

# Testing Psicologico

## Lezione 1A - Comandi di base e vettori

Filippo Gambarota

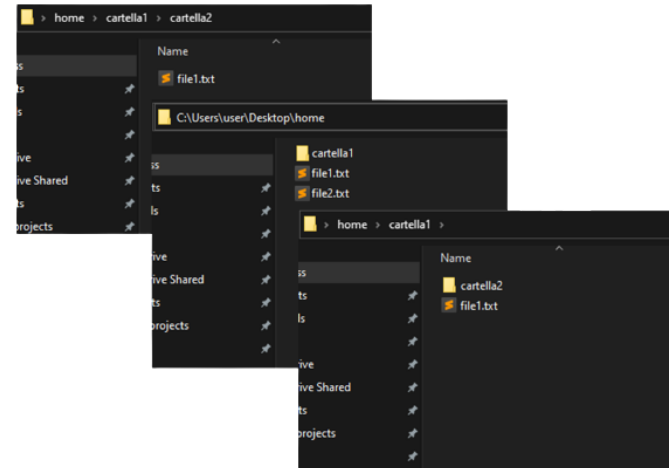
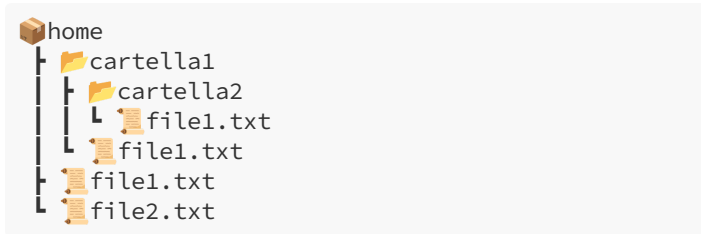
@Università di Padova

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



# 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()`

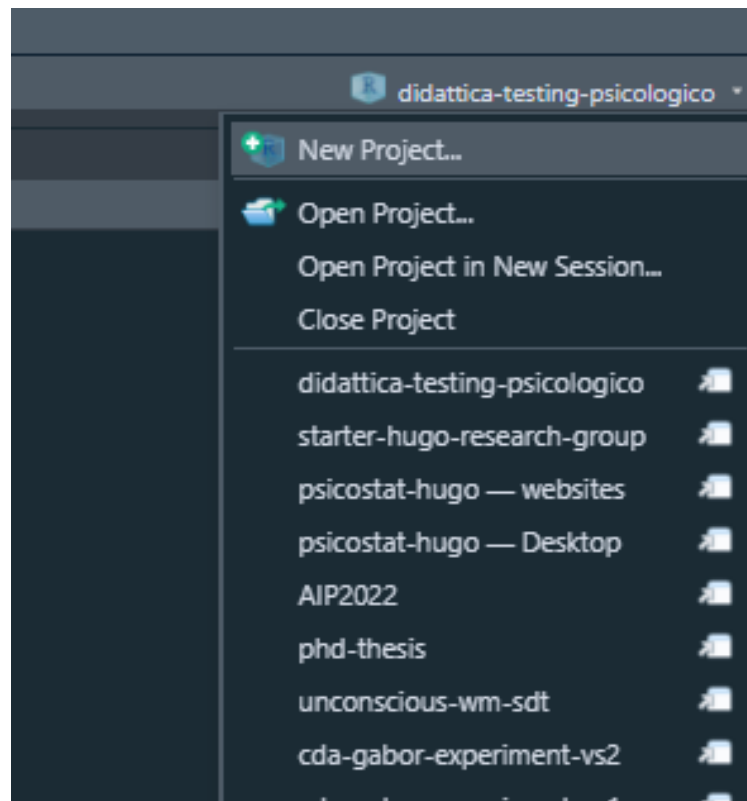
```
setwd("cartella/sub-cartella/...")
```

# Extra: R Projects

# Extra: R Projects

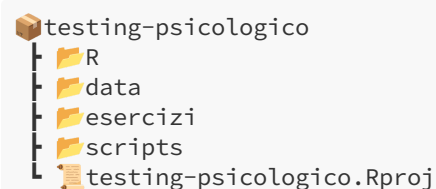
Gli R Projects sono una funzionalità di R Studio e permettono di impostare automaticamente la **working directory** nella cartella dove è contenuto il file \*.Rproj. In questo modo, ogni volta che R Studio viene aperto caricando un R Project, tutti i percorsi sono relativi alla *root* del progetto.

- [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:

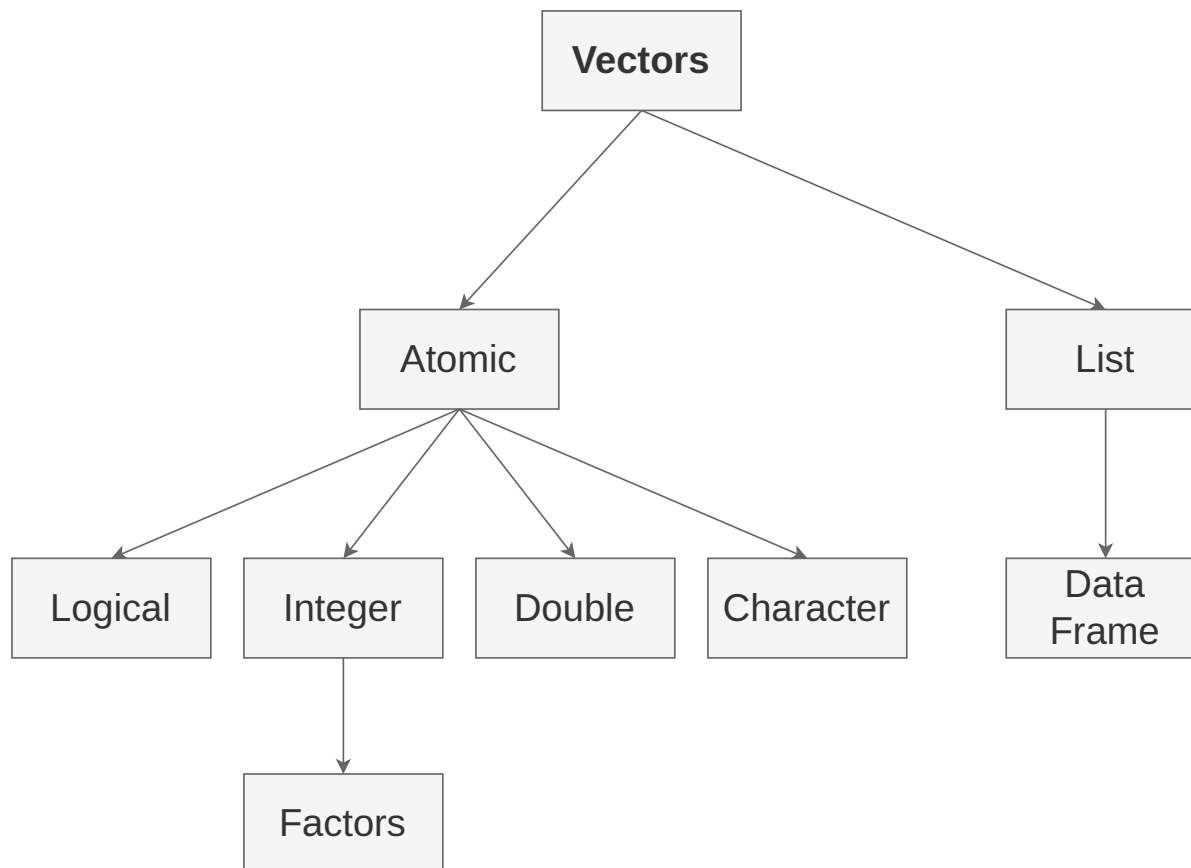
- presenza di **vincoli** (e.g., il vettore può essere solo numerico o di stringhe)
- presenza di **metodi** (i.e., funzioni) per **accedere**, **estrarre** e **modificare** i dati

# Strutture dati

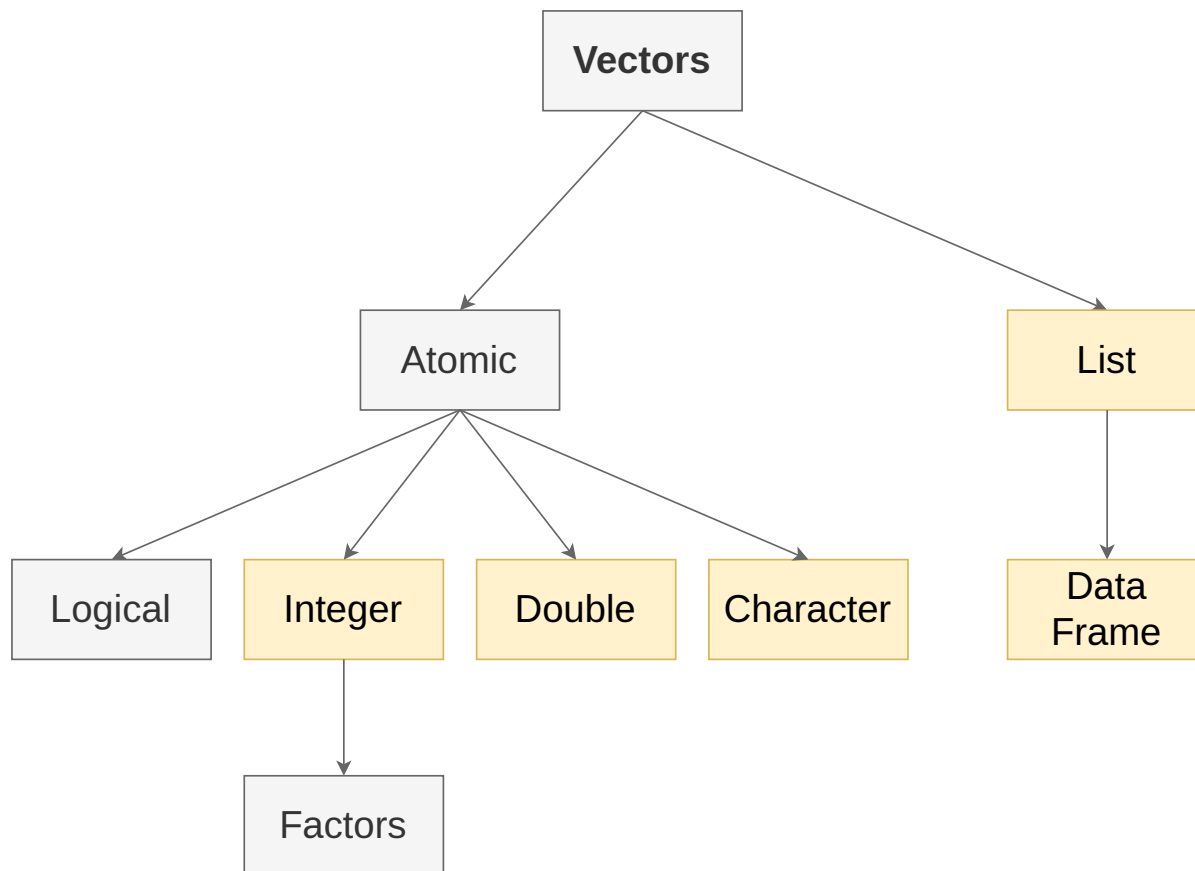
Esiste una struttura dati che abbiamo sicuramente usato. Quale? 🤔

	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  $n$

Posizione:	[1]	[2]	[3]	...	[i]	...	[n-1]	[n]
Valore:	$x_1$	$x_2$	$x_3$	...	$x_i$	...	$x_{n-1}$	$x_n$

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

```
my_vec[my_selection]
```

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

# 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 abbiamo 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 2 3 4 5 7 8 9 10
```

```
# Questo
```

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

```
## [1] -2.47637639 -0.70056650 0.13770189 0.03100410 0.09335986 -0.14315109
## [7] -0.48432950 0.45455251 0.34486660
```

```
# e questo sono equivalenti
```

```
my_vec[my_vec < 0.5]
```

```
## [1] -2.47637639 -0.70056650 0.13770189 0.03100410 0.09335986 -0.14315109
## [7] -0.48432950 0.45455251 0.34486660
```

# 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] -0.7005665  0.1377019  0.0310041  1.2981937 -0.4843295  0.4545525  0.3448666
```

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

```
## [1] -2.47637639 -0.70056650  0.13770189  0.03100410  0.09335986 -0.14315109  
## [7] -0.48432950  0.45455251  0.34486660
```

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] -0.7005665  0.1377019  0.0310041  1.2981937 -0.4843295  0.4545525  0.3448666
```

# 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] -2.47637639 -0.70056650 0.13770189 0.03100410 0.09335986 999.00000000
## [7] -0.14315109 -0.48432950 0.45455251 0.34486660
```

```
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 0.1377019 0.0310041 888.0000000 1.2981937
## [7] -0.1431511 -0.4843295 0.4545525 0.3448666
```

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

```
## [1] "-2.47637639483999" "ciao" "0.137701894772188"
## [4] "0.0310040981032199" "0.0933598568093401" "1.29819370870798"
## [7] "-0.143151090476452" "-0.484329497000884" "0.454552507639268"
## [10] "0.344866596944343"
```

# Esercizi

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

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

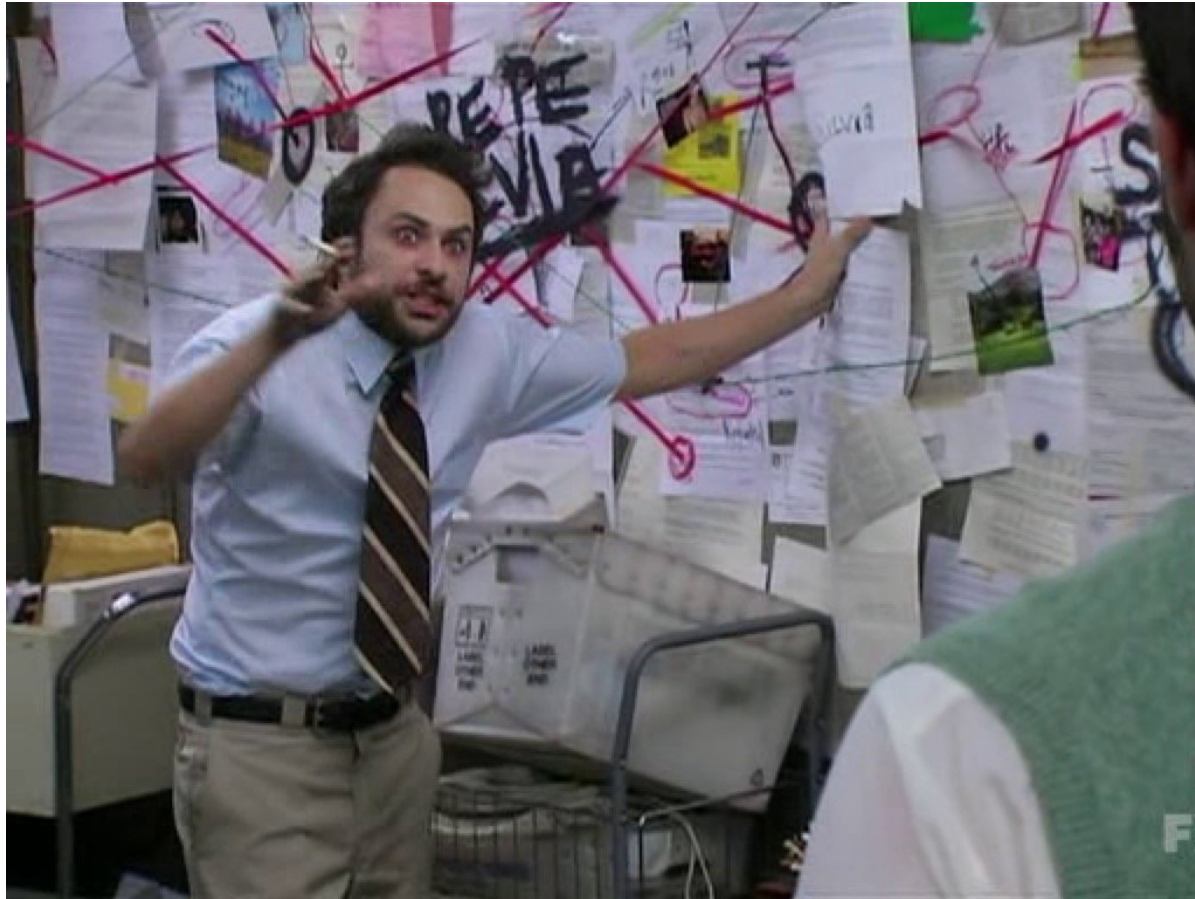
3. Create un vettore (`vec_let`) di caratteri con 30 lettere random dell'alfabeto (vedi il comando `sample()` ed l'oggetto `letters`) usando però solo le prime 5

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

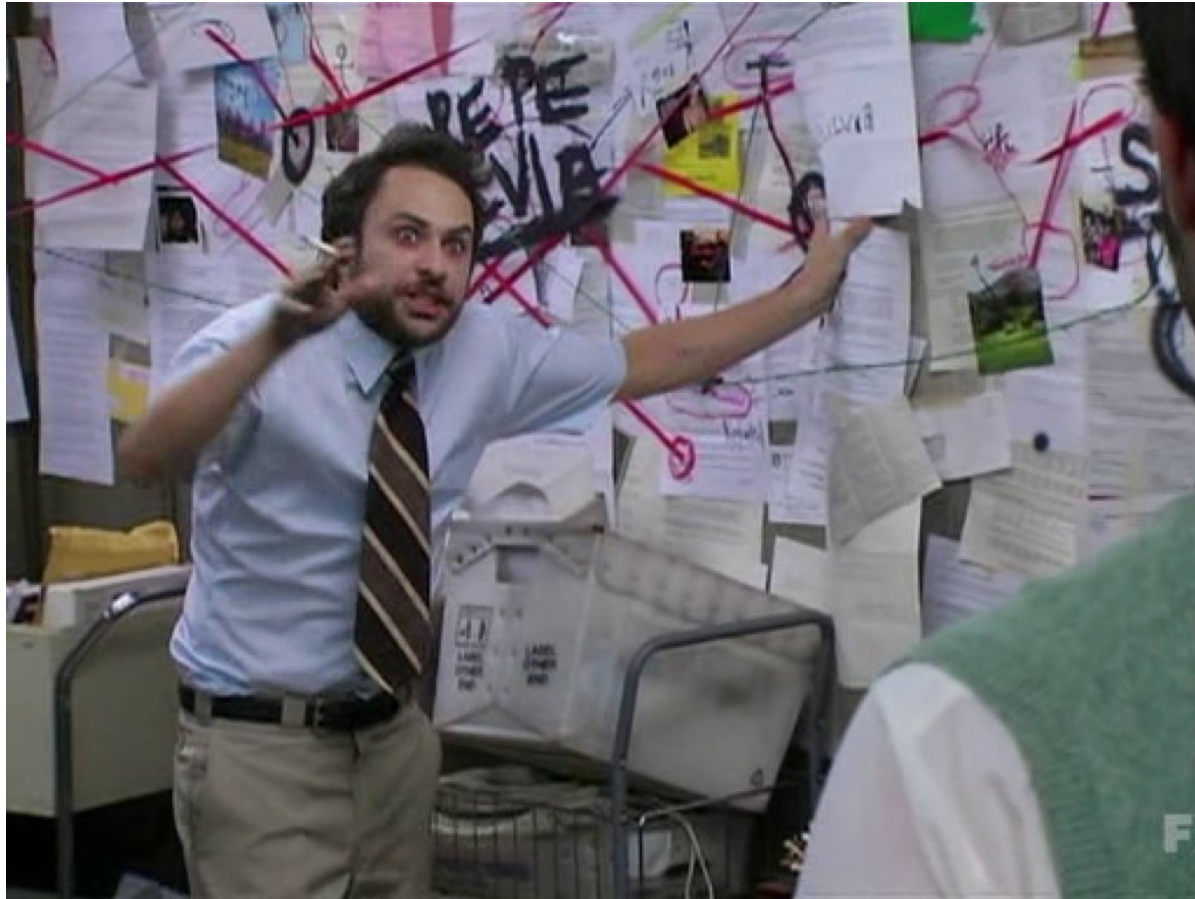
4. Create un vettore (`vec_int`) di 50 elementi con numeri casuali (**interi**) con un range tra 20 e 150 (vedi il comando `runif` e cerca di capire come trasformarli in interi):

- 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

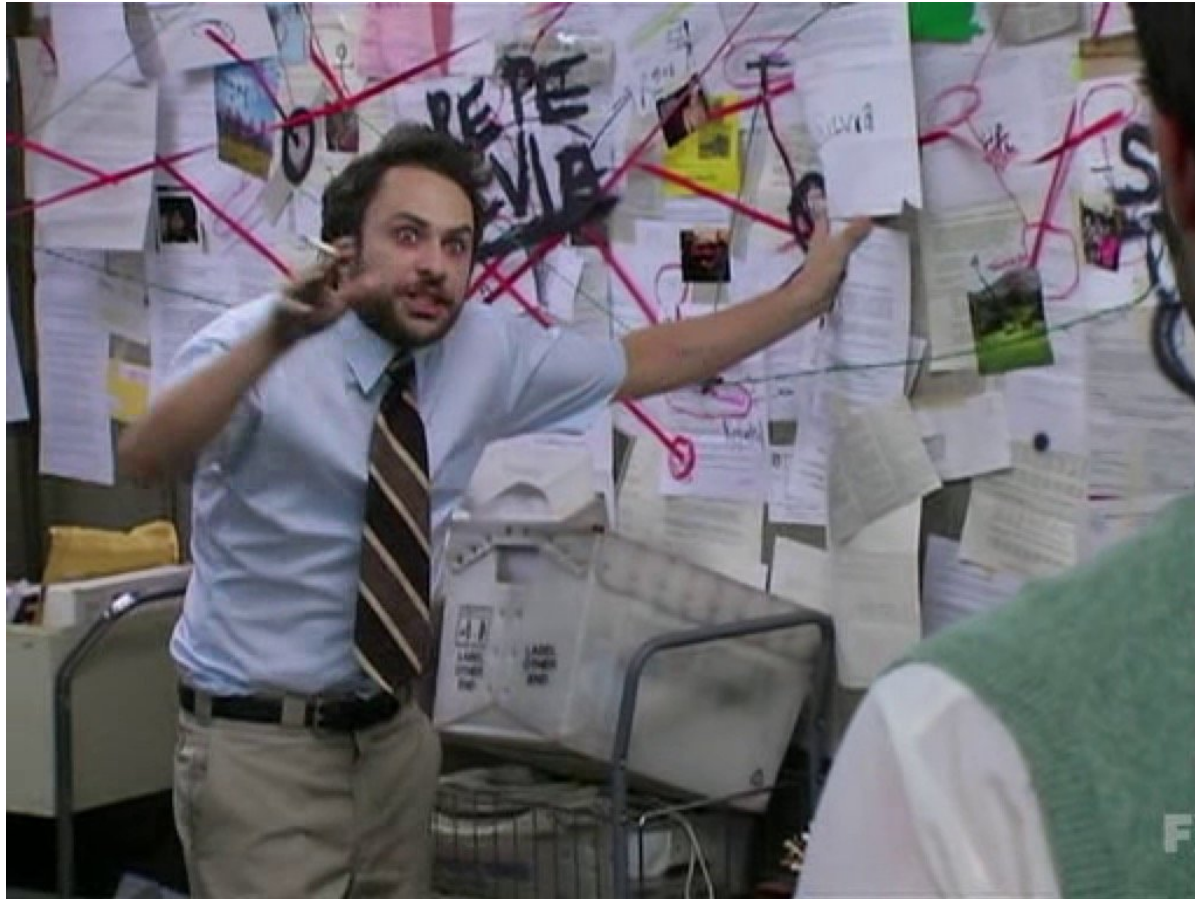
No solutions here 😱



No solutions here 😱



No solutions here 😱





# Esercizi (Advanced)<sup>1</sup>

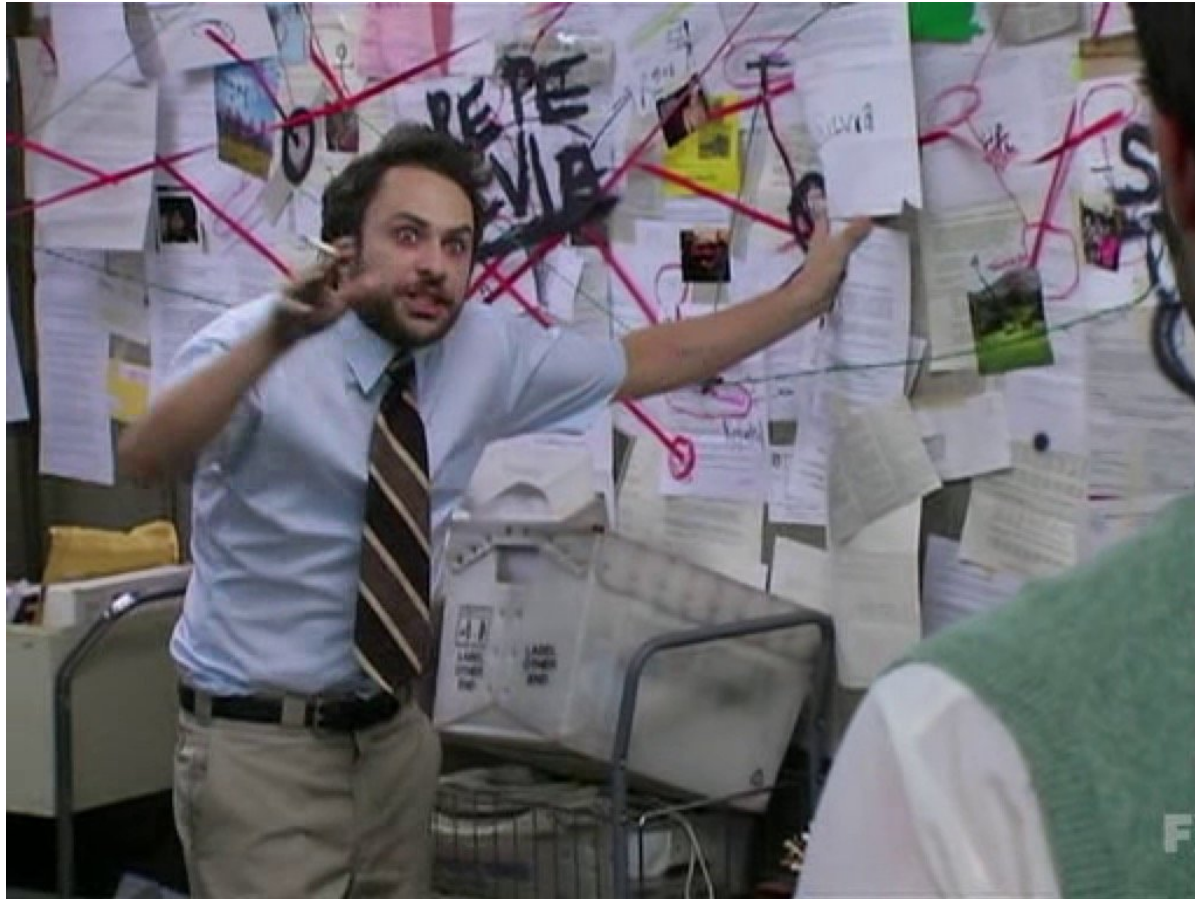
1. Si utilizzi il comando `letters`, con gli opportuni indici, per produrre le seguenti parole: (tip: vedete il comando `match()` per trovare gli indici)
  - albero, cane, ermeneutica, orologio, patologico
2. Quando abbiamo un dataset e dati sensibili spesso è utile generare un codice identificativo unico per ogni soggetto. Create un codice identificativo unico in questo modo "id\_3numericasuali\_4lettere casuali" (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 😊



No solutions here 😱



## 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] 5 2 5 1 2 2 4 4 2 3
## [ 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, soprattutto 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] 1 4 1 5 4 4 2 2 4 3  
## [ 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] 1 4 1 5 4 4 2 2 4 3  
## [ reached getOption("max.print") -- omitted 20 entries ]
```