Testing Psicologico

Lezione 1A - Comandi di base e vettori

Filippo Gambarota

@Università di Padova

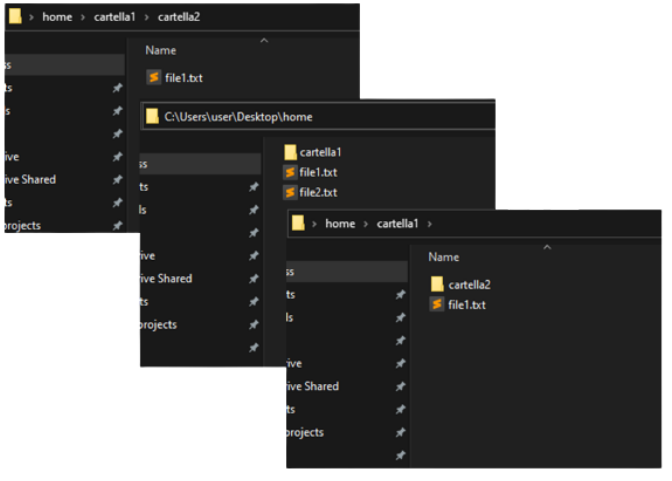
2022-2023

Organizzazione R ed R Studio

Working Directory e Percorsi

Il nostro computer è composto da file e cartelle organizzati in modo **gerarchico** tra loro

```
📦home
 ├ 📂cartella1
 │  ├ 📂cartella2
 │  │  └ 📜file1.txt
 │  └ 📜file1.txt
 ├ 📜file1.txt
 └ 📜file2.txt
```



Working Directory e Percorsi

Nel momento in cui usiamo **R**, lui si colloca automaticamente in un dato percorso:

```
getwd()
```

```
## [1] "/home/filippogambarota/Documents/teaching-projects/didattica-testing-psicologico/slides/lezione1a"
```

Noi possiamo modificare il collocamento di R usando il comando `setwd()`

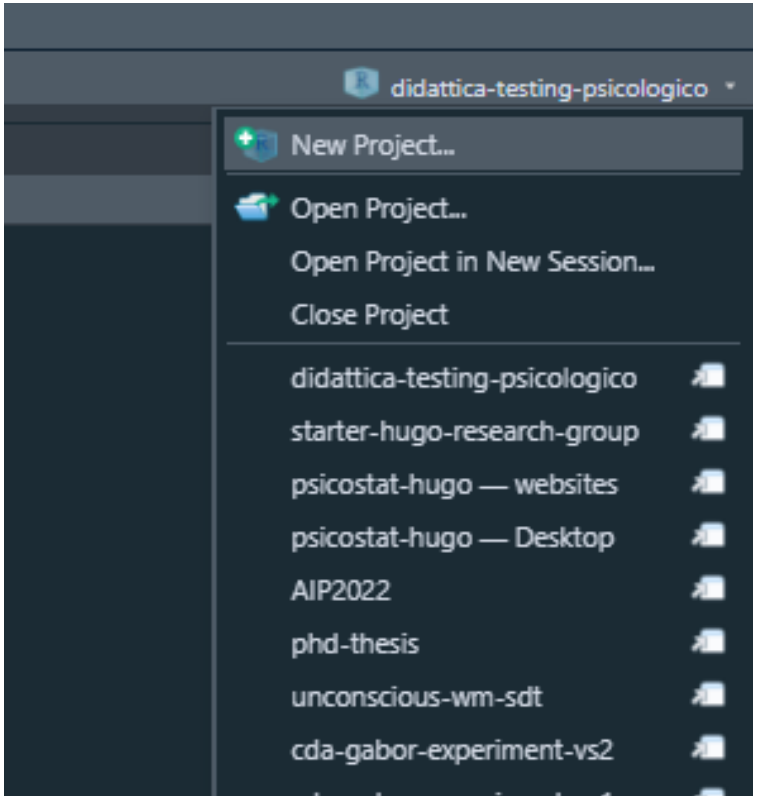
```
setwd("cartella/sub-cartella/...")
```

Extra: R Projects

Extra: R Projects

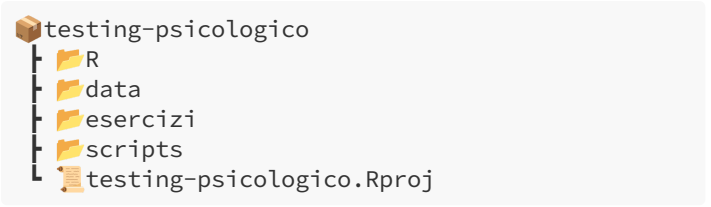
Gli R Projects sono una funzionalità di R Studio e permettono di impostare automaticamente la **working directory** nella cartella dove è contenuto il file `*.Rproj`. In questo modo, ogni volta che R Studio viene aperto caricando un R Project, tutti i percorsi sono relativi alla *root* del progetto.

- Video Tutorial sui percorsi e R Projects



Organizziamo la cartella...

Create un R Project chiamato `testing-psicologico` dentro una cartella chiamata `testing-psicologico` con la seguente struttura:



1. create il progetto R
2. scaricate la cartella `data` da questo link
3. scaricate la cartella `scripts` da questo link
4. scaricate la cartella `R` da questo link
5. create una cartella `esercizi`

R comandi di base

Oggetti

> Everything that exists in R is an object - John M. Chambers

Tutto quello che **creiamo** in R (vettori, matrici, funzioni, variabili, etc.) sono considerati come oggetti:

```
x <- 10
y <- 1:10
z <- "ciao"
f <- function(x) x + 3
class(x)
```

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

```
class(z)
```

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

```
class(f)
```

```
## [1] "function"
```

Funzioni

> Everything that happens in R is the result of a function call - John M. Chambers

Tutto quello che **eseguiamo** in R è il risultato di una funzione:

```
seq(1, 10, 1)
```

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

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

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

```
`+`(3, 2) # equivale a 3 + 2
```

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

```
`<-`(ciao, 2) # equivale a ciao <- 2
ciao
```

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

Importare una funzione

In R tutto (vettore, dataframe, lista, etc.) è un oggetto, anche le funzioni. Per caricare una funzione salvata in un file `.R` possiamo usare il comando `source(file)`. Il file verrà caricato e tutto il codice lanciato. Se qualche oggetto o funzione è stato creato sarà disponibile globalmente:

```
source("../../R/rsummary.R")
ls()
```

```
##  [1] "ciao"    "f"       "file"    "online"  "out_dir" "params"  "pdf"
##  [8] "x"       "y"       "z"
```

Strutture dati

Strutture dati

Le strutture dati sono modalità tramite cui un linguaggio di programmazione **organizza** tipologia e **struttura** dei vari tipi possibili di dato. Il vettore e la matrice sono delle strutture dati.

Aspetti principali di una struttura dati:

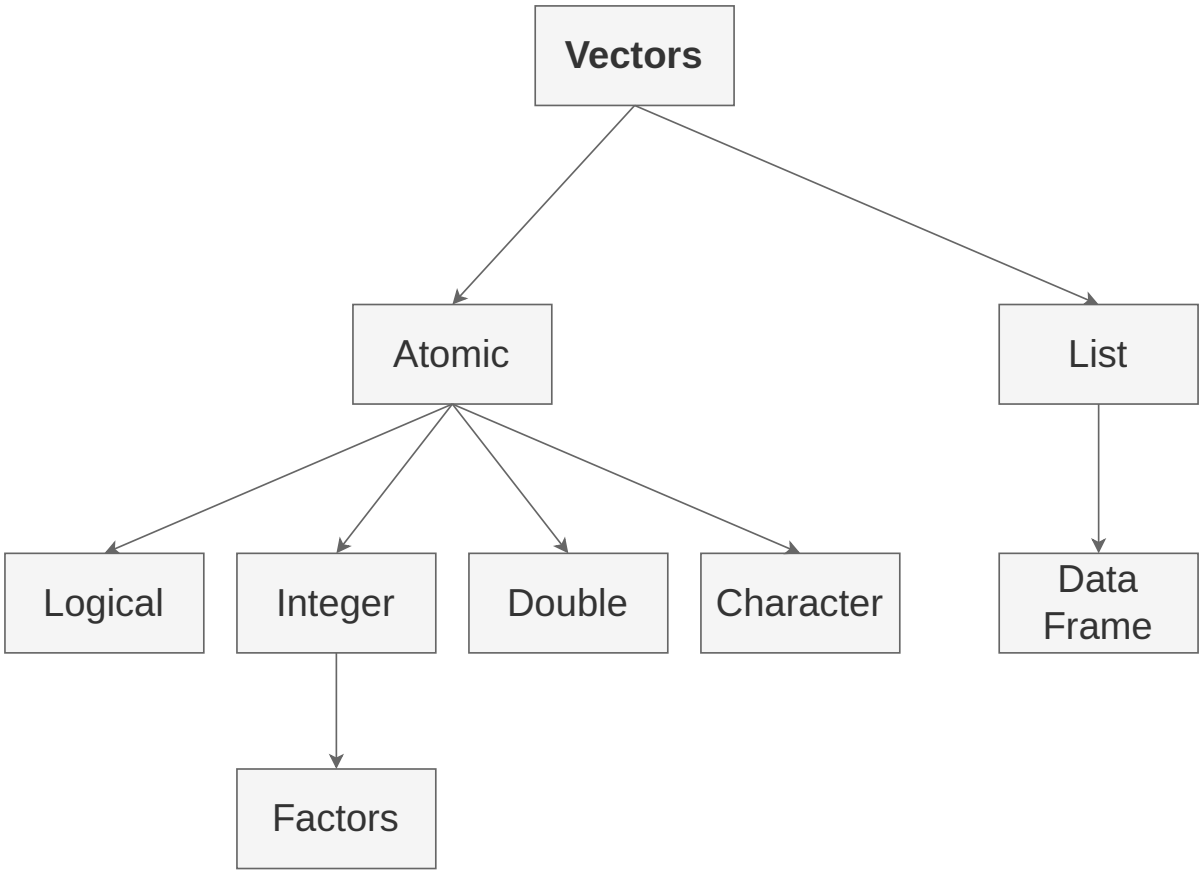
- presenza di **vincoli** (e.g., il vettore può essere solo numerico o di stringhe)
- presenza di **metodi** (i.e., funzioni) per **accedere**, **estrarre** e **modificare** i dati

Strutture dati

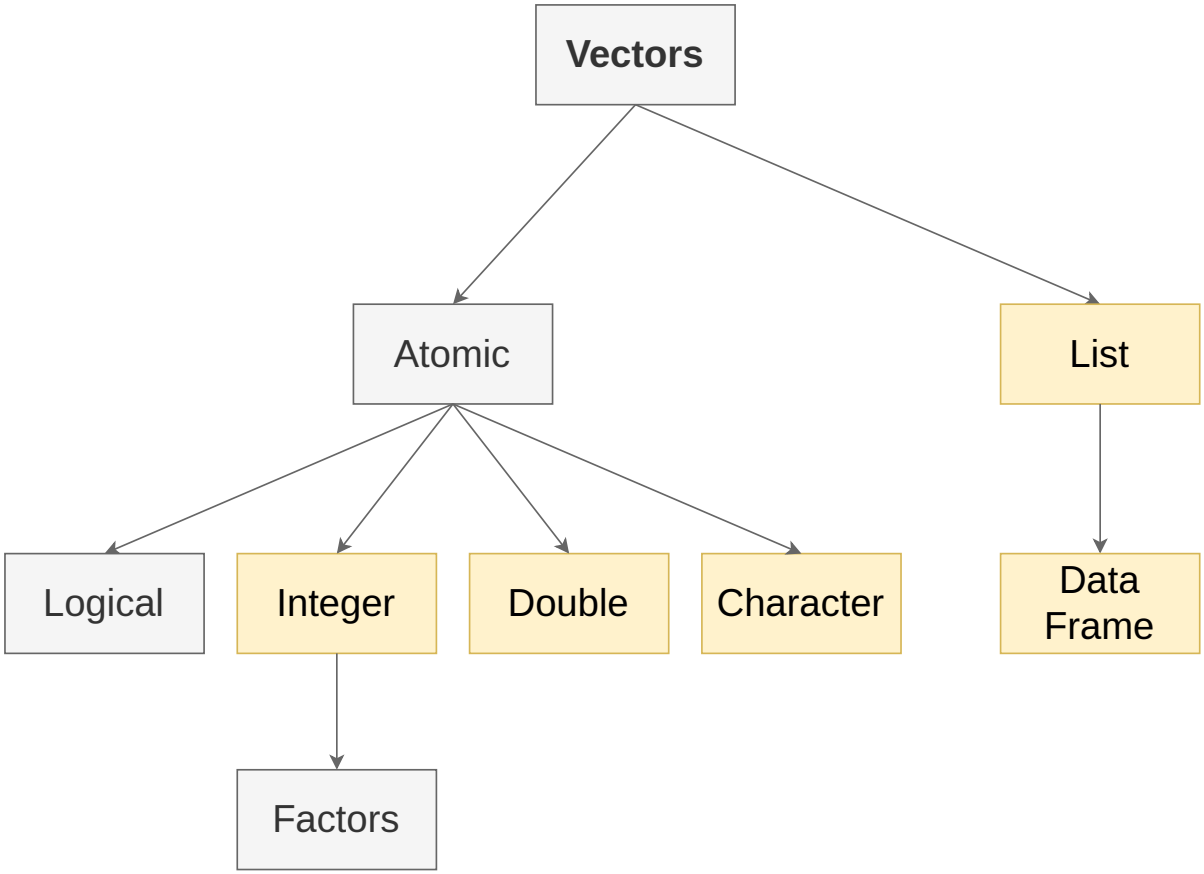
Esiste una struttura dati che abbiamo sicuramente usato. Quale? 🤔

	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



Strutture dati in R

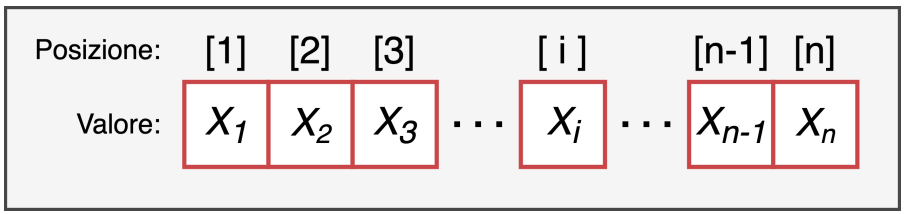


Vettori

Dubbi/Domande? 🤔

Piccolo ripasso...

- Il vettore è una struttura **unidimensionale**, che contiene **un solo tipo** di dato. Le proprietà fondamentali sono la tipologia (`str(vettore)`) e la sua lunghezza (`length(vettore)`)
- Ogni elemento (a prescindere dalla tipologia) è indicizzato partendo da 1 fino alla lunghezza del vettore n



Indicizzazione Vettori

Indicizzazione Vettori

Indicizzare una struttura dati è un'operazione fondamentale e complessa. Ma la logica sottostante è molto semplice. La sezione 10.2 del libro `Introduction2R` è un buon riferimento.

Il modo più semplice è quello di usare l'**indice di posizione**

```
my_vec <- 1:10
my_vec[1] # primo elemento
```

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

```
my_vec[2:5] # dal secondo al 5
```

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

```
my_vec[length(my_vec)] # ultimo elemento
```

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

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]  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE
```

```
my_vec[my_selection]
```

```
## [1] 1 2 6 8 9
```

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
```

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
```

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]  1  3  4  9 10
```

```
# Questo

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

```
## [1] -0.96179633  0.33068543  0.01703712 -0.54595247  0.25931586
```

```
# e questo sono equivalenti

my_vec[my_vec < 0.5]
```

```
## [1] -0.96179633  0.33068543  0.01703712 -0.54595247  0.25931586
```

Indicizzazione negativa

Possiamo anche escludere degli elementi sia usando l'indicizzazione *logica* che quella *intera*:

```
my_vec[-c(1, 5, 7)] # escludo gli elementi alla posizione 1, 5 e 7
```

```
## [1]  0.65509603  0.33068543  0.01703712  0.78666009  1.72839854 -0.54595247
## [7]  0.25931586
```

```
my_vec[!my_vec > 0.5] # escludo gli elementi > di 0.5
```

```
## [1] -0.96179633  0.33068543  0.01703712 -0.54595247  0.25931586
```

Ovviamente assegnando il risultato `<-` delle operazioni precedenti ad un nuovo oggetto o allo stesso creiamo un subset del vettore iniziale

```
my_vec_new <- my_vec[-c(1, 5, 7)] # escludo gli elementi alla posizione 1, 5 e 7
my_vec_new
```

```
## [1]  0.65509603  0.33068543  0.01703712  0.78666009  1.72839854 -0.54595247
## [7]  0.25931586
```

Sostituire

Possiamo anche *sovrascrivere* degli elementi di un vettore ancora utilizzando sia l'indicizzazione *logica* che quella *intera*:

```
my_vec_new <- my_vec # creo un nuovo vettore per fare queste operazioni
my_vec_new[my_vec_new > 0.5] <- 999
my_vec_new
```

```
##  [1]  -0.96179633 999.00000000   0.33068543   0.01703712 999.00000000
##  [6] 999.00000000 999.00000000 999.00000000  -0.54595247   0.25931586
```

```
my_vec_new <- my_vec # creo un nuovo vettore per fare queste operazioni
my_vec_new[c(1,2,5)] <- 888
my_vec_new
```

```
##  [1] 888.00000000 888.00000000   0.33068543   0.01703712 888.00000000
##  [6]   0.78666009   1.86703799   1.72839854  -0.54595247   0.25931586
```

```
my_vec_new <- my_vec # creo un nuovo vettore per fare queste operazioni
my_vec_new[2] <- "ciao"
my_vec_new  # cosa notate?
```

```
##  [1] "-0.961796334395623" "ciao"               "0.330685426108418"
##  [4] "0.0170371226712847" "0.781568423259614"  "0.786660086379985"
##  [7] "1.86703799128984"   "1.72839854093367"   "-0.545952468699917"
## [10] "0.259315861735512"
```

Esercizi

1.Create il seguente **vettore** $V = (2, 3.5, 5, 6.5, 8, 9.5)$

2.Create il seguente **vettore di caratteri** $V = (x, x, x, y, y, z, z, z, z, z, z)$

3.Create un vettore (`vec_let`) di caratteri con 30 lettere random dell'alfabeto (vedi il comando `sample()` ed l'oggetto `letters`) usando però solo le prime 5

1. selezionate solo le lettere "a"
2. selezionate solo le lettere "a", "b" e "c"
3. selezionate tutte le lettere TRANNE la "a"

4.Create un vettore (`vec_int`) di 50 elementi con numeri casuali (**interi**) con un range tra 20 e 150 (vedi il comando `runif` e cerca di capire come trasformarli in interi):

- selezionare tutti i numeri > 50
- selezionare tutti i numeri < 100
- selezionare i numeri che occupano posizioni in sequenza di 2. Ad esempio $x = [10, 30, 40, 2, 3, 1]$, seleziono $[30, 2, 1]$
- seleziono tutti i numeri pari (vedi l'operatore modulo ?`%%`) E minori di 100

Soluzioni

```
seq(2, 10, 1.5) # 1
```

```
## [1] 2.0 3.5 5.0 6.5 8.0 9.5
```

```
rep(c("x", "y", "z"), c(3, 2, 6)) # 2
```

```
##  [1] "x" "x" "x" "y" "y" "z" "z" "z" "z" "z" "z"
```

```
vec_let <- sample(x = letters[1:5], size = 30, replace = TRUE) # 3, cosa fa replace?
vec_let
```

```
##  [1] "d" "d" "b" "a" "a" "c" "b" "a" "d" "e"
##  [ reached getOption("max.print") -- omitted 20 entries ]
```

```
vec_let[vec_let == "a"] # 3.1
```

```
## [1] "a" "a" "a" "a" "a" "a"
```

```
vec_let[vec_let == "a" | vec_let == "b" | vec_let == "c"] # 3.2 oppure vec_let[vec_let %in% c("a", "b", "c
```

```
##  [1] "b" "a" "a" "c" "b" "a" "b" "a" "c" "a"
##  [ reached getOption("max.print") -- omitted 8 entries ]
```

Soluzioni

```
vec_let[vec_let != "a"] # 3.3
```

```
##  [1] "d" "d" "b" "c" "b" "d" "e" "b" "d" "c"
##  [ reached getOption("max.print") -- omitted 14 entries ]
```

```
vec_int <- round(runif(n = 50, min = 20, max = 150), 0) # 4 vedi anche as.integer()
vec_int
```

```
##  [1]  91 119  30 104 146  21 130  78 119 125
##  [ reached getOption("max.print") -- omitted 40 entries ]
```

```
vec_int[vec_int > 50] # 4.1
```

```
##  [1]  91 119 104 146 130  78 119 125  70 137
##  [ reached getOption("max.print") -- omitted 32 entries ]
```

```
vec_int[vec_int < 100] # 4.2
```

```
##  [1] 91 30 21 78 70 41 76 61 84 95
##  [ reached getOption("max.print") -- omitted 19 entries ]
```

Soluzioni

```
vec_int[seq(2, 50, 2)] # 4.3 # indicizzazione intera
```

```
##  [1] 119 104  21  78 125 137 136 146 112  76
##  [ reached getOption("max.print") -- omitted 15 entries ]
```

```
vec_int[vec_int %% 2 == 0 & vec_int < 100] # 4.4
```

```
##  [1] 30 78 70 76 84 60 70 48 54 90
##  [ reached getOption("max.print") -- omitted 7 entries ]
```

Esercizi (Advanced)[1]

1. Si utilizzi il comando letters, con gli opportuni indici, per produrre le seguenti parole: (tip: vedete il comando `match()` per trovare gli indici)
    - albero, cane, ermeneutica, orologio, patologico
2. Quando abbiamo un dataset e dati sensibili spesso è utile generare un codice identificativo unico per ogni soggetto. Create un codice identificativo unico in questo modo "id_3numericasuali_4letterecasuali" (vedi il comando `sample()`) ad esempio `1_324_aeiz`. Per farlo usate il comando `paste0()` (veramente molto utile).
3. Usando lo stesso approccio, componiamo i nostri identificativi basandoci su informazioni già presenti. Copiate e incollate il codice qui sotto e create gli identificativi in questo modo `id_mesenascita_annonascita`:

```
id <- 1:20
mese <- sample(1:12, 20, replace = TRUE)
anno <- round(runif(20, 1994, 2002))
```

[1] Thanks to Professor Massimiliano Pastore 😄

Soluzioni

```r
# il segreto è usare il comando match e selezionare le lettere dalla lista
word <- c("a", "l", "b", "e", "r", "o")
match(word, letters)
```

```
## [1]  1 12  2  5 18 15
```

```r
letters[match(word, letters)]
```

```
## [1] "a" "l" "b" "e" "r" "o"
```

```r
# il comando paste0() incolla delle stringhe insieme per comporre delle nuove stringhe

numeri <- paste0(sample(0:9, 3, replace=TRUE), collapse = "")
lettere <- paste0(sample(letters, 3, replace=TRUE), collapse = "")
paste0("1_", numeri, "_", lettere)
```

```
## [1] "1_708_iyq"
```

```r
codice <- paste0(id, "_", mese, "_", anno) # il comando paste è vettorizzato quindi funziona anche su vett
codice
```

```
##  [1] "1_7_1998"  "2_5_1999"  "3_7_1997"  "4_4_1998"  "5_6_2001"  "6_9_1997"
##  [7] "7_2_1998"  "8_5_2001"  "9_6_1997"  "10_5_2000"
##  [ reached getOption("max.print") -- omitted 10 entries ]
```

Extra - Manipolazione Avanzata Vettori

Ricodifica

- Spesso abbiamo bisogno non solo di accedere/selezionare ma di **modificare** in modo condizionale gli elementi di un vettore
- Quando facciamo lo *scoring* dei questionari (lo vedrete più avanti) alcuni elementi delle risposte devono essere cambiate
- Ad esempio, **invertire** i valori delle risposte per alcuni item: 1 --> 5, 2 --> 4, 3 --> 3, 4 --> 2, 5 --> 1
- Si può fare manualmente (😱) in Excel o in modo più robusto in R

```
item1 <- sample(1:5, 30, replace = TRUE)
item1
```

```
##  [1] 4 4 2 4 2 4 4 3 3 2
##  [ reached getOption("max.print") -- omitted 20 entries ]
```

Ricodifica

Il principio è quello della selezione/sostituzione logica ma ci serve un modo più compatto

```
item1[item1 == 1] <- 5
item1[item1 == 2] <- 4
item1[item1 == 3] <- 3
item1[item1 == 4] <- 2
item1[item1 == 5] <- 1
```

Funziona? Vedete alcuni problemi? Provate a utilizzarlo...

Ricodifica

Problema 1: Lungo e ripetitivo da scrivere, sopratutto se dobbiamo cambiare molti valori

Problema 2: Deve essere eseguito in **parallelo** altrimenti il risultato è sbagliato (2 diventa 4 ma poi ritorna 2)

Ricodifica - `car::recode()`

Il pacchetto `car` ha una funzione `recode` che esegue esattamente questa operazione `recode(vettore, c("old = new", ...))`

```
car::recode(item1, "1=5; 2=4; 3=3; 4=2; 5=1")
```

```
##  [1] 2 2 4 2 4 2 2 3 3 4
##  [ reached getOption("max.print") -- omitted 20 entries ]
```

Ricodifica - `dplyr::case_when()`

Il pacchetto `dplyr` ha una funzione `case_when()` che in modo più intuitivo ricodifica un vettore:

```
dplyr::case_when(item1 == 1 ~ 5,
                 item1 == 2 ~ 4,
                 item1 == 3 ~ 3,
                 item1 == 4 ~ 2,
                 item1 == 5 ~ 1)
```

```
##  [1] 2 2 4 2 4 2 2 3 3 4
##  [ reached getOption("max.print") -- omitted 20 entries ]
```