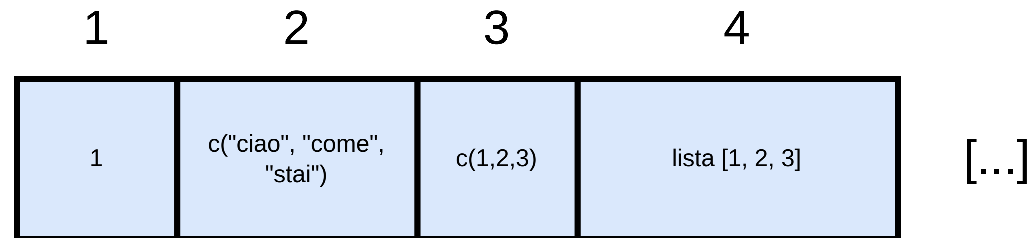
Alcuni modelli statistici o rappresentazioni grafiche sono applicabili a variabili ordinali e in R possono essere esplicitamente creati.

Liste

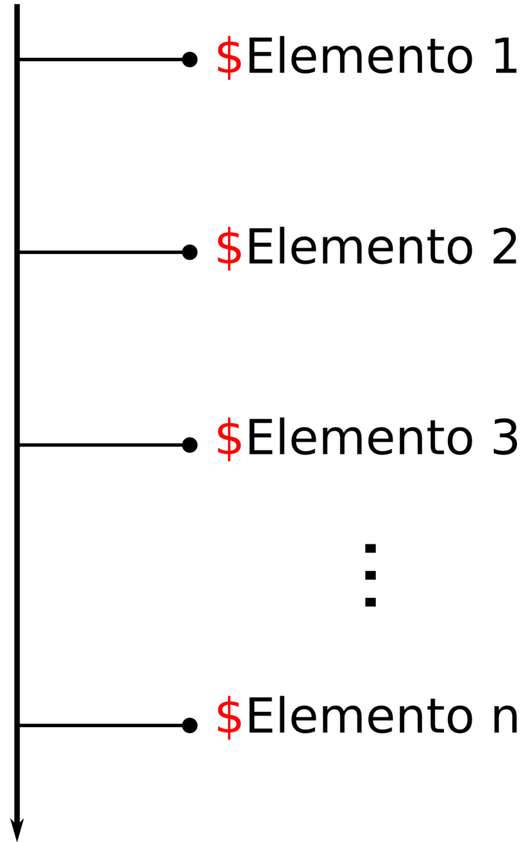
Liste

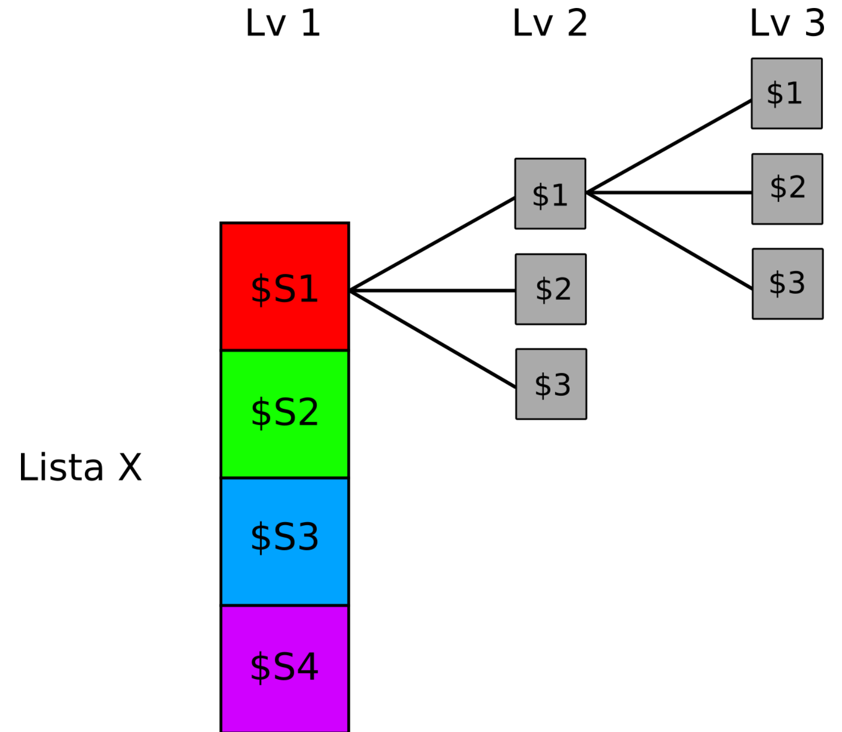
La lista è una generalizzazione dell' **atomic vector** dove:

- i **dati** possono essere di **diversa tipologia**
- ogni elemento può essere **a sua volta una lista**
- sono una struttura dati *unidimensionale* ma possono svilupparsi in **profondità** (ci arriviamo)



Liste





Liste - Creazione

Per creare una lista si può usare il comando list, specificando gli elementi ed eventualmente un nome associato ad ogni elemento:

```
my_list <- list(
    1:10,
    rep(c("a", "b", "c"), each = 3),
    my_fac
)

my_list
```

```
## [[1]]
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## [[2]]
## [1] "a" "a" "a" "b" "b" "b" "c" "c" "c"
##
## [[3]]
## [1] a a a b b b b c c
## Levels: a b c
```

Liste - Attributi

Come per i vettori anche le liste hanno una lunghezza (`lenght()`) ed eventualmente dei nomi (`names()`). Il comando `str()` (struttura) è molto utile per le liste perchè fornisce una visione sulla struttura:

```
names(my_list)
```

```
## NULL
```

```
length(my_list)
```

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

```
str(my_list)
```

```
## List of 3
##  $ : int [1:10] 1 2 3 4 5 6 7 8 9 10
##  $ : chr [1:9] "a" "a" "a" "b" ...
##  $ : Factor w/ 3 levels "a","b","c": 1 1 1 2 2 2 2 3 3
```

```
names(my_list) <- c("elemento1", "elemento2", "elemento3")
```

Liste - Indicizzazione

L'indicizzazione è più "complessa" ma molto simile ai vettori. Possiamo indicizzare con parentesi quadre `lista[n]` o con le doppie parentesi quadre `lista[[n]]`. Se a lista è una `named list` quindi con associati i nomi, possiamo usare `$` con il nome associato `lista$nome_elemento`

```r
my_list[1] # accedo al primo elemento COME LISTA
```

```
## $elemento1
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
my_list[[1]] # accedo al primo elemento
```

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

```r
my_list$elemento1 # accedo con il nome
```

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

```r
my_list["elemento1"]
```

```
## $elemento1
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
my_list[["elemento1"]]
```

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

Liste - Indicizzazione

La differenza tra le parentesi quadre riguarda il fatto se vogliamo fare un subset della lista ottenendo un'altra lista oppure se vogliamo accedere direttamente all'elemento interno.

Se vogliamo selezionare più elementi (quindi fare un vero e proprio subset della lista) dobbiamo sempre usare le paarentesi quadre singole:

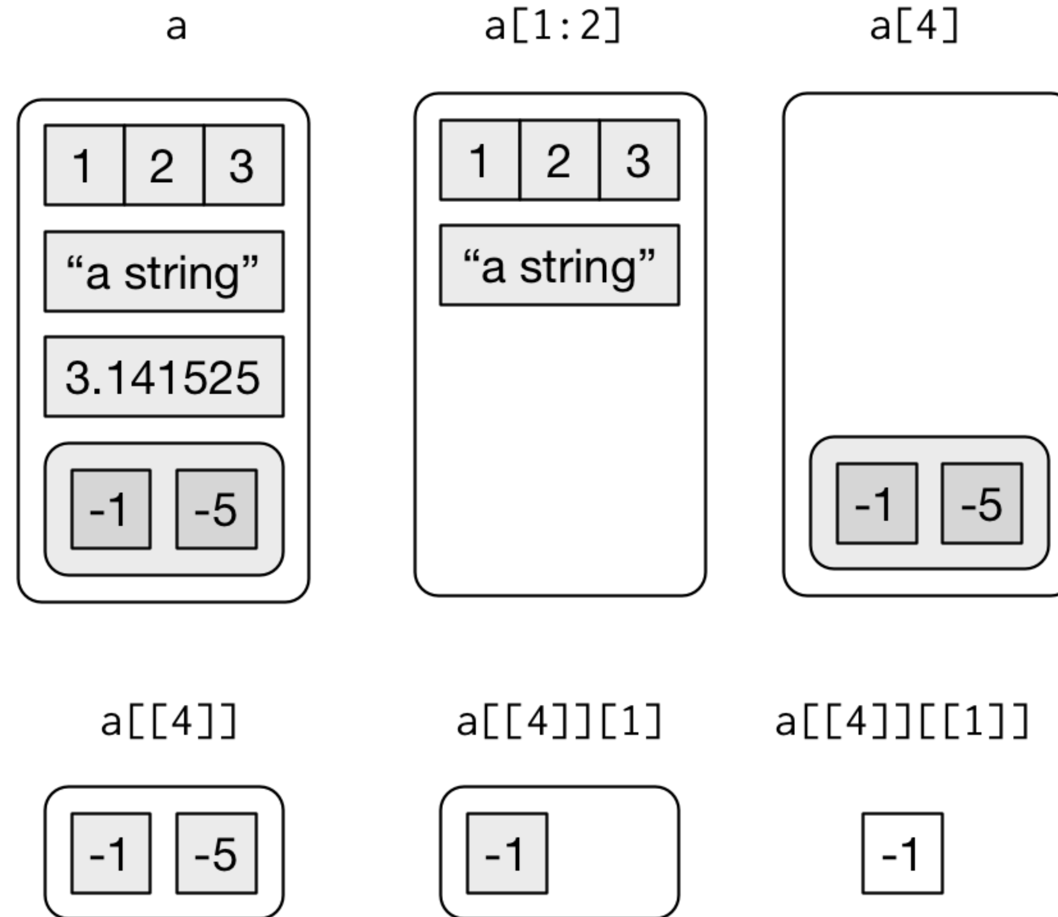
```
my_list[1:2]
```

```
## $elemento1
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $elemento2
## [1] "a" "a" "a" "b" "b" "b" "c" "c" "c"
```

```
my_list[[1:2]]
```

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

Liste - Indicizzazione



> Thanks to Hadley 😄 https://r4ds.had.co.nz/vectors.html#lists-of-condiments

Liste - Indicizzazione Nested

Come abbiamo visto le liste possono contenere altre liste, ottendo una struttura *unidimensionale* ma che si può sviluppare in profondità. Per selezionare elementi *nested* si possono concatenare più parentesi:

```
my_list <- list(1:10, letters[1:10], rnorm(10))
my_list <- list(my_list, rnorm(10), rnorm(10))
str(my_list)
```

```
## List of 3
##  $ :List of 3
##   ..$ : int [1:10] 1 2 3 4 5 6 7 8 9 10
##   ..$ : chr [1:10] "a" "b" "c" "d" ...
##   ..$ : num [1:10] -1.9484 1.9616 0.1679 0.8952 0.0601 ...
##  $ : num [1:10] -0.75742 -0.00756 0.94999 -0.13091 0.3617 ...
##  $ : num [1:10] -0.0602 -0.4641 -0.0623 0.0829 0.0184 ...
```

Liste - Indicizzazione Nested

```
my_list[[1]][[1]] # primo elemento della prima lista
```

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

```
my_list[[1]][[1]][[1]] # primo elemento del primo elemento della prima lista
```

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

C'è anche un modo meno intuitivo ma equivalente per indicizzare in modo ricorsivo:

```
my_list[[c(1,2)]] # equivalente a my_list[[1]][[2]]
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

Modificare elementi

Per modificare elementi posso usare l'approccio selezione e riassegnazione:

```
my_list[1] <- list(oggetto) # se uso le parentesi singole [] devo assegnare una lista
my_list[[1]] <- oggetto # se uso le doppie, riassegno direttamente l'oggetto
```

In generale valgono gli stessi principi dei vettori (atomics) quindi selezione negativa `lista[-index]` e selezione multipla `lista[1:n]`

Matrici

Matrici - Creazione

Le matrici sono una struttura dati *bidimensionale* caratterizzate da `righe` e `colonne` ovvero le dimensioni `dim()` dove il numero di righe rappresenta la dimensione 1 e il numero di colonne la dimensione 2. La matrice si crea con il comando `matrix(data, nrow, ncol, byrow =, dimnames=)`

```
# usare indici progressivi come dato è utile per capire l'argomento byrow

my_mat <- matrix(data = 1:9, ncol = 3, nrow = 3, byrow = FALSE)
my_mat
```

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

```
t(my_mat) # inverte righe e colonne, equivalente a mettere byrow = TRUE
```

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

```
dim(my_mat)
```

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

```
attributes(my_mat)
```

```
## $dim
## [1] 3 3
```

Matrici

Come per i vettori ci sono alcune cose rilevanti:

- Possono contenere una sola tipologia di dati
- Essendo bidimensionali, abbiamo bisogno di due indici di posizione (righe e colonne) per identificare un elemento
- Possono essere viste come un insieme di singoli vettori

Matrici - Creazione

Il numero di righe e colonne non deve essere lo stesso necessariamente (matrice quadrata) ma il numero di righe deve essere compatibile con il vettore `data`:

```
matrix(data = 1:100, ncol = 3, nrow = 3)
```

```
## Warning in matrix(data = 1:100, ncol = 3, nrow = 3): data length [100] is not a sub-
## multiple or multiple of the number of rows [3]
```

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

```
matrix(data = 1:9, ncol = 5, nrow = 5)
```

```
## Warning in matrix(data = 1:9, ncol = 5, nrow = 5): data length [9] is not a sub-multiple
## or multiple of the number of rows [5]
```

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

Cosa è successo alle matrici? E' un errore? Cosa fa R di default?

Matrici - Creazione

Tendenzialmente le matrici sono usate per calcolo e statistica e non è comune usare dei nomi per le colonne/righe (vedi `dataframe`) ma usando il comando `dimnames =` o `dimnames(matrix) <- list(rownames, colnames)`:

```
# Ci serve una lista con nomi di righe e colonne
dim_names <- list(
    c("row1", "row2", "row3"),
    c("col1", "col2", "col3")
)

my_mat <- matrix(data = 1:9, ncol = 3, nrow = 3, dimnames = dim_names)
dimnames(my_mat) <- dim_names
my_mat
```

```
##      col1 col2 col3
## row1    1    4    7
## row2    2    5    8
## row3    3    6    9
```

Matrici - Operazioni

Senza andare nei meandri dell'algebra, le matrici (come i vettori) possono essere usati per operazioni matematiche

```
my_mat + my_mat
```

```
##      col1 col2 col3
## row1    2    8   14
## row2    4   10   16
## row3    6   12   18
```

```
my_mat * my_mat
```

```
##      col1 col2 col3
## row1    1   16   49
## row2    4   25   64
## row3    9   36   81
```

```
my_mat %*% my_mat
```

```
##      col1 col2 col3
## row1   30   66  102
## row2   36   81  126
## row3   42   96  150
```

```
my_mat + 1
```

```
##      col1 col2 col3
## row1    2    5    8
## row2    3    6    9
## row3    4    7   10
```

Matrici - Indicizzazione

Anche l'indicizzazione è un'estensione di quella per i vettori adattata alle due dimensioni. Per identificare uno o più elementi nella matrice abbiamo bisogno di **indici/e di riga e/o colonna** separati da **virgola**, sempre con le parentesi quadre: `matrice[riga, colonna]`:

```
my_mat[1,1] # primo elemento della prima riga e colonna
```

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

```
my_mat[3,3]
```

```
## [1] 9
```

```
my_mat[1, ] # tutta la prima riga
```

```
## col1 col2 col3
##    1    4    7
```

```
my_mat[ ,1] # tutta la prima colonna
```

```
## row1 row2 row3
##    1    2    3
```

Matrici - Indicizzazione Logica

Come per i vettori anche la matrice può essere usata per operazioni *relazionali* ed essere indicizzata in modo logico:

```
my_mat > 4 # matrice logica
```

```
##       col1  col2 col3
## row1 FALSE FALSE TRUE
## row2 FALSE  TRUE TRUE
## row3 FALSE  TRUE TRUE
```

```
my_mat[my_mat > 4]
```

```
## [1] 5 6 7 8 9
```

Come vedete restituisce un vettore che rispetta la selezione, ma non una matrice.

Matrici non numeriche

Questa (abbastanza inutile) variante delle matrici è possibile perchè come per i vettori possiamo avere matrici logiche, di stringhe etc. ma dati diversi non possono coesistere:

```
my_mat > 3
```

```
##       col1 col2 col3
## row1 FALSE TRUE TRUE
## row2 FALSE TRUE TRUE
## row3 FALSE TRUE TRUE
```

```
is.logical(my_mat > 3)
```

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

```
matrix("R", nrow = 3, ncol = 3) # notate che se l'argomento data è un solo elemento questo viene riciclato
```

```
##      [,1] [,2] [,3]
## [1,] "R"  "R"  "R"
## [2,] "R"  "R"  "R"
## [3,] "R"  "R"  "R"
```

L'indicizzazione è la medesima mentre le operazioni (matematiche) sono possibili solo per matrici numeriche.

Matrici - Attributi

In parte lo abbiamo già visto ma le matrici hanno come attributi la dimensione `dim()` ovvero il numero di righe `nrow()` e il numero di colonne `ncol()`. Inoltre le dimensioni possono avere anche un nome `dimnames()`:

```
attributes(my_mat)
```

```
## $dim
## [1] 3 3
##
## $dimnames
## $dimnames[[1]]
## [1] "row1" "row2" "row3"
##
## $dimnames[[2]]
## [1] "col1" "col2" "col3"
```

```
dim(my_mat)
```

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

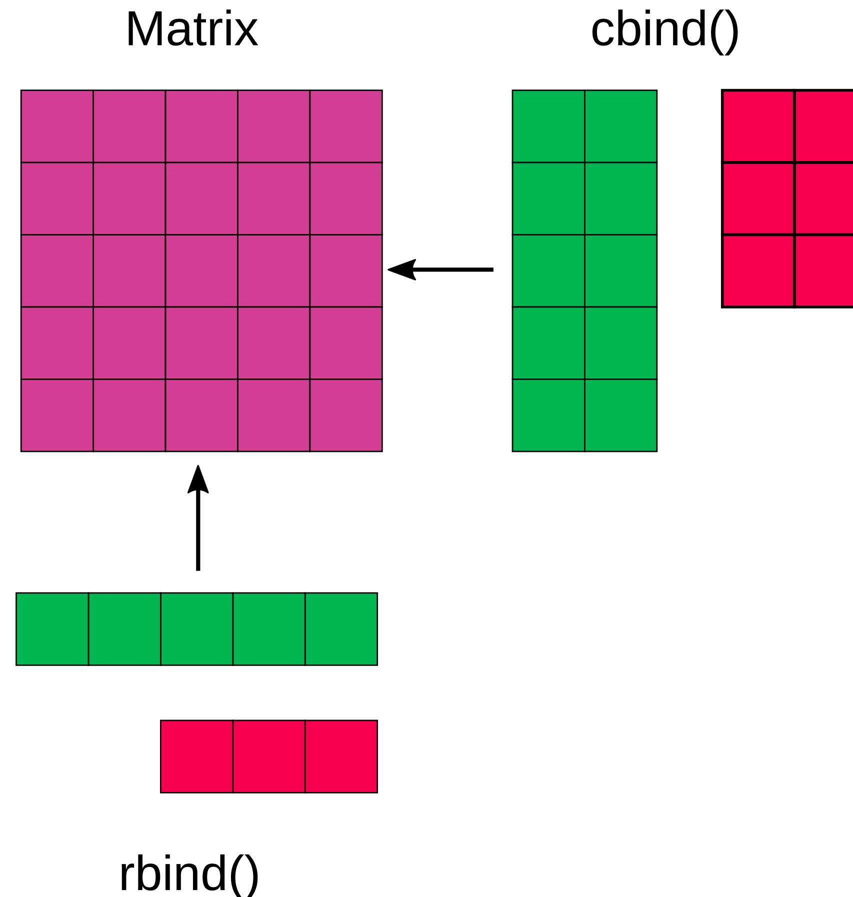
```
ncol(my_mat)
```

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

```
nrow(my_mat)
```

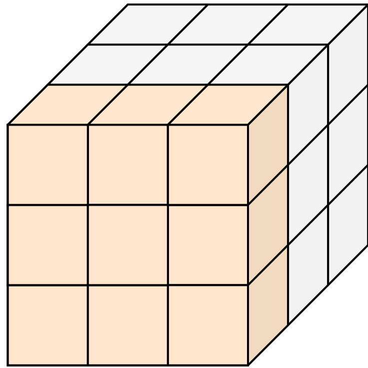
Matrici

Se per i vettori possiamo unire elementi con `c(1,2,3)` o unire vettori `c(vec1, vec2)` con le matrici possiamo usare i comandi `cbind()` e `rowbind()`:



Extra: Array

Gli array sono degli oggetti *n-dimensionali*. Se la matrice è un quadrato un array è un cubo. Valgono le stesse proprietà della matrice chiaramente scalate alle *n dimensioni*:



```r
my_array <- array(1:27, dim = c(3,3,3)) # esempio tridimensionale
my_array
```

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

Extra: Array

L'indicizzazione avviene allo stesso modo delle matrici aggiungendo una dimensione: `my_array[riga, colonna, dimensione]`

```
my_array[1,1,1] # prima riga, prima colonna, prima "fetta"
```

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

Anche gli attributi sono gli stessi ma chiaramente scalati su *n* dimensioni. Le dimensioni sono potenzialmente "infinite" ma aumenta anche la complessità e la praticità della struttura dati. Ad esempio un array a 4 dimensioni è un insieme di array a 3 dimensioni (e già diventa complesso).

Dataframe

Dataframe

Il dataframe è la struttura più "complessa", utile e potente di R. Da un punto di vista intuitivo è un *foglio excel* mentre da un punto di vista di R è una tipologia di lista con alcune caratteristiche/restrizioni[1]

- ogni elemento della lista è un **vettore con un nome associato** (aka una colonna)
- ogni **lista/colonna** deve avere lo **stesso numero di elementi**
- di conseguenza ogni **riga** ha lo **stesso numero di elementi** (struttura *rettangolare*)

[1] Advanced R - Hadley Wickam

Dataframe - Creazione

La creazione di un `dataframe` è molto simile alla lista tramite la funzione `data.frame(colonna_1, colonna_2, colonna_n)` dove ogni colonna è un vettore di uguale lunghezza.

```
my_df <-data.frame(
    colonna1 = 1:10,
    colonna2 = letters[1:10],
    colonna3 = rnorm(10),
    colonna4 = runif(10)
)
my_df
```

```
##    colonna1 colonna2   colonna3  colonna4
## 1         1        a  1.1130766 0.4750487
## 2         2        b  0.1263024 0.4096702
## 3         3        c -0.9829902 0.3685319
## 4         4        d  0.2618617 0.1915239
## 5         5        e -0.6573282 0.6239650
## 6         6        f -0.5027528 0.3740979
## 7         7        g  1.4884774 0.9714233
## 8         8        h -0.8613276 0.5984966
## 9         9        i -0.1493563 0.8672094
## 10       10        j -0.9855539 0.5696519
```

Dataframe - Attributi

Vediamo che il `dataframe` ha infatti sia gli attributi della `lista` ovvero i `names` ma anche gli attributi della `matrice` ovvero le dimensioni (righe e colonne):

```
typeof(my_df) # è una lista
```

```
## [1] "list"
```

```
attributes(my_df) # ma ha anche una classe dedicata
```

```
## $names
## [1] "colonna1" "colonna2" "colonna3" "colonna4"
##
## $class
## [1] "data.frame"
##
## $row.names
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
dim(my_df)
```

```
## [1] 10  4
```

```
nrow(my_df)
```

```
## [1] 10
```

Dataframe - Indicizzazione

Anche per l'indicizzazione il dataframe acquisisce le proprietà della `matrice` e della `lista` con nomi associati. In particolare posso usare le parentesi quadre `[]` oppure il simbolo del dollaro `$`:

```
# Indicizzazione come matrice
my_df[1,1]
```

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

```
my_df[1, ]
```

```
##   colonna1 colonna2 colonna3  colonna4
## 1        1        a 1.113077 0.4750487
```

```
my_df[, 1]
```

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

```
# Indicizzazione come matrice
my_df$colonna1
```

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

Dataframe - Indicizzazione

Per dimostrare che il dataframe è essenzialmente una matrice, possiamo usare la doppia parentesi quadra per estrarre una colonna e non `[riga,colonna]`:

```
my_df[[1]] # estraggo la prima colonna
```

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

```
my_df[["colonna1"]] # estraggo la prima colonna usando il nome
```

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

Dataframe - Indicizzazione

In generale, l'indicizzazione del dataframe è quella più complessa ed efficiente, sopratutto combinata con **operazioni relazionali**:

```
my_df[my_df$colonna1 > 4, ]
```

```
##    colonna1 colonna2   colonna3  colonna4
## 5         5        e -0.6573282 0.6239650
## 6         6        f -0.5027528 0.3740979
## 7         7        g  1.4884774 0.9714233
## 8         8        h -0.8613276 0.5984966
## 9         9        i -0.1493563 0.8672094
## 10       10        j -0.9855539 0.5696519
```

```
my_df[my_df$colonna1 > 4 & my_df$colonna4 > 0.1, ]
```

```
##    colonna1 colonna2   colonna3  colonna4
## 5         5        e -0.6573282 0.6239650
## 6         6        f -0.5027528 0.3740979
## 7         7        g  1.4884774 0.9714233
## 8         8        h -0.8613276 0.5984966
## 9         9        i -0.1493563 0.8672094
## 10       10        j -0.9855539 0.5696519
```

```
my_df[my_df$colonna1 > 4, "colonna4"]
```

```
## [1] 0.6239650 0.3740979 0.9714233 0.5984966 0.8672094 0.5696519
```

```
my_df[my_df$colonna1 > 4, 1]
```

```
## [1]  5  6  7  8  9 10
```

Dataframe - Indicizzazione

L'idea generale è semplice ovvero quando seleziono faccio operazioni sulle **righe** (seleziono alcune righe) ed eventualmente sulle **colonne** (seleziono alcune colonne). Attenzione però a cosa viene restituito!

```
typeof(my_df[1,1])
```

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

```
typeof(my_df[1, ])
```

```
## [1] "list"
```

```
typeof(my_df[, 1])
```

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

Viene restituito non sempre un dataframe, per evitare questo (se è necessario) possiamo usare `drop = FALSE` per ottenere sempre un dataframe:

```
typeof(my_df[, 1, drop = FALSE])
```

```
## [1] "list"
```

```
typeof(my_df[1,1, drop = FALSE])
```

```
## [1] "list"
```

Dataframe - Indicizzazione `subset()`

Ci sono anche dei modi alternativi e più compatti di indicizzare. Ad esempio usando la funzione `subset()`:

```
head(subset(iris, subset = Species == "setosa" & Sepal.Length > 4))
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

E' equivalente a:

```
head(iris[iris$Species == "setosa" & iris$Sepal.Length > 4, ])
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

Dataframe - Indicizzazione `subset()`

Possiamo anche selezione delle colonne invece che righe:

```r
# usando i nomi delle colonne
head(subset(iris, select = c(Sepal.Length, Species))) # equivale a iris[, c("Sepal.
```

```
##   Sepal.Length Species
## 1          5.1  setosa
## 2          4.9  setosa
## 3          4.7  setosa
## 4          4.6  setosa
## 5          5.0  setosa
## 6          5.4  setosa
```

```r
# oppure usando indici
head(subset(iris, select = c(1, 5)))
```

```
##   Sepal.Length Species
## 1          5.1  setosa
## 2          4.9  setosa
## 3          4.7  setosa
## 4          4.6  setosa
## 5          5.0  setosa
## 6          5.4  setosa
```

Dataframe - Indicizzazione `subset()`

Possiamo anche combinare le due cose facendo una selezione di righe e colonne:

```
head(subset(iris, Species == "setosa" & Sepal.Length > 4, c(Sepal.Length, Species)))
```

```
##   Sepal.Length Species
## 1          5.1  setosa
## 2          4.9  setosa
## 3          4.7  setosa
## 4          4.6  setosa
## 5          5.0  setosa
## 6          5.4  setosa
```

E' equivalente a:

```
head(iris[iris$Species == "setosa" & iris$Sepal.Length > 4, c("Sepal.Length", "Spec
```

```
##   Sepal.Length Species
## 1          5.1  setosa
## 2          4.9  setosa
## 3          4.7  setosa
## 4          4.6  setosa
## 5          5.0  setosa
## 6          5.4  setosa
```

Dataframe - Indicizzazione - *data mask*

Cosa fa `subset` di speciale? essenzialmente prende delle espressioni `Species == "setosa"` che vengono eseguite all'interno dell'ambiente dataframe specificato come primo argomento. Un più generico esempio del data masking è la funzione `with`:

```
Sepal.Length <- "ciao"
with(iris, Sepal.Length + Sepal.Width) # se usasse la variabile definita prima darebbe errore
```

```
##   [1]  8.6  7.9  7.9  7.7  8.6  9.3  8.0  8.4  7.3  8.0  9.1  8.2  7.8  7.3  9.8 10.1
##  [17]  9.3  8.6  9.5  8.9  8.8  8.8  8.2  8.4  8.2  8.0  8.4  8.7  8.6  7.9  7.9  8.8
##  [33]  9.3  9.7  8.0  8.2  9.0  8.5  7.4  8.5  8.5  6.8  7.6  8.5  8.9  7.8  8.9  7.8
##  [49]  9.0  8.3 10.2  9.6 10.0  7.8  9.3  8.5  9.6  7.3  9.5  7.9  7.0  8.9  8.2  9.0
##  [65]  8.5  9.8  8.6  8.5  8.4  8.1  9.1  8.9  8.8  8.9  9.3  9.6  9.6  9.7  8.9  8.3
##  [81]  7.9  7.9  8.5  8.7  8.4  9.4  9.8  8.6  8.6  8.0  8.1  9.1  8.4  7.3  8.3  8.7
##  [97]  8.6  9.1  7.6  8.5  9.6  8.5 10.1  9.2  9.5 10.6  7.4 10.2  9.2 10.8  9.7  9.1
## [113]  9.8  8.2  8.6  9.6  9.5 11.5 10.3  8.2 10.1  8.4 10.5  9.0 10.0 10.4  9.0  9.1
## [129]  9.2 10.2 10.2 11.7  9.2  9.1  8.7 10.7  9.7  9.5  9.0 10.0  9.8 10.0  8.5 10.0
## [145] 10.0  9.7  8.8  9.5  9.6  8.9
```

L'espressione `Sepal.Length + Sepal.Width` viene eseguita all'interno (`with`) del dataframe `iris` in modo da eseguire anche operazioni complesse tra le colonne di `iris` senza dover ogni volta specificare `iris$variabile`.

Extra - Importazione dati

Importare i dati (solitamente come `dataframe`) è un operazione complessa e che deve gestire numerosi casi particolari. Come indicazione generale:

1. Capire il formato in **input** (`csv`, `txt`, `sav`, etc.)
2. Trovare la funzione/pacchetto di R che gestisce quel tipo di formato. `read.csv()`, `read.table()`, etc.
3. Leggere la documentazione della funzione e trovare gli argomenti appropriati (`header`, `sep`, etc.)
4. Organizzare i dati nel proprio PC (**working directory**)
5. Importare i dati e controllare il risultato
    - vedere anomalie
    - leggere eventuali errori/warnings
    - controllare la tipologia di variabili (caratteri vs numeri) ed eventualmente sistemare

> Per approfondire ho scritto un documento sull'argomento https://arca-dpss.github.io/course-R/extra/importing_data.html