

# Introduzione a R

Primi passi con R



ARCA - @DPSS

Filippo Gambarota

# Primi passi con R

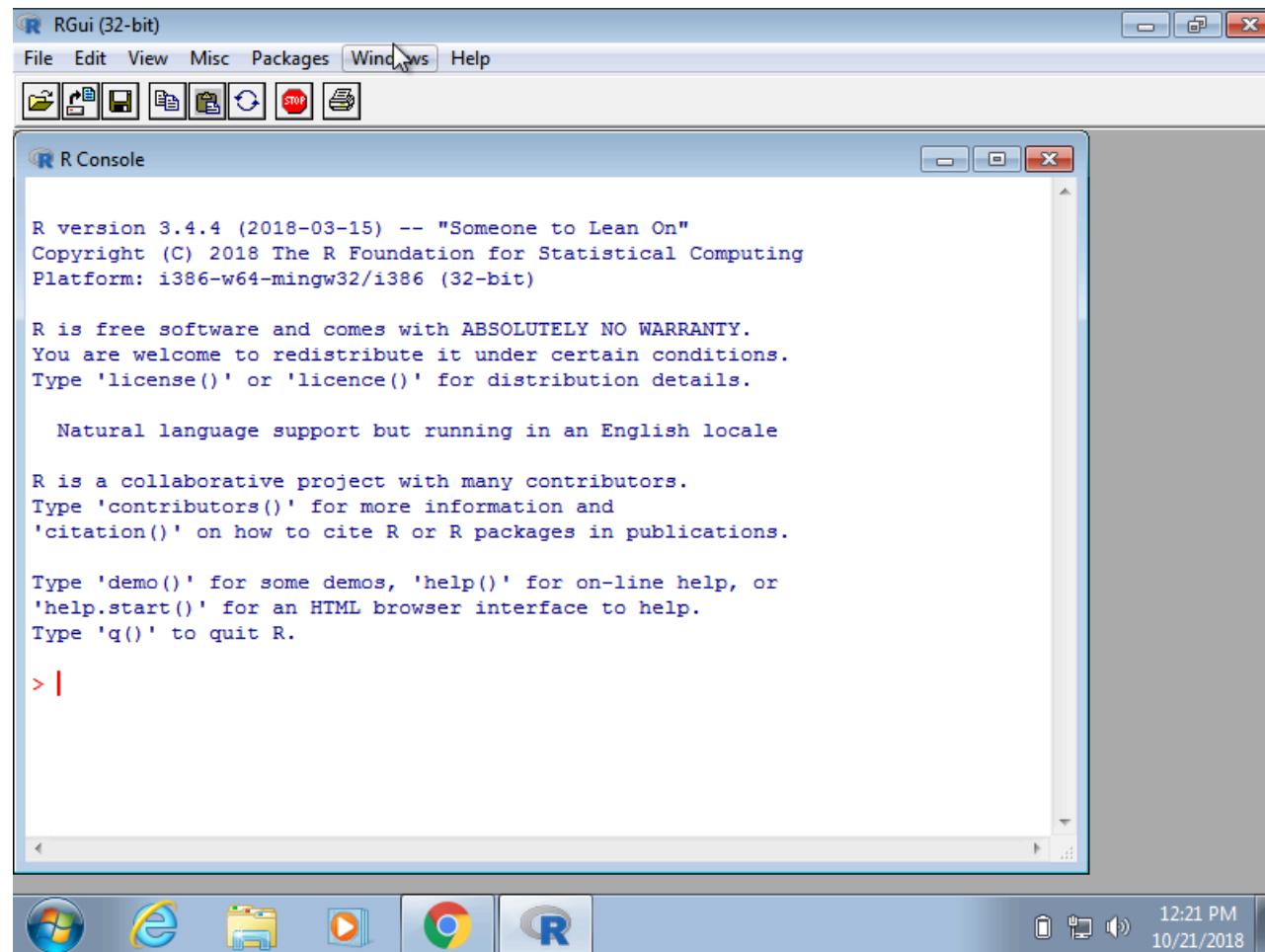
# Installazione

Per l'installazione trovate le indicazioni nella sezione **Installare R e RStudio** del libro. In generale i passaggi sono:

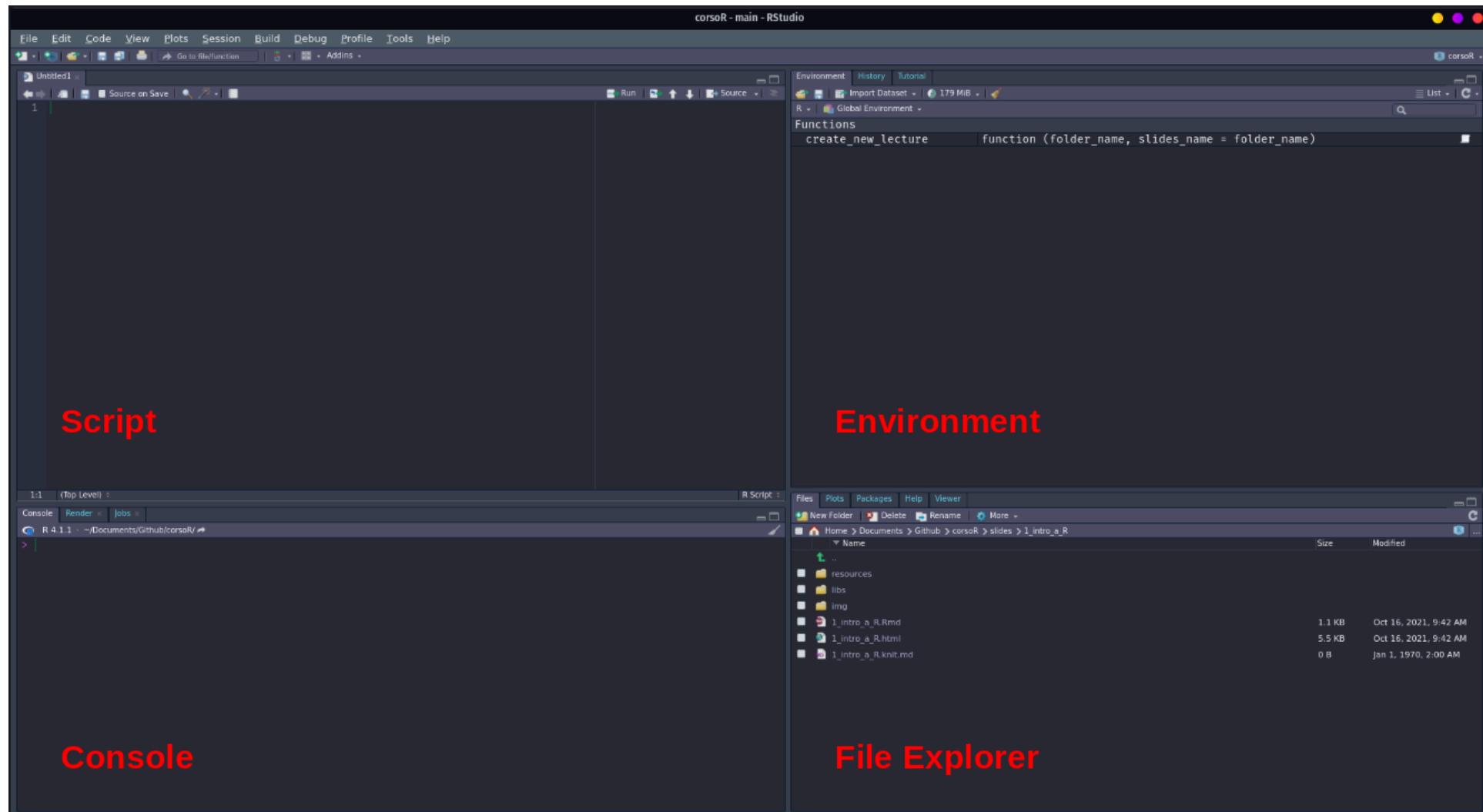
- scaricare R e installare **R** per il vostro sistema operativo
- scaricare e installare **RStudio**

Come si presenta R

# Console



# RStudio



# Questions?



# I primi passi in R

# R come calcolatrice

In R è possibile effettuare tutte le **operazioni matematiche** e algebriche dalle più semplici alle più avanzate

Funzione	Nome	Esempio
$x + y$	Addizione	<pre>&gt; 5 + 3 [1] 8</pre>
$x - y$	Sottrazione	<pre>&gt; 7 - 2 [1] 5</pre>
$x * y$	Moltiplicazione	<pre>&gt; 4 * 3 [1] 12</pre>
$x / y$	Divisione	<pre>&gt; 8 / 3 [1] 2.666667</pre>
$x \%% y$	Resto della divisione	<pre>&gt; 7 \%% 5 [1] 2</pre>

# Operatori matematici

- Importante considerare l'**ordine delle operazioni** analogo alle regole della matematica:  $2 \times 3 + 1$  prima  $2 \times 3$  e poi  $+ 1$ . Analogamente in R:

```
# Senza parentesi
```

```
2 * 3 + 1
```

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

```
# Con le parentesi
```

```
(2 * 3) + 1
```

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

```
# Con le parentesi forzando un ordine diverso
```

```
2 * (3 + 1)
```

```
## [1] 8
```

# Operatori relazionali

Gli operatori relazionali sono molto utili dentro le **funzioni**, per **selezionare elementi dalle strutture dati** (vedremo più avanti) e in generale per **controllare** alcune sezioni del nostro codice:

```
3 > 4
```

```
40 == 40
```

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

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

```
3 >= 3
```

```
10 != 50
```

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

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

```
10 < 100
```

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

# Operatori logici

Gli operatori logici permettono di **combinare espressioni relazionali** e ottenere sempre un valore TRUE o FALSE:

```
3 > 4 & 10 < 100
```

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

```
10 < 100 | 50 > 2
```

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

```
!5 > 4
```

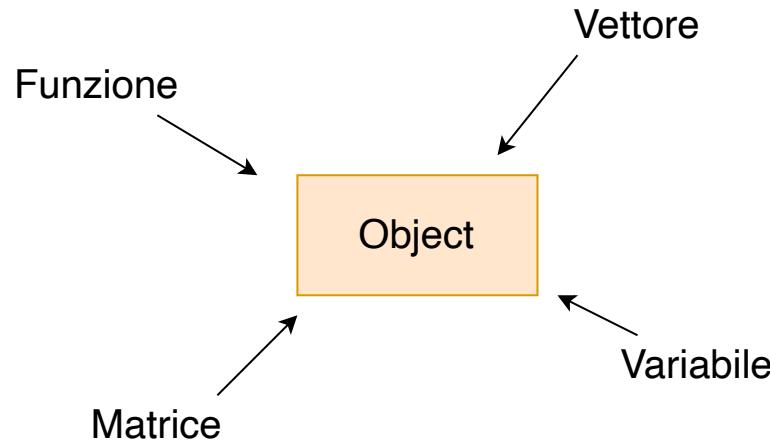
```
## [1] FALSE
```

R e gli oggetti

# R e gli oggetti

| “Everything that exists in R is an object” - John Chambers

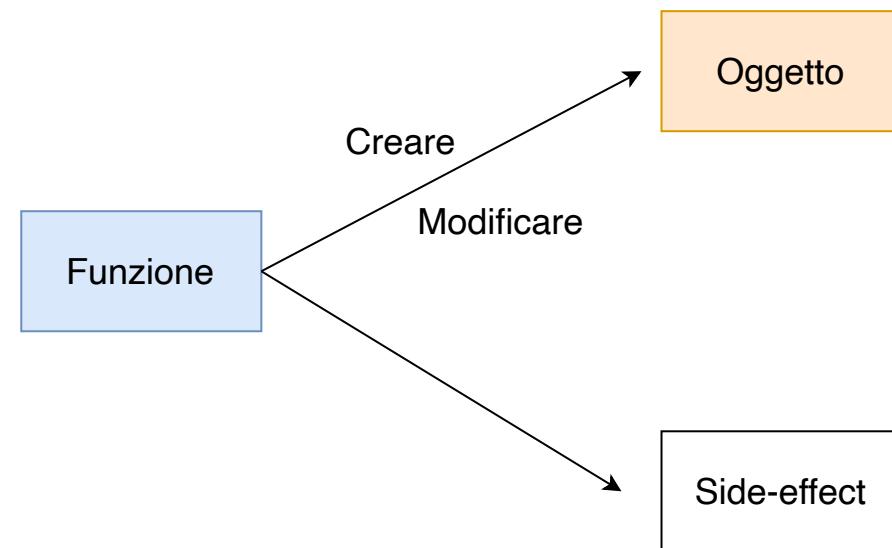
Il concetto di **oggetto** è fondamentale in R. Essenzialmente tutto quello che possiamo creare o utilizzare in R come un numero, un vettore, dei caratteri o delle funzioni sono creati come oggetti.



# R e le funzioni

| “Everything that happen in R is a function call” - John Chambers

Anche il concetto di **funzione** è fondamentale in R. Essenzialmente tutto quello che facciamo è chiamare **funzioni** su oggetti ottenendo un nuovo oggetto o modificando un oggetto esistente



# Cosa possiamo usare/creare in R?

- **Numeri**: 100, 20, 6, 5.6 sono tutti numeri interpretati e trattati come tali
- **Stringhe**: "ciao", "1" sono *caratteri* che vengono interpretati letteralmente devono essere dichiarati con ""
- **Nomi**: ciao, x sono nomi (senza virgolette) e sono utilizzati per essere associati ad un oggetto (variabile, funzione, etc.)
  - **operatori**: sono delle funzioni (e quindi oggetti con un nome associato) che si utilizzano in modo particolare. `3 + 4` in questo caso `+` è un operatore (funzione) che si può usare anche come `+(3, 4)`

# R e gli oggetti

- Come creare un oggetto?
- Oggetti e nomi
- Dove viene creato l'oggetto?

# Come creare un oggetto?

La creazione di un oggetto avviene tramite il comando `<-` oppure `=` in questo modo: `nome <- oggetto`:

```
x
```

```
## Error in eval(expr, envir, enclos): object 'x' not found
```

```
10 # questo non è un oggetto, non è salvato
```

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

```
x <- 10 # ora il valore numerico 10 è associato al nome "x"
```

```
x
```

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

## Convenzioni vs regole

Ci sono alcune cose da considerare quando si scrive codice ed in particolare si creano oggetti:

- **alcune modalità sono errate** --> R ci fornisce un messaggio di errore
- **alcune modalità sono sconsigliate** --> funziona tutto ma ci potrebbero essere problemi
- **alcune modalità sono stilisticamente errate** --> funziona tutto, nessun problema ma... anche l'occhio vuole la sua parte

# Oggetti e nomi

Il nome di un oggetto è importante sia per l'utente che per il software stesso:

```
1 <- 10 # errore  
  
_ciao <- 10 # errore  
  
mean <- 10 # possibile ma pericoloso  
  
'1` <- 10 # con i backticks si può usare qualsiasi nome ma poco pratico
```

```
## Error: <text>:4:2: unexpected symbol  
## 3:  
## 4: _ciao  
##      ^
```

```
my_obj <- 10  
  
my.obj <- 10  
  
My_obj <- 10 # attenzione a maiuscole e minuscole
```

# Oggetti e nomi (proibiti)

In R ci sono anche dei nomi non solo sconsigliati ma proprio **proibiti** che nonostante siano sintatticamente corretti, non possono essere usati (per ovvie ragioni):

```
mean <- 10 # ok ma sconsigliato
```

```
function <- 10
```

```
## Error: <text>:4:10: unexpected assignment
## 3:
## 4: function <-
##          ^
```

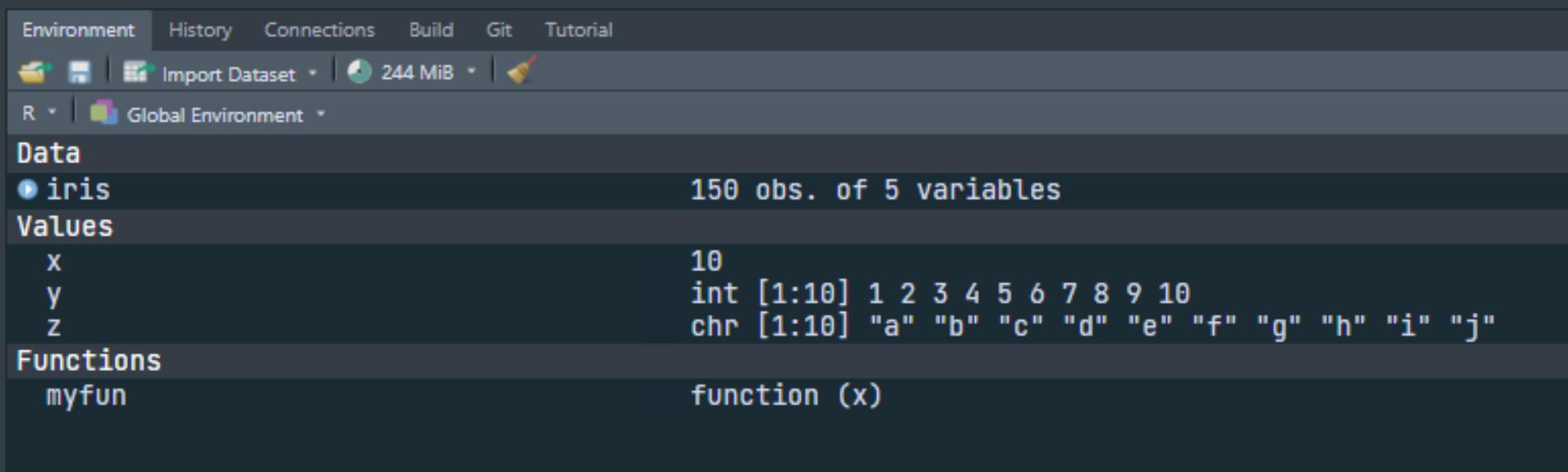
```
TRUE <- 4
```

```
## Error in TRUE <- 4: invalid (do_set) left-hand side to assignment
```

```
T <- 2 # attenzione
```

# Dove viene creato l'oggetto?

Di default gli oggetti sono creati nel **global environment** accessibile con `ls()` o visibile in R Studio con anche alcune informazioni aggiuntive:



The screenshot shows the RStudio interface with the Global Environment tab selected. The pane displays the following information:

- Data**:
  - `iris`: 150 obs. of 5 variables
- Values**:
  - `x`: 10
  - `y`: int [1:10] 1 2 3 4 5 6 7 8 9 10
  - `z`: chr [1:10] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
- Functions**:
  - `myfun`: function (x)

**Non solo numeri (anticipazione)**

# Non solo numeri (anticipazione)

In R possiamo usare oltre ai numeri (in senso matematico) anche le **stringhe** ovvero parole, lettere intepretate così come sono:

```
"ciao" # stringa formata da 5 caratteri
```

```
## [1] "ciao"
```

```
x <- "ciao" # associo la stringa ad un oggetto
```

```
x + 1 # operazioni matematiche con stringhe (ha senso?)
```

```
## Error in x + 1: non-numeric argument to binary operator
```

```
x == "ciao"
```

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

```
x > 10
```

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

# Funzioni

# Funzioni

Le funzioni sono un argomento relativamente complesso ed avanzato. Lo tratteremo più avanti nella sezione [Funzioni](#) del capitolo Programmazione in R. Siccome le usiamo fin da subito è importante avere chiari alcuni aspetti:

- Funzioni come oggetti
- Argomenti obbligatori, opzionali e default
- Ordine degli argomenti
- Documentazione

# Funzioni come oggetti

Abbiamo già visto che ogni cosa in R è un oggetto. Anche le funzioni seppur molto diverse da altri elementi sono creati e trattati in R come oggetti:

```
myfun <- function(x) {  
  return(x + 3)  
}  
  
ls()  
  
## [1] "file"          "math_operators" "my_obj"        "My_obj"  
## [5] "my.obj"        "myfun"         "names_function" "pdf"  
## [9] "T"              "x"
```

Possiamo crearle, eliminarle o sovrascriverle come un normale oggetto. Vedremo più avanti come crearle ma tenete in considerazione che tutte le funzioni che usiamo sono create come oggetti e salvati nell'ambiente (quando facciamo `library()` sono rese disponibili)

# Argomenti

Gli argomenti delle funzioni sono quelli che da *utenti* dobbiamo conoscere ed impostare nel modo corretto per fare in modo che la funzioni faccia quello per cui è stata pensata. Vediamo l'`help` della funzione `mean()`

RDocumentation

Search all packages and functio



Learn R

base (version 3.6.2)

## mean: Arithmetic Mean

### Description

Generic function for the (trimmed) arithmetic mean.

### Usage

```
mean(x, ...)
```

# Argomenti

- `x` è un oggetto (ovviamente 😊). [...] "currently there are methods for numeric/logical vectors...". Quindi `x` deve essere numerico o logico. Ha senso fare la media di caratteri? 🤔
- `trim`
- `na.rm`

Per impostare questi argomenti ci sono 2 regole:

- l'ordine non conta SE DEFINISCO NOME DELL'ARGOMENTO con `x = vettore`, `na.rm = TRUE`, etc.
- l'ordine conta SE NON DEFINISCO IL NOME DELL'ARGOMENTO. Posso quindi omettere `argomento = valore` ma devo rispettare l'ordine con cui è stata scritta la funzione

# Argomenti

In questo caso proviamo ad usare la funzione `mean()`:

```
myvec <- rnorm(100, 10, 5)
mean(myvec) # x definito, trim non definito, na.rm non definito

## [1] 9.917387

mean(myvec, trim = 0.10) # x definito, trim definito, na.rm non definito

## [1] 9.934376

mean(myvec, na.rm = TRUE) # x definito trim non definito, na.rm definito

## [1] 9.917387

mean(myvec, TRUE) # cosa succede?

## Error in mean.default(myvec, TRUE): 'trim' must be numeric of length one
```

# Formula Syntax (extra, but useful)

- In R vedrete spesso l'utilizzo dell'operatore `~` per fare grafici, statistiche descrittive, modelli lineari etc. L'utilizzo di `y ~ x` permette di creare del codice R che non viene eseguito subito ma può essere eseguito successivamente in un ambiente specifico.
- è l'unico caso dove nomi non assegnati possono essere utilizzati senza errori
- questo tipo di programmazione si chiama **non-standard evaluation** perchè appunto non funziona come il solito codice R

```
y # y non esiste e quindi ho un errore
```

```
## Error in eval(expr, envir, enclos): object 'y' not found
```

```
y ~ x # usando ~ non ho errori perchè il codice non viene eseguito
```

```
## y ~ x  
## <environment: 0x127945be0>
```

```
`~`(y, x) # l'operatore ~ non è altro che una funzione come quelle che abbiamo visto fino ad ora
```

```
## y ~ x  
## <environment: 0x127945be0>
```

# Formula Syntax (extra, but useful)

Le formule vengono utilizzate in tantissimi contesti.

Per fare modelli di regressione

```
# un modello linare -> dipendente ~ indipendenti  
lm(y ~ x1 + x2)
```

Per fare aggregare un dataset

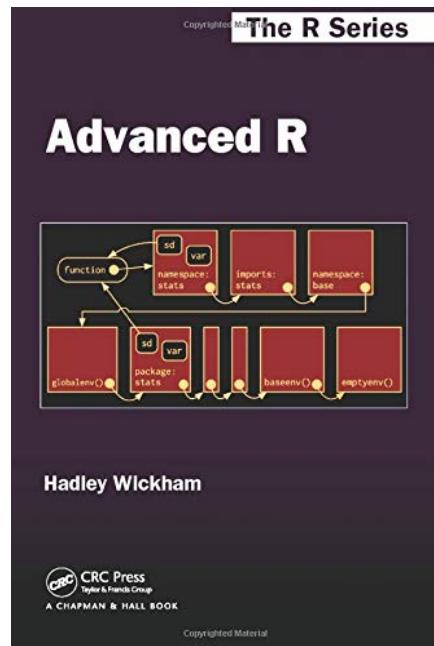
```
# per aggregare un dataset (vedremo più avanti : )  
aggregate(y ~ x, data = data, FUN = mean)
```

Per fare grafici

```
# per fare grafici  
boxplot(y ~ x, data = data)
```

# Formula Syntax (extra, but useful)

In generale, ogni volta che usate delle variabili *unquoted* (senza virgolette) e queste non sono dichiarate nell'ambiente, state probabilmente usando la **non-standard evaluation** e c'è una formula da qualche parte 😅 . Per approfondire:



Capitolo Metaprogramming

Ambiente di lavoro 

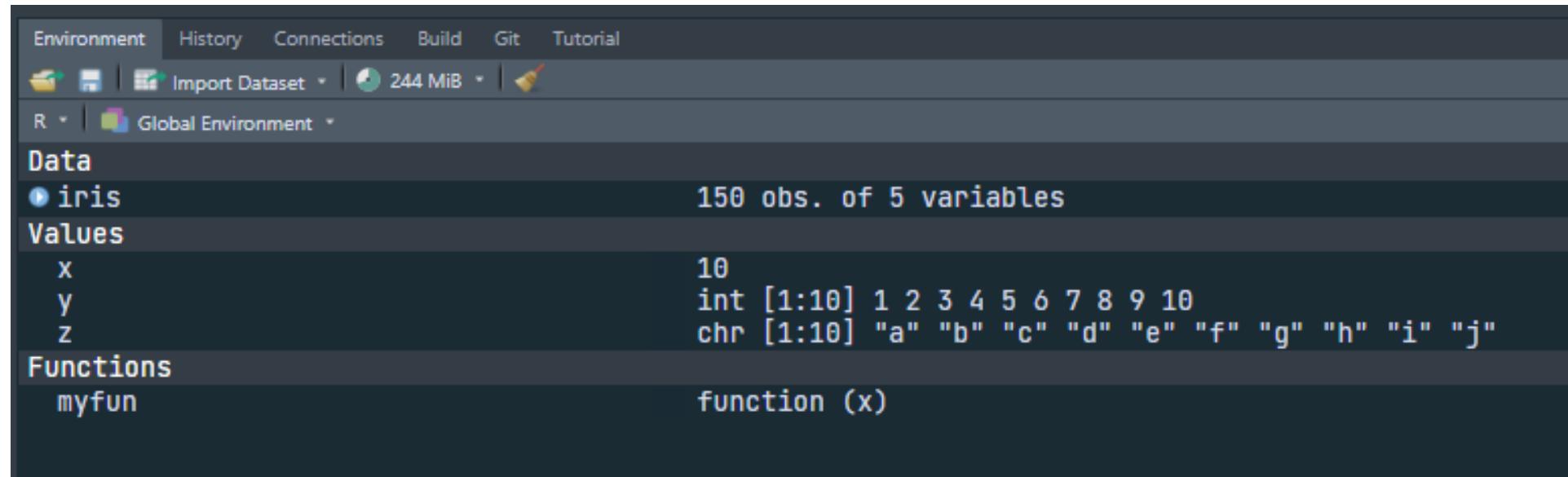
# Ambiente di lavoro



- Environment
- Working directory
- Packages

# Environment

Il **working environment** è la vostra *scrivania* quando lavorate in R. Contiene tutti gli oggetti (variabili) creati durante la sessione di lavoro.

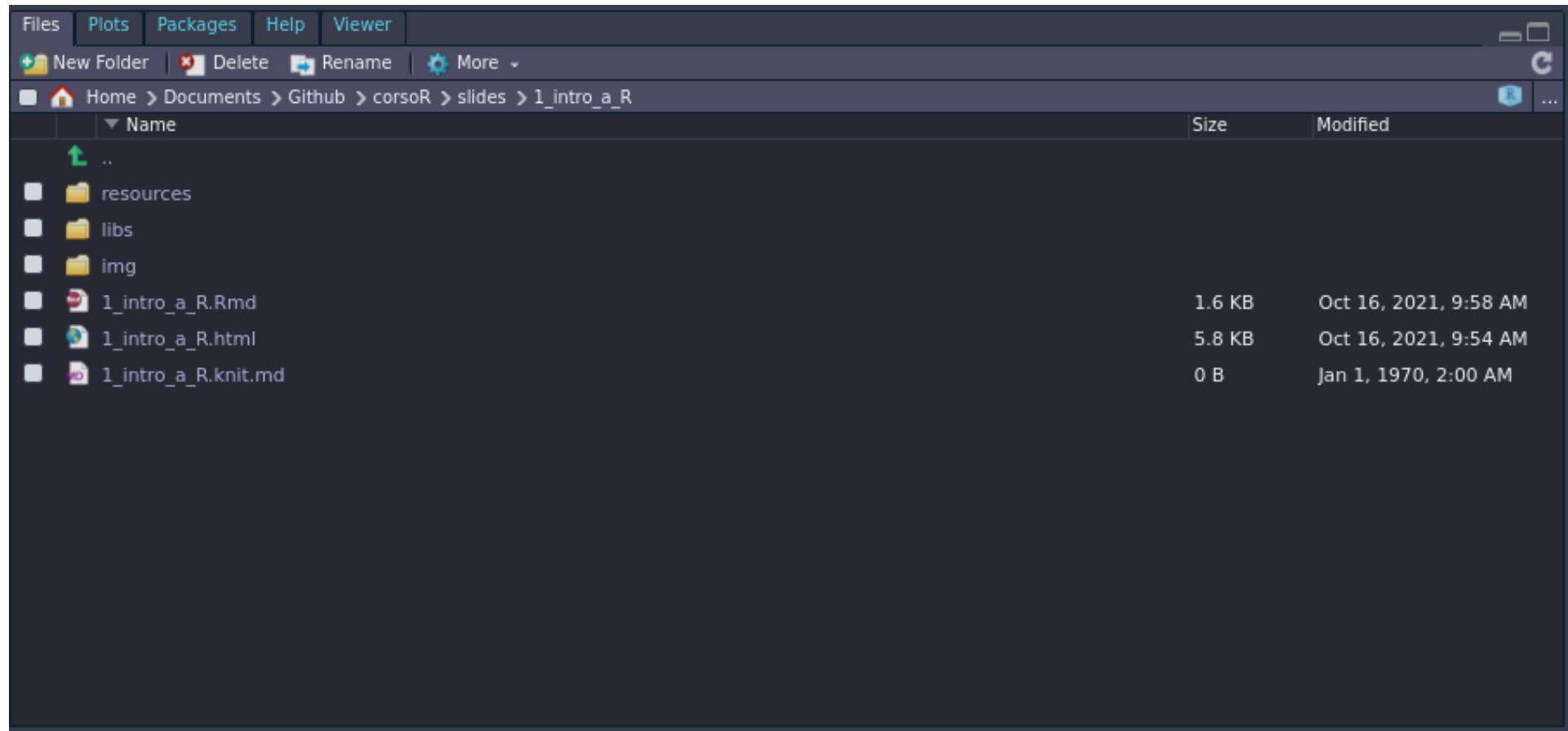


The screenshot shows the RStudio interface with the 'Environment' tab selected. The Global Environment pane displays the following objects:

- Data**:
  - iris: 150 obs. of 5 variables
- Values**:
  - x: 10
  - y: int [1:10] 1 2 3 4 5 6 7 8 9 10
  - z: chr [1:10] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
- Functions**:
  - myfun: function (x)

# Working Directory

La working directory è la posizione (cartella) sul vostro PC dove R sta lavorando e nella quale R si aspetta di trovare i vostri file, se non specificato altrimenti



# Packages

In R è possibile installare e caricare pacchetti aggiuntivi che non fanno altro che rendere disponibili librerie di funzioni create da altri utenti. Per utilizzare un pacchetto:

- Installare il pacchetto con `install.packages("nomepacchetto")`
- Caricare il pacchetto con `library(nomepacchetto)`
- Accedere ad una funzione senza caricare il pacchetto `nomepacchetto::nomefunzione()`. Utile se serve solo una funzione o ci sono conflitti

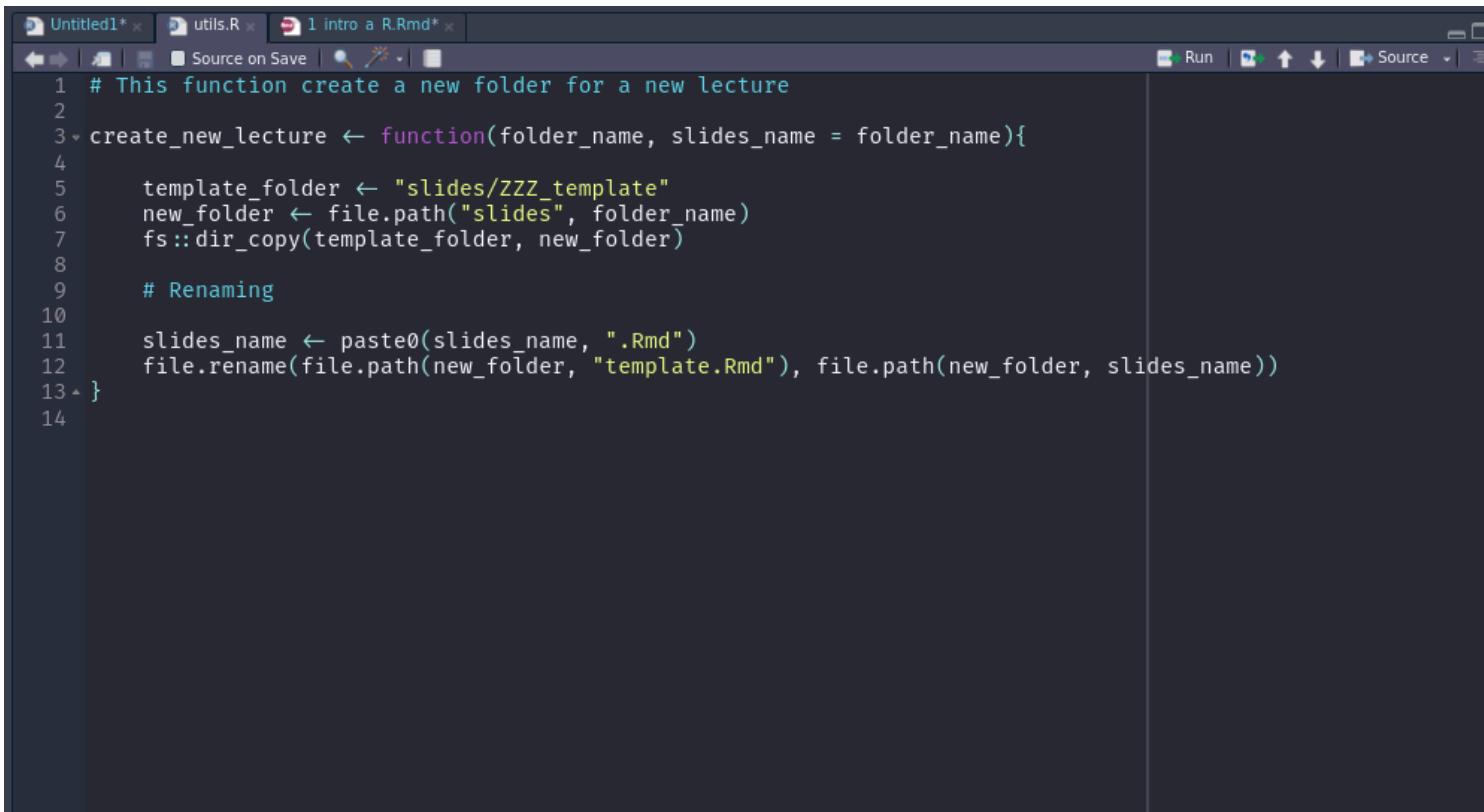
# Packages

Name	Description	Version	
<b>User Library</b>			
abind	Combine Multidimensional Arrays	1.4-5	● ●
anytime	Anything to 'POSIXct' or 'Date' Converter	0.3.9	● ●
arrayhelpers	Convenience Functions for Arrays	1.1-0	● ●
AsioHeaders	'Asio' C++ Header Files	1.16.1-1	● ●
askpass	Safe Password Entry for R, Git, and SSH	1.1	● ●
assertthat	Easy Pre and Post Assertions	0.2.1	● ●
backports	Reimplementations of Functions Introduced Since R-3.0.0	1.2.1	● ●
base64enc	Tools for base64 encoding	0.1-3	● ●
bayesplot	Plotting for Bayesian Models	1.8.1	● ●
BayesRS	Bayes Factors for Hierarchical Linear Models with Continuous Predictors	0.1.3	● ●
bayestestR	Understand and Describe Bayesian Models and Posterior Distributions	0.11.0	● ●
BH	Boost C++ Header Files	1.75.0-0	● ●
binom	Binomial Confidence Intervals For Several Parameterizations	1.1-1	● ●
bitops	Bitwise Operations	1.0-7	● ●
blob	A Simple S3 Class for Representing Vectors of Binary Data ('BLOBS')	1.2.2	● ●
bookdown	Authoring Books and Technical Documents with R Markdown	0.24	● ●
bridgesampling	Bridge Sampling for Marginal Likelihoods and Bayes Factors	1.1-2	● ●
brio	Basic R Input Output	1.1.2	● ●
brms	Bayesian Regression Models using 'Stan'	2.16.1	● ●
Broddingnag	Very Large Numbers in R	1.2-6	● ●
broom	Convert Statistical Objects into Tidy Tibbles	0.7.9	● ●

Come lavorare in R

# Scrivere e organizzare script

- Lo script è un file di testo dove il codice viene salvato e può essere lanciato in successione
- Nello script è possibile combinare **codice** e **commenti**



The screenshot shows the RStudio interface with three tabs at the top: "Untitled1\*", "utils.R\*", and "1 intro a R.Rmd\*". The main area displays the following R code:

```
1 # This function creates a new folder for a new lecture
2
3 create_new_lecture <- function(folder_name, slides_name = folder_name){
4
5   template_folder <- "slides/ZZZ_template"
6   new_folder <- file.path("slides", folder_name)
7   fs::dir_copy(template_folder, new_folder)
8
9   # Renaming
10
11  slides_name <- paste0(slides_name, ".Rmd")
12  file.rename(file.path(new_folder, "template.Rmd"), file.path(new_folder, slides_name))
13}
14
```

# R Projects

Gli `R projects` sono una feature implementata in R Studio per organizzare una cartella di lavoro

- permettono di impostare la **working directory** in automatico
- permettono di usare **relative path** invece che **absolute path**
- rendono più **riproducibile** e **trasportabile** il progetto
- permettono un **veloce accesso** ad un determinato progetto

# R Projects

Per capire meglio il funzionamento degli R projects e di come sono organizzati i file ho fatto un video che può chiarire la questione:



Come risolvere i problemi ~~nella vita~~ in R

# Come risolvere i problemi ~~nella vita~~ in R

In R gli errori sono:

- inevitabili
- parte del codice stesso
- educativi

Resta solo da capire come affrontarli



# R ed errori

Ci sono diversi livelli di **allerta** quando scriviamo codice:

- **messaggi**: la funzione ci restituisce qualcosa che è utile sapere, ma tutto liscio
- **warnings**: la funzione ci informa di qualcosa di *potenzialmente* problematico, ma (circa) tutto liscio
- **error**: la funzione non solo ci informa di un **errore** ma le operazioni richieste non sono state eseguite

Ne vedremo e vedrete molti usando R 😊

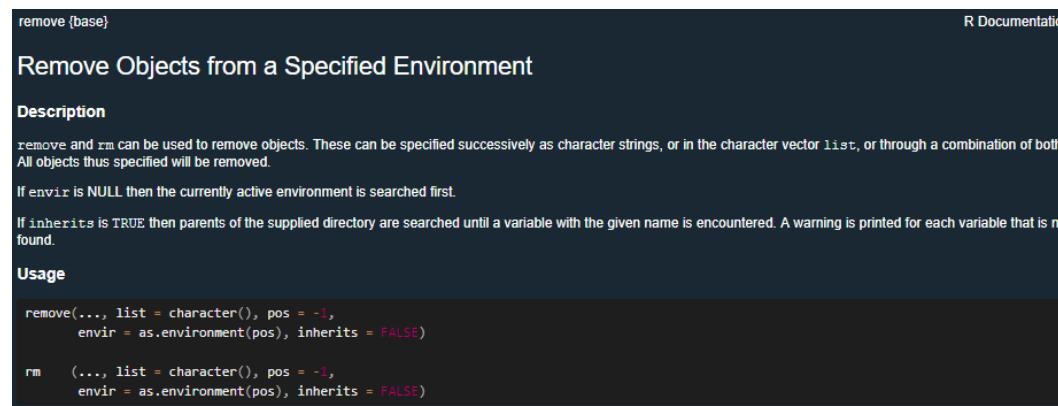
# Come risolvere un errore?

- capire il messaggio
- leggere la documentazione della funzione
- cercare il messaggio su Google
- chiedere aiuto nei forum dedicati



# Come risolvere un errore?

- Ogni funzione ha una pagina di documentazione accessibile con `?nomefunzione`, `??nomefunzione` oppure `help(nomefunzione)`
- Possiamo cercare anche la documentazione del pacchetto
- Possiamo cercare su Google il nome della funzione o l'eventuale messaggio che riceviamo



*help(rm)*

# Stack overflow

**Stack overflow** è un forum di discussione riguardo qualsiasi cosa coinvolga codice (statistica, programmazione, etc.). E' pieno di errori comuni, *How to do ...* e di risposte/soluzioni estremamente utili. Nel 90% dei casi il problema che avete è comune ed è già presente una soluzione.

The screenshot shows a Stack Overflow page for a question titled "How to make a great R reproducible example". The question has 2468 upvotes and was asked 11 years, 5 months ago. It has been modified 2 months ago and viewed 414k times. A note indicates that answers are a community effort and not accepting new ones. The question text discusses the importance of reproducible examples in R. The sidebar includes links for Home, PUBLIC, Questions, Tags, Users, Companies, COLLECTIVES, TEAMS, and various blog posts from The Overflow Blog.

stackoverflow Products Search... 171 1 7 +20 ? Ask Question

Home PUBLIC Questions Tags Users Companies COLLECTIVES Explore Collectives TEAMS

How to make a great R reproducible example

Asked 11 years, 5 months ago Modified 2 months ago Viewed 414k times

2468

This question's answers are a [community effort](#). Edit existing answers to improve this post. It is not currently accepting new answers or interactions.

When discussing performance with colleagues, teaching, sending a bug report or searching for guidance on mailing lists and here on Stack Overflow, a [reproducible example](#) is often asked and always helpful.

What are your tips for creating an excellent example? How do you paste data structures from [r](#) in a text format? What other information should you include?

The Overflow Blog

- Introducing the Ask Wizard: Your guide to crafting high-quality questions
- How to get more engineers entangled with quantum computing (Ep. 501)

Featured on Meta

- The 2022 Community-a-thon has begun!

# Se non trovo una soluzione?

Se non trovo una soluzione posso chiedere. Fare una domanda riguardo un errore o un problema di codice non è semplice come sembra. Le fonti dell'errore possono essere molteplici (il mio specifico computer, un pacchetto che ho installato, il codice sorgente etc.). Ecco una guida per chiedere in modo efficace:

The screenshot shows the Stack Overflow website's "How do I ask a good question?" guide. The left sidebar includes links for Home, PUBLIC (Questions, Tags, Users, Companies), COLLECTIVES (Explore Collectives), and TEAMS (Stack Overflow for Teams). The main content area has a breadcrumb trail: Stack Overflow > Help center > Asking. The title is "How do I ask a good question?". Below it, text says: "We're happy to help you, but in order to **improve your chances** of getting an answer, here are some **guidelines to follow**:". A section titled "Make sure your question is on-topic and suitable for this site" explains that Stack Overflow only accepts certain types of questions about programming and software development, and that questions must be written in English. It also notes that closure is not the end of the road for questions. The right sidebar is titled "Asking" and lists various related topics.

Stack Overflow > Help center > Asking

## How do I ask a good question?

We're happy to help you, but in order to **improve your chances** of getting an answer, here are some **guidelines to follow**:

**Make sure your question is on-topic and suitable for this site**

Stack Overflow only accepts certain types of questions about programming and software development, and your question must be written in English. If your question is not on-topic or is otherwise unsuitable for this site, then it will likely be closed.

Closure is not the end of the road for questions; it is intended to be a temporary state until the question is revised to meet our requirements. However, if you fail to do that, or it is impossible to do so, then the question will stay closed and will not be answered.

Since you're reading this page, hopefully you will post a suitable, on-topic question from the outset, thus eliminating the need for the closure and reopening process!

**Asking**

- What types of questions should I avoid asking?
- What topics can I ask about here?
- What does it mean if a question is "closed"?
- What is the Ask Wizard?
- How do I ask a good question?**
- Why do I see a message that my question does not meet quality standards?
- Why are questions no longer being accepted from my account?
- What should I do when someone answers my question?
- Why is the system asking me to wait a day or more before asking another question?
- What are tags, and how should I use them?

<https://stackoverflow.com/help/how-to-ask>