Introduction to R

Corso per imparare le basi di **R**

Claudio Zandonella Callegher and Filippo Gambarota members of Psicostat

08-12-2021

Contents

Presentazione

In questo libro impareremo le basi di *R*, uno dei migliori software per la visualizzazione e l'analisi statistica dei dati. Partiremo da zero intorducendo gli aspetti fondamentili di R e i concetti alla base di ogni linguaggio di programmazione che ti pemetteranno in seguito di approfondire e sviluppare le tue abilità in questo bellissimo mondo.

Perchè R

Ci sono molte ragioni per cui scegliere R rispetto ad altri programmi usati per condurre le analisi statistiche. Innanzitutto è un linguaggio di programmazione (come ad esempio Python, Java, C++, o Julia) e non semplicemente un'interfaccia punta e clicca (come ad esempio SPSS o JASP). Questo comporta si maggiori difficoltà iniziali ma ti ricompenserà in futuro poichè avari imparato ad utilizza uno strumennto molto potente.

Inoltre, R è:

- nato per la statistica
- open-source
- ricco di pacchetti
- supportato da una grande community
- gratis

Struttura del libro

Il libro è suddiviso in quattro sezioni principali:

- **Get started**. Una volta installato R ed RStudio, famiglierizzeremo con l'ambiente di lavoro introducendo alcuni aspetti generali e le funzioni principali. Verranno inoltre descritte alcune buone regole per iniziare una sessione di lavoro in R.
- **Struttura dei dati**. Impareremo gli oggetti principali che R utilizza al suo interno. Variabili, vettori, matrici, dataframe e liste non avranno più segreti e capiremo come manipolarli e utlizzarli a seconda delle varie necessità.
- **Algoritmi**. Non farti spaventare da questo nome. Ne avrai spesso sentito parlarne come qualcosa di molto complicato, ma in realtà gli algoritmi sono semplicemente una serie di istruzioni che il computer segue quando deve eseguire un determinato compito. In questa sezione vedremo i principali comandi di R usati per definire degli algoritmi. Questo è il vantaggio di conoscere un linguaggio di programmazione, ci permette di creare nuovi programmi che il computer eseguirà per noi.

- **Case study**. Eseguiremo passo per passo un analisi che ci permetterà di imparare come importare i dati, codificare le variabili, manipolare e preprare i dati perle analisi, condurre delle analisi descrittive e creare dei grafici.

Alla fine di questo libro probabilmente non sarete assunti da Google, ma speriamo almeno che R non vi faccia più così paura e che magari a qualcuno sia nato l'interesse di approfondire questo fantastico mondo fatto di linee di codice.

Risorse Utili

Segnaliamo qui per il lettore interessato del materiale online (in inglese) per approfondire le conoscenze sull'uso di R.

Materiale introduttivo:

- *R for Psychological Science* di Danielle Navarro https://psyr.djnavarro.net/index.html
- *Hands-On Programming with R* di Garrett Grolemund https://rstudio-education.github.io/hopr/

Materiale intermedio:

- *R for Data Science* di Hadley Wickham e Garrett Grolemund https://r4ds.had.co.nz/

Materiale avanzato:

- *R Packages* di Hadley Wickham e Jennifer Bryan https://r-pkgs.org/
- *Advanced R* di Hadley Wickham https://adv-r.hadley.nz/

Psicostat

Questo libro è stato prodotto da Claudio Zandonella Callegher and Filippo Gambarota, membri di **Psicostat**. Un gruppo di ricerca interdisciplinare dell'universita di Padova che unisce la passione per la statistica e la psicologia. Se vuoi conoscere di più riguardo le nostre attività visita il nosto sito https://psicostat.dpss.psy.unipd.it/ o aggiungiti alla nostra mailing list https://lists.dpss.psy.unipd.it/postorius/lists/psicostat.lists.dpss.psy.unipd.it/.

Collaborazione

Se vuoi collaborare alla revione e scrittura di questo libro (ovviamente è tutto in R) visita la nostra repository di Github https://github.com/psicostat/Introduction2R.

Riconoscimenti

Il template di questo libro è basato su Rstudio Bookdown-demo rilasciato con licenza CC0-1.0 e rstudio4edu-book rilasciato con licenza CC BY. Nota che le illustrazioni utilizzate nelle vignette appartengono sempre a rstudio4edu-book e sono rilasciate con licenza CC BY-NC.

Licenza

Get Started

Introduzione

In questa sezione verranno prima presentate le istruzioni per installare R ed RStudio. Successivamente, svolgeremo le prime oprazioni in R e famiglierizzeremo con dei concetti di base della programmazioene quali gli oggetti e le funzioni. Introdurremmo infine altri concetti relativi alle sessioni di lavoro in R e descriveremo alcune buone regole per nell'utilizzo di R.

I capitoli sono così organizzati:

- **Capitolo 1 - Installare R e RStudio**. Instruzioni passo a passo per installare R e RStudio
- **Capitolo 2 - Interfaccia RStudio**. Introduzione all'interfaccia utente di RStudio.
- **Capitolo 3 - Primi Passi in R**. Operatori matematici, operatori relazionali, operatori logici.
- **Capitolo 4 - Due Compagni Inseparabili**. Introduzione dei concetti di oggetti e funzioni in R.
- **Capitolo 5 - Ambiente di Lavoro**. Introduzione dei concetti di Envieronment, working directory e dei pacchetti di R.
- **Capitolo 6 - Sessione di Lavoro**. Descrizione di buone pratiche nelle sessioni di lavoro e gestione degli errori.

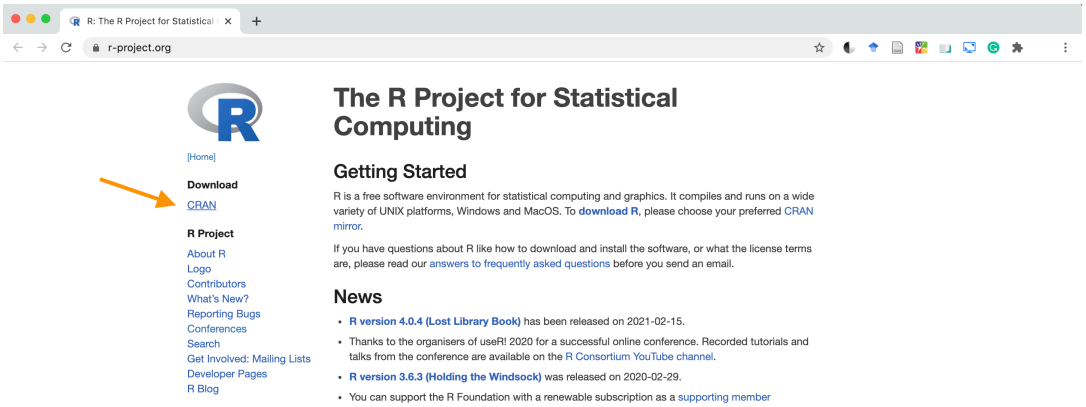
Chapter 1

Installare R e RStudio

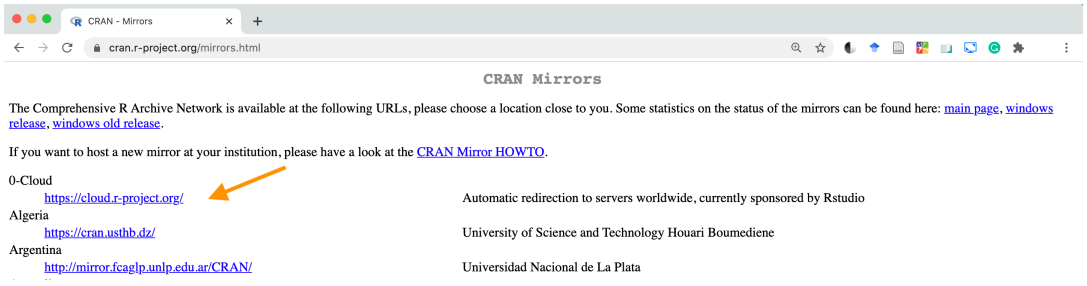
R ed R-studio sono due software distinti. R è un linguaggio di programmazione usato in particolare in ambiti quali la statistica. R-studio invece è un'interfaccia *user-friendly* che permette di utilizzare R. R può essere utilizzato autonomamente tuttavia è consigliato l'utilizzo attraverso R-studio. Entrambi vanno installati separatamente e la procedura varia a seconda del proprio sistema operativo (Windows, MacOS o Linux). Riportiamo le istruzioni solo per Windows e MacOS Linux (Ubuntu). Ovviamente R è disponibile per tutte le principali distribuzioni di Linux. Le istruzioni riportate per Ubuntu (la distribuzione più diffusa) sono valide anche per le distribuzioni derivate.

1.1 Installare R

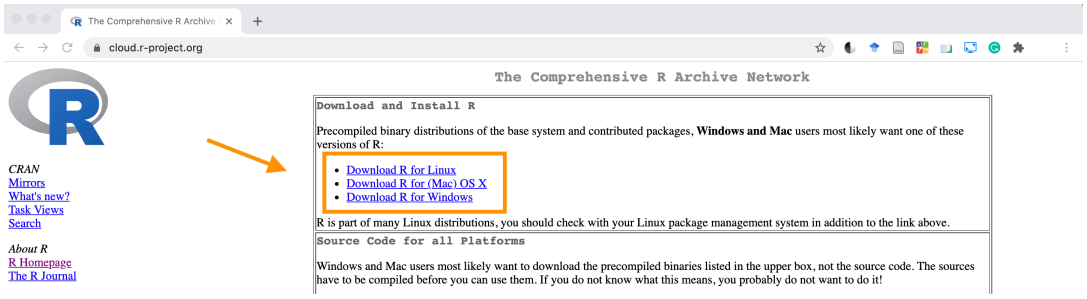
1. Accedere al sito https://www.r-project.org
2. Selezionare la voce **CRAN** (Comprehensive R Archive Network) dal menù di sinistra sotto **Download**



3. Selezionare il primo link https://cloud.r-project.org/

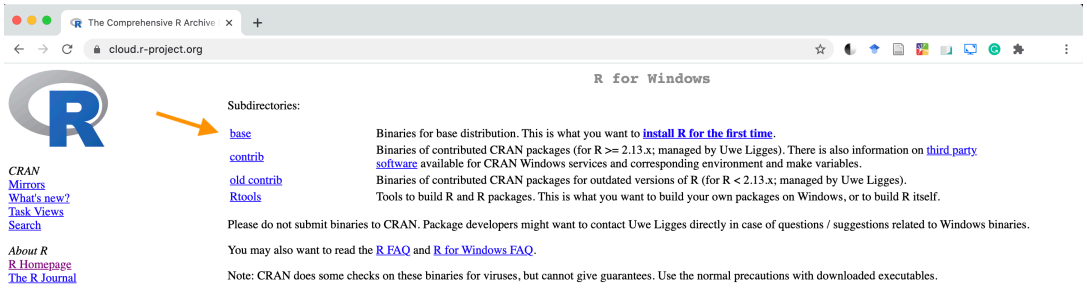


4. Selezionare il proprio sistema operativo

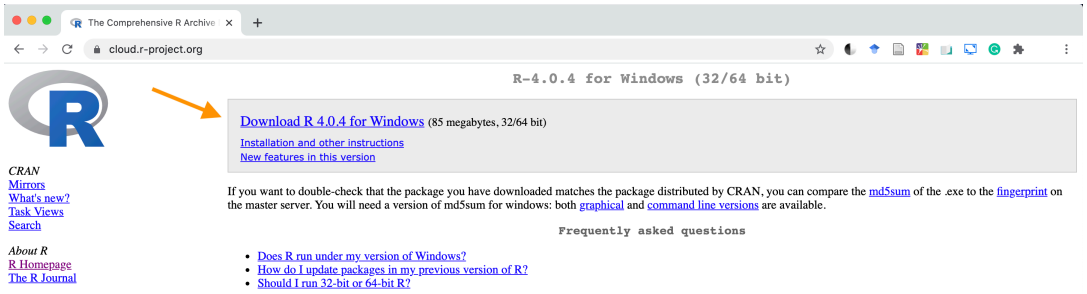


1.1.1 R Windows

1. Selezionare la voce **base**



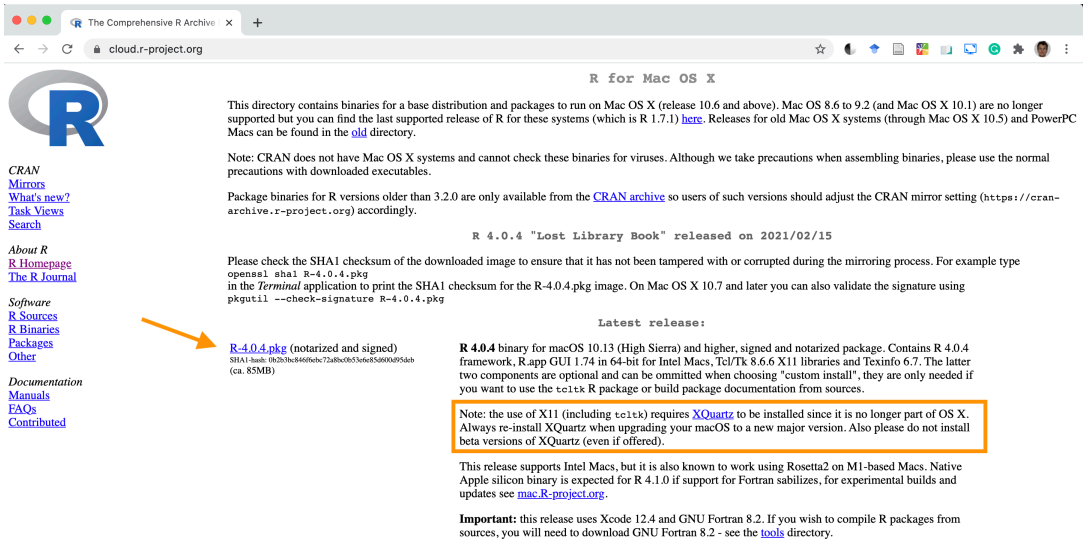
2. Selezionare la voce **Download** della versione più recente di R disponibile



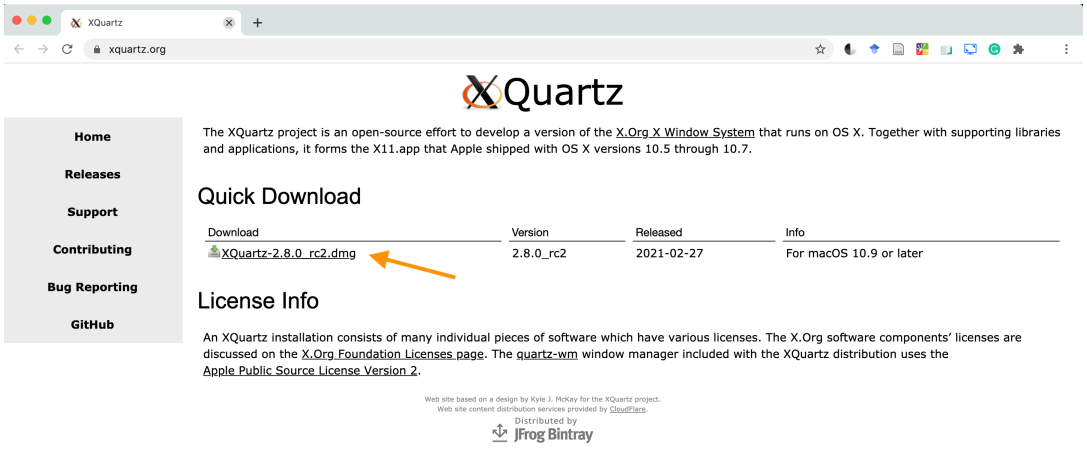
3. Al termine del download, eseguire il file e seguire le istruzioni fino al termine dell'installazione

1.1.2 R MacOS

1. Selezionare della versione più recente di R disponibile



2. Al termine del download, eseguire il file e seguire le istruzioni fino al termine dell'installazione di R
3. Successivamente è necessario installare anche una componente aggiuntiva **XQuartz** premendo il link all'interno del riquadro arancione riportato nella figura precedente
4. Selezionare la voce Download



5. Al termine del download, eseguire il file e seguire le istruzioni fino al termine dell'installazione

1.1.3 R Linux

Nonostante la semplicità di installazione di pacchetti su Linux, R a volte potrebbe essere più complicato da installare per via delle diverse distribuzioni, repository e chiavi per riconoscere la repository come sicura.

Sul **CRAN** vi è la guida ufficiale con tutti i comandi `apt` da eseguire da terminale. Seguendo questi passaggi non dovrebbero esserci problemi.

1. Andate sul CRAN
2. Cliccate `Download R for Linux`
3. Selezionate la vostra distribuzione (Ubuntu in questo caso)
4. Seguite le istruzioni, principalmente eseguendo i comandi da terminale suggeriti

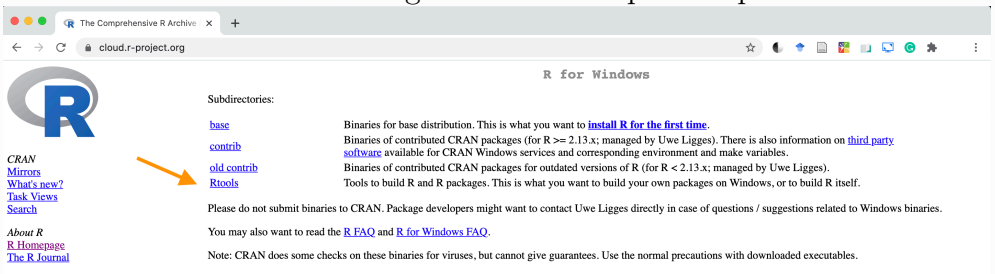
Per qualsiasi difficoltà o errore, sopratutto con il mondo Linux, una ricerca su online risolve sempre il problema.

Approfondimento: R Tools

Utilizzi avanzati di R richiedono l'insallazione di una serie ulteriore software definiti **R tools**.

Windows

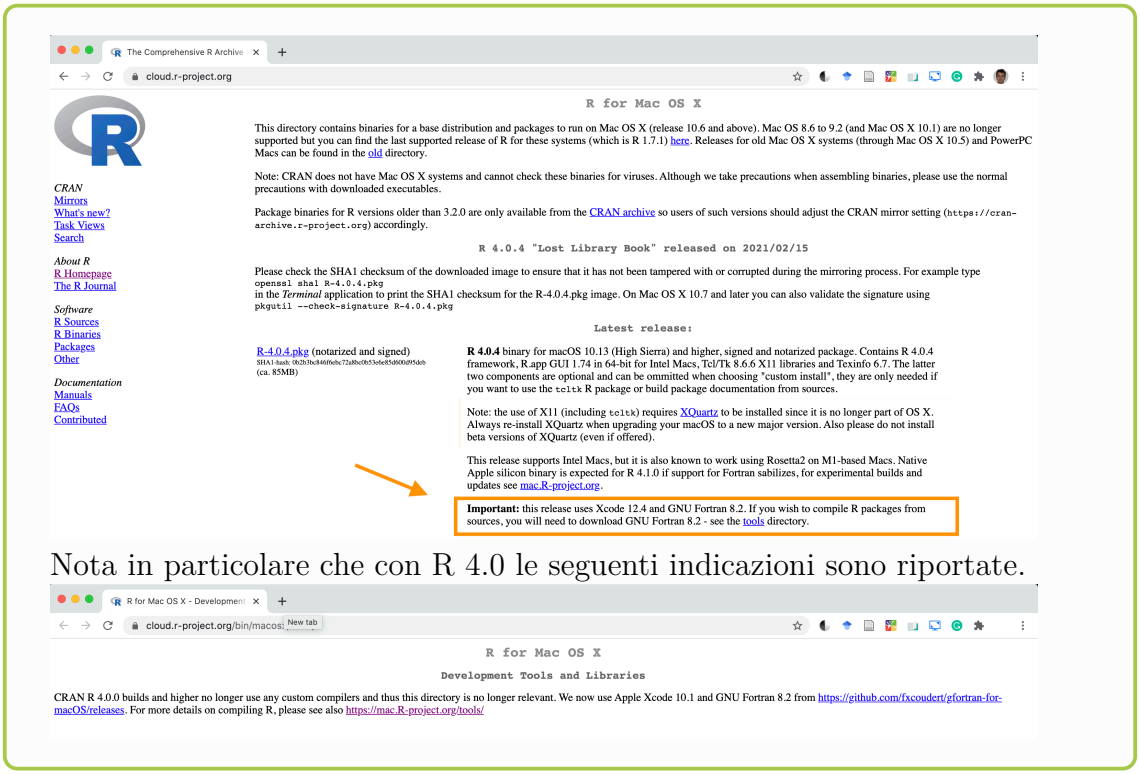
Seleziona la voce **Rtools** e segui le istruzioni per completare l'installazione.



Nota che sono richieste anche delle operazioni di configurazione affinchè tutto funzioni correttamente.

MacOS

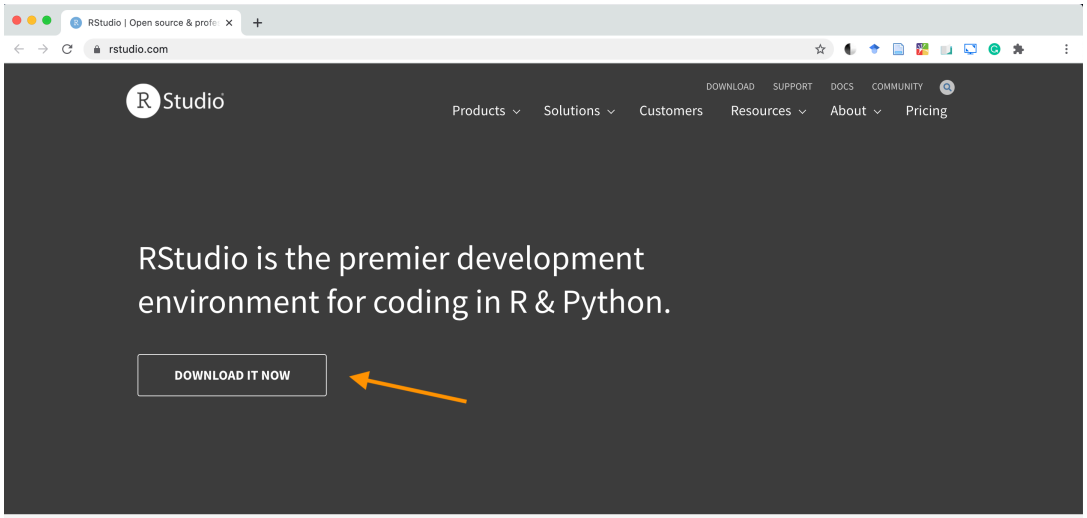
Seleziona la voce **tools** e segui le istruzioni riportate.



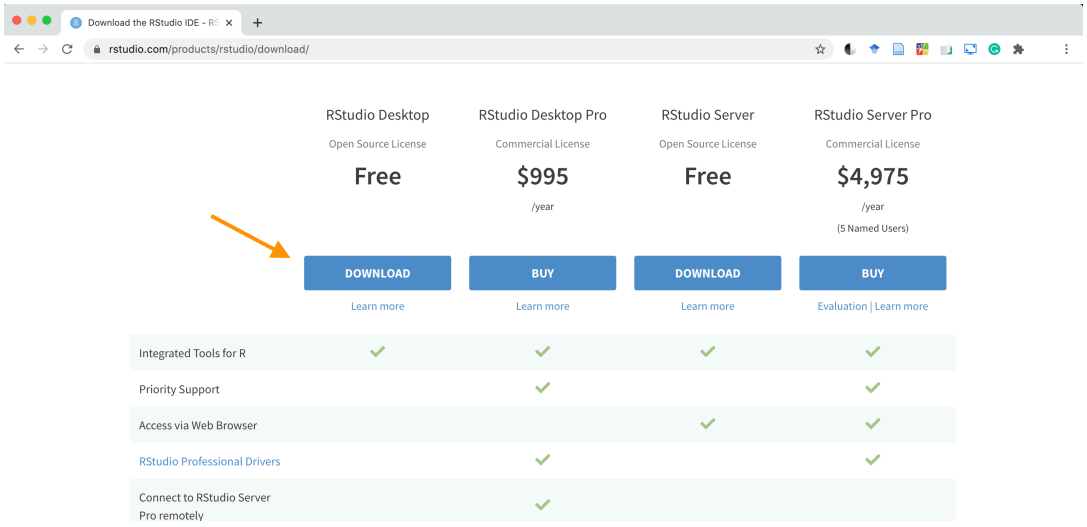
Nota in particolare che con R 4.0 le seguenti indicazioni sono riportate.

1.2 Installare R Studio

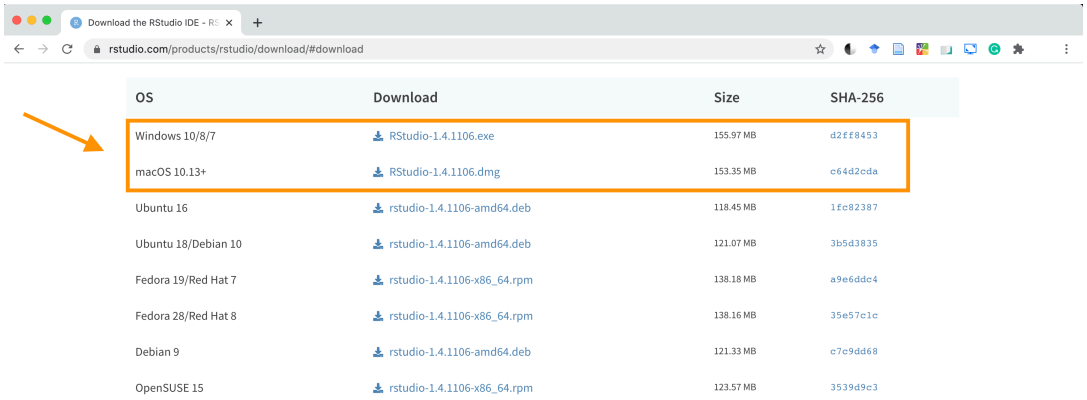
1. Accedere al sito https://rstudio.com
2. Selezionare la voce **DOWNLOAD IT NOW**



3. Selezionare la versione gratuita di RStudio Desktop



4. Selezionare la versione corretta a seconda del proprio sistema operativo



5. Al termine del download, eseguire il file e seguire le istruzioni fino al termine dell'installazione

1.2.1 R Studio in Linux

In questo caso, come su Windows e MacOS l'installazione consiste nello scaricare ed eseguire il file corretto, in base alla distribuzione (ad esempio `.deb` per Ubuntu e derivate). Importante, nel caso di Ubuntu (ma dovrebbe valere anche per le altre distribuzioni) anche versioni successive a quella indicata (es. Ubuntu 16) sono perfettamente compatibili.

Chapter 2

Interfaccia RStudio

In questo capitolo presenteremo l'interfaccia utente di RStudio. Molti aspetti che introdurremo brevemente qui verranno discussi nei sucessivi capitoli. Adesso ci interessa solo famigliarizzare con l'interfaccia del nostro strumento di lavoro principale ovvero RStudio.

Come abbiamo visto nel Capitolo 1, R è il vero "motore computazionale" che ci permette di compiere tutte le operazioni di calcolo, analisi statistiche e magie varie. Tuttavia l'interfaccia di base di R, definita **Console** (vedi Figura 2.1), è per così dire *démodé* o meglio, solo per veri intenditori.

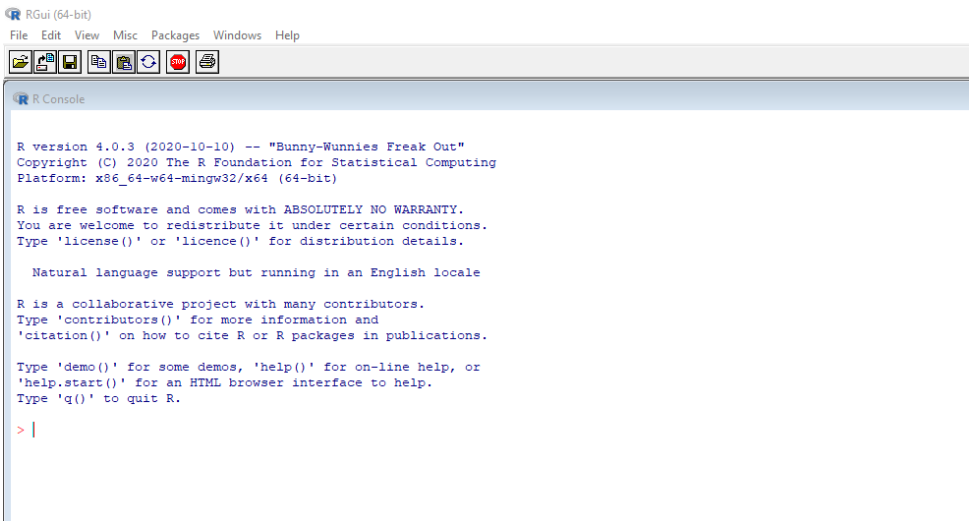


Figure 2.1: La console di R, solo per veri intenditori

In genere, per lavorare con R viene utilizzato RStudio. RStudio è un programma (IDE - Integrated Development Environment) che integra in un unica interfaccia utente (GUI - Graphical User Interface) diversi strumenti utili per la scrittura ed esecuzione di codici. L'interfaccia di RStudio è costituita da 4 pannelli principali (vedi Figura 2.2):

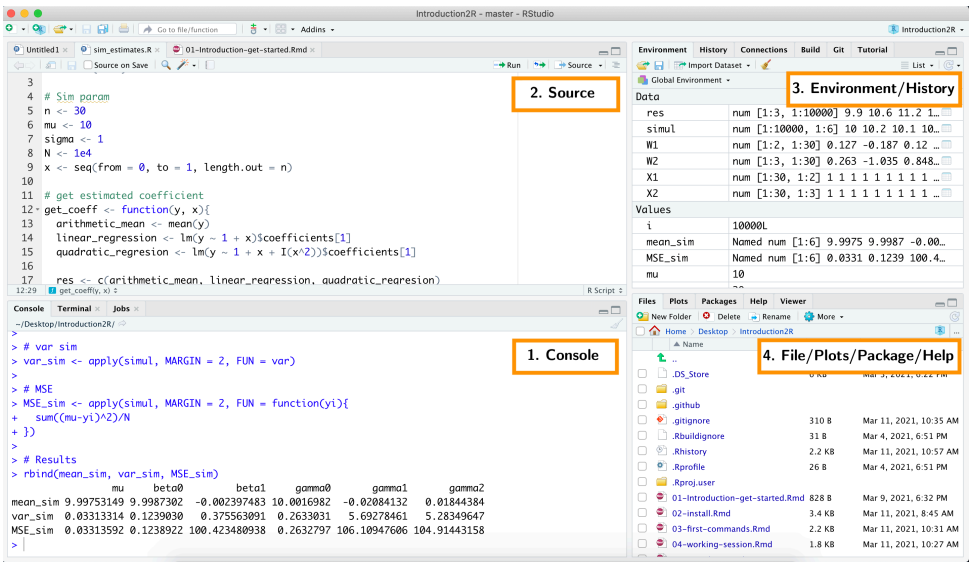


Figure 2.2: Interfaccia utente di Rstudio con i suoi 4 pannelli

Approfondimento: R-Basic vs RStudio

L'utilizzo di R attraverso l'interfaccia di base piuttosto che RStudio, non è uno scontro tra due scuole di pensiero (o generazioni). Entrambe hanno vantaggi e svantaggi e pertanto vengono scelte a seconda delle dieverse necessità. Quando si è alla ricerca della massima ottimizzazione, l'uso dell'interfaccia di base, grazie alla sua semplicità, permette di minimizzare l'utilizzo della memoria limitandosi allo stretto e necessario.
In altri casi, invece, le funzionalità e strumenti aggiuntivi di RStudio permettono una maggiore efficacia nel proprio lavoro.

1. Console: il cuore di R

Qui ritroviamo la *Console* di R dove vengono effetivemente eseguiti tutti i tuoi codici e comandi. Nota come nell'ulitma riga della *Console* appaia il carattere `>`. Questo è definito *prompt* è ci indica che R in attesa di nuovi comandi da eseguire.

La *Console* di R è un'interfaccia a linea di comando. A differenza di altri programmi "*punta e clicca*", in R è necessario digitare i comandi utilizzando la tastiera. Per eseguire dei comandi possiamo direttamnte scrivere nella *Console* le operazioni da eseguire e premere `invio`. R eseguirà immediatamente i nostro comando, riporterà il risultato e nella linea successiva apparirà nuovamente il *prompt* indicando che R è pronto ad eseguire un altro comando (vedi Figura 2.3).

Nel caso di comandi scritti su più righe, vedi l'esempio di Figura 2.4, è possibile notare come venga mostrato il simbolo `+` come *prompt*. Questo indica che R è in attesa che l'intero comando venga digitato prima che esso venga eseguito.

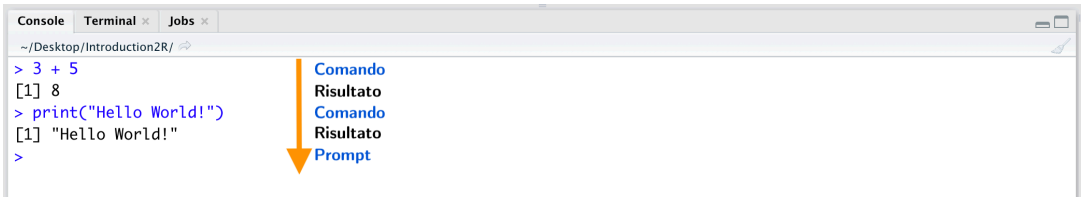


Figure 2.3: Esecuzione di comandi direttamente nella console

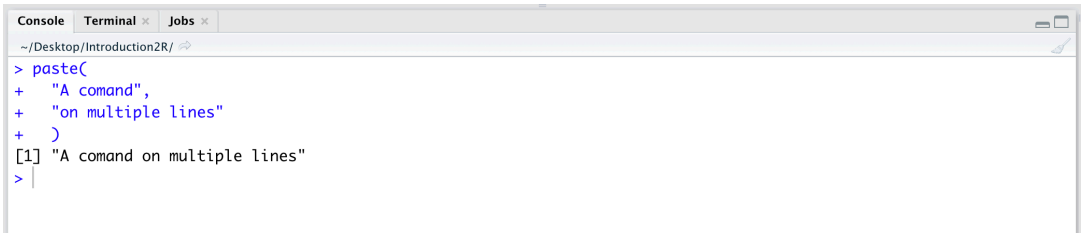


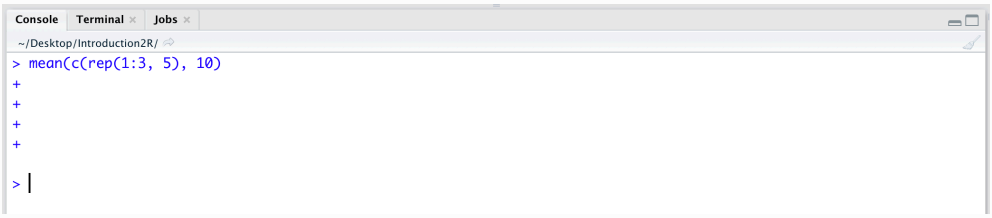
Figure 2.4: Esecuzione di un comando su più righe

Come avrai notato facendo alcune prove, i comandi digitati nella *Console* vengono eseguiti immediatamente ma non sono salvati. Per rieseguire un comando, possiamo navigare tra quelli precedentemente eseguiti usando le freccie della tastiera ↑↓. Tuttavia, in caso di errori dovremmo riscrivere e rieseguire tutti i comandi. Siccome scrivere codici è un continuo "*try and error*", lavorare unicamente dalla *Console* diventa presto caotico. Abbiamo bisogno quindi di una soluzione che ci permetta di lavrorare più comodamente sui nostri codici e di poter salvare i nostri comandi da eseguire all'occorrenza con il giusto ordine. La soluzione sono gli *Scripts* che introdurremo vedremo nella prossima sezione.

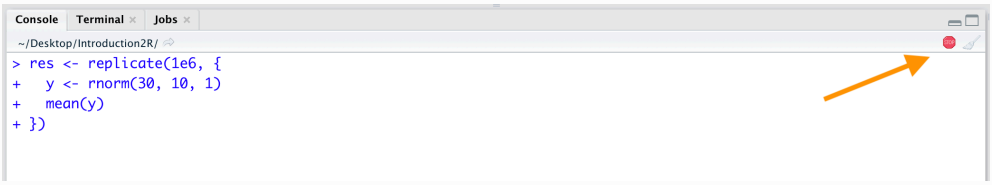
Tip-Box: Interrompere un comando

Potrebbe accadere che per qualche errore nel digitare un comando o perchè sono richiesti lunghi tempi computazionali, la *Console* di R diventi non responsiva. In questo caso è necessario interrompere la scrittura o l'esecuzione di un comando. Vediamo due situazioni comuni:

1. **Continua a comparire il prompt `+`**. Specialmente nel caso di utilizzo di parentesi e lunghi comandi, accade che una volta premuto `invio` R non esegua alcun comando ma resta in attesa mostrando il *prompt* `+` (vedi Figure seguente). Questo è in genere dato da un errore nella sintassi del comando (e.g., un errore nell'uso delle parentesi o delle virgole). Per riprendere la sessione è necessario premere il tasto `esc` della tastiera. L'apprire del *prompt* `>`, indica che R è nuovamente in ascolto pronto per esequire un nuovo comando ma attento a non ripetere lo stesso errore.



2. **R non risponde**. Alcuni calcoli potrebbero richiedere molto tempo o semplicemnte un qualche problema ha mandato in loop la tua sessione di lavoro. In questa situazione la *Console* di R diventa non responsiva. Nel caso fosse necessario interrompere i processi attualmente in esecuzione devi premere il pulsante *STOP* come indicato nella Figura seguente. R si fermerà e ritornerà in attesa di nuovi comandi (*prompt* >).



Trick-Box: Force Quit

In alcuni casi estremi in cui R sembra non rispondere, usa i comandi `Ctrl-C` per forzare R a interrompere il processo in esecuzione.
Come ultima soluzione ricorda uno dei principi base dell'informatica "*spegni e riaccendi*" (a volte potrebbe bastare chiudere e riaprire RStudio).

2. Source: il tuo blocco appunti

In questa parte vengono mostrati i tuoi *Scripts*. Questi non sono altro che degli speciali documenti (con estensione "**.R**") in cui sono salvati i tuoi codici e comandi che potrai eseguire quando necessario in R. Gli *Scripts* ti permetteranno di lavorare comodamente sui tuoi codici, scrivere i comandi, corregerli, organizzarli, aggiungere dei commenti e soprattutto salvarli.

Dopo aver terminato di scrivere i comandi, posiziona il cursore sulla stessa linea del comando che desideri eseguire e premi `command + invio` (MacOs) o `Ctrl+R` (Windows). Automaticamente il comando verà copiato nella *Console* ed eseguito. In alternativa potrai premere il tasto **Run** indicato dalla freccia in Figura 2.5.

Tip-Box: Commenti

Se hai guardato con attenzione lo script rappresentato in Figura 2.5, potresti

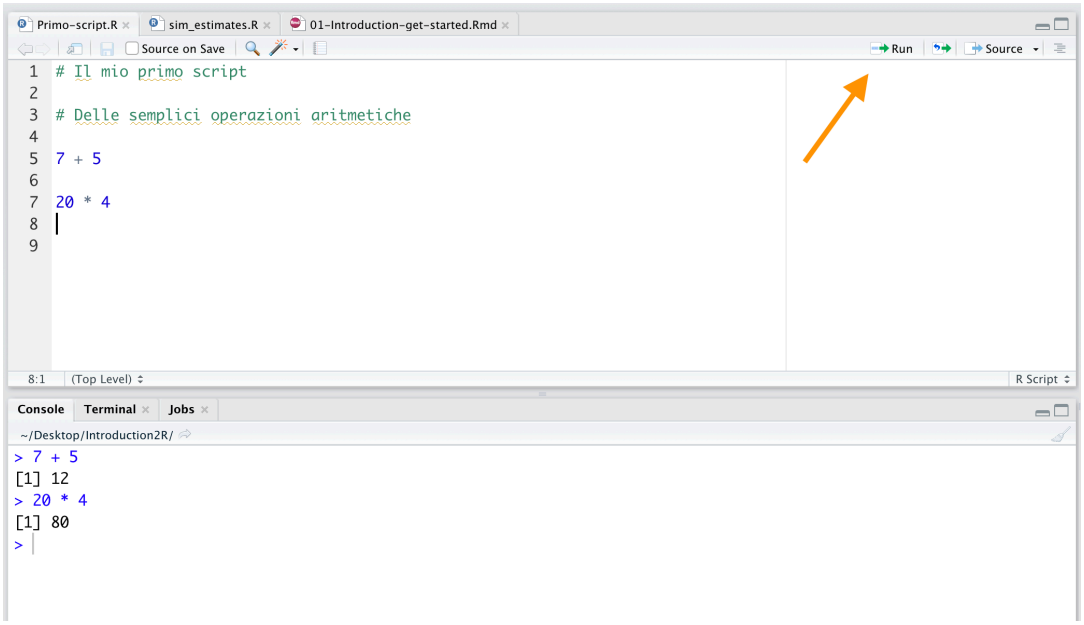
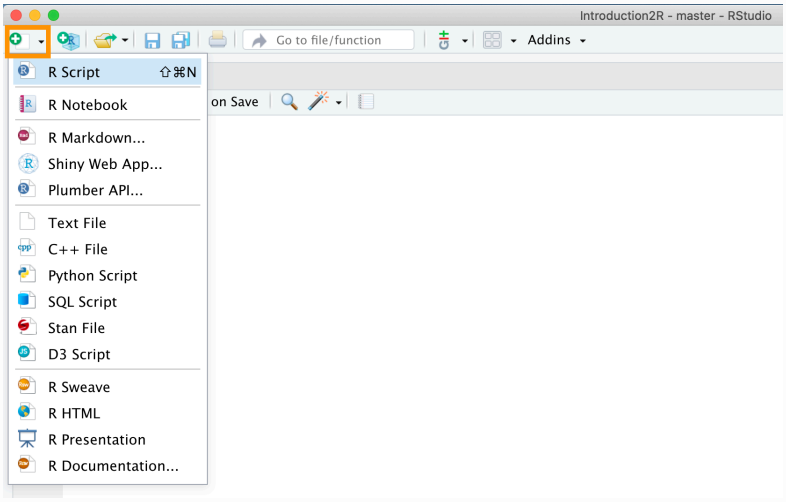


Figure 2.5: Esecuzione di un comando da script premi 'command + invio' (MacOs)/ 'Ctrl+R' (Windows) o premi il tasto indicato dalla freccia

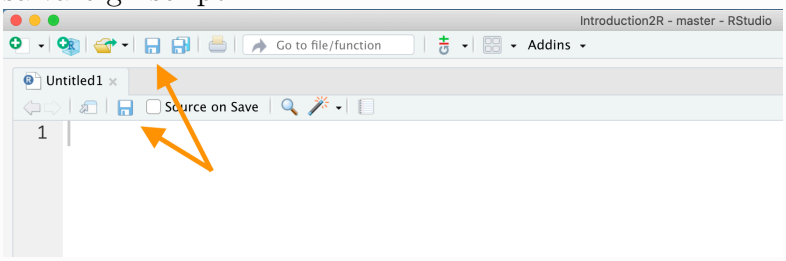
aver notato delle righe di testo verde precedute dal simbolo **#**. Questo simbolo può essere utlizzato per inserire dei *commenti* all'interno dello script. R ignorerà qualsiasi commento ed eseguirà soltato le parti di codici.
L'utilizzo dei commenti è molto importante nel caso di script complessi poichè ci permette di spiegare e documentare il codice che viene eseguito. Nel Capitolo 6.1.2 approfondiremo il loro utilizzo.

Approfondimento: Creare e Salvare uno Script

Per creare un nuovo script è sufficiente premere il pulsante in alto a sinistra come mostrato in Figura e selezionare *"R Script"*.



Unn nuovo script senza nome verrà creato. Per salvare lo script premere l'icona del floppy e indicare il nome. Ricorda di usare l'estensione "**.R**" per salvare gli script.



3. Environment e History: la sessione di lavoro

Qui sono presentati una serie di pannelli utili per valutare informazioni inerenti alla propria sessione di lavoro. I pannelli principali sono *Environment* e *History* (gli altri pannelli presenti in Figura 2.6 riguardanno funzioni avanzate di RStudio).

- **Environment**: elenco tutti gli oggetti e variabili attualmente presenti nel'ambiente di lavoro. Approfondiremo i concetti di variabili e di ambiente di lavoro rispettivamente nel Capitolo 4.1 e Capitolo 5.1.

- **History**: elenco di tutti i comandi precedentemente eseguiti nella console. Nota che questo no equivale ad uno script, anzi, è semplicemente un elenco non modificabile (e quasi mai usato).

4. File, Plots, Package, Help: system management

In questa parte sono raccolti una serie di pannelli utilizzatti per interfacciarsi con ulteriori risorse del sistema (e.g., file e pacchetti) o produrre output quali grafici e tabelle.

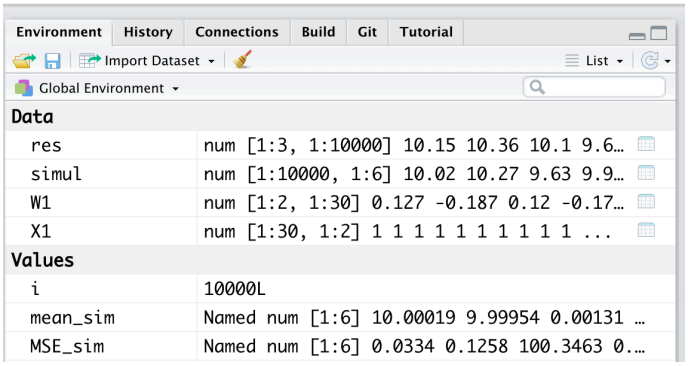


Figure 2.6: *Environment* - Elenco degli oggetti e variabili presenti nel'ambiente di lavoro

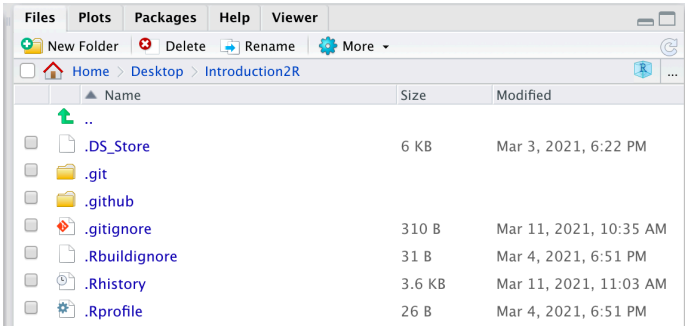


Figure 2.7: *Files* - permette di navigare tra i file del proprio computer

- **Files**: pannello da cui è possibile navigare tra tutti i file del proprio computer

- **Plots**: pannello i cui vengono prodotti i grafici e che è possibil esportare cliccando *Export*.

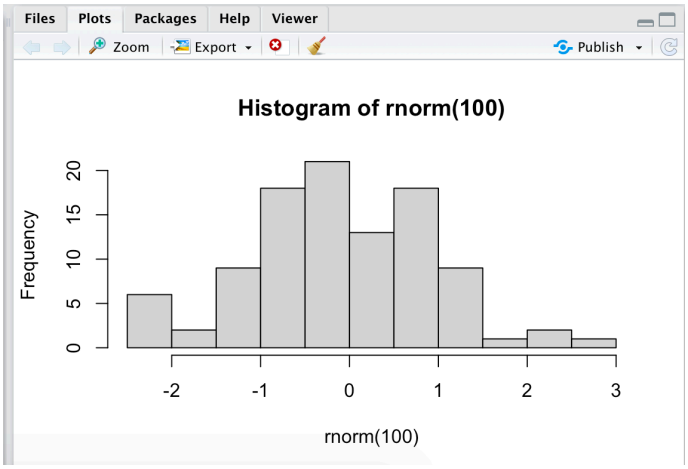


Figure 2.8: *Plots* - presentazione dei grafici

- **Packages**: elenco dei pacchetti di R (questo argomento verrà approfondito nel Capitolo 5.3).

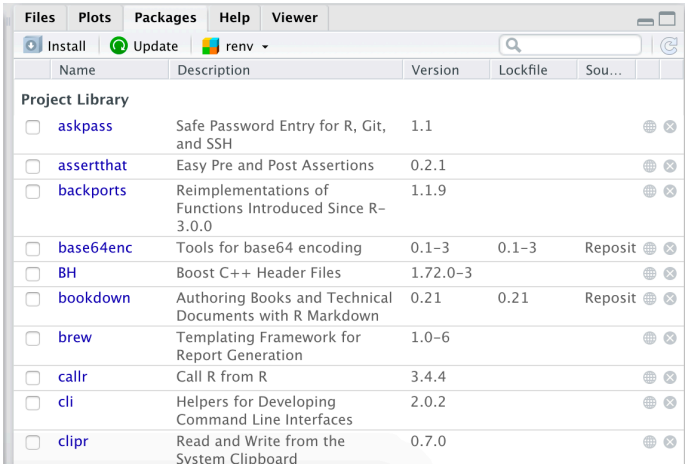


Figure 2.9: *Packages* - elenco dei pacchetti di R

- **Help**: utilizzato per navigare la documentazione interna di R (questo argomento verrà approfondito nel Capitolo 4.2.2).

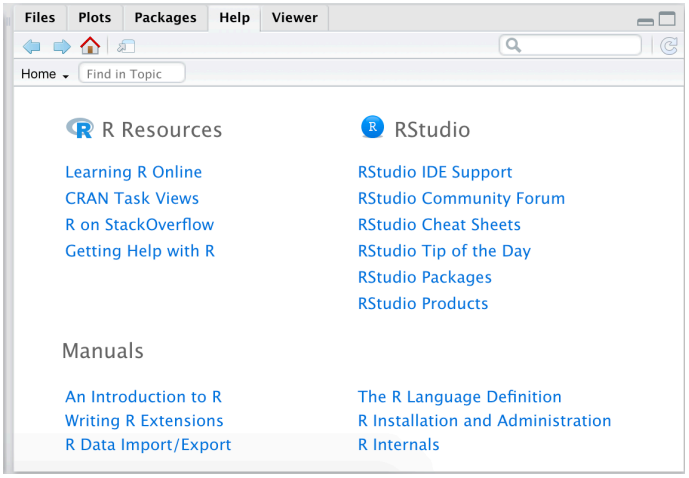
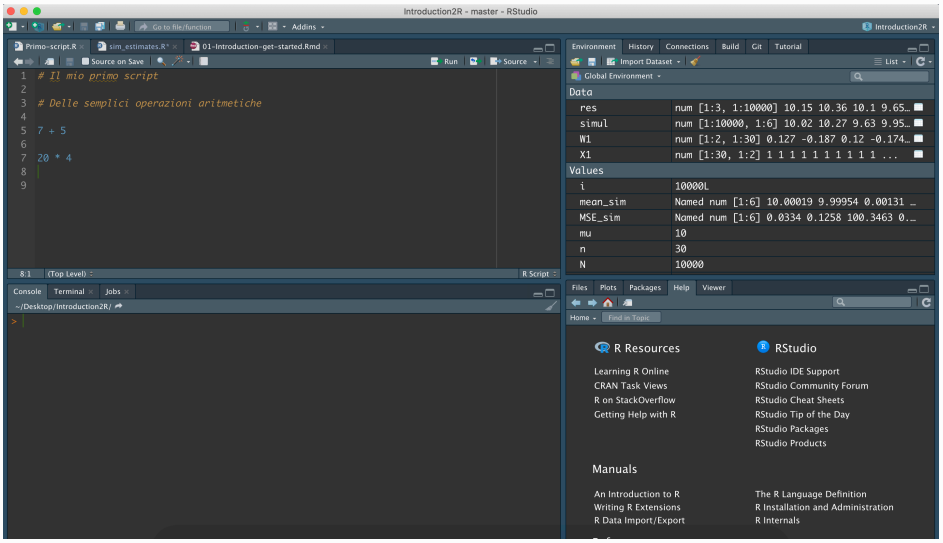


Figure 2.10: *Help* - documentazione di R

Tip-Box: Personalizza tema e layout

RStudio permette un ampio grado di personalizzazione dell'intrafaccia grafica utilizzata. E' possibile cambiare tema, font e disposizione dei pannelli a seconda dei tuoi gusti ed esigenze.
Prova a cambiare il tema dell editor in *Idle Fingers* per utlizzare on background scuro che affatichi meno la vista (vedi Figura seguente). Clicca su RStudio > Preferenze > Appearence (MacOS) o Tools > Options > Appearence (Windows).



Chapter 3

Primi Passi in R

Ora che abbiamo iniziato a famigliarizzare con il nostro stumento di lavoro possiamo finalmente dare fuoco alle polveri e concentraci sulla scrittura di codici!

In questo capitolo muoveremo i primi passi in R. Inizieremo vedendo come utilizzare operatori matematici, relazionali e logici per compiere semplici operazioni in R. Imparare R è un lungo percorso (scoop: questo percorso non termina mai dato che R è sempre in continuo sviiluppo). Soprattutto all'inizio può sembrare eccessivamente difficile poichè è si incontrano per la prima volta molti comandi e concetti di programmazione. Tuttavia, una volta famigliarizzato con gli apetti di base, la progressione diventa sempre più veloce (inarrestabile direi!).

In questo capitolo introdurremo per la prima volta molti elementi che saranno poi ripresi e approfonditi nei seguenti capitoli. Quindi non preoccuparti se non tutto ti sarà chiaro fin da subito. Imparare il tuo primo linguaggio di programmazione è difficile ma da qualche parte bisogna pure iniziare. Pronto per le tue prime linee di codice? Let's become a useR!

3.1 Operatori Matematici

R è un'ottima calcolatrice. Nella Tabella 3.1 sono elencati i principali operatori matematici e funzioni usate in R.

> 💡 Tip-Box: Le prime funzioni
>
> Nota come per svolgere operazioni come la radice quadrata o il valore assoluto vengono utlizzate delle specifiche funzioni. In R le funzioni sono richiamate digitando `<nome-funnzione>()` (e.g., `sqrt(25)`) indicando all'interno delle parentesi tonde gli argomenti della funzione. Approfondiremo le funzioni nel Capitolo 4.2.

3.1.1 Ordine Operazioni

Nello svolgere le operazioni, R segue lo stesso l'ordine usato nelle normali espressioni matematiche. Quindi l'ordine di precedenza degli operatori è:

Table 3.1: Operatori Matematici

Funzione	Nome	Esempio
`x + y`	Addizione	`> 5 + 3` `[1] 8`
`x - y`	Sottrazione	`> 7 - 2` `[1] 5`
`x * y`	Moltiplicazione	`> 4 * 3` `[1] 12`
`x / y`	Divisione	`> 8 / 3` `[1] 2.666667`
`x %% y`	Resto della divisione	`> 7 %% 5` `[1] 2`
`x %/% y`	Divisione intera	`> 7 %/% 5` `[1] 1`
`x ^ y`	Potenza	`> 3^3` `[1] 27`
`abs(x)`	Valore assoluto	`> abs(3-5^2)` `[1] 22`
`sign(x)`	Segno di un'espressione	`> sign(-8)` `[1] -1`
`sqrt(x)`	Radice quadrata	`> sqrt(25)` `[1] 5`
`log(x)`	Logaritmo naturale	`> log(10)` `[1] 2.302585`
`exp(x)`	Esponenziale	`> exp(1)` `[1] 2.718282`

`sin(x)`
`cos(x)`
`tan(x)`
`asin(x)`
`acos(x)`
`atan(x)`
	Funzioni trigonometriche	`>sin(pi/2)`
		`[1]1`
		`>cos(pi/2)`
		`[1]6.123234e-17`

`factorial(x)`	Fattoriale	`> factorial(6)` `[1] 720`
`choose(n, k)`	Coefficiente binomiale	`> choose(5,3)` `[1] 10`

1. `^` (potenza)
2. `%%` (resto della divisione) e `%/%` (divisione intera)
3. `*` (moltiplicazione) e `/`(divisione)
4. `+` (addizione) e `-`(sotttrazione)

Nota che in presenza di funzioni (e.g., `abs()`, `sin()`), R per prima cosa sostituisca le funzioni con il loro risultato per poi procedere con l'esecuzione delle operazioni nell'ordine indicato precedentemente.

L'ordine di esecuzione delle operazioni può essere controllato attraverso l'uso delle **parentesi tondone** `()`. R eseguirà tutte le operazioni incluse nelle parentesi seguendo lo stesso ordine inndicato sopra. Utilizzando più gruppi di parentesi possiamo ottenere i risultati desiderati.

Warning-Box: Le parentesi

Nota che in R solo le **parentesi tonde** `()` sono utilizzate per gestire l'ordine con cui sono eseguite le oprazioni.
Parentesi quadre `[]` e **parentesi graffe** `{}` sono invece speciali operatori utilizzati in R per altre ragioni come la selezione di elementi e la definizione di blocchi di codici. Argomenti che approfondiremo rispettivamente nel Capitolo 7.2 e Capitolo TODO.

Esercizi

Calcola il risultato delle seguenti operazioni utilizzando R (soluzioni):

1. $\frac{(45+21)^3+\frac{3}{4}}{\sqrt{32-\frac{12}{17}}}$
2. $\frac{\sqrt{7-\pi}}{3\ (45-34)}$
3. $\sqrt[3]{12-e^2}+\ln(10\pi)$
4. $\frac{\sin(\frac{3}{4}\pi)^2+\cos(\frac{3}{2}\pi)}{\log_7 e^{\frac{3}{2}}}$
5. $\frac{\sum_{n=1}^{10} n}{10}$

Note per la risoluzione degli esercizi:

- In R la radice quadrata si ottine con la funzione `sqrt()` mentre per radici di indici diversi si utilizza la notazione esponenziale ($\sqrt[3]{x}$ è dato da `x^(1/3)`).
- Il valore di π si ottiene con `pi`.
- Il valore di e si ottiene con `exp(1)`.
- In R per i logaritmi si usa la funzione `log(x, base=a)`, di base viene considerato il logaritmo naturale.

3.2 Operatori Relazionali e Logici

Queste operazioni al momento potrebbero sembrare non particolrmente interessanti ma si riveleranno molto utili nei capitoli successivi ad esempio per la selezione di elementi (vedi Capitolo 7.2.1) o la definizionne di algoritmi (vedi Capitolo TODO).

3.2.1 Operatori Relazionali

In R è possibile valutare se una data relazione è vera o falsa. Ad esempio, posiamo valutare se "*2 è minore di 10*" o se "*4 numero è un numero pari*".

R valuterà le proposizioni e ci restituirà il valore `TRUE` se la proposizione è vera oppure `FALSE` se la proposizione è falsa. Nella Tabella 3.2 sono elencati gli operatori relazionali.

Table 3.2: Operatori Relazionali

Funzione	Nome	Esempio
`x == y`	Uguale	`> 5 == 3` `[1] FALSE`
`x != y`	Diverso	`> 7 != 2` `[1] TRUE`
`x > y`	Maggiore	`> 4 > 3` `[1] TRUE`
`x >= y`	Maggiore o uguale	`> -2 >= 3` `[1] FALSE`
`x < y`	Minore	`> 7 < 5` `[1] FALSE`
`x <= y`	Minore o uguale	`> 7 <= 7` `[1] TRUE`
`x %in% y`	inclusione	`> 5 %in% c(3, 5, 8)` `[1] TRUE`

Warning-Box: '==' non è uguale a '='

Attenzione che per valutare l'uguaglianza tra due valori non bisogna utilizzare = ma ==. Questo è un'errore molto comune ceh si commmette in continuazione.
L'operatore = è utilizzato in R per assegnare un valore ad una variablie. Argomento che vederemo nel Capitolo 4.1.1.

Tip-Box: TRUE-T-1; FALSE-F-0

Nota che in qualsiasi linguaggio di Programmazione, ai valori TRUE e FALSE sono associati rispettivament i valori numerici 1 e 0. Questi sono definiti valori booleani.

```
TRUE == 1   # TRUE
TRUE == 2   # FALSE
TRUE == 0   # FALSE
FALSE == 0  # TRUE
FALSE == 1  # FALSE
```

In R è possibile anche abbreviare TRUE e FALSE rispettivamente in T e F, sebbene sia una pratica non consigliata poichè potrebbe nonn essere chiara e creare fraintendimenti. Infatti mentre TRUE e FALSE sono parole riservate (vedi Capitolo 4.1.2) T a F non lo sono.

```
T == 1      # TRUE
T == TRUE   # TRUE
F == 0      # TRUE
F == FALSE  # TRUE
```

3.2.2 Operatori Logici

In R è possibile congiungere più relazioni per valutare una desiderata proposizione. Ad esempio potremmo valutare se "*17 è maggiore di 10 e minore di 20*". Per unire più relazioni in un'unica proposizione che R valuterà come `TRUE` o `FALSE`, vengono utilizati gli operatori logici riportati in Tabella 3.3.

Table 3.3: Operatori Logici

Funzione	Nome	Esempio
`!x`	Negazione	`> !TRUE` `[1] FALSE`
`x & y`	Congiunzione	`> TRUE & FALSE` `[1] FALSE`
`x | y`	Disgiunzione Inclusiva	`> TRUE | FALSE` `[1] TRUE`

Questi operatori sono anche definiti operatori booleani e seguono le comuni definizioni degli operatori logici. In particolare abbiamo che:

- Nel caso della **congiunzione logica** `&`, affinchè la proposizione sia vera è necessario che entrambe le relazioni siano vere. Negli altri casi la proposizione sarà valutarta falsa (vedi Tabella 3.4).

Table 3.4: Congiunzione '&'

x	y	x \& y
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

- Nel caso della **disgiunzione inclusiva logica** `|`, affinchè la proposizione sia vera è necessario che almeno una relaziona sia vera. La proposizione sarà valutarta falsa solo quando entrambe le relazioni sono false (vedi Tabella 3.5).

Table 3.5: Disgiunzione inclusiva '|'

x	y	x | y
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

Approfondimento: Disgiunzione esclusiva

Per completezza ricordiamo che tra gli operatori logici esiste anche la **disgiunzione esclusiva**. La proposizione sarà valutata falsa se entrambe le relazioni sono vere oppure false. Affinchè la proposizione sia valutata vera una sola delle relazioni deve essere vera mentre l'altra deve essere falsa.
In R la disgiunzione esclusiva tra due ralazioni (x e y) è indicata con la funzione `xor(x, y)`. Tuttavia tale funzione è raramente usata.

Table 3.6: Disgiunzione esclusiva 'xor(x, y)'

x	y	xor(x, y)
TRUE	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

3.2.3 Ordine valutazione relazioni

Nel valutare le veridicità delle proposizioni R esegue le operazioni nel seguente ordine:

1. operatori matematici (e.g., `^`, `*`, `/`, `+`, `-`, etc.)
2. operatori relazionali (e.g., `<`, `>`, `<=`, `>=`, `==`, `!=`)
3. operatori logici (e.g., `!`, `&`, `|`)

La lista completa dell'ordine di esecuzione delle operazioni è riportata al seguente link https://stat.ethz.ch/R-manual/R-devel/library/base/html/Syntax.html. Ricordiamo che, in caso di dubbi riguardanti l'ordine di esecuzione delle operazioni, la cosa migliore è utilizzare le parentesi tonde `()` per disambiguare ogni possibile fraintendimento.

Warning-Box: L'operatore '%in%'

Nota che l'operatore `%in%` che abbiamo precedentemente indicato tra gli op-

eratori relazionali in realtà è un operatore speciale. In particolare, non segue le stesse regole degli altri operatori relazionlali per quanto riguarda l'ordine di esecuzione.
La soluzione migliore? Usa le parentesi!

Esercizi

Esegui i seguenti esercizi utilizzando gli operatori relazionali e logici (soluzioni):

1. Definisici due relazioni false e due vere che ti permettano di valutare i risultati di tutti i possibili incroci che puoi ottenere con gli operatori logici `&` e `|`.
2. Definisci una proposizione che ti permetta di valutare se un numero è pari. Definisci un'altra proposizione per i nueri dispari (tip: cosa ti ricorda `%%`?).
3. Definisci una proposizione per valutare la seguente condizione (ricordati di testare tutti i possibili scenari) "*x è un numero compreso tra -4 e -2 oppure è un numero compreso tra 2 e 4*".
4. Esegui le seguenti operazioni `4 ^ 3 %in% c(2,3,4)` e `4 * 3 %in% c(2,3,4)`. Cosa osservi nell'ordine di esecuzione degli operatori?

Chapter 4

Due Compagni Inseparabili

In questo capitolo introdurremmo i concetti di oggetti e funzioni, due elementi alla base di R (e di ogni linguaggio di programmazione). Potremmo pensare agli oggetti in R come a delle variabili che ci permettono di mantenere in memoria dei valori (e.g., i risultati dei nostri calcoli o i nostri dati). Le funzioni in R, invece, sono analoghe a delle funzioni matematiche che, ricevuti degli oggetti in input, compiono delle azioni e restituiscono dei nuovi oggetti in output.

Questa è una iper-semplificazione (e pure tecnicamente non corretta) che ci permettere però di capire come, partendo dai nostri dati o valori iniziali, possiamo manipolarli applicando delle funzioni per ottenere, attraverso differenti step, i risultati desiderati (e.g., analisi statistiche o grafici e tabelle).

Qui valuteremo gli aspetti fondamentali riguardanti l'utilizzo degli oggetti e delle funzioni che saranno successivamente approfonditi rispettivamente nel corso della Seconda Sezione e della Terza Sezione del libro.

4.1 Oggetti

Quando eseguiamo un commando in R, il risultato ottenuto viene immediatamente mostrato in *Console*. Tale risultato, tuttavia, non viene salvato in memoria e quindi non potrà essere riutilizzato in nessuna operazione futura. Condurre delle analisi in questo modo sarebbe estremamente complicato ed inefficiente. La soluzione più ovvia è quella di salvare in memoria i nostri risultati intermedi per poterli poi riutilizzare nel corso delle nostre analisi. Si definisce questo processo come *assegnare* un valore ad un oggetto.

4.1.1 Assegnare e Richiamare un oggetto

Per assegnare il valore numerico 5 all'oggetto `x` è necessario eseguire il seguente comando:

```
x <- 5
```

La funzione `<-` ci permette di assegnare i valori che si trovano alla sua destra all'oggetto il cui nome è definito alla sinistra. Abbiamo pertanto il seguente pattern: `<nome-oggetto> <- <valore-assegnato>`.Notate come in *Console* appaia solo il comando appena eseguito ma non venga mostrato alcun risultato.

Per utilizzare il valore contenuto nell'oggetto sarà ora sufficiente richiamare nel proprio codice il nome dell'oggetto desiderato.

```
x + 3
## [1] 8
```

E' inoltre possibile "aggiornare" o "sostituire" il valore contenuto in un oggetto. Ad esempio:

```
# Aggiornare un valore
x <- x*10
x
## [1] 50

# Sostituire un valore
x <- "Hello World!"
x
## [1] "Hello World!"
```

Nel primo caso, abbiamo utilizzato il vecchio valore contenuto in `x` per calcolare il nuovo risultato che è stato assegnato a `x`. Nel secondo caso, abbiamo sostituito il vecchio valore di `x` con un nuovo valore (nell'esempio una stringa di caratteri).

Approfondimento: Assegnare valori '<-' vs '='

Esistono due operatori principali che sono usati per assegnare un valore ad un oggetto: l'operatore `<-` e l'operatore `=`. Entrambi sono validi e spesso la scelta tra i due diventa solo una questione di stile personale.

```
x_1 <- 45
x_2 = 45

x_1 == x_2
## [1] TRUE
```

Esistono, tuttavia, alcune buone ragioni per preferire l'uso di `<-` rispetto a `=` (attenti a non confonderlo con l'operatore relazionale `==`). L'operazione di assegnazione è un'operazione che implica una direzionalità, il chè è reso esplicito dal simbolo `<-` mentre il simbolo `=` non evidenzia questo aspetto e anzi richiama la relazione di uguaglianza in matematica.
La decisione su quale operatore adottare è comunque libera, ma ricorda che una buona norma nella programmazione riguarda la *consistenza*: una volta presa una decisione è bene mantenerla per facilitare la comprensione del codice.

4.1.2 Nomi degli oggetti

La scelta dei nomi degli oggetti sembra un aspetto secondario ma invece ha una grande importanza per facilitare la chiarezza e la comprensione dei codici.

Ci sono alcune regole che discriminano nomi validi da nomi non validi. Il nome di un oggetto:

- deve iniziare con una lettera e può contenere lettere, numeri, underscore (`_`), o punti (`.`).
- potrebbe anche iniziare con un punto (`.`) ma in tal caso non può essere seguito da un numero.
- non deve contenere caratteri speciali come `#`, `&`, `$`, `?`, etc.
- non deve essere una parola riservata ovvero quelle parole che sono utilizzate da R con un significato speciale (e.g, `TRUE`, `FALSE`, etc.; esegui il comando `?reserved` per la lista di tutte le parole riservate in R).

Warning-Box: CaSe-SeNsItIvE

Nota come R sia **Case-Sensitive**, ovvero distingua tra lettere minuscole e maiuscole. Nel seguente esempio i due nomi sono considerate diversi e pertanto non avviene una sovrascrizione ma due differenti oggetti sono creati:

```
My_name <- "Monty"
my_name <- "Python"

My_name
## [1] "Monty"
my_name
## [1] "Python"
```

Inoltre, il nome ideale di un oggetto dovrebbe essere:

- **auto-descrittivo**: dal solo nome dovrebbe essere possibile intuire il contenuto dell'oggetto. Un nome generico quale `x` o `y` ci sarebbero di poco aiuto poichè potrebbero contenere qualsiasi informazione. Invece un nome come `weight` o `gender` ci suggerirebbe chiaramente il contenuto dell'oggetto (e.g., il peso o il gender dei partecipanti del nostro studio).
- **della giusta lunghezza**: non deve essere ne troppo breve (evitare sigle incomprensibili) ma neppure troppo lunghi. La lunghezza corretta è quella che permette al nome di esssere sufficientemente informativo senza aggiungere inutili dettagli. In genere sono sufficienti 2 o 3 parole.

Approfondimento: CamelCase vs snake_case

Spesso più parole sono usate per ottenere un nome sufficientemente chiaro. Dato che però non è possibile includere spazi in un nome, nasce il problema di

come unire più parole senza che il nome diventi incomprensibile, ad esempio `mediatestcontrollo`.
Esistono diverse convenzioni tra cui:

- **CamelCase**. L'inizio di una nuova parole viene indicata con l'uso della prima lettera maiuscola. Ad esempio `mediaTestControllo`.
- **snake_case**. L'inizio di una nuova parola viene indicata con l'uso carattere `_`. Ad esempio `media_test_controllo`.
- una variante al calssico **snake_case** riguarda l'uso del `.`, ad esempio `media.test.controllo`. Questo approccio in genere è evitato poichè in molti linguaggi di programmazione (ed anche in R in alcune condizioni) il carattere `.` è un carattere speciale.

In genere viene raccomandato di seguire la convenzione **snake_case**. Tuttavia, la decisione su quale convenzione adottare è libera, ma ricorda ancora che una buona norma nella programmazione riguarda la *consistenza*: una volta presa una decisione è bene mantenerla per facilitare la comprensione del codice.

4.1.3 Tipologie Dati e Strutture Dati

Per lavorare in modo ottimale in R, è fondamentale conoscere bene e distinuere chiaramente quali sono le tipologie di dati e le strutture degli oggetti usati.

In R abbiamo 4 principali tipologie di dati, ovvero tipologie di valori che possono essere utilizzati:

- `character` - *Stringhe di caratteri* i cui valori alfannumerici vengono delimitati dalle doppie vigolette `"Hello world!"` o virgolette singole `'Hello world!'`.
- `double` - *Valori reali* con o senza cifre decimali ad esempio `27` o `93.46`.
- `integer` - *Valori interi* definiti apponendo la lettera `L` al numero desiderato, ad esempio `58L`.
- `logical` - *Valori logici* `TRUE` e `FALSE` usati nelle operazioni logiche.

```
typeof("Psicostat")
## [1] "character"
typeof(24.04)
## [1] "double"
typeof(1993L)
## [1] "integer"
typeof(TRUE)
## [1] "logical"
```

In R abbiamo inoltre differenti tipologie di oggetti, ovvero diverse strutture in cui possono essere organizzati i dati:

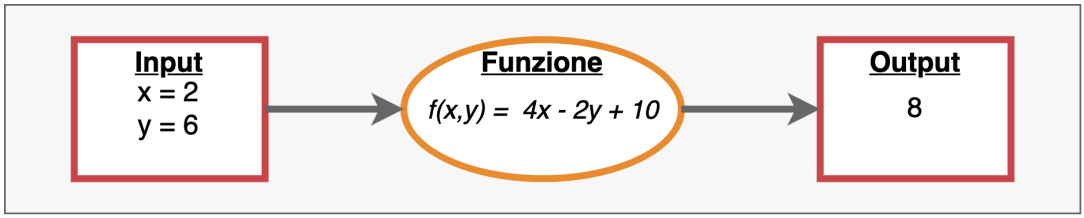
- **Vettori**
- **Matrici**

- **Dataframe**
- **Liste**

Approfondiremo la loro definizione, le loro caratteristiche ed il loro utilizzo nel corso di tutta la Seconda Sezione di questo libro.

4.2 Funzioni

Possiamo pensare alle funzioni in R in modo analogo alle classiche funzioni matematiche. Dati dei valori in input, le funzioni eseguono dei specifici calcoli e restituiscono in output il risultato ottenuto.



Abbiamo già incontrato le nostre prime funzioni per eseguire specifiche operazioni matematiche nel Capitolo 3.1 come ad esempio `sqrt()` o `abs()` usate per ottenere ripettivamente la radice quadrata o il valore assoluto di un numero. Ovviamente le funzioni in R non sono limitate ai soli calcoli matematici ma possono eseguire qualsiasi genere di compito come ad esempio creare grafici e tabelle o manipolare dei dati o dei file. Tuttavia il concetto rimane lo stesso: ricevuti degli oggetti in input, le funzioni compiono determinate azioni e restituiscono dei nuovi oggetti in output.

In realtà incontreremo delle funzioni che non richiedono input o non produrre degli output. Ad esempio `getwd()` non richiede input oppure la funzione `rm()` non produce output. Tuttavia questo accade nella minoranza dei casi.

Per eseguire una funzione in R è necessario digitare il nome della funzione ed indicare tra parentesi i valori che vogliamo assegnare agli **argomenti** della funzione, ovvero i nostri input, separati da virgole. Generalmente si utilizza quindi la seguente sintassi:

```
<nome-funzione>(<nome-arg1> = <valore-arg1>, <nome-arg2> = <valore-arg2>,...)
```

Ad esempio per creare una sequenza di valori con incrementi di 1 posso usare la funzione `seq()`, i cui argomenti sono `from` e `to` ed indicano rispettivamente il valore iniziale ed il valore massimo della sequenza.

```
# creo una sequenza di valori da 0 a 10 con incrementi di 1
seq(from = 0, to = 10)
##  [1]  0  1  2  3  4  5  6  7  8  9 10
```

4.2.1 Argomenti di una Funzione

Nel definire gli argomenti di una funzione non è necessario specificare il nome degli argomenti. Ad esempio il comando precedente può essere eseguito anche specificando solamente i valori.

```
# creo una sequenza di valori da 0 a 10 con incrementi di 1
seq(0, 10)
##  [1]  0  1  2  3  4  5  6  7  8  9 10
```

Tuttavia, questo rende più difficile la lettura e la comprensione del codice poichè non è chiaro a quali argomenti si riferiscono i valori. L'ordine con cui vengono definiti i valori in questo caso è iportante, poichè R assume rispetti l'ordine prestabilito degli argomenti. Osserva come invertendo invertendo i valori ovviamente otteniamo risultati differenti da quelli precedenti, ma questo non avviene quando il nome dell'argomento è specificato.

```
# inverto i valori senza i nomi degli argomenti
seq(10, 0)
##  [1] 10  9  8  7  6  5  4  3  2  1  0

# inverto i valori con i nomi degli argomenti
seq(to = 10, from = 0)
##  [1]  0  1  2  3  4  5  6  7  8  9 10
```

Vediamo inoltre come le funzioni possano avere molteplici argomenti, ma che non sia necessario specificare il valore per ognuno di essi. Molti argomenti, infatti, hanno già dei valori prestabiliti di *default* e non richiedo quindi di essere specificati almeno che ovviamente non si vogliano utilizzare impostazioni diverse da quelle di *default*. Oppure lo specificare un dato argomento rispetto ad un altro può definire il comportamento stesso della funzione.

Ad esempio la funzione `seq()` possiede anche gli argomenti `by` e `length.out` che prima non erano stati specificati. `by` permette di definire l'incremento per ogni elemento successivo della sequenza mentre `length.out` permette di definire il numero di elementi della sequenza. Vediamo come allo specificare dell'uno o dell'altro agromento (o di entrambi) il comportamento della funzione vari.

```
seq(from = 0,  to = 10, by = 5)
## [1]  0  5 10
seq(from = 0,  to = 10, length.out = 5)
## [1]  0.0  2.5  5.0  7.5 10.0
seq(from = 0,  to = 10, length.out = 5, by = 4)
## Error in seq.default(from = 0, to = 10, length.out = 5, by = 4): too many arguments
```

E' pertanto cosigliabile esplicitare sempre gli argomenti di una funzione per rendere chiaro a che cosa si riferiscono i valori indicati. Questo è utlile anche per evitare eventuali comportamenti non voluti delle funzioni ad individuare più facilmente possibili errori.

Gli argomenti di una funzione, inoltre, richiedono specifiche tipologie e strutture di dati e sta a noi assicuraci che i dati siano forniti nel modo corretto. Vediamo ad esempio come la funzione `mean()` che calcola la media di un insieme di valori, richieda come input un vettore di valori numerici. Approfondiremo il concetto di vettori nel Capitolo 7, al momento ci basta sapere che possiamo usare la funzione `c()` per combinare più valori in un unico vettore.

```
# Calcolo la media dei seguenti valori (numerici)
mean(c(10, 6, 8, 12)) # c() combina più valori in un unico vettore
## [1] 9

mean(10, 6, 8, 12)
## [1] 10
```

Notiamo come nel primo caso il risultato sia corretto mentre nel secondo è sbagliato. Questo perchè `mean()` richiede come primo argomento il vettore su cui calcolare la media. Nel primo caso abbiamo correttamente specificato il vettore di valori usando la funzione `c()`. Nel secondo caso invece, il primo argomento risulta essere solo il valore `10` ed R calcola la media di `10` ovvero `10`. Gli altri valori sono passati ad altri argomenti che non alterano il comportameto ma neppure ci segnalano di questo importante errore.

Nel seguente esempio, possiamo vedere come `mean()` richieda che i valori siano numerici. Seppur `"1"` `"2"`, e `"3"` siano dei numeri, l'utilizzo delle doppie virgolette li rende delle stringhe di caratteri e non dei valori numerici e giustamente R non può eseguire una media su dei caratteri.

```
# Calcolo la media dei seguenti valori (caratteri)
mean(c("1", "2", "3"))
## Warning in mean.default(c("1", "2", "3")): argument is not numeric or logical:
## returning NA
## [1] NA
```

Capiamo quindi che per usare correttamente le funzioni è fondamentale conoscerne gli argomenti e rispettare le tipologie e strutture di dati richieste.

4.2.2 Help! I need Somebody...Help!

Conoscere tutte le funzioni e tutti i loro argomenti è impossibile. Per fortuna R ci viene in soccorso fornendoci per ogni funzione la sua documentazione. Qui vengono fornite tutte le informazioni riguardanti la finalità della funzione, la descrizione dei suoi argomenti, i dettagli riguardanti i suoi possibili utilizzi.

Per accedere alla documentazione possiamo utilizzare il comando `?<nome-funzione>` oppure `help(<nome-funzione>)`. Ad esempio:

```
?seq
help(seq)
```

Una pagina si aprirà nel pannello "Help" in basso a destra con la documentazione della funzione in modo simile a quanto rappresentato in Figura 4.1.

Il formato e le informazioni presenti nella pagina seguono delle norme comuni ma non obbligatorie. Infatti, non necessariamente vengono usati sempre tutti i campi e comunque all'autore delle funzioni è lasciato un certo grado di libertà nel personalizzare la documentazione. Tra i campi principali e più comunemente usati abbiamo:

- **Titile** - Titolo esplicativo della finalità della funzione

- **Description** - Descrizione coincisa della funzione
- **Usage** - Viene mostrata la struttura della funzione con i suoi argomenti e valori di default
- **Arguments** - Elenco con la descrizione dettagliata di tutti gli argomenti. Qui troviamo per ogni argomento sia le opzioni utilizzabili ed il loro effetto, che la tipologia di valori richiesti
- **Details** - Descrizione dettagliata della funzione considerando i casi di utilizzo ed eventuali note tecniche
- **Value** - Descrizione dell'output dalla funzione. Qui troviamo sia la descrizione della struttura dei dati dell'output che la descrizione dei suei elementi utile per interpretare ed utilizzare i rsultati ottenuti

seq {base} R Documentation

Sequence Generation

Description

Generate regular sequences. `seq` is a standard generic with a default method. `seq.int` is a primitive which can be much faster but has a few restrictions. `seq_along` and `seq_len` are very fast primitives for two common cases.

Usage

```
seq(...)

## Default S3 method:
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
    length.out = NULL, along.with = NULL, ...)

seq.int(from, to, by, length.out, along.with, ...)

seq_along(along.with)
seq_len(length.out)
```

Arguments

`...`	arguments passed to or from methods.
`from, to`	the starting and (maximal) end values of the sequence. Of length `1` unless just `from` is supplied as an unnamed argument.
`by`	number: increment of the sequence.
`length.out`	desired length of the sequence. A non-negative number, which for `seq` and `seq.int` will be rounded up if fractional.
`along.with`	take the length from the length of this argument.

Details

Numerical inputs should all be finite (that is, not infinite, `NaN` or `NA`).

Figure 4.1: Help-page della funzione seq()

- **See Also** - Eventuali link ad altre funzioni simili o in relazione con la nostra funzione
- **Examples** - Esempi di uso della funzione

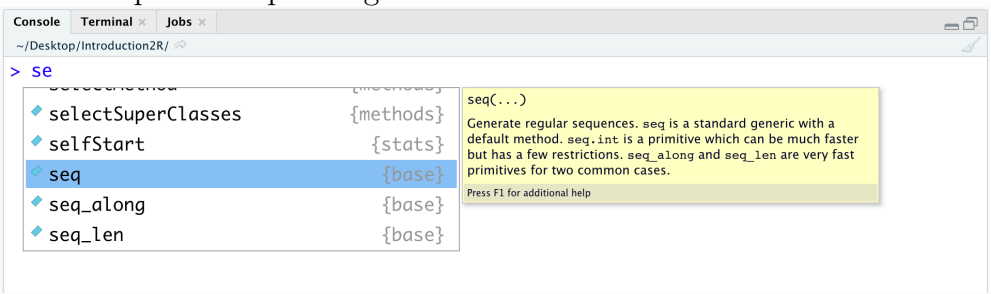
Ricerca per Parola

Quando non si conosce esattamente il nome di una funzione o si vuole cercare tutte le funzioni e pagine che includono una certa parola, è possibile utilizzare il comando `??<parola>` oppure `help.search(<parola>)`.

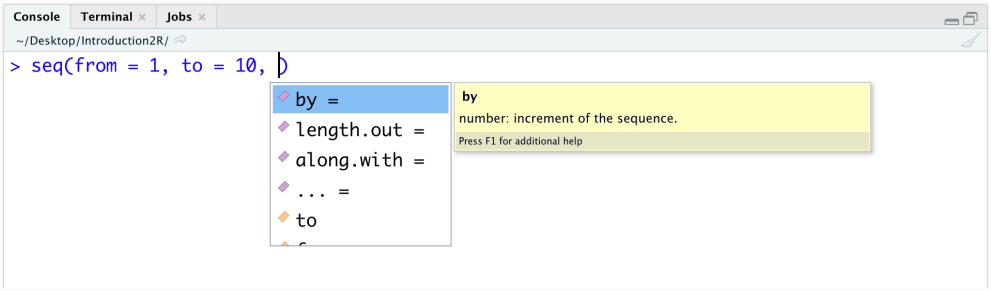
R eseguirà una ricerca tra tutta la documentazione disponibile è fornirà un elenco delle pagine che contengono la parola desiderata nel titolo o tra le keywords.

Trick-Box: Autocompletamento with 'Tab'

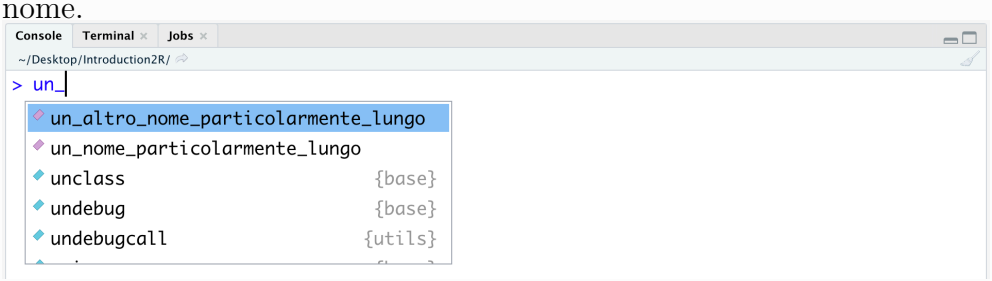
La natura dei programmatori è essere pigri e smemorati. Per fortuna ogni *code editor* che si rispetti (i.e., programma per la scrittura di codici) possiede delle utli funzioni di autocompletamento e suggerimento dei comandi che semplificano la scrittura di codici.
In Rstudio, i suggerimenti compaino automaticamente durante la scrittura di un comando oppure possono essere richiamati premendo il tasto `Tab` in alto a sinistra della tastiera (). Comparirà una finestra con possibili soluzioni di autocompletamento del nome della funzione. Utilizzando le frecce della tastiera possiamo evidenziare lq funzione voluta e premere `Invio` per autocompletare il comando. Nota come accanto al nome della funzione appare anche un piccolo riquadro giallo con la descrizione della funzione.



Per inserire gli argomenti della funzione possiamo fare affidamento nuovamente ai suggerimenti e alla funzione di autocompletamento. Sarà sufficiente premere nuovamente il tasto `Tab` e questa volta comparirà una lista degli argomenti con la relativa descrizione. Sarà quindi sufficiente selezionare con le frecce l'argomento desiderato e premere `Invio`.



Notate come la funzione di autocompletamento non sia utilizzata solo per le funzioni ma anche per i nomi degli oggetti. Questo ci consentirà di richiamare velocemente oggetti precedentemente creati evitando di digitare l'intero nome.



Chapter 5

Ambiente di Lavoro

In queso capitolo introdurremo alcuni concetti molto importanti che riguardano l'ambiente di lavoro in R o RStudio. In particolare parleremo dell'*environment*, della *working directory* e dell'utilizzo dei pacchetti.

5.1 Environment

Nel Capitolo 4.1, abbiamo visto come sia possibile assegnare dei valori a degli oggetti. Questi oggetti vengono creati nel nostro ambiente di lavoro (o meglio *Environment*) e potranno essere utilizzati in seguito.

Il nostro Enviroment raccoglie quindi tutti gli oggetti che vengono creati durante la nostra sessione di lavoro. E' possibile valutare gli oggetti attualmente presenti osservando il pannello *Environment* in alto a destra (vedi Figura 5.1) oppure utilizzadno il comando `ls()`, ovvero *list objects*.

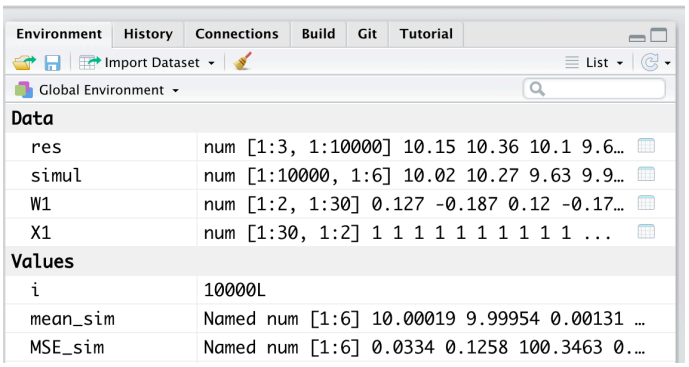


Figure 5.1: *Environment* - Elenco degli oggetti e variabili presenti nel'ambiente di lavoro

All'inizio della sessione di lavoro il nostro Environment sarà vuoto (vedi Figura 5.2). Il comando `ls()` non restituirà alcun oggeto ma per indicare l'assenza di oggerri userà la risposta `character(0)`, ovvero un vettore di tipo caratteri di lunghezza zero (vedi Capitolo 7).

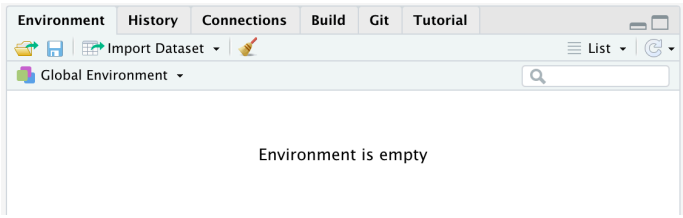


Figure 5.2: *Environment* vuoto ad inizio sessione di lavoro

```
# Environment vuoto
ls()
## character(0)
```

5.1.1 Aggiungere Oggetti all'Environment

Una volta creati degli oggetti, questi saranno presenti nel nostro Environment e il comando `ls()` restituirà un vettore di caretteri in cui vengono elencati tutti i loro nomi.

```
# Creo oggetti
x <-  c(2,4,6,8)
y <-  27
word <- "Hello Word!"

# Lista nomi oggetti nell'Environment
ls()
## [1] "word" "x"    "y"
```

Nel pannello in alto a destra (vedi Figura 5.3), possiamo trovare un elenco degli oggetti attualmente presenti nel nostro Environment. Insieme al nome vengono riportate anche alcune utili inforamzioni a seconda del tipo di oggetto. Vediamo come nel nostro esempio, nel caso di variabili con un singolo valore (e.g., `word` e `y`) venganno presentati direttamente gli stessi valori. Mentre, nel caso di vettori (e.g., `x`) vengano fornite anche informazioni riguardanti la tipologia di vettore e la sua dimensione (vedi Capitolo 7), nell'esempio abbiimao un vettore numerico (`num`) di 4 elementi (`[1:4]`).

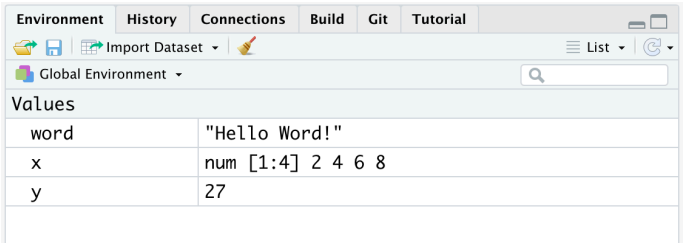


Figure 5.3: *Environment* contenente gli oggetti creati

5.1.2 Rimuovere Oggetti dall'Environment

Per rimuovere un oggetto dal proprio environment è possibile utilizzare il comando `remove()` oppure la sua abbreviazione `rm()`, indicando tra parentesi il nome dell'oggetto che si intende

rimuovere. E' possibile indicare più di un oggetto separando i loro nomi con la virgola.

```
# Rimuovo un oggetto
rm(word)
ls()
## [1] "x" "y"

# Rimuovo più oggetti contemporaneamente
rm(x,y)
ls()
## character(0)
```

Trick-Box: rm(list=ls())

Qualora fosse necessario eliminare tutti gli oggetti attualmete presenti nel nosto ambiente di lavoro è possibile ricorrere alla formula `rm(list=ls())`. In questo modo si avrà la certezza di pulire l'ambiente da ogni oggetto e di ripristinarlo alle condizioni iniziali della sessione.

Approfondimento: Mantenere Ordinato l'Environment

Avere cura di mantenre il proprio Environment ordinato ed essere consapevoli degli oggetti attualmente presenti è importante. Questo ci permette di evitare di compiere due errori comuni.

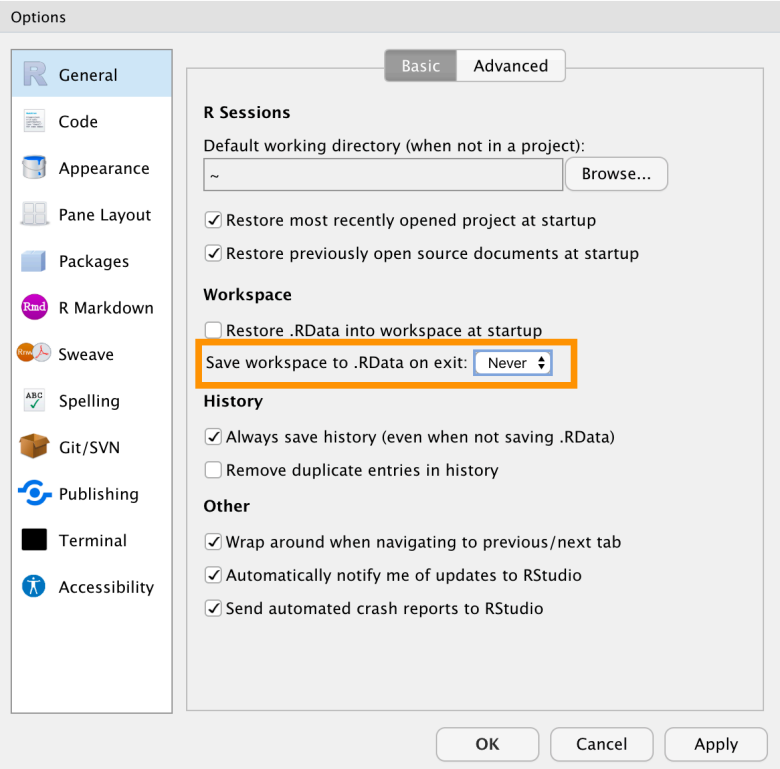
- **Utilizzare oggetti non ancora creati**. In questo caso l'errore è facilemente individuabile dat che sarà lo stesso R ad avvisarci che *"object '' not found"*. In questo caso dovremmo semplicemente eseguire il comando per creare l'oggetto richieto.

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

- **Utilizzare oggetti con "*vecchi*" valori**. Se non si ha cura di mantenere ordinato il proprio ambiente di lavoro potrebbe accadere che diversi oggetti vengano creati durante successive sessioni di lavoro. A questo punto si corre il rischio di perdere il controllo rispetto al vero contenuto degli oggetti e potremmo quindi utilizzare degli oggetti pensando che contengano un certo valore, quando invece si riferisono a tutt'altro. Questo comporta che qualsiasi nostro risultato perda di significato. Bisogna prestare molta attenzione perchè R non potrà avvisarci di questo errore (per lui sono solo numeri), siamo noi che dobbiamo es-

sere consapevoli del fatto che i comandi eseguiti abbiano senso oppure no.

Per mantere un Environmet ordinato vi consigliamo innanzitutto di non salvare automaticamente il vostro *workspace* quando terminate una sessione di lavoro. E' possibile settare tale opzione nelle impostazioni generali di R selezionando *"Never"* alla voce *"save workspace to .RData on exit"* come riportato nella Figura seguente.



Questo vi permetterà di iniziare ogni nuova sessione di lavoro in un Environment vuoto, evitando che vecchi oggetti si accumulino nel corso delle diverse sesssioni di lavoro. Durante le vostre sessioni, inoltre, sarà utile eseguire il comando `rm(list=ls())` quando inizierete un nuovo compito in modo da eliminare tutti i vecchi oggetti.

Environment una Memoria a Breve Termine

Notiamo quindi come l'Environment sia qualcosa di transitorio. Gli oggetti vengono salvati nella memoria primaria del computer (RAM, possiamo pensarla in modo analogo alla memoria a breve termine dei modelli cognitivi) e verranno cancellati al comando `rm(list=ls())` o al termine di ogni sessione di lavoro.
Il fatto di partire ogni volta da un Environment vuoto, vi costringerà a rac-

cogliere tutti i passi delle vostre analisi all'interno di uno script in modo ordinato evitando di fare affidamento su vecchi oggetti. Tutti gli oggetti necessari durante le analisi, infattii, dovranno essere ricreati ad ogni sessione, garantendo la riproducibilità e correttezza del lavro (almeno dal punto di vista di programmazione). Idealmente dovrebbe essere posibile, in una sessione di lavoro, partire da un Environment vuoto ed eseguire in ordine tutti i comandi contenuti in uno script fino ad ottenere i risultati desiderati.
E' facile intuire come in certe situazioni questa non sia la soluzione più efficiente. Alcuni comandi, infatti, potrebbero richiedere molti minuti (o anche giorni) per essere eseguiti. In questi casi sarebbe conveniente, pertanto, salvare i risultati ottenuti per poterli utilizzare anche in sessioni successive, senza la necessità di dover eseguire nuovamente tutti i comadi. Vedremo nel Capitolo TODO come sarà possibile salvare permanentemente gli oggetti creati nella memoria secondaria del computer (hard-disk, nella nostra analogia la memoria a lungo termine) e come caricarli in una successiva sessione di lavoro.

5.2 Working Directory

Il concetto di *working directory* è molto importante ma spesso poco conosciuto. La *working directory* è la posizione all'interno del computer in cui ci troviamo durante la nostra sessione di lavoro e da cui eseguiamo i nostri comandi.

5.2.1 Organizzazione Computer

L'idea intuitiva che abbiamo comunemente del funzionamento del computer è fuorviante. Spesso si pensa che il Desktop rispecchi l'organizzazione del nostro intero computer e che tutte le azioni siano gestite attraverso l'interfaccia punta-e-clicca a cui ormai siamo abituati dai moderni sistemi operativi.

Senza entrare nel dettaglio, è più corretto pensare all'organizzazione del cumputer come ad un insieme di cartelle e sottocartelle che contengono tutti i nostri file e al funzionamento del computer come ad un insieme di processi (o comandi) che vengono eseguiti. Gli stessi programmi che installiamo non sono altro che delle cartelle in cui sono contenuti tutti gli script che determinano il loro funzionamento. Anche il Desktop non è altro che una semplice cartella mentre quello che vediamo noi è un programma definito dal sistema operativo che visualizza il contenuto di quella cartella sul nostro schermo e ci permette di interfacciarci con il mouse.

Tutto quello che è presente nel nostro computer, compresi i nostri file, i programmi e lo stesso sistema operativo in uso, tutto è organizzato in un articolato sistema di cartelle e sottocartelle. Approsimativamente possiamo pensare all'organizzazione del nostro computer in modo simile alla Figura 5.4 (da: https://en.wikipedia.org/wiki/Operating_system).

Ai livelli più bassi troviamo tutti i file di sistema ai quali gli utenti possono accedere solo con speciali autorizzazioni. Al livello superiore troviamo tutte i file riguardanti i programmi e applicazioni installati che in genere sono utilizzabili da più utenti sullo stesso computer. Infine troviamo tutte le cartelle e file che riguardano lo specifico utente.

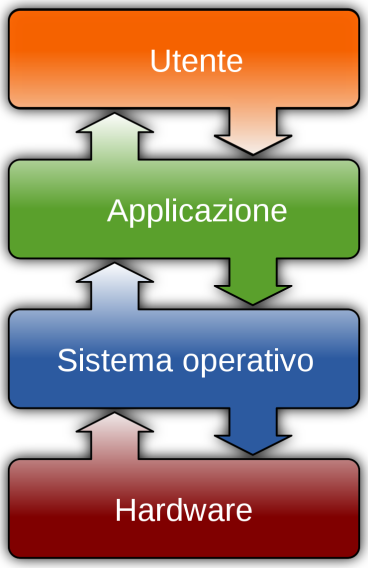


Figure 5.4: Organizzazione Computer (da Wikipedia vedi link nel testo)

5.2.2 Absolute Path e Relative Path

Questo ampio preambolo riguardante l'organizzazione in cartelle e sottocartelle, ci serve perchè è la struttura che il computer utilizza per orientarsi tra tutti file quando esegue dei comandi attraverso un'interfaccia a riga di comando (e.g., R). Se vogliamo ad esempio caricare dei dati da uno specifico file in R devo fornire il *path* (o indirizzo) corretto che mi indichi esattamente la posizione del file all'interno della struttura di cartelle del computer. Ad esempio, immaginiamo di avere dei dati `My-data.Rda` salvato nella cartella `Introduction2R` nel proprio Desktop.

```
Desktop/
 |
 |-  Introduction2R/
 |     |
 |     |- Dati/
 |     |    |- My-data.Rda
```

Per indicare la posizione del File potrei utilizzare un:

- **absolute path** - la posizione *"assoluta"* del file rispetto alla *root directory* del sistema ovvero la cartella principale dell'intero computer.

```
# Mac
"/Users/<username>/Desktop/Introduction2R/Dati/My-data.Rda"

# Windows Vista
"c:\Users\<username>\Desktop\Introduction2R\Dati\My-data.Rda"
```

- **relative path** - la posizione del file rispetto alla nostra attuale posizione nel computer da cui stiamo eseguendo il comando, ovvero rispetto alla **working directory** della nostra sessione

di lavoro. In questo riprendendo il precedente esempio se la nostra working directory fosse la cartella `Desktop/Introduction2R` avremmo i seguenti relative path:

```
# Mac
"Dati/My-data.Rda"

# Windows Vista
"Dati\My-data.Rda"
```

Nota come sia preferibile l'utilizzo dei relative path poichè gli absolute path sono unici per il singolo computer di riferimento e non possono essere quindi utilizzati su altri computer.

Warning-Box: "Error: No such file or directory"

Qualora si utilizzasse un relative path per indicare la posizione di un file, è importante che la working directory attualment in uso sia effettivamente quella prevista. Se ci trovassimo in una divesa cartella, ovviamente il "relative path" indicato non sarebbe più valido e R ci mostrerebbe un messaggio di errore.
Riprendendo l'esempio precedente, supponiamo che la nostra attuale working directory sia `Desktop` invece di `Desktop/Introduction2R`. Eseguendo il comadno `load()` per caricare i dati utilizzando il relative path ora non più valido ottengo:

```
load("Dati/My-data.Rda")
## Warning in readChar(con, 5L, useBytes = TRUE): cannot open compressed file
## 'Dati/My-data.Rda', probable reason 'No such file or directory'
## Error in readChar(con, 5L, useBytes = TRUE): cannot open the connection
```

Il messaggio di errore mi indica che R non è stato in grado di trovare il file seguendo le mie indicazioni. E' come se chiedessi al computer di aprire il frigo ma attualmente si trovasse in camera, devo prima dargli le indicazioni per raggiungere la cucina altrimenti mi risponderebbe *"frigo non trovato"*. Risulta pertanto fondamentale essere sempre consapevoli di quale sia l'attuale working directory in cui si sta svolgendo la sessione di lavoro.
Ovviamente otterrei lo stesso errore anche usando un absolute path se questo contenesse degli errori.

Approfondimento: The Garden of Forking Paths

Come avrai notato dagli esempi precedenti, sia la struttura in cui vengono organizzati i file nel computer sia la sintassi utilizzata per indicare i path è

differente in base al sistema operativo utilizzato.

Mac OS e Linux

- Il carattere utilizzato per separare le cartelle nella definizione del path è `"/"`:

```
"Introduction2R/Dati/My-data.Rda"
```

- La root-directory viene indicata iniziando il path con il carattere `"/"`:

```
"/Users/<username>/Desktop/Introduction2R/Dati/My-data.Rda"
```

- La cartella *home* dell'utente (ovvero `/Users/<username>/`) viene indicata iniziando il path con il carattere `"~"`:

```
"~/Desktop/Introduction2R/Dati/My-data.Rda"
```

Windows

- Il carattere utilizzato per separare le cartelle nella definizione del path è `"\"`:

```
"Introduction2R\Dati\My-data.Rda"
```

- La root-directory viene indicata con `"c:\"`:

```
"c:\Users\<username>\Desktop\Introduction2R\Dati\My-data.Rda"
```

5.2.3 Working Directory in R

Vediamo ora i comandi utilizzati in R per valutare e cambiare la working directory nella propria sessione di lavoro.

> **Tip-Box: One "/" to Rule Them All**
>
> Nota negli esempi successivi, come in R il caratere `"/"` sia sempre utilizzato per separare le cartelle nella definizione del path indipendentemente dal sistema operativo.

Attuale Working Directory

In R è possibile valutare l'attuale working directory utilizzando il comando `getwd()` che restituirà l'absolute path dell'attuale posizione.

```
getwd()
## [1] "/Users/<username>/Desktop/Introduction2R"
```

In alternativa, l'attuale working directory è anche riportata in alto a sinistra della Console come mostrato in Figura 5.5.

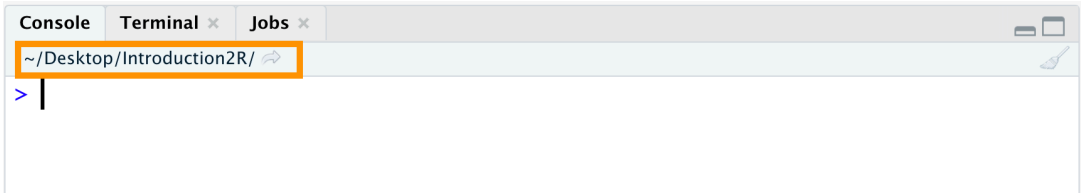


Figure 5.5: Workig directory dell'attuale sessione di lavoro

Premendo la freccia al suo fianco il pannello *Files* in basso a destra sarà reindirizzato direttamento alla workig directory dell'attuale sessione di lavoro. In questo modo sarà facile navigare tra i file e cartelle presenti al suo interno (vedi Figura 5.6).

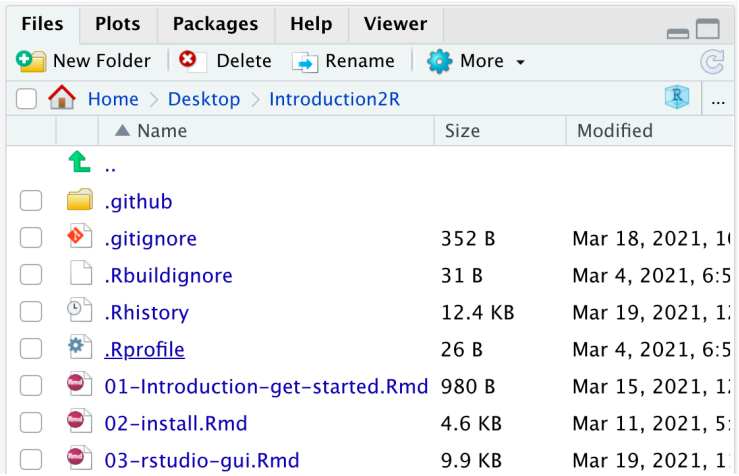


Figure 5.6: Workig directory dell'attuale sessione di lavoro

Cambiare Working Directory

Per cambiare la working directory è possibile utilizzare il comando `setwd()` indicando il path (absolute o relative) della nuova working directory. Nota come, nel caso in cui venga indicato un relative path, questo dovrà indicare la posizione della nuova working directory rispetto alla vecchia working directory.

```
getwd()
## [1] "/Users/<username>/Desktop/Introduction2R"

setwd("Dati/")

getwd()
## [1] "/Users/<username>/Desktop/Introduction2R/Dati"
```

In alternativa è possibile selezionare l'opzione *"Choose Directory"* dal menù *"Session"* > *"Set Working Directory"* come mostrato in Figura 5.7. Verrà quindi richiesto di selezionare la working directory desiderata e preme "*Open*".

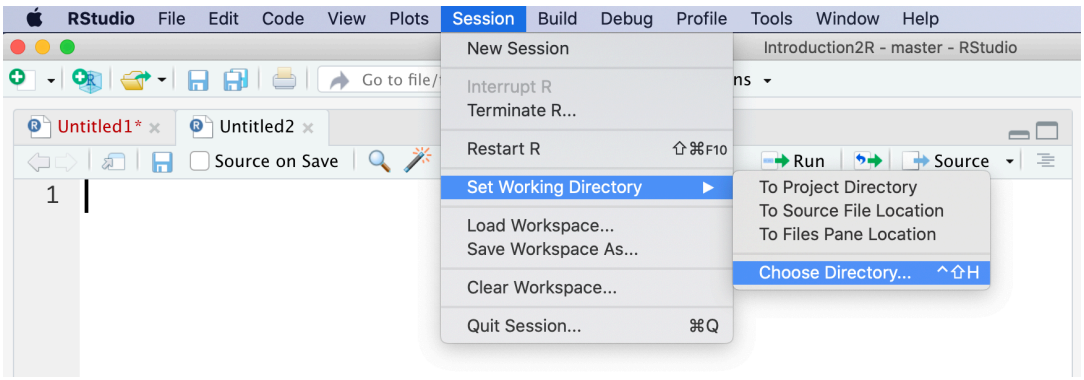
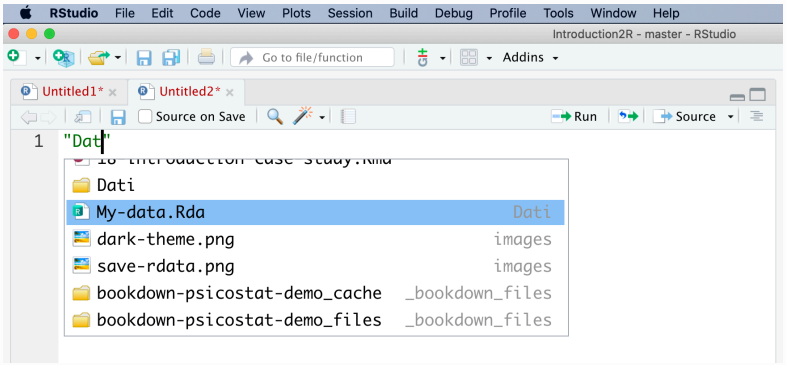


Figure 5.7: Definire la working directory

Trick-Box: Show me the Path

Nota come sia possibile nel digitare il path sfruttare l'autocompletamento. All'interno delle virgolette `""` premi il tasto `Tab` per visualizzare i suggerimenti dei path relativi alla attuale working directory.



E' possibile inoltre utilizzare i caratteri speciali `"./"` e `"../"` per indicare rispettivamente l'attuale working directory e la cartella del livello superiore (i.e., *parent folder*) che include l'attuale working directory. `"../"` ci permette quindi di navigare a ritroso dalla nostra attuale posizione tra le cartelle del computer.

```
getwd()
## [1] "/Users/<username>/Desktop/Introduction2R"

setwd("../")

getwd()
## [1] "/Users/<username>/Desktop/"
```

5.3 R-packages

Uno dei grandi punti di forza di R è quella di poter estendere le proprie funzioni di base in modo semplice ed intuitivo utilizzando nuovi pacchetti. Al momento esistono oltre **17'000** pachetti disponibili gratuitamente sul CRAN (la repository ufficiale di R). Questi pacchetti sono stati sviluppati dall'immensa comunity di R per svolgere ogni sorta di compito. Si potrebbe dire quindi che in R ogni cosa sia possibile, basta trovare il giusto pacchetto (oppure crearlo!).

Quando abbiamo installato R in automatico sono stati installati una serie di pacchetti che costituiscono la **system library**, ovvero tutti quei pacchetti di base che permettono il fuzionamento di R. Tuttavia, gli altri pacchetti non sono disponibili da subito. Per utilizzare le funzioni di altri pacchetti, è necessario seguire una procedura in due step come rappresentato in Figura 5.8:

1. **Scaricare ed installare i pacchetti sul nostro computer**. I pacchetti sono disponibili gratuitamente online nella reopsitory del CRAN, una sorta di archivio. Vengono quindi scaricati ed installati nella nostra *library*, ovvero la raccolta di tutti i pacchetti di R disponibili sul nostro computer.

2. **Caricare il paccheto nella sessione di lavoro**. Anche se il paccheto è installato nella nostra library non siamo ancora pronti per utilizzare le sue funzioni. Sarà necessario prima

caricare il pacchetto nella nostra sessione di lavoro. Solo ora le funzionni del pacchetto saranno effetivamente disponibili per essere usate.

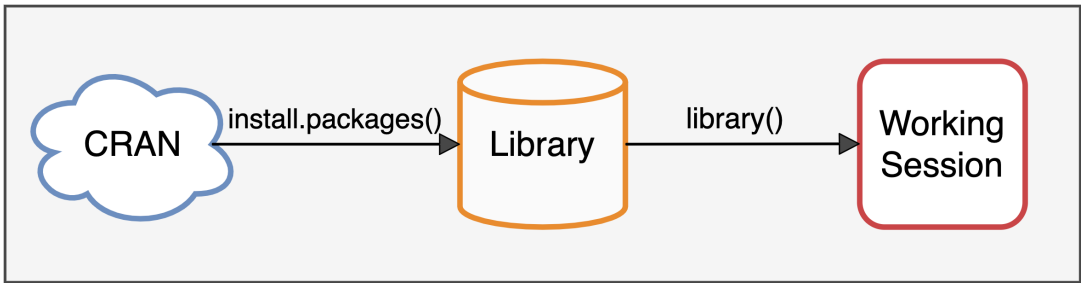


Figure 5.8: Utilizzare i paccheti in R

Questo procedimento in due step potrebbe sembrare poco intuitivo. *"Perchè dover caricare qualcosa che è già installato?"* La risposta è molto semplice ci serve per mantenere efficiente e sotto controllo la nostra sessione di lavoro. Infatti non avremo mai bisogno di tutti i pacchetti installati ma a seconda dei compiti da eseguire utilizzeremo di volta in volta solo alcuni pacchetti specifici. Se tutti i pacchetti fossero caricati automaticamente ogni volta sarebbe un inutile spreco di memoria e si creerebbero facilmente dei conflitti. Ovvero, alcune funzioni di diversi pacchetti potrebbero avere lo stesso nome ma scopi diversi. Sarebbe quindi molto facile ottenere errori o comunque risultati non validi.

Vediamo ora come eseguire queste operazioni in R.

5.3.1 install.packages()

Per installare dei pacchetti dal CRAN nella nostra library è possibile eseguire il comando `install.packages()` indicando tra parentesi il nome del pacchetto desiderato.

```
# Un ottimo pacchetto per le analisi statistiche di John Fox
# un grandissimo statistico...per gli amici Jonny la volpe ;)
install.packages("car")
```

In alternativa è possibile utilizzare il pulsante *"Install"* nella barra in alto a sinistra del pannello Packages (vedi Figura 5.9), indicando successivamente il nome del pacchetto desiderato.

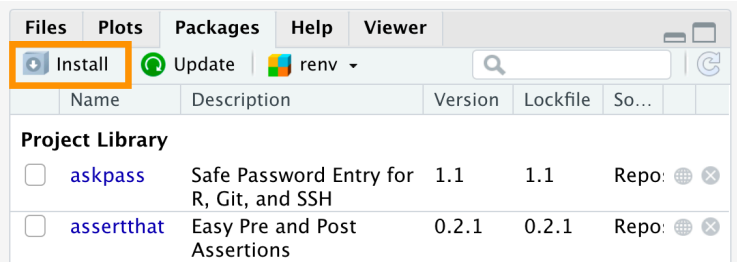


Figure 5.9: Installare paccektti tramite interfacci RStudio

Nota come installare un pacchetto potrebbe comportare l'installazione di più pacchetti. Questo perchè verranno automaticamente installate anche le *dependencies* del pacchetto, ovverro, tutti

i pacchetti usati internamente dal pacchetto di interesse che quindi necessari per il suo corretto funzionaemnto (come in un gioco di matrioske).

Una volta installato il pacchetto, questo comarirà nella library ovvero la lista dei pacchetti disponibili mostrata nel pannello Packages (vedi Figura 5.10).

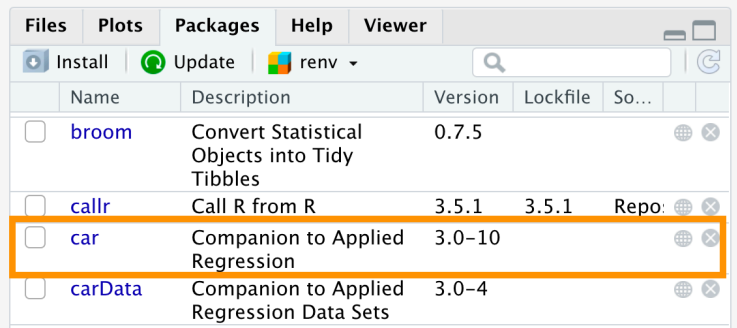


Figure 5.10: Il pacchetto car è ora disponibile nella library

Approfondimento: Binary or Source Version?

Nell'installare dei pacchetti, potrebbe accadere che R presenti un messaggio simile al seguente:

```
There are binary versions available but the
  source versions are later:
          binary source needs_compilation
devtools   1.13.4  2.0.1             FALSE
    [... una lista di vari pacchetti...]

Do you want to install from sources the packages which need compilation?
```

In breve, la risposta da dare è **NO** (`"n"`). Ma che cosa ci sta effetivamente chiedendo R? Esistono diversi modi in cui un pacchetto è disponibile, tra i principali abbiamo:

- **Versione Binary** - pronta all'uso e semplice da installare
- **Versione Source** - richiede una particolare procedura per essere installata detta compilazione

In genere quindi, è sempre preferibile installare la versione *Binary*. Tuttavia, in questo caso R ci avverte che, per alcuni pacchetti, gli aggiornameni più recenti sono disponibili solo nella versione *Source* e ci chiede quindi se installarli attraverso la procedura di compilazione.
E' preferibile rispondere *"no"*, installando così la versione *Binary* pronta all'uso anche se meno aggiornata. Qualora fosse richiesto obbligatoriamente

di installare un pacchetto nella version *Source* (perchè ci servono gli ultimi aggiornamenti o perchè non disponibile altrimenti) dovremmo avere prima installato **R tools** (vedi *"Approfondimento: R Tools"* nel Capitolo 1.1), che ci fornirà gli strumenti necessari per compilare i pacchetti.
Per una discussione dettagliata vedi https://community.rstudio.com/t/meaning-of-common-message-when-install-a-package-there-are-binary-versions-available-but-the-source-versions-are-later/2431 e https://r-pkgs.org/package-structure-state.html

5.3.2 library()

Per utilizzare le funzioni di un pacchetto già presente nella notra library, dobbiamo ora caricarlo nella nostra sessione di lavoro. Per fare ciò, posssiamo utilizzare il comando `library()` indicando tra parentesi il nome del pacchetto richiesto.

```
library(car)
```

In alternativa è possibile spuntare il riquadro alla sinistra del nome del pacchetto dal pannello Packages come mostrato in Figura 5.11. Nota tuttavia come questa procedura sia sconsigliata. Infatti, ogni azione punta-e-clicca dovrebbe essere eseguita ad ogni sessione mentre l'utilizzo di comandi inclusi nello script garantisce la loro esecuzione automatica.

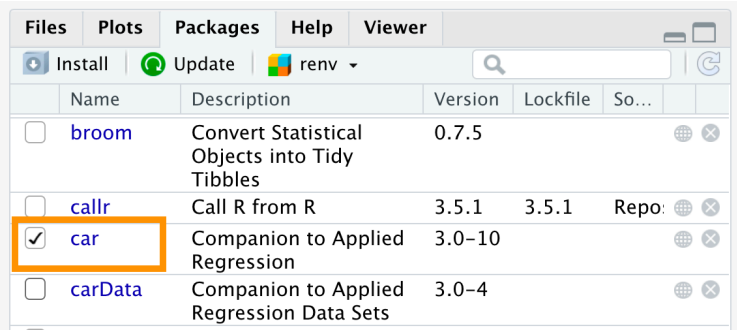


Figure 5.11: Caricare un pacchetto nella sessione di lavoro

Ora siamo finalmente pronti per utilizzare le funzioni del pacchetto nella nostra sessione di lavro.

Trick-Box: package::function()

Esiste un piccolo trucco per utilizzare la funzione di uno specifico pacchetto senza dover caricare il pacchetto nella propria sessione. Per fare questo è possibile usare la sintassi:

```
<nome-pacchetto>::<nome-funzione>()

# Esempio con la funzione Anova del pacchetto car
car::Anova()
```

L'utilizzo dei :: ci permette di richiamare direttamente la funzione desiderata. La differennza tra l'uso di `library()` e l'uso di :: riguarda aspetti abbastanza avanzati di R (per un approfondimento vedi https://r-pkgs.org/namespace.html). In estrema sintesi, possiamo dire che in alcuni casi è preferibile non caricare un'intero pacchetto se di questo abbiamo bisogno di un'unica funzione.

5.3.3 Aggiornare e Rimuovere Pacchetti

Anche i pacchettti come ogni altro software vengono aggiornati nel corso del tempo fornendo nuove funzionalità e risolvendo eventuli problemi. Per aggiornare i pacchetti alla versione più recente è possibile eseguire il comando `update.packages()` senza inidare nulla tra le parentesi.

In alternativa è possibile premere il pulsante *"Update"* nella barra in alto a sinistra del pannello Packages (vedi Figura 5.12), indicando successivamente i pachetti che si desidera aggiornare. Nota come nella lista dei pacchetti venga riportata l'attuale versione alla voce *"Version"*.

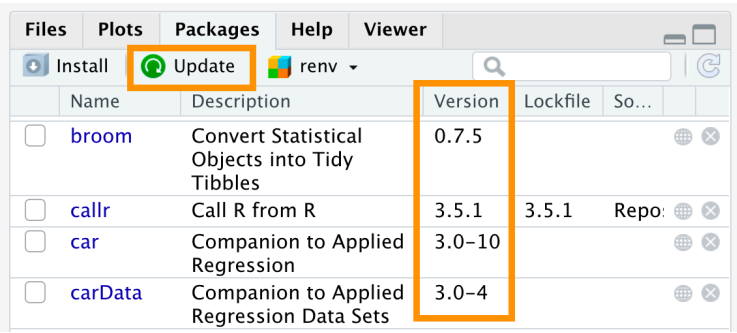


Figure 5.12: Aggiornare i pacchetti

Nel caso in cui si vogli invece rimuover uno specifico pacchetto, è possibile eseguire il comando `remove.packages()` indicando tra le parentesi il nome del pacchetto.

In alternativa è possibile premere il pulsante `x` alla destra del paccehettto nel pannello Packages come mostrao in Figura 5.13.

5.3.4 Documentazione Pacchetti

Ogni pacchetto include la documentazione delle proprie funzioni e delle *vignette* ovvero dei brevi tutorial che mostrano degli esempi di applicazione e utilizzo del pacchetto.

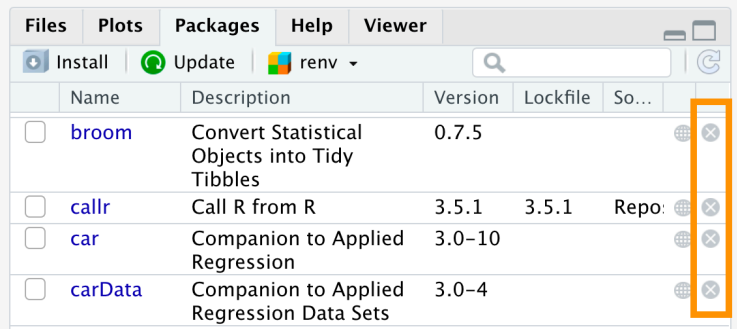


Figure 5.13: Rimuovere un pacchetto

- **Documentazione funzione** - Per accedere alla documentazione di una funzione è sufficiente utilizzare il comando `?<nome-funzione>` oppure `help(<nome-funzione>)`. Ricorda è necessario avere prima caricato il pacchetto altrimenti la funzione non risulta ancora disponibile. In alternativa si potrebbe estendere la ricerca utilizzando il comando `??`.
- **Vignette** - Per ottenere la lista di tutte le vignette di un determinato pacchetto è possibile utilizzare il comando `browseVignettes(package = <nome-pacchetto>)'`. Mentre, per accedere ad una specifica vignetta, si utilizza il comando `vignette("<name-vignetta>")`.
- **Documentazione intero pacchetto** - Premendo il nome del pacchetto dal pannello Packages in basso a destra, è possibile accedere alla lista di tutte le informazioni relative al pacchetto come riportato in Figura 5.14. Vengono prima forniti i link per le vignette ed altri file relativi alle caratteristiche del pacchetto. Successivamente sono presentate in ordine alfabetico tutte le funzioni.

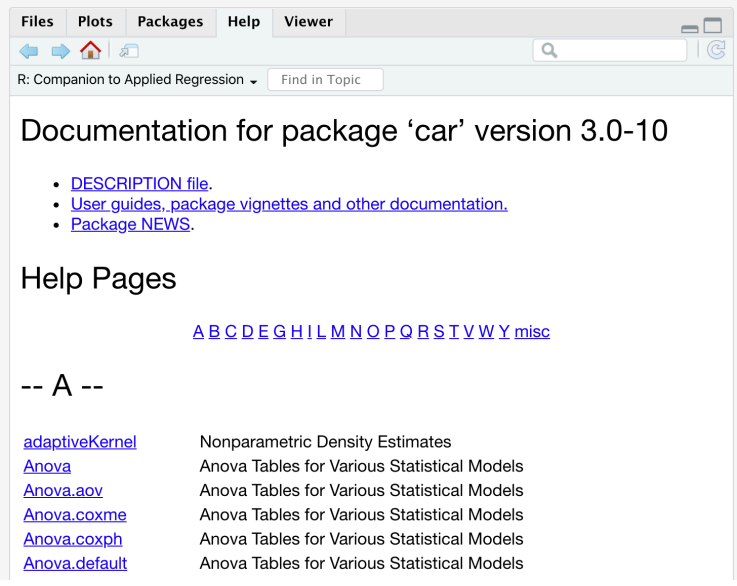


Figure 5.14: Documetazione del pacchetto car

Ricordate tuttavia che in ogni caso la più grande risorsa di informazioni è come sempre google. Spesso i pacchetti più importanti hanno addirittura un proprio sito in cui raccolgono molto ma-

teriale utile. Ma comunque in ogni caso in internet sono sempre disponibili moltissimi tutorial ed esempi.

Approfondimento: Github

Il CRAN non è l'unica risorsa da cui è possibile installare dei pacchetti di R tuttavia è quella ufficiale e garantisce un certo standard e stabilità dei pacchetti presenti. In internet esistono molte altre repository che raccolgono paccetti di R (e software in generale) tra cui una delle più popolari è certamente GitHub (https://github.com/).
Github viene utilizzato come piattaforma di sviluppo per molti pacchetti di R ed è quindi possibile trovarve le ultime versioni di sviluppo dei pacchetti con gli aggirnamenti più recenti o anche nuovi pacchetti non ancora disponibili sul CRAN. Va sottolineato tuttavia, come quete siano appunto delle versioni di sviluppo e quindi potrebbero presentare maggiori problemi. Inoltre per installare i pacchetti in questo modo, è richiesta l'installzione di **R tools** (vedi *"Approfondimento: R Tools"* nel Capitolo 1.1).
Per installare un pacchetto direttamente da Github è posibile utilizzare il comando `install_github()` del pacchetto `devtools`, indicanto tra parentesi la l'url della repository desiderata.

```
install.packages("devtools")

# ggplot2 il miglior pacchetto per grafici
devtools::install_github("https://github.com/tidyverse/ggplot2")
```

Chapter 6

Sessione di Lavoro

In questo capitolo, discuteremo di alcuni aspetti generali delle sessioni di lavoro in R. Descriveremo delle buone abitudini riguardanti l'organizzazione degli scripts per essere ordinati ed efficaci nel proprio lavoro. Inoltre descriveremo come organizzare i propri progetti in cartelle ed in particolare introdurremo l'uso degli *RStudio Projects*. Infine approfondiremo l'uso dei messaggi di R ed in particolare come comportarsi in caso di errori.

6.1 Organizzazione Script

Abbiamo visto che idealmente tutti i passaggi delle nostre analisi devono essere raccolti in modo ordinato all'interno di uno script. Eseguendo in ordine linea per linea i comandi, dovrebbe essere possibile svolgere tutte le analisi fino ad ottenere i risultati desiderati.

Vediamo ora una serie di buone regole per organizzare in modo ordinato il codice all'interno di uno script e facilitare la sua lettura.

6.1.1 Creare delle Sezioni

Per mantenere chiara l'oreganizzazione degli script e facilitare la sua comprensione, è utile suddividere il codice in sezioni dove vengono eseguiti i diversi step delle analisi. In RStudio è possibile creare una sezione aggiungendo al termine di una linea di commento i caratteri `####` o `----`. Il testo del commento verrà considerao il titolo della sezione e comparirà una piccola freccia a lato del numero di riga. E' possibile utilizzare a piacere i caratteri `#` o `-` per creare lo stile desiderato, l'importante è che la linea si concluda con almeno quattro caratteri identici.

```
# Sezione 1 ####

# Sezione 2 ----

#----    Sezione 3    ----

####   Sezione non valida   --##
```

A titolo del tutto esemplificativo prendiamo in esempio la divisione in sezioni utilizzata nello script in Figura 6.1.

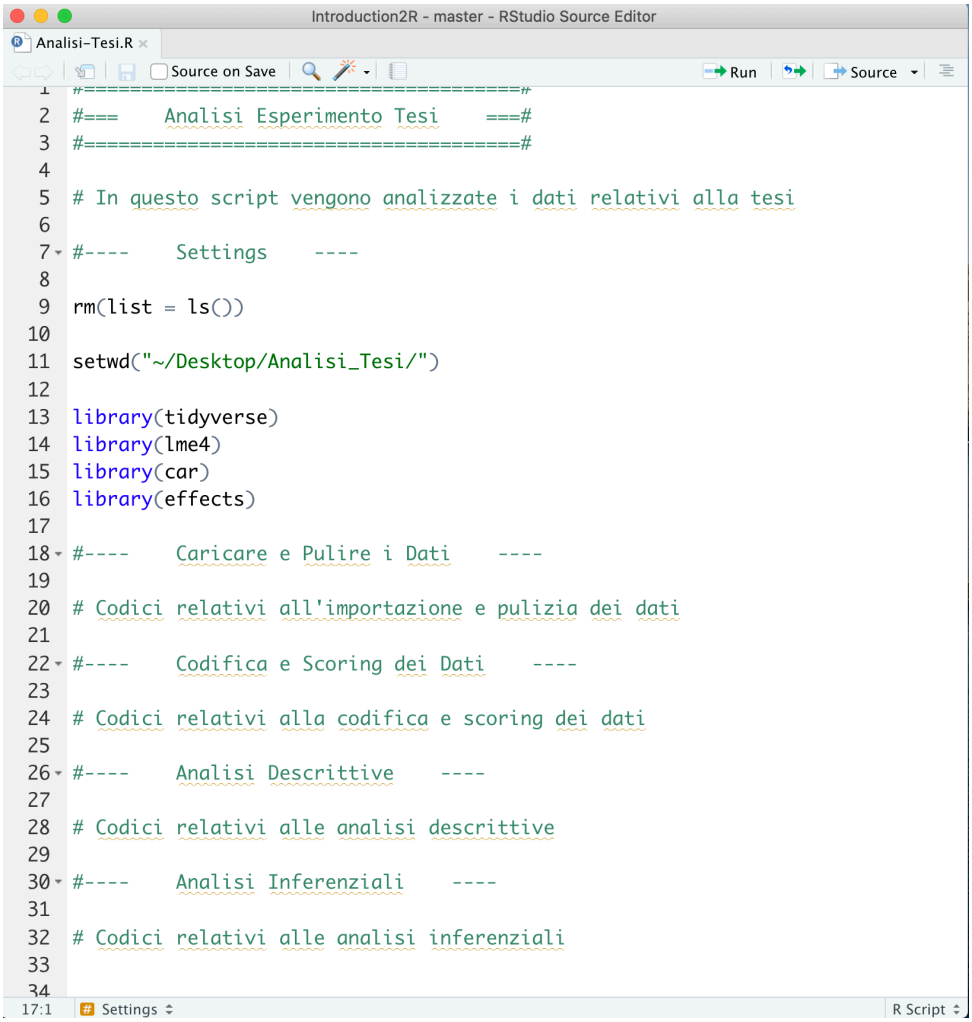


Figure 6.1: Esempio di suddivisione in sezioni di uno script

- **Titolo** - Un titolo esplicativo del contenuto dello script. E' possibile utilizzare altri caratteri all'interno dei commenti per creare l'effetto desiderato.
- **Intorduzione** - Descrizione e utili informazioni che riguardano sia l'obbiettivo del lavoro che l'esecuzione del codice (e.g., dove sono disponibili i dati, eventuali specifiche tecniche). Potrebbe essere utile anche indicare l'autore e la data del lavoro.
- **Setting** - Sezione fondamentale in cui si predispone l'abiente di lavoro. Le operazioni da svolgere sono:

  1. `rm(list = ls())` per pulire l'Environment da eventuali oggetti in modo da eseguire lo script partendo da un ambiente vuoto (vedi Capitolo 5.1).
  2. `setwd()` per settare la working directory per assicuraci che i comandi siano eseguiti dalla correttta posizione nel nostro computer (vedi Capitolo 5.2).
  3. `library()` per caricare i pacchetti utilizzati nel corso delle analisi (vedi Capitolo 5.3).

- **Caricare e Pulire i Dati** - Generica sezione in cui eseguire l'importazione e pulizia dei dati.
- **Codifica e Scoring dei Dati** - Generica sezione in cui eseguire la codifica ed eventuale scoring dei dati.
- **Analisi Descrittive** - Generica sezione in cui eseguire le analisi descrittive.
- **Analisi Inferenziali** - Generica sezione in cui eseguire le analisi inferenziali

Oltre che a mantenere ordinato e chiaro il codice, suddividere il proprio script in sezioni ci permette anche di navigare facilmente tra le diverse parti del codice. Possiamo infatti sfruttare l'indice che automaticamente viene creato. L'indice è consultabile premendo il tasto in alto a destra dello script da cui successivamente selezionare la sezione desiderata (vedi Figura 6.2).

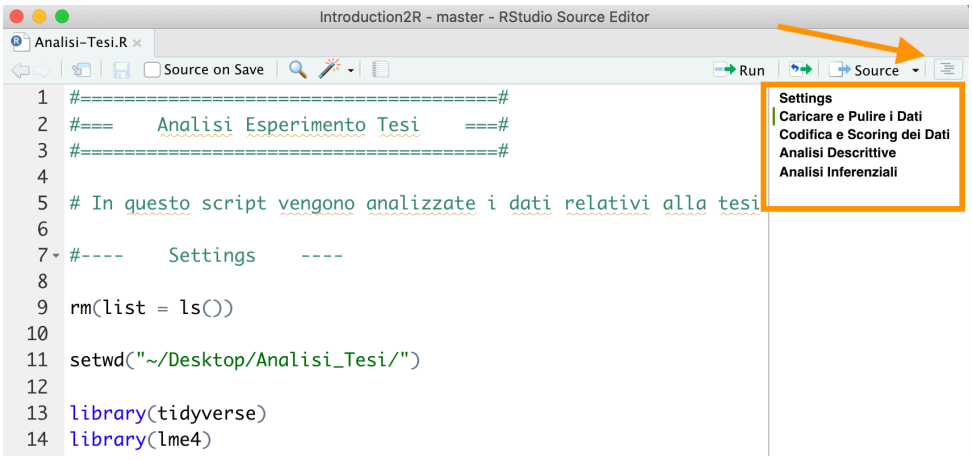


Figure 6.2: Indice in alto per navigazione sezioni

In alternativa, è possibile utilizzare il menù in basso a sinistra dello script (vedi Figura 6.3).

Infine, un altro vantaggio è quello di poter compattare o espandere le sezioni di codice all'interno dell'editor, utilizzando le frecce a lato del numero di riga (vedi Figura 6.4).

6.1.2 Sintassi

Elenchiamo qui altre buone norme nella scrittura del codice che ne facilitano la comprensione.

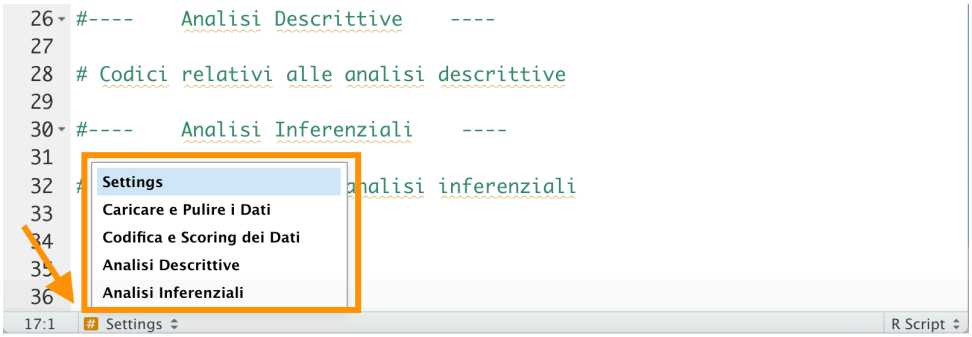


Figure 6.3: Menù in basso per navigazione sezioni

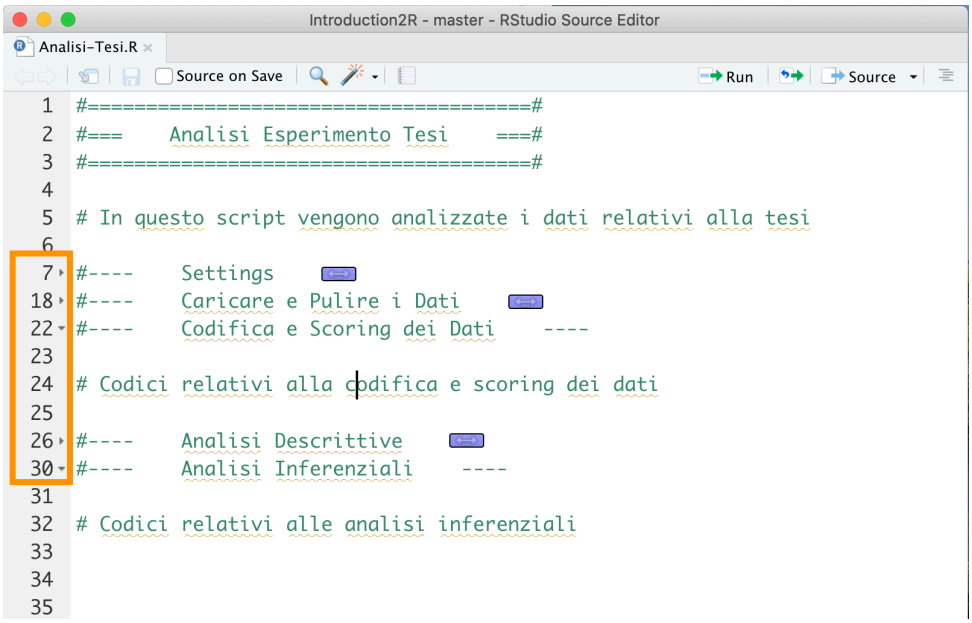


Figure 6.4: Compattare ed espandere le sezioni di codice

Commenti

L'uso dei commenti è molto importante, ci permette di documentare le varie parti del codice e chiarire eventuli comandi difficili da capire. Tuttavia, non è necessario commentare ogni singola riga di codice ed anzi è meglio evitare di commentare laddove i comandi sono facilemtne interpetabili semplicemnte leggendo il codice.

La capacità di scrivere commenti utili ed eviare quelli rindondanti si impara con l'esperienza. In generale un commento non dovrebbe indicare *"che cosa"* ma piuttosto il *"perchè"* di quella parte di codice. Infatti il cosa è facilmente interpretabile dal codice stesso mentre il perchè potrebbe essere meno ovvio e soprattutto più utile per la comprensione dell'intero script. Ad esempio:

```
x <- 10 # assegno a x il valore 10
x <- 10 # definisco massimo numero risposte
```

Il primo commento è inutile poichè è facilmente comprensibile dal codice stesso, mentre il secondo commento è motlo utile perchè chiarisce il significato della variabile e mi faciliterà nella comprensione del codice.

Nomi Oggetti

Abbiamo visto nel Capitolo 4.1.2 le regole che discriminano nomi validi da nomi non validi e le convenzioni da seguire nella definizione di un nome. Ricordiamo qui le caratterisiche che un nome deve avere per facilitare la comprensione del codice. Il nome di un oggetto deve essere:

- **auto-descrittivo** - Dal solo nome dovrebbe essere possibile intuire il contenuto dell'oggetto. E' meglio quindi evitare nomi generici (quali `x` o `y`) ed utilizzare invece nomi che chiaramente descrivano il contenuto dell'oggetto.
- **della giusta lunghezza** - Non deve essere ne troppo breve (evitare sigle incomprensibili) ma neppure troppo lunghi. In genere sono sufficienti 2 o 3 parole per descrivere chiaramente un oggetto.

E' inoltre importante essere **consistenti** nella scelta dello stile con cui si nominano le variabili. In genere è preferibile usare lo **snake_case** rispetto al **CamelCase**, ma la scelta è comunque libera. Tuttavia, una volta presa una decisione, è bene mantenerla per facilitare la comprensione del codice.

Esplicitare Argomenti

Abbiamo visto nel Capitolo 4.2.1 l'importanza di esplicitare il nome degli argomenti quando vengono utilizzati nelle funzioni. Specificando a che cosa si riferiscono i vari valori facilitiamo la lettura e la comprensione del codice. Ad esempio:

```
seq(0, 10, 2)
```

Potrebbe non essere chiaro se intendiamo una sequennza tra 0 e 10 di lunghezza 2 o a intervalli di 2. Specificando gli argomenti evitiamo incomprensioni e possibili errori.

```
seq(from = 0, to = 10, by = 2)
seq(from = 0, to = 10, length.out = 2)
```

Spazi, Indentazione ed allineamento

Al contrario di molti altri software, R non impone regole severe nell'utilizzo di spazi, indentazioni ed allineamenti ed in genere è molto permissivo per quanto riguarda la sintassi del codice. Tuttavia è importante ricordare che:

> Good coding style is like using correct punctuation. You can manage without it, but it sure makes things easier to read. Hadley Wickham

Prendiamo ad esempio le seguenti linee di codice, che includono delle funzioni avanzate di R:

```
# Stile 1
k=10;if(k<5){x<-5:15}else{x<-seq(0,16,4)};y=7*2-12;mean(x/y)
## [1] 4

# Stile 2
k <-  10

if (k < 5){
  x <- 5:15
} else {
  x<-seq(from = 0, to = 16, by = 4)
}

y <- 7 * 2 - 12

mean(x / y)
## [1] 4
```

Come puoi notare otteniamo in entrambi i casi gli stessi risultati, per R non c'è alcuna differenza. Tuttavia, l'uso di spazi, una corretta indentazione ed un appropriato allineamento facilita la lettura e comprensione del codice.

In genere sono valide le seguenti regole:

- Aggiungi degli **spazi** intorno agli operatori (`+`, `-`, `<-`, etc.) per separargli dagli argomenti ad eccezione di `:`.

```
# Good
35 / 5 + 7
x <- 0:10

# Bad
35/5+7
x<-0 : 10
```

- Nelle funzioni aggiungi degli **spazi** intorno al simbolo = che separa il nome degli argomenti e il loro valore. Aggiungi uno spazio dopo ogni virogola ma non separare il nome della funzione dalla parentesi sinistra.

```
# Good
seq(from = 0, to = 10, by = 2)

# Bad
seq (from=0,to=10,by=2)
```

- Usa la corretta **indentazione** per i blocchi di codice posti all'interno delle parentesi graffe. Il livello di indentazione deve rispecchiare la struttura di annidamento del codice.

```
# Good
for (...) {       # loop più esterno
  ...
  for (...) {     # loop interno
    ...
    if (...) {    # isruzione condizionale
      ...
    }
  }
}

# Bad
for (...) {       # loop più esterno
...
for (...) {     # loop interno
...
if (...) {    # isruzione condizionale
...
}
}
}
```

- **Allinea** gli argomenti di una funzione se questi spaziano più righe.

```
# Good
data.frame(id = ...,
           name = ...,
           age = ...,
           sex = ...)

# Bad
data.frame(id = ..., name = ...,
age = ..., sex = ...)
```

Approfondimento: Tutta una Questione di Stile

Potete trovare ulteriori regole e consigli riguardanti lo stile nella scrittura di codici al seguente link https://irudnyts.github.io/r-coding-style-guide/.

6.2 Organizzazione Progetti

All'aumentare della complessità di un'analisi vi troverete presto a dover gestire molti file di deversa tipologia (e.g., dati, report, grafici) e a dover suddividere le varie parti del lavoro in differenti script. A questo punto sarà opportuno organizzare in modo ordinato tutto il materiale necessario per l'analisi e i relativi risultati in un'unica cartella.

Idealmente, infatti, ogni analisi verrà salvata in una differente cartella e possiamo ad esempio organizzare i file utilizzando la seguente struttura di sottocartelle:

```
My_analysis/
 |-  Data/
 |-  Documents/
 |-  Outputs/
 |-  R/
```

- `Data/` - tutti i file relativi ai dati usati nell'analisi. Sarà importante mantenere sia una copia dei dati *raw*, ovvero i dati grezzi senza alcuan manipolazione, sia i dati effetivamente usati nelle analisi che in genere sono stati già puliti e codificati.
- `Documents/` - tutti i file di testo (e.g., Word o simili) e report utilizzati per descrivere le analisi (vedi in particolare R-markdown).
- `Outputs/` - eventuali outputs creati durante le analisi come ad esempio grafici e tabelle.
- `R/` - tutti gli script utilizzati per le analisi. E' possibile numerare gli script per suggerire il corretto ordine in cui debbano essere eseguiti, ad esempio `01-import-data.R`, `02-munge-data.R`, `03-descriptive-analysis.R` etc.

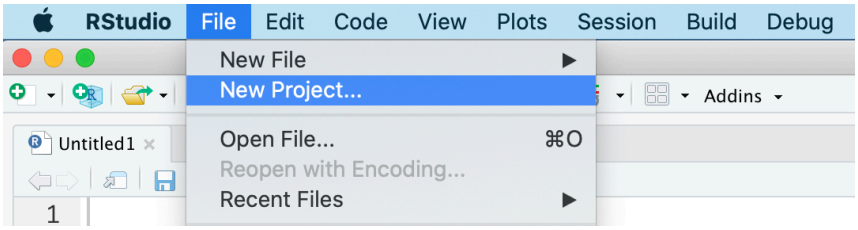
Questa struttura è puramente esemplificativa ma può essere un utile base di partenza che è possibile adattare a seconda delle particolari esigenze di ogni lavoro.

6.2.1 RStudio Projects

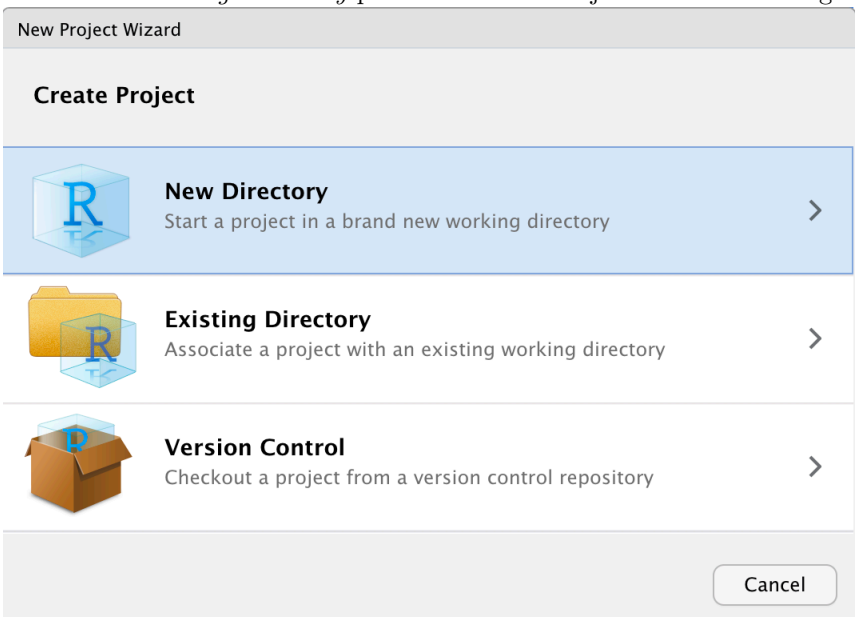
RStudio permette inoltre di creare degli **R Projects**, ovvero degli spazi di lavoro che permettono di gestire indipendentemente diversi progetti. Ogni progetto avrà la propria working directory, workspace, history e settaggi personalizzati. Questo consente di passare velocemente da un progetto ad un altro riprendento immediatamente il lavoro da dove si era arrivati senza altre preoccupazioni.

Vediamo quindi come creare un **R Projects**.

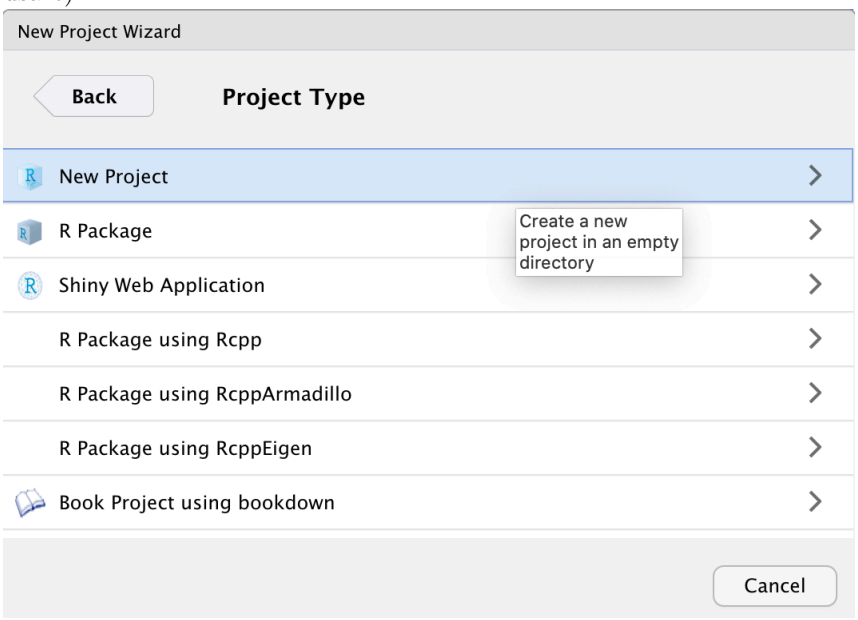
1. Selezionare *File* > *New Project...*



2. Selezionare *New Directory* per creare un R Project in una nuova cartella (in alternativa selezionare *Existing Directory* per creare un R Project in una cartella già esistente)

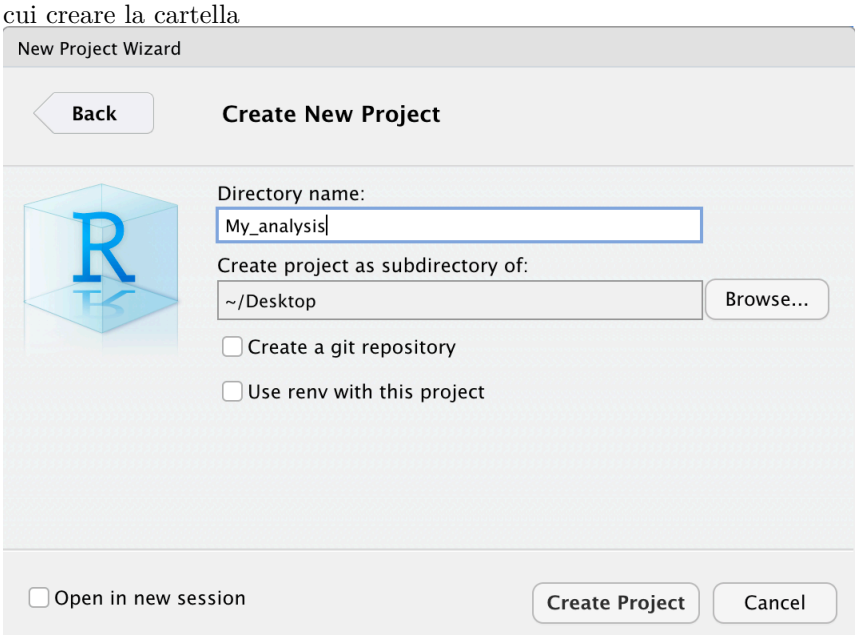


3. Selezionare *New Project* (in utilizzi avanzati è possibile scegliere particolari template da usare)



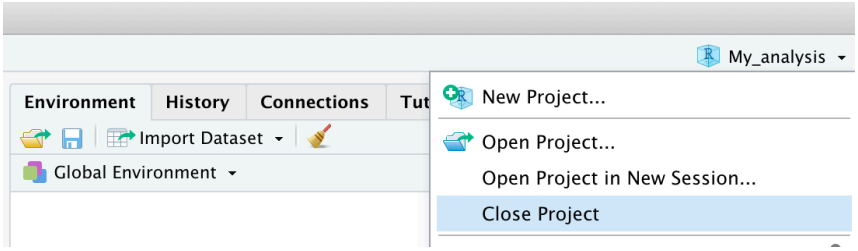
4. Indicare il nome della cartella (utilizzato anche come nome dell'R Project) e la posizione in

cui creare la cartella

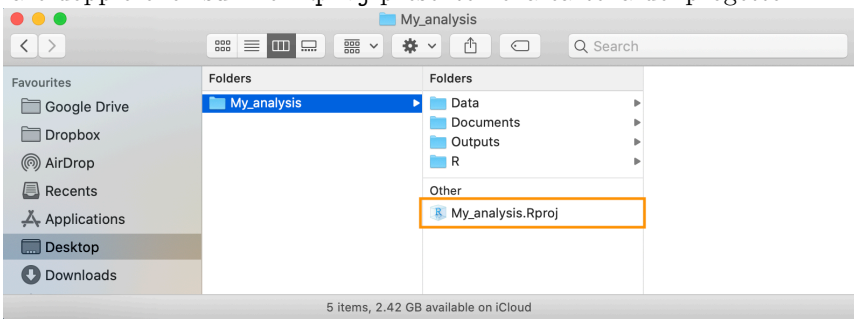


5. Una volta creato il progetto RStudio aprirà direttamente il progetto. Notate come nell'icona di RStudio comparirà ora anche il nome del progetto attualmente aperto

6. Per chiudere un progetto è sufficiente selezionare *File* > *Close Project* o in alterantiva utilizzare il menù in alto a destra



7. Per aprire un progetto precedente, è sufficiente selezionare *File* > *Open Project* o in alterantiva fare doppio click sul file `.Rproj` presente nella cartella del progetto



Elenchiamo ora alcuni dei principali vantaggi dell'utilizzare gli R Project:

- La **working directory** viene autoamticamente settata nella cartella del progetto. Questo ci permette di non doverci più preoccupare della sua definizione e possiamo definire ogni *path* in relazione alla cartella del progetto.
- Quando apriamo un progetto in una successiva **sessione di lavoro** gli script e i documenti di lavoro verranno automaticamente aperti come li avevamo lasciati nella sessione precedente. E' come avere una scrivania per ogni progetto dove lasciare tutti i documenti utili e riprendere il lavoro immediatamete.
- Utilizzando i progetti è possibile **personnalizzare e automatizzare** molte funzioni come ad esempio caricare i paccheti o eseguire determinati codici. Tuttavia richiedono una buona conoscenza di R.
- Esistono molti **template** per i progetti che implementano utili funzionalità, in particolare la struttura degli R Projects usata per lo sviluppo dei pacchetti è molto utile anche nel caso di analisi statistiche. Tuttavia richiedono una buona conoscenza di R.

6.3 Messages, Warnings e Errors

R utilizza la console per comunicare con noi durante le nostre sessioni di lavoro. Oltre a fornirci i risultati dei nostri comandi, R ci segnala anche altre utili informazioni attraverso diverse tipologie di messaggi. In particolare abbiamo:

- **Messages**: dei semplici messaggi che ci possono aggiornare ad esempio sullo stato di avanzamento di un dato compito oppure fornire suggerimenti sull'uso di una determinata funzione o pacchetto (spesso vengono mostrati quando viene caricato un pacchetto).
- **Warnings**: questi messaggi sono utlizzati da R per dirci che c'è stato qualche cosa di strano che ha messo in allerta R. R ci avvisa che, sebbene il comando sia stato eseguito ed abbiamo ottenuto un risultato, ci sono stati dei comportamenti inusuali o magari eventuali correzioni apportate in automatico. Nel caso di warnings non ci dobbiamo allramre, è importante controllare che i comandi siano corretti e che abbiamo effetivamente ottenuto il risultato desiderato. Una volta sicuri dei risultati possiamo procedere tranquillamente.
- **Errors**: R ci avvisa di eventuali errori e problemi che non permettono di eseguire il comando. in questo caso non otterremo nnessun risultato ma sarà necessario capire e risolvere il problema per poi rieseguire nuovamente il comando e procedere.

Notiamo quindi come non tutti i messaggi che R ci manda sono dei messaggi di errore. E' quindi importante non spaventarsi ma leggere con attenzione i messaggi, molte volte si tratta semplicemente di avvertimenti o suggerimenti.

Tuttavia gli errori rappresentano sempre il maggiore dei problemi perchè non è possibile procedere nel lavoro senza averli prima risolti. E' importante ricordare che i messaggi di errore non sono delle critiche che R ci rivolge perchè sbagliamo. Al contrario, sono delle richieste di aiuto fatte da R perchè non sa come comportarsi. Per quanto super potente, R è un seplice programma che non può interpretare le nostre richieste ma si basa sull'uso dei comandi che seguono una rigida sintassi. A volte è sufficiente una virgola mancante o un charattere al posto di un numero per mandare in confusione R e richiedere il nostro intervento risolutore.

6.3.1 Risolvere gli Errori

Quando si approccia la scrittura di codice, anche molto semplice, la cosa che sicuramente capiterà più spesso sarà riscontrare messaggi di **errore** e quindi trovare il modo per risolverli.

> Qualche programmatore esperto direbbe che l'essenza stessa di programmare è in realtà risolvere gli errori che il codice produce.

L'**errore non è quindi un difetto o un imprevisto**, ma parte integrante della scrittura del codice. L'importante è capire come gestirlo.

Abbiamo tutti le immagini in testa di programmatori da film che scrivono codice alla velocità della luce, quando nella realtà dobbiamo spesso affrontare **bug**, **errori di output** o altri problemi vari. Una serie di skills utili da imparare sono:

- Comprendere a fondo gli **errori** (non banale)
- Sapere **come e dove cercare una soluzione** (ancora meno banale)
- In caso non si trovi una soluzione direttamente, chiedere aiuto in modo efficace

Comprendere gli errori

Leggere con attenzione i messaggi di errore è molto importante. R è solitamente abbastanza esplicito nel farci capire il problema. Ad esempio usare una funzione di un pacchetto che non è stato caricato di solito fornisce un messaggio del tipo `Error in funzione : could not find function "funzione"`.

Tuttavia, in altre situazioni i messaggi potrebbero non essere altrettanto chiari. Seppur esplicito R è anche molto sintetico e quindi l'utilzzo di un linguaggio molto specifico (e almeno inizialmente poco familiare), potrebbe rendere difficile capire il loro significato o addirittura renderli del tutto incomprensibili. Man mano che diventerete più esperti in R, diventerà sempre più semplice ed immmediato capire quale sia il problema e anche come risolverlo. Ma nel caso non si conosca la soluzione è necessario cercarla in altro modo.

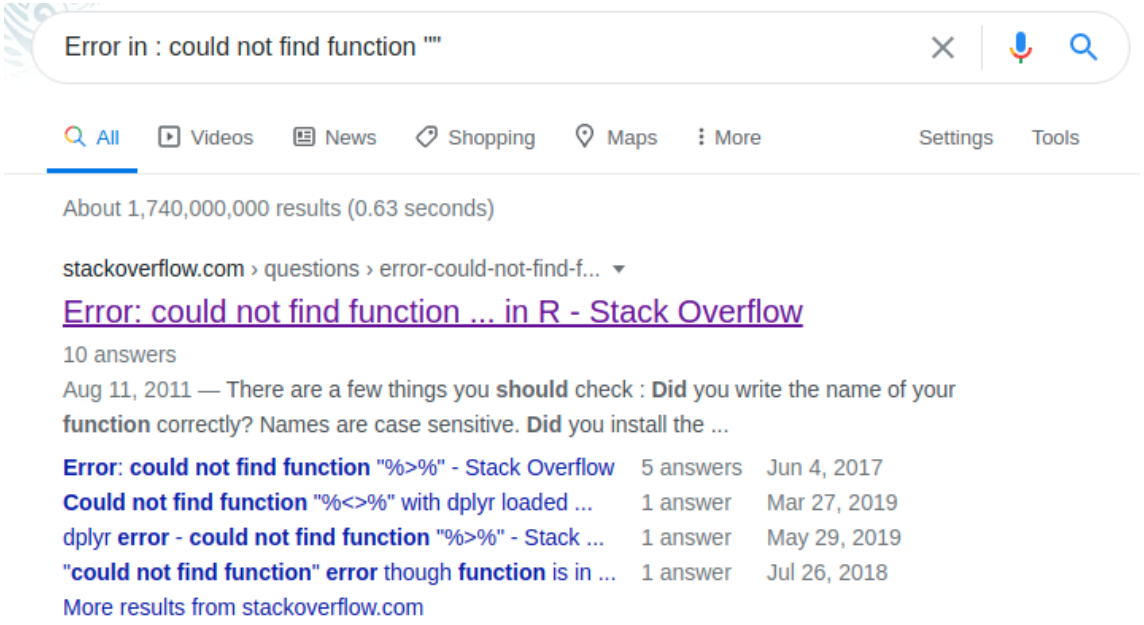
Problema + Google = Soluzione

In qualsiasi situazione Google è il nostro miglior amico.

Cercando infatti il messaggio di errore/warning su Google, al 99% avremo altre persone che hanno avuto lo stesso problema e probabilmente anche una soluzione.

> Tip-Box: Ricerca su Google
>
> Il modo migliore per cercare è copiare e incollare su Google direttamente l'output di errore di R come ad esempio `Error in funzione : could not find function "funzione"` piuttosto che descrivere a parole il problema. I messaggi di errore sono standard per tutti, la tua descrizione invece no.

Cercando in questo modo vedrete che molti dei risultati saranno esattamente riferiti al vostro errore:



Chiedere una soluzione

Se invece il vostro probelma non è un messaggio di errore ma un utilizzo specifico di R allora il consiglio è di usare una ricerca del tipo: `argomento + breve descrizione problema + R`. Nelle sezioni successive vedrete nel dettaglio altri aspetti della programmazione ma se volete ad esempio calcolare la **media** in R potrete scrivere `compute mean in R`. Mi raccomando, fate tutte le ricerche in **inglese** perchè le possibilità di trovare una soluzione sono molto più alte.

Dopo qualche ricerca, vi renderete conto che il sito che vedrete più spesso si chiama **Stack Overflow**. Questo è una manna dal cielo per tutti i programmatori, a qualsiasi livello di expertise. E' una community dove tramite domande e risposte, si impara a risolvere i vari problemi ed anche a trovare nuovi modi di fare la stessa cosa. E' veramente utile oltre che un ottimo modo per imparare.

L'ultimo punto di questa piccola guida alla ricerca di soluzioni, riguarda il fatto di dover non solo cercare ma anche chiedere. Dopo aver cercato vari post di persone che richiedevano aiuto per un problema noterete che le domande e le risposte hanno sempre una struttura simile. Questo non è solo un fatto stilistico ma anzi è molto utile per uniformare e rendere chiara la domanda ma sopratutto la risposta, in uno spirito di condivisione. C'è anche una guida dedicata per scrivere la domanda perfetta.

In generale[1]:

- Titolo: un super riassunto del problema
- Contesto: linguaggio (es. R), quale sistema operativo (es. Windows)
- Descrizione del problema/richiesta: in modo chiaro e semplice ma non troppo generico
- Codice ed eventuali dati per capire il problema

L'ultimo punto di questa lista è forse il più importante e si chiama in gergo tecnico **REPREX** (**Rep**roducible **Ex**ample). E' un tema leggermente più avanzato ma l'idea di fondo è quella di

[1] Fonte: Writing the perfect question - Jon Skeet

fornire tutte le informazioni possibili per poter riprodurre (e quindi eventualmente trovare una soluzione) il codice di qualcuno nel proprio computer.

Se vi dico "R non mi fa creare un nuovo oggetto, quale è l'errore?" è diverso da dire "il comando `oggetto -> 10` mi da questo errore `Error in 10 <- oggetto : invalid (do_set) left-hand side to assignment`, come posso risolvere?"

Trick-Box: reprex

Ci sono anche diversi pacchetti in R che rendono automatico creare questi esempi di codice da poter condividere, come il pacchetto `reprex`.

Struttura Dati

Introduzione

In questa sezione verranno introdotte le principali tipologie di oggetti usati in R . Ovvero le principali strutture in cui possono essere organizzati i dati: Vettori, Matrici, Dataframe e Liste.

Per ognuna di esse descriveremo le loro caretteristiche e vedremo come crearle, modificarle e manipolarle a seconda delle necessità

I capitoli sono così organizzati:

- **Capitolo 7 - Vettori**. Impareremo le caratterisiche e l'uso dei vettori soffermandoci anche sulle diverse tipologie di dati.
- **Capitolo 8 - Fattori**. Impareremo le caratterisiche e l'uso dei fattori, ovvero un particolare tipo di vettori usati per le variabili categoriali ed ordinali.
- **Capitolo 9 - Matrici**. Impareremo le caratterisiche e l'uso delle matrici introducendo anche gli array.
- **Capitolo 10 - Dataframe**. Impareremo le caratterisiche e l'uso dei datafraeme, l'oggetto maggiormmente utilizzato nell'analisi di dati.
- **Capitolo 11 - Liste**. Impareremo le caratterisiche e l'uso delle liste.

Chapter 7

Vettori

I vettori sono la struttura dati più semplice tra quelle presenti in R. Un vettore non è altro che un insieme di elementi disposti in uno specifico ordine e possiamo quindi immaginarlo in modo simile a quanto rappresentato in Figura 7.1.

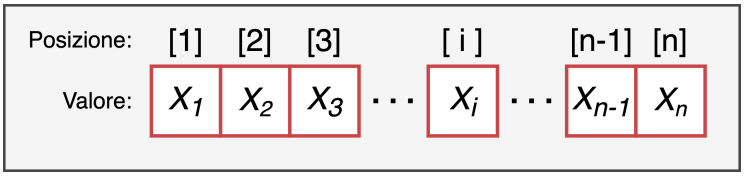


Figure 7.1: Rappresentazione della struttura di un vettore di lunghezza *n*

Due caratteristiche importanti di un vettore sono:

- la **lunghezza** - il numero di elementi da cui è formato il vettore
- la **tipologia** - la tipologia di dati da cui è formato il vettore. Un vettore infatti deve esssere formato da **elementi tutti dello stesso tipo** e pertanto esistono diversi vettori a seconda della tipologia di dati da cui è formato (valori numerici, valori interi, valori logici, valori carattere).

E' fondamentale inoltre sottolineare come ogni **elemento** di un vettore sia caratterizzato da:

- un **valore** - ovvero il valore dell'elemento che può essere di qualsiasi tipo ad esempio un numero o una serie di caratteri.
- un **indice di posizione** - ovvero un numero intero positivo che identifica la sua posizione all'interno del vettore.

Notiamo quindi come i vettori x e y così definiti:

$$x = [1, 3, 5]; \quad y = [3, 1, 5],$$

sebbene includano gli stessi elementi, non sono identici poichè differiscono per la loro disposizione. Tutto questo ci serve solo per ribadire come l'ordine degli elementi sia fondamentale per la valutazione di un vettore.

Vedimao ora come creare dei vettori in R e come compiere le comuni operazini di selezione e manipolazione di vettori. Successivamente approfondiremo le caratteristiche dei vettori valutandone le diverse tipologie.

7.1 Creazione

In realtà abbiamo già incontrato dei vettori nei precedenti capitoli poichè anche le variabili con un singolo valore altro non sono che un vettore di lunghezza 1. Tuttavia, per creare dei vettori di più elementi dobbiamo utilizzare il comando `c()`, ovvero *"combine"*, indicando tra le parentesi i valori degli elementi nella sucessione desiderata e separati da una virgola. Avremo quindi la seguente sintassi:

```
nome_vettore <- c(x_1, x_2, x_3, ..., x_n)
```

Nota come gli elementi di un vettore debbano essere tutti della stessa tipologia ad esempio valori numerici o valori carattere.

Approfondimento: Sequenze - ':', seq() e rep()

In altrentativa è possibile utilizzare qualsiasi funzione che restituisca come output una sequenza di valori sotto forma di vettore. Tra le funzioni più usate per creare delle sequenze abbiammo:

- `<from>:<to>` - Genera una sequenza di valori numerici crescenti (o decrescenti) dal primo valore indicato (`<from>`) al secondo valore indicato (`<to>`) a step di 1 (o -1).

```
# sequenza crescente
1:5
## [1] 1 2 3 4 5

# sequenza decrescente
2:-2
## [1]  2  1  0 -1 -2

# sequenza con valori decimali
5.3:10
## [1] 5.3 6.3 7.3 8.3 9.3
```

- `seq(from = , to = , by = , length.out = )` - Genera una sequenza regolare di valori numerici compresi tra `from` e `to` con incrementi indicati da `by`, oppure di lunghezza conplessiva indicata da `length.out` (vedi `?seq()` per maggiori dettagli).

```r
# sequenza a incrementi di 2
seq(from = 0, to = 10, by = 2)
## [1]  0  2  4  6  8 10

# sequenza di 5 elementi
seq(from = 0, to = 1, length.out = 5)
## [1] 0.00 0.25 0.50 0.75 1.00
```

- `rep(x, times = , each = )` - Genera una sequenza di valori ripetendo i valori contenuti in `x`. I valori di `x` posssono essere ripetuti nello stesso ordine più volte specificando `times` oppure ripetuti ciascuno più volte specificando `each` (vedi `?rep()` per maggiori dettagli).

```r
# sequenza a incrementi di 2
rep(c("foo", "bar"), times = 3)
## [1] "foo" "bar" "foo" "bar" "foo" "bar"

# sequenza di 5 elementi
rep(1:3, each = 2)
## [1] 1 1 2 2 3 3
```

Esercizi

Famigliarizza con la creazione di vettori (soluzioni):

1. Crea il vettore `x` contenente i numeri 4, 6, 12, 34, 8
2. Crea il vettore `y` contenente tutti i numeri pari compresi tra 1 e 25 (`?seq()`)
3. Crea il vettore `z` contenente tutti i primi 10 multipli di 7 partendo da 13 (`?seq()`)
4. Crea il vettore `s` in cui le lettere `"A"`,`"B"` e `"C"` vengono ripetute nel medesimo ordine 4 volte (`?rep()`)
5. Crea il vettore `t` in cui le letter `"A"`,`"B"` e `"C"` vengono ripetute ognuna 4 volte (`?rep()`)
6. Genera il seguente output in modo pigro, ovvero scrivendo meno codice possibile ;)

```
## [1] "foo" "foo" "bar" "bar" "foo" "foo" "bar" "bar"
```

7.2 Selezione Elementi

Una volta creato un vettore potrebbe essere necessario selezionare uno o più dei suoi elementi. In R per selezionare gli elementi di un vettore si utilizzano le **parentesi quadre** `[]` dopo il nome del vettore, indicando al loro interno **l'indice di posizione** degli elementi desiderati:

```
nome_vettore[<indice-posizione>]
```

Attenzione, non devo quindi indicare il valore dell'elemento desiderato ma il suo indice di posizione. Ad esempio:

```
# dato il vettore
my_numbers <- c(2,4,6,8)

# per selezionare il valore 4 utilizzo il suo indice di posizione ovvero 2
my_numbers[2]
## [1] 4

# Se utilizzassi il suo valore (ovvero 4)
# otterrei l'elemento che occupa la 4° posizione
my_numbers[4]
## [1] 8
```

Per selezionare più elementi è necessario indicare tra le parentesi quadre tutti gli indici di posizione degli elementi desiderati. Nota come non sia possibile fornire semplicemente i singoli indici numerici ma questi devono essere raccolti in un vettore, ad esempio usando la funzione `c()`. Praticamente usiamo un vetore di indici per selezionare gli elemeni desiderati dal nostro vettore iniziale.

```
# ERRATA selezione più valori
my_numbers[1,2,3]
## Error in my_numbers[1, 2, 3]: incorrect number of dimensions

# CORRETTA selezione più valori
my_numbers[c(1,2,3)]
## [1] 2 4 6
my_numbers[1:3]
## [1] 2 4 6
```

Tip-Box: Selezionare non è Modificare

Nota come l'operazione di selezione non modifichi l'oggetto iniziale. Pertanto è necessario salvare il risultato della selezione se si desidera mantenere le modifiche.

```r
my_words <- c("foo", "bar", "baz", "qux")

# Seleziono i primi 2 elementi
my_words[1:2]
## [1] "foo" "bar"

# Ho ancora tutti gli elementi nell'oggetto my_words
my_words
## [1] "foo" "bar" "baz" "qux"

# Salvo i risultati
my_words <- my_words[1:2]
my_words
## [1] "foo" "bar"
```

Warning-Box: Casi Estremi nella Selezione

Cosa accade se utiliziamo un indice di posizione maggiore del numero di elementi del nostro vettore?

```r
# Il mio vettore
my_numbers <- c(2,4,6,8)

my_numbers[10]
## [1] NA
```

R non restituisce un errore ma il valore `NA` ovvero *Not Available*, per indicare che nessun valore è disponibile.
Osserviamo infine anche altri comportamenti particolari o possibili errori nella selezione di elementi.

- L'indice di posizione deve essere un valore numerico e non un carattere.

```
# ERRATA selezione più valori
my_numbers["3"]
## [1] NA

# CORRETTA selezione più valori
my_numbers[3]
## [1] 6
```

- I numeri decimali vengono ignorati e non "arrotondati"

```
my_numbers[2.2]
## [1] 4
my_numbers[2.8]
## [1] 4
```

- Utilizzando il valore 0 ottengo un vettore vuoto

```
my_numbers[0]
## numeric(0)
```

7.2.1 Utilizzi Avanzati Selezione

Vediamo ora alcuni utilizzi avanzati della selezione di elementi di un vettore. In particolare impareremo a:

- utilizzare gli operatori relazionali e logici per selezionare gli elementi di un vettore
- modificare l'ordine degli elementi
- creare nuove combinazioni
- sostituire degli elementi
- eliminare degli elementi

Operatori Relazionali e Logici

Un'utile funzione è quella di selezionare tra gli elementi di un vetore quelli che rispetano una certa condizione. Per fare questo dobbiamo specificare all'interno delle parentesi quadre la proposizione di interesse utilizzando gli operatori relazionali e logici (vedi Capitolo 3.2).

Possiamo ad esempio selezionare da un vettore numerico tutti gli elementi maggiori di un certo valore, oppure selezionare da un vettore di caratteri tutti gli elementi uguali ad una data stringa.

```
# Vettore numerico - seleziono elemeni maggiori di 0
my_numbers <- -5:5
my_numbers[my_numbers >= 0]
## [1] 0 1 2 3 4 5

# Vettore caratteri - seleziono elemeni uguali a "bar"
my_words <- rep(c("foo", "bar"), times = 4)
my_words[my_words == "bar"]
## [1] "bar" "bar" "bar" "bar"
```

Per capire meglio questa operazione è importante notare come nello stesso comando ci siano in realtà due passaggi distinti:

- **Vettore logico** (vedi Capitolo 7.4.4) - quando un vettore è valutato in una proposizione, R restituisce un nuovo vettore che contiene per ogni elemento del vettore iniziale la risposta (`TRUE` o `FALSE`) alla nostra proposizione.
- **Selezione** - utilizziamo il vettore logico ottenuto per selezionare gli elementi dal vettore iniziale. Gli elementi associati al valore `TRUE` sono selezionati mentre quelli associati al valore `FALSE` sono scartati.

Rendiamo espliciti questi due passaggi nel seguente codice:

```
# Vettore logico
condition <- my_words == "bar"
condition
## [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE

# Selezione
my_words[condition]
## [1] "bar" "bar" "bar" "bar"
```

Ordinare gli Elementi

Gli indici di posizione possono essere utilizzati per ordinare gli elementi di un vettore a seconda delle necessità.

```
messy_vector <- c(5,1,7,3)

# Altero l'ordine degli elementi
messy_vector[c(4,2,3,1)]
## [1] 3 1 7 5

# Ordino gli elementi per valori crescenti
messy_vector[c(2,4,1,3)]
## [1] 1 3 5 7
```

Trick-Box: sort() vs order()

Per ordinare gli elementi di un vettore in ordine crescente o decrescente (sia alfabetico che numerico), è possibile utilizzare la funzione `sort()` specificando l'argomento `decreasing`. Vedi l'help page della funzione per ulteriori informazioni (`?sort()`).

```
# Ordine alfabetico
my_letters <- c("cb", "bc", "ab", "ba", "cb", "ab")
sort(my_letters)
## [1] "ab" "ab" "ba" "bc" "cb" "cb"

# ordine decrescente
sort(messy_vector, decreasing = TRUE)
## [1] 7 5 3 1
```

Nota come esista anche la funzione `order()` ma questa sia un false-friend perchè non ci fornisce direttamente un vettore con gli elementi ordinati ma bensì gli indici di posizione per riordinare gli elementi (`?order()`). Vediamo nel seguente esempio come utilizzare questa funzione:

```
# Indici di posizione per riordinare gli elementi
order(messy_vector)
## [1] 2 4 1 3
# Riordino il vettore usando gli indici di posizione
messy_vector[order(messy_vector)]
## [1] 1 3 5 7
```

Combinazioni di Elementi

Gli stessi indici di posizione possono essere richiamati più volte per ripetere gli elementi nelle combinazioni desiderate formando un nuovo vettore.

```
my_numbers <- c(5,6,7,8)
# Ottengo un nuovo vettore con la combinazioen desiderata
my_numbers[c(1,2,2,3,3,3,4)]
## [1] 5 6 6 7 7 7 8
```

Modificare gli Elementi

Un importante utilizzo degli indici riguarda la modifica di un elemento di un vettore. Per sostituire un vecchio valore con un nuovo valore, posso utilizzare la funzione *assign* (`<-` o `=`) come nell'esempio:

```r
my_names <- c("Andrea", "Bianca", "Carlo")

# Modifico il  nome "Bianca" in "Beatrice"
my_names[2] <- "Beatrice"
my_names
## [1] "Andrea"   "Beatrice" "Carlo"
```

Per sostituire il valore viene indicato alla sinistra dell'operatore *assign* il valore che si vuole modificare e alla destra il nuovo valore. Nota come questa operazione possa essere usata per aggiungere anche nuovi elementi al vettore.

```r
my_names[4]
## [1] NA

# Aggiungo il nome "Daniela"
my_names[4] <- "Daniela"
my_names
## [1] "Andrea"   "Beatrice" "Carlo"    "Daniela"
```

Eliminare gli Elementi

Per **eliminare degli elementi** da un vettore, si indicano all'interno delle parentesi quadre gli indici di posizione degli elementi da eliminare preceduti dall'operatore `-` (*meno*). Nel caso di più elementi è anche possibile indicare il meno solo prima del comando `c()`, ad esempio il comando `x[c(-2,-4)]` diviene `x[-c(2,4)]`.

```r
my_words <- c("foo", "bar", "baz", "qux")

# Elimino "bar"
my_words[-2]
## [1] "foo" "baz" "qux"

# Elimino "foo" e "baz"
my_words[-c(1,3)]    # oppure my_words[c(-1, -3)]
## [1] "bar" "qux"
```

Nota come l'operazione di eliminazione sia comunque un'operazione di selezione. Pertanto è necessario salvare il risultato ottenuto se si desidera mantenere le modifiche.

```r
# Elimino "foo" e "baz"
my_words[-c(1,3)]
## [1] "bar" "qux"

# Ho ancora tutti gli elementi nell'oggetto my_words
my_words
## [1] "foo" "bar" "baz" "qux"

# Salvo i risultati
my_words <- my_words[-c(1,3)]
```

```
my_words
## [1] "bar" "qux"
```

7.2.1.1 which()

La funzione `which()` è molto utile per ottenere la **posizione** all'interno di un vettore associata ad una certa condizione logica. In altri termini se vogliamo sapere in quale posizione sono i valori `> 5` in un certo vettore numerico possiamo usare la funzione `which(x > 5)` dove `x` è chiaramente il nostro vettore numerico.

```
##  [1]  TRUE  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE
## [1]  1  2  4  7 10
```

Come vedete, la funzione `which()` essenzialmente restituisce la **posizione** (e non il valore) dove la condizione testata è `TRUE`. E' importante notare che queste due scritture sono equivalenti:

```
## [1] 8.331936 9.200411 5.541441 9.894681 7.138463
## [1] 8.331936 9.200411 5.541441 9.894681 7.138463
```

Infatti, come abbiamo visto possiamo indicizzare un vettore sia con un'altro vettore che ci indica la posizione degli elementi da estrarre che un vettore **logico** di lunghezza uguale al vettore originale.

Esercizi

Esegui i seguenti esercizi (soluzioni):

1. Del vettore `x` seleziona il 2°, 3° e 5° elemento
2. Del vettore `x` seleziona i valori 34 e 4
3. Dato il vettore `my_vector = c(2,4,6,8)` commenta il risultato del comando `my_vector[my_vector]`
4. Del vettore `y` seleziona tutti i valori minori di 13 o maggiori di 19
5. Del vettore `z` seleziona tutti i valori compresi tra 24 e 50
6. Del vettore `s` seleziona tutti gli elementi uguali ad `"A"`
7. Del vettore `t` seleziona tutti gli elementi diversi da `"B"`
8. Crea un nuovo vettore `u` identico a `s` ma dove le `"A"` sono sostituite con la lettera `"U"`
9. Elimina dal vettore `z` i valori 28 e 42

7.3 Funzioni ed Operazioni

Vediamo ora alcune utili funzioni e comuni operazioni che è possibile svolgere con i vettori (vedi Tabella 7.1).

Nota che l'esecuzione di operazioni matematiche (e.g., `+`, `-`, `*`, `/` etc.) è possibile sia rispetto ad un singolo valore sia rispetto ad un altro vettore:

- **Singolo valore** - l'operazione sarà svolta per ogni elemento del vettore rispetto al singolo valore fornito.

Table 7.1: Funzioni e operazioni con vettori

Funzione	Descrizione
`nuovo_vettore <- c(vettore1, vettore2)`	Unire più vettori in un unico vettore
`length(nome_vettore)`	Valutare il numero di elementi contenuti in un vettore
`vettore1 + vettore2`	Somma di due vettori
`vettore1 - vettore2`	Differenza tra due vettori
`vettore1 * vettore2`	Prodotto tra due vettori
`vettore1 / vettore2`	Rapporto tra due vettori

- **Altro vettore** - l'operazione sarà svolta per ogni coppia di elementi dei due vettori. E' quindi necessario che i due vettori abbiano la **stessa lunghezza**, ovvero lo stesso numero di elementi.

```
x <- 1:5
y <- 1:5

# Sommo un valore singolo
x + 10
## [1] 11 12 13 14 15

# Somma di vettori (elemento per elemento)
x + y
## [1]  2  4  6  8 10
```

Warning-Box: Vettori di Diversa Lunghezza

Qualora i vettori differiscano per la loro lunghezza, R ci presenterà un warning avvisandoci del problema ma eseguirà comunque l'operazione utilizzando più volte il vettore più corto.

```
x + c(1, 2)
## Warning in x + c(1, 2): longer object length is not a multiple of shorter ob
## length
## [1] 2 4 4 6 6
```

Tuttavia, compiere operazioni tra vettori di diversa lunghezza (anche se multipli) dovrebbe essere evitato poichè è facile causa di errori ed incomprensioni.

Approfondimento: Vectorized Operations

In R la maggior parte degli operatori sono *vettorizzati*, ovvero calcolano diret-

tamente il risultato per ogni elemento di un vettore. Questo è un grandissimo vantaggio poichè ci permette di essere molto efficienti e coincisi nel codice. Senza vettorizzazione, ogni operazione tra due vettori richiederebbe di specificare l'operazione per ogni elemento del vettore. Nel precedente esempio della somma tra `x` e `y` avremmo dovuto usare il seguente codice:

```
z <- numeric(length(x))
for(i in seq_along(x)) {
      z[i] <- x[i] + y[i]
}
z
## [1]  2  4  6  8 10
```

Nota come questo sia valido anche per gli **operatori relazionali e logici**. Infatti valutando una proposizione rispetto ad un vettore, otterremmo una risposta per ogni elemento del vettore

```
my_values <- 1:8

# Valori compresi tra 4 e 7
my_values >= 4 & my_values <= 7
## [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE
```

Esercizi

Esegui i seguenti esercizi (soluzioni):

1. Crea il vettore `j` unendo i vettori `x` ed `z`.
2. Elimina gli ultimi tre elementi del vettore `j` e controlla che i vettori `j` e `y` abbiano la stessa lunghezza.
3. Calcola la somma tra i vettori `j` e `y`.
4. Moltiplica il vettore z per una costante `k=3`.
5. Calcola il prodotto tra i primi 10 elementi del vettore `y` ed il vettore `z`.

7.4 Data Type

Abbiamo visto come sia necessario che in un vettore tutti gli elementi siano della stessa tipologia. Avremo quindi diversi tipi di vettori a seconda della tiopologia di dati che contengono.

In R abbiamo 4 principali tipologie di dati, ovvero tipologie di valori che possono essere utilizzati:

- `character` - *Stringhe di caratteri* i cui valori alfannumerici vengono delimitati dalle doppie vigolette `"Hello world!"` o virgolette singole `'Hello world!'`.
- `double` - *Valori reali* con o senza cifre decimali ad esempio `27` o `93.46`.
- `integer` - *Valori interi* definiti apponendo la lettera `L` al numero desiderato, ad esempio `58L`.
- `logical` - *Valori logici* `TRUE` e `FALSE` usati nelle operazioni logiche.

Possiamo verificare la tipologia di un valore utlizzando la funzione `typeof()`.

```
typeof("foo")
## [1] "character"

typeof(2021)
## [1] "double"

typeof(2021L) # nota la lettera L
## [1] "integer"

typeof(TRUE)
## [1] "logical"
```

Esistono molte altre tipologie di dati tra cui `complex` (per rappresentare i numeri complessi del tipo $x + yi$) e `Raw` (usati per rappresenttare i valori come bytes) che però riguardano usi poco comuni o comunque molto avanzati di R e pertanto non verranno trattati.

Approfondimento: Tutta una Questione di Bit

Questa distinzione tra le varie tipologie di dati deriva dalla modalità con cui il computer rappresenta internamente i diversi valori. Sappiamo infatti che il computer non possiede caratteri ma solamente bits, ovvero successioni di 0 e 1 ad esempio 01000011.
Senza scendere nel dettaglio, per ottimizzare l'uso della memoria i diversi valori vengono *"mappati"* utilizzando i bits in modo differente a seconda delle tipologie di dati. Pertanto in R il valore `24` sarà rappresentato diversamente a seconda che sia definitio come una stringa di caratteri (`"24"`), un numero intero (`24L`) o un numero double (`24`).

Integer vs Double

In particolare un aspetto poco intuitivo riguarda la differenza tra valori `double` e `integer`. Mentre i valori interi possono essere rappresentati con precisione dal computer, non tutti i valori reali posssono essere rappresentati esattamente utilizzando il numero massimo di 64 bit. In questi casi i loro valori vengonno quindi approsimati e, sebbene questo venga fatto con molta precisione, a volte potrebbe portare a dei risultati inaspettati. Nota infatti come nell'esempio seguente non otteniamo zero, ma osserviamo un piccolo errore dovuto all'approsimazione dei valori `double`.

```r
my_value <- sqrt(2)^2 # dovrei ottenere 2
my_value - 2          # dovrei ottenre 0
## [1] 4.440892e-16
```

E' importante tenere a mente questo problema nei test di ugualianza dove l'utilizzo dell'operatore `==` potrebbe generare delle risposte inaspettate. In genere viene quindi preferita la funzione `all.equal()` che prevede un certo margine di tolleranza (vedi `?all.equal()` per ulteriori dettagli).

```r
my_value == 2          # Problema di approsimazione
## [1] FALSE
all.equal(my_value, 2) # Test con tolleranza
## [1] TRUE
```

Ricorda infine che i computer hanno un limite rispetto al massimo valore e minimo valore che possono rappresentare sia per quanto riguarda i valori interi che i valori reali. Per approfondire vedi https://stat.ethz.ch/pipermail/r-help/2012-January/300250.html.

Vediamo ora i diversi tipi di vettori a seconda della tipologia di dati utilizzati.

7.4.1 Character

I vettori formati da stringhe di caratteri sono definiti vettori di caratteri. Per valutare la tipologia di un oggetto possiamo utilizzare la funzione `class()`, mentre ricordiamo che la funzionne `typeof()` valuta la tipologia di dati. In questo caso otteniamo per entrambi il valore `character`.

```r
my_words<-c("Foo","Bar","foo","bar")

class(my_words) # tipologia oggetto
## [1] "character"

typeof(my_words) # tipologia dati
## [1] "character"
```

Non è possibile eseguire operazioni aritmetiche con vettori di caratteri ma solo valutare relazioni di uguaglianza o disuguaglianza ripetto ad un'altra stringa.

```r
my_words + "foo"
## Error in my_words + "foo": non-numeric argument to binary operator

my_words == "foo"
## [1] FALSE FALSE  TRUE FALSE
```

7.4.2 Numeric

In R, se non altrimenti specificato, ogni valore numerico viene rappresentato come un `double` indipendentemente che abbia o meno valori decimali. I vettori formati da valori double sono definiti vettori numerici. In R la tipologia del vettore è indicata con `numeric` mentre i dati sono `double`.

```
my_values <- c(1,2,3,4,5)
class(my_values)  # tipologia oggetto
## [1] "numeric"

typeof(my_values) # tipologia dati
## [1] "double"
```

I vettori numerici sono utilizzati per compiere qualsiasi tipo di operazioni matematiche o logico-relazionali.

```
my_values + 10
## [1] 11 12 13 14 15

my_values <= 3
## [1]  TRUE  TRUE  TRUE FALSE FALSE
```

7.4.3 Integer

In R per specificare che un valore è un numero intero viene aggiunta la lettera `L` immediatamente dopo il numero. I vettori formati da valori interi sono definiti vettori di valori interi. In R la tipologia del vettore è indicata con `integer` allo stesso modo dei dati.

```
my_integers <- c(1L,2L,3L,4L,5L)
class(my_integers)  # tipologia oggetto
## [1] "integer"

typeof(my_integers) # tipologia dati
## [1] "integer"
```

Come per i vettori numerici, i vettori di valori interi possono esssere utilizzati per compiere qualsiasi tipo di operazioni matematiche o logico-relazionali. Nota tuttavia che operazioni tra integer e doubles restituiranno dei doubles ed anche nel caso di operazioni tra integers il risultato potrebbe non essere un integer.

```
is.integer(5L * 5)   # integer e double
## [1] FALSE

is.integer(5L * 5L)  # integer e integer
## [1] TRUE

is.integer(5L / 5L)  # integer e integer
## [1] FALSE
```

7.4.4 Logical

I vettori formati da valori logici (`TRUE` e `FALSE`) sono definiti vettori logici. In R la tipologia del vettore è indicata con `logical` allo stesso modo dei dati.

```
my_logical <- c(TRUE, FALSE, TRUE)
class(my_logical)  # tipologia oggetto
## [1] "logical"

typeof(my_logical) # tipologia dati
## [1] "logical"
```

I vettori di valori logici possono esssere utilizzati con gli operatori logici.

```
my_logical & c(FALSE, TRUE, TRUE)
## [1] FALSE FALSE  TRUE

my_logical & c(0, 1, 3)
## [1] FALSE FALSE  TRUE
```

Tuttavia ricordiamo che ai valori `TRUE` e `FALSE` sono associati rispettivamente i valori numerici 1 e 0 (o più precisamente i valori interi `1L` e `0L`). Pertanto è possibile eseguire anche operazioni matematiche dove verrano automaticamente considerati i rispettivi valori numerici. Ovviamente il risultato ottenuto sarà un valore numerico e non logico.

```
TRUE * 10
## [1] 10

FALSE * 10
## [1] 0
```

Trick-Box: sum() e mean()

Utilizzando le funzioni `sum()` e `mean()` con un vettore logico, possiamo valutare rispettivamente il numero totale e la percentuale di elementi che hanno soddisfatto una certa condizione logica.

```
my_values <- rnorm(50)  # genero dei numeri casuali

sum(my_values > 0)      # totale numeri positivi
## [1] 29

mean(my_values > 0)      # percentuale numeri positivi
## [1] 0.58
```

Approfondimento: is.* adn as.* Function Families

Esistono due famiglie di funzioni che permettono rispettivamene di testare e di modificare la tipologia dei dati.

is.* Family

Per testare se un certo valore (o un vettore di valori) appartiene ad una specifica tipologia di dati, possimao utilizzare una tra le seguenti funzioni:

- `is.vector()` - valuta se un oggetto è un generico vettore di qualsiasi tipo

```
is.vector("2021")    # TRUE
is.vector(2021)      # TRUE
is.vector(2021L)     # TRUE
is.vector(TRUE)      # TRUE
```

- `is.character()` - valuta se l'oggetto è una stringa

```
is.character("2021") # TRUE
is.character(2021)   # FALSE
is.character(2021L)  # FALSE
is.character(TRUE)   # FALSE
```

- `is.numeric()` - valuta se l'oggetto è un valore numerico indipendentemente che sia un double o un integer

```
is.numeric("2021")   # FALSE
is.numeric(2021)     # TRUE
is.numeric(2021L)    # TRUE
is.numeric(TRUE)     # FALSE
```

- `is.double()` - valuta se l'oggetto è un valore `double`

```
is.double("2021")    # FALSE
is.double(2021)      # TRUE
is.double(2021L)     # FALSE
is.double(TRUE)      # FALSE
```

- `is.integer()` - valuta se l'oggetto è un valore intero

```
is.integer("2021")   # FALSE
is.integer(2021)     # FALSE
is.integer(2021L)    # TRUE
is.integer(TRUE)     # FALSE
```

- `is.logical()` - valuta se l'oggetto è un valore logico

```
is.logical("2021")   # FALSE
is.logical(2021)     # FALSE
is.logical(2021L)    # FALSE
is.logical(TRUE)     # TRUE
```

as.* Family

Per modificare la tipologia di un certo valore (o un vettore di valori), possimao utilizzare una tra le seguenti funzioni:

- `as.character()` - trasforma l'oggetto in una stringa

```
as.character(2021)
## [1] "2021"
as.character(2021L)
## [1] "2021"
as.character(TRUE)
## [1] "TRUE"
```

- `as.numeric()` - trasforma l'oggetto in un `double`

```
as.numeric("foo")  # Non valido con stringhe di caratteri
## Warning: NAs introduced by coercion
## [1] NA
as.numeric("2021") # Valido per stinghe di cifre
## [1] 2021
as.numeric(2021L)
## [1] 2021
as.numeric(TRUE)
## [1] 1
```

- `as.double()` - trasforma l'oggetto in un `double`

```
as.double("2021") # Valido per stinghe di cifre
## [1] 2021
as.double(2021L)
## [1] 2021
as.double(TRUE)
## [1] 1
```

- `as.integer()` - trasforma l'oggetto in un `integer`

```
as.integer("2021") # Valido per stinghe di cifre
## [1] 2021
as.integer(2021.6) # Tronca la parte decimale
## [1] 2021
as.integer(TRUE)
## [1] 1
```

- `as.logical()` - trasforma un oggetto numerico in un valore logico qualsiasi valore diverso da 0 viene considerato `TRUE`

```
as.logical("2021") # Non valido per le stringhe
## [1] NA
as.logical(0)
## [1] FALSE
as.logical(0.5)
## [1] TRUE
as.logical(2021L)
## [1] TRUE
```

7.4.5 Valori speciali

Vediamo infine alcuni valori speciali utilizzatti in R con dei particolari significati e che richiedono specifici accorgimenti quando vengono manipolati:

- `NULL`- rappresenta l'oggetto nullo, ovvero l'assenza di un oggetto. Spesso viene restituito dalle funzioni quano il loro output è indefinito.
- `NA` - rappresenta un dato mancate (*Not Available*). E' un valore costante di lunghezza 1 che può essere utlizzato per qualsiasi tipologia di dati.

- `NaN` - indica un risulato matematico che non può essere rappresentato come un valore numerico (*Not A Number*). E' un valore costante di lunghezza 1 che può essere utlizzato come valore numerico (non intero).

```
0/0
## [1] NaN
sqrt(-1)
## Warning in sqrt(-1): NaNs produced
## [1] NaN
```

- `Inf` (o `-Inf`) - indica un risultato matematico inifinito (o infinito negativo). E' anche utilizzato per rappresentare numeri estremamente grandi.

```
pi^650
## [1] Inf

-pi/0
## [1] -Inf
```

E' importante essere consapevoli delle caratteristiche di questi valori poichè presentano dei comportamenti peculiari che, se non corretamente gestiti, possono generare conseguenti errori nei codici. Descriviamo ora alcune delle carateristiche più importanti.

Lunghezza Elementi

Notiamo innanzitutto come mentre `NULL` sia effettivamente un oggetto nullo, ovvero privo di dimensione, `NA` sia uno speciale valore che rappresenta la presenza di un dato mancate. Pertanto `NA`, a differenza di `NULL`, è effettivamente un valore di lunghezza 1.

```
# Il valore NULL è un oggetto nullo
values_NULL <- c(1:5, NULL)
length(values_NULL)
## [1] 5
values_NULL # NULL non è presente
## [1] 1 2 3 4 5

# Il valore NA è un oggetto che testimonia un'assenza
values_NA <- c(1:5, NA)
length(values_NA)
## [1] 6
values_NA   # NA è presente
## [1]  1  2  3  4  5 NA
```

Allo stesso modo, anche i valori `NaN` e `Inf` sono effettivamente dei valori di lunghezza 1 usati per testimoniare speciali risultati numerici.

```
length(c(1:5, NaN))
## [1] 6
length(c(1:5, Inf))
## [1] 6
```

Propagazione Valori

Altra importante caratteristica è quella che viene definita *propagazione* dei valori ovvero le operazioni che includono questi speciali valori resituiscono a loro volta lo stesso speciale. Ciò significa che questi valori si propagheranno di risultato in risultato all'interno del nostro codice se non opportunamente gestiti.

- `NULL`- osserviamo come se il valore `NULL` viene utilizzato in una qualsiasi operazione matematica il risultato sarà un vettore numerico vuoto di dimensione 0, il quale può essere interpretato in modo simile (seppur non identico) al valore `NULL`

```
res_NULL <- NULL * 3
length(res_NULL)
## [1] 0
res_NULL
## numeric(0)
```

- `NA` - quando `NA` viene utilizzato in una qualsiasi operazione matematica il risultato sarà nuovamene un `NA`.

```
NA * 3
## [1] NA
```

- `NaN` - quando `NaN` viene utilizzato in una qualsiasi operazione matematica il risultato sarà nuovamene un `NaN`.

```
NaN * 3
## [1] NaN
```

- `Inf` (o `-Inf`) - qualora `Inf` (o `-Inf`) siano utilizzati in un'operazione matematica il risultato seguirà le comuni regole delle operazioni tra infiti.

```
Inf - 3     # Inf

Inf * -3    # - Inf

Inf + Inf   # Inf

Inf + -Inf  # NaN

Inf * -Inf  # -Inf

Inf / Inf   # NaN
```

Testare Valori

E' importante ricordare come per testare l apresenza di uno di questi valori speciali siano presenti delle funzioni specifiche della famiglia `is.*`. Non deve mai essere utlizzato il comune operatore di uguaglianza `==` poichè non fornisce i risultai corretti.

- is.null

```
NULL == NULL      # logical(0)
is.null(NULL)     # TRUE
```

- is.na

```
NA == NA     # NA
is.na(NA)    # TRUE
```

- is.nan

```
NaN == NaN   # NA
is.nan(NaN) # TRUE
```

- Inf

```
Inf == Inf        # TRUE considero anche il segno
is.infinite(Inf) # TRUE sia per Inf che -Inf
```

Operatori Logici

Un particolare comportamento riguarda i risultati ottenute con gli operatori logici dove la *propagazione* del valore non segue sempre le attese. Osserviamo i diversi casi:

- `NULL`- ottenimo come da attese un vettore logico vuoto di dimensione 0

```
TRUE & NULL  # logical(0)
TRUE | NULL  # logical(0)

FALSE & NULL # logical(0)
FALSE | NULL # logical(0)
```

- `NA` - non otteniamo come da attese sempre il valore `NA` ma in alcune condizioni la proposizione sara `TRUE` o `FALSE`

```
TRUE & NA  # NA
TRUE | NA  # TRUE

FALSE & NA # FALSE
FALSE | NA # NA
```

- `NaN` - otteniamo gli stessi risultati del caso precedente utilizzando il valore `NA`

```
TRUE & NaN  # NA
TRUE | NaN  # TRUE

FALSE & NaN # FALSE
FALSE | NaN # NA
```

- `Inf` - essendo un valore numerico diverso da zero otteniamo i rissultati secondo le attese

```
TRUE & Inf  # TRUE
TRUE | Inf  # TRUE

FALSE & Inf # FALSE
FALSE | Inf # TRUE
```

Tip-Box: A Logical Solution

Un comportamento tanto strano per quanto riguarda l'utilizzo del valore `NA` con gli operatori logici può essere spiegato dal fatto che il valore `NA` in realtà sia un valore logico che indica la mancanza di una risposta.

```
is.logical(NA)
## [1] TRUE
```

Pertanto le proposizioni vengono correttamente seguendo le comuni regole. Nel caso di `TRUE | NA` la proposizione è giudicata `TRUE` perchè con l'operatore di disgiunzione è sufficiente che una delle due parti sia vera avvinchè la proposizione sia vera. Nel caso di `FALSE & NA`, invece, la proposizione è giudicata `FALSE` perchè con l'operatore di congiunzione è sufficiente che una delle due parti sia falsa avvinchè la proposizione sia falsa. LA non risposta indicata da `NA` i questi casi è ininfluente, mentre determina il risultato nei restanti casi quando la seconda parte della proposizione deve essere necessariamente valutata. A questo punto gli operatori restituiscono `NA` poichè incapaci di determinare la risposta.
Per quanto riguarda il caso del valore `NaN` è sufficinete ricordare che tale valore sia comunque un valore numerico di cui però non è possibile identificare il valore.

```
is.numeric(NaN)
## [1] TRUE
```

Tutti i valori numerici sono considerati validi nelle operazioni logiche, dove qualsiais numero diveros da zero è valutato `TRUE`. Pertanto viene seguito lo stesso ragionamento precedente, quando non è necessario valutare entrambe

le parti della proposizione viene fornita una risposta, mentre si ottiene `NA` negli altri casi quando R è obbligato a valutare entrambe la parti ma è incapace di fornire una risspossta poichè non può determinare il valore di `NaN`.

Approfondimento: L'importanza dei Dati Mancanti

Lavorare in presenza di dati mancanti accadrà nella maggior parte dei casi. Molte delle funzioni presenti in R hanno già delle opzioni per rimuovere automaticamete eventuali dati mancanti così da poter ottenre correttamete i risultati.

```
my_sample <- c(2,4,6,8, NA)
mean(my_sample)
## [1] NA
mean(my_sample, na.rm = TRUE)
## [1] 5
```

Tuttavia, è importante non avvalersi in modo automatico di tali opzioni ma avere cura di valutare attentamente la presenza di dati mancanti. Questo ci permetterà di indagare possibili pattern riguardanti i dati mancanti e valutare la loro possibile influenza sui nostri risultati e la validità elle conclusioni.
Inoltre sarà fondamentale controllare sempre l'effittiva dimensione del campione utilizzato nelle vari analisi. Ad esempio se non valutato attentamente potremmo non ottenere il numero effettivo di valori su cui è stata calcolata precedentemente la media.

```
length(my_sample)    # NA incluso
## [1] 5
length(my_sample[!is.na(my_sample)]) # NA escluso
## [1] 4
```

Chapter 8

Fattori

In questo capitolo vedremo i *fattori*, una speciale tipologia di vettori utilizzata per salvare informazioni riguardanti una variabile categoriale (nominale o ordinale). Tuttavia, prima di introdurre i fattori, descriveremo che cosa sono gli attributi di un oggetto. Questi ci permetteranno successivamente di capire meglio il funzionamento dei fattori.

8.1 Attributi di un Oggetto

In R, gli oggetti possiedono quelli che sono definiti *attributi*, ovvero delle utili informazioni riguardanti l'oggetto stesso, una sorta di *metadata*. Queste informazioni non interferiscono con i valori contenuti negli oggetti nè vengono normalmente mostrati nell'output di un oggetto. Tuttavia, si rivelano particolarmente utili in alcune circostanze e permettono di fornire speciali informazioni associate ad un determinato oggetto.

Gli oggetti possiedono diversi attributi a seconda della loro tipologia. Tuttavia, tra quelli principalmente usati troviamo:

- **Classe** (`class`) - la classe (o tipologia) di un oggetto. Ci permette di verificare la tipologia di struttura dati di un particolare oggetto.
- **Nomi** (`names`) - nomi degli elementi di un oggetto. Permette ad esempio di assegnare dei nomi agli elementi un vettore o alle righe e colonne di una matrice o dataframe.
- **Dimensione** (`dim`) - la dimensione dell'oggetto. Questo attributo non è disponibile per i vettori ma sarà particolarmente importante nel caso delle matrici e dataframe.

Per valutare ee eventualmente modifiacare gli attributi di un oggetto esistono delle specifiche funzioni dedicate. Ad esempio abbiamo:

- `attributes()` - elenca tutti gli attributi di un oggetto
- `class()` - accede all'attributo `class` di un oggetto
- `names()` - accede all'attributo `names` di un oggetto
- `dim()` - accede all'attributo `dim` di un oggetto

Vediamo ora alcuni utilizzi degli attributi con i vettori. Gli attributi nel caso delle altre tipologie di oggetti, invece, saranno trattati nei rispettivi capitoli.

8.1.1 Attributi di un Vettore

Vediamo come inizialmente un generico vettore non possiede alcun attributo vettore.

```
my_vector <- 1:10

attributes(my_vector)
## NULL
```

Classe

Eseguendo la funzione `class()`, tuttavia, otteniamo comunque la precisa tipologia di vettore, nel presente caso `"integer"`.

```
class(my_vector)
## [1] "integer"
```

Dimensione

Abbiamo anticipato come l'attributo `dim` non sia disponibile per i vettori, mentre diverrà molo importante nel caso di matrici e dataframe. Tuttavia un analogo valore della dimensione di un vettore è dato dalla sua lunghezza, valutata con la funzione `length()`.

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

Nomi Elementi

Inizialmente gli elementi di un vettore non possiedono nomi.

```
names(my_vector)
## NULL
```

Per impostare i nomi degli elementi, sarà quindi necessario assegnare a `names(nome_vettore)` un vettore di caratteri, contenente i nomi desiderati, della stessa lunghezza del vettore che stiamo rinominando.

```
names(my_vector) <- paste0("Item_", 1:10)
my_vector
##  Item_1  Item_2  Item_3  Item_4  Item_5  Item_6  Item_7  Item_8  Item_9 Item_10
##       1       2       3       4       5       6       7       8       9      10
```

Questa procedura ci pemette di ottenere quello che viene definito un *named vector*. Possiamo vedere come i nomi degli elementi compaiano ora tra gli attributi dell'oggetto.

```
## $names
##  [1] "Item_1"  "Item_2"  "Item_3"  "Item_4"  "Item_5"  "Item_6"  "Item_7"
##  [8] "Item_8"  "Item_9"  "Item_10"
```

Trick-Box: Selezione Named Vectors

Una particolare utlizzo dei named vectors riguarda la selezione dei valori tramite i nomi degli elementi. Nota che per un corretto funzionameto è necessario che tutti gli elementi possiedano nomi differenti.

```
my_vector[c("Item_3", "Item_5")]
## Item_3 Item_5
##      3      5
```

Nel caso dei vettori questo approccio è raramente utilizzato mentre vedremo che sarà molto comune per la selezione delle variabili di un dataframe.

8.2 Fattori

I fattori sono quindi una speciale tipologia di vettori che, attraverso l'uso degli attributi, permettono di salvare in modo efficiente le variabili categoriali (nominali o ordinali). Il comando usato per creare un fattore in R è `factor()` e contiene diversi argomenti:

```
nome_fattore <- factor(x, levels = , ordered = FALSE)
```

- `x` - i dati della nostra variabile categoriale
- `levels`- i possibili livelli della nostra variabile categoriale
- `ordered` - valore logico che indica se si tratta di una variabile nominale (`FALSE`) o ordinale (`TRUE`)

Ad esempio potremmo creare la variabile `colore_occhi` in cui registrare il colore degli occhi dei membri di una classe:

```
# Creo i dati
my_values <- rep(c("verde", "marrone", "azzurro"), times = 3)
my_values
## [1] "verde"   "marrone" "azzurro" "verde"   "marrone" "azzurro" "verde"
## [8] "marrone" "azzurro"

# Creo il fattore
my_factor <- factor(my_values)
my_factor
## [1] verde   marrone azzurro verde   marrone azzurro verde   marrone azzurro
## Levels: azzurro marrone verde
```

Nota come non sia necessario specificare l'argomento `levels`. I livelli della variabile, infatti, vengono determinati automaticamente a partire dai dati presenti e ordinati in ordine alfabetico. Tuttavia, specificare i livelli nella creazione di un fattore ci permette di definire a piacere l'ordine dei livelli ed anche includere eventuali livelli non presenti nei dati.

8.2.1 Funzionamento Fatori

Cerchiamo ora di capire meglio la struttura dei fattori e l'utilizzo degli attributi.

```
attributes(my_factor)
## $levels
## [1] "azzurro" "marrone" "verde"
##
## $class
## [1] "factor"
```

Vediamo come la classe dell'oggetto sia `factor` e abbiamo aunceh un ulteriore attributo `levels` dove sono salvati i posibili livelli della nostra variabile. Ma attenzione adesso a cosa otteniamo quando valutiamo la tipologia di dati contenuti nel fattore e la sua sruttura.

```
# Tipologia dati
typeof(my_factor)
## [1] "integer"

# Sstrutura
str(my_factor)
##  Factor w/ 3 levels "azzurro","marrone",..: 3 2 1 3 2 1 3 2 1
```

Ci saremmo aspettati di ottenere `character` pensando che all'interno del fattore fossero salvati vari valori della nostra variabile come stringhe. Invece il fattore è formato da `integer` e possimao osservare come effetivamente l'output del comando `str()` riporta dei valori numerici (oltre ai livelli della variabile). Come spiegarsi tutto questo?

La soluzione è molto semplice. Nel creare un fattore, R valuta i livelli presennti creando l'attributo `levels` e poi sostituisce ad ogni elemento un valore numerico che indica il livello della variabile. Pertanto nel nostro esempio avremmo che il valore `1` è associato al livello `"azzurro"`, il valore `2` a `"marrone"` e il valore `3` a `"verde"`. Questo approccio permette di ottimizzare l'uso della memoria tuttavia inizialmete potrebbe risultare poco intuitivo e causa di errori.

> 🌵 Warning-Box: Attenti alla Conversione
>
> Uno dei principali errori riguarda la conversione da un fattore ad un normale vettore. Nel caso volessimo ottenere un vettore di caratteri possimo usare la funzione `as.character()` ottenendo il risultato voluto.
>
> ```
> as.character(my_factor)
> ## [1] "verde"   "marrone" "azzurro" "verde"   "marrone" "azzurro" "verde"
> ## [8] "marrone" "azzurro"
> ```
>
> Tuttavia, se volessimo ottenere un vettore numerico, dobbiamo prestare particolare attenzione. Considera il seguente esempio dove abbiamo gli anni di istruzione di alcuni partecipanti ad uno studio. Potremmo eseguire alcune

analissi considerando questa variable come categoriale per poi ritrasformarla in una variebile numerica per compiere altre analisi. Osserva cosa succede

```
# Creo la mia variabile come fattore
school_years<-factor(c(11, 8, 4, 8, 11, 4, 11, 8))
school_years
## [1] 11 8  4  8  11 4  11 8
## Levels: 4 8 11

# Trasformo in un vettore numerico
as.numeric(school_years)
## [1] 3 2 1 2 3 1 3 2
```

In modo forse inaspettato non otteniamo i valori originali (i.e., 4, 8, 11) ma dei valori differenti. Questi in realtà sono i valori numerici che R ha usato per associare ogni elemento al corripondente livello. Per ottenre i valori corretti dobbiamo eseguire il seguente comando:

```
as.numeric(as.character(school_years))
## [1] 11  8  4  8 11  4 11  8
```

Questo ci permette di sostituire prime i valori con le giuste *etichette*, ovvero i livelli della variabile, e successivamente convertire il tutto in un vettore numerico.
E' importante prestare molta attenzione in questi casi poichè un eventuale errore potrebbe non risultare subito evidente.

8.2.2 Operazioni Fattori

Ora che abbiamo capito il funzionamento dei fattori vediamo alcune comuni operazioni.

Rinominare i Livelli

E possibile rinominare i livelli di un fattore utilizzando la funzione `levels()` quesa ci permette di accedere agli attuali livelli ed eventualmente ssostituirli.

```
my_factor
## [1] verde   marrone azzurro verde   marrone azzurro verde   marrone azzurro
## Levels: azzurro marrone verde
# Attuli livelli
levels(my_factor)
## [1] "azzurro" "marrone" "verde"
```

```r
# Rinomino i livelli
levels(my_factor) <- c("brown", "blue", "green")
my_factor
## [1] green blue  brown green blue  brown green blue  brown
## Levels: brown blue green
```

Ordinare i Livelli

E' importante non confondere l'ordinare i livelli con il rinominarli. Infatti, mentre nel primo caso viene solo modificato l'ordine dei livelli, nel secondo caso verrebbero modificati anche tutti gli effettivi valori degli elementi. In genere è preferibile quindi ridefinire il fattore specificando l'argomento `levels`.

Vediamo un esempio dove raccogliamo i dati riguardanti i dosaggi di un farmaco:

```r
dosage <- factor(rep(c("medium", "low", "high"), times = 2))
dosage
## [1] medium low    high   medium low    high
## Levels: high low medium
```

Non avendo specificato l'attributo `levels` i livelli siano stati definiti automaticamente in ordine alfabetico. Osserviamo cosa succede se per erroe rinominiamo i livelli invece di riordinarli correttamente.

```r
# Creo una copia
dosage_wrong <- dosage

# ERRORE: rinomino i livelli
levels(dosage_wrong) <- c("low", "medium", "high")
dosage_wrong
## [1] high   medium low    high   medium low
## Levels: low medium high
```

Nota come questo sia un grave errore poichè rinominado i livelli abbiamo modificato anche gli effettivi valori degli elementi. Adesso i valori sono tutti diversi e insensati.

Per riordinare corretamente i livelli riutilizziamo la funzione `factor()` specificando i livelli nell'ordine desiderato.

```r
dosage <- factor(dosage, levels = c("low", "medium", "high"))
dosage
## [1] medium low    high   medium low    high
## Levels: low medium high
```

Così facendo abbiamo riordinato i livelli a nostro piacimento senza modificare gli effettivi valori.

Extra

Sono possibili diverse operazioni con i fattori. Ad esempio, possiamo essere interessati a eliminare un certo livello di un fattore:

- se facciamo un subset di un fattore e non abbiamo più valori associati a quel livello
- se vogliamo semplicemente rimuovere un livello e le rispettive osservazioni

In questi casi possiamo usare la funzione `droplevels(x, exclude = ...)` che permette di rimuovere determinati livelli:

```
## [1] "a" "b" "c"
## [1] "a" "b" "c"
```

Come vedete nonostante non ci siano più valori `c`, abbiamo il fattore ha comunque associati tutti i valori iniziali:

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

Possiamo anche direttamente eliminare un livello ma in corrispondenza dei valori associati avremmo degli `NA`:

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

Come è possibile eliminare un livello, è anche possibile aggiungere un livello ad un fattore usando semplicemente il comando `factor()` e specificando tutti i livello più quello/i aggiuntivi:

```
##  [1] <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> b    b    b    b    b
## [16] c    c
## Levels: b c nuovolivello
```

In questo caso abbiamo usato la funzione `c(vecchi_livelli, nuovo_livello)` per creare un vettore di nuovi livelli da usare in questo caso. In alternativa possiamo anche usare il metodo di assegnazione `levels(x) <-` specificando ancora un vettore di livelli oppure specificando un singolo valore assegnando al nuovo indice:

E' possibile inoltre combinare due fattori in modo da ottenerne uno unico unendo quindi i livelli e i rispettivi valori numerici. Semplicemente usando il comando `c(fac1, fac2)`:

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

8.2.3 Fattori Ordinali

Vediamo infine un esempio di variabile categoriale ordinale. Riprendendo l'esempio precedente rigurdo i dosaggio del farmaco, è chiaro che esiste una relazionee ordinale tra i veri livelli della variabile. Per creare una variabile ordinale possiamo specificare l'argomento `ordered = TRUE`:

```
dosage_ord <- factor(dosage, levels = c("low", "medium", "high"), ordered = TRUE)
dosage_ord
## [1] medium low    high   medium low    high
## Levels: low < medium < high
```

Notiamo come la natura ordinale dei livelli sia specificata sia quando vengono riportati i livelli sia nella classe dell'oggetto

```
# Categoriale nominale
class(dosage)
## [1] "factor"

# Categoriale ordinale
class(dosage_ord)
## [1] "ordered" "factor"
```

Tip-Box: Codifica Tipologia Variabli

In R è importante codificare correttamente le differenii varibili specificando la loro tipologia. Distinnguendo appropriatamente le variabili categoriali (nominali e ordinali) rispetto alle varibili numeriche e alle semplici variabili di caratteri abbiamo numerosi vantaggi. In R, infatti, molti pacchetti e funzioni adottano particolari accorgimenti a seconda della tipologia di variabile fornendoci output e risultati coerenti alla natura della variabile.
Nota ad esempio comee l'output della funzione `summary()` cambi a seconda della tipologia di varibile.

```
# Variabile numerica
summary(1:15)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     1.0     4.5     8.0     8.0    11.5    15.0

# Variabile Categoriale
summary(dosage)
##    low medium   high
##      2      2      2
```

Questo risulterà particolarmente importante nell'esecuzione di analisi statistiche e nella creazione di grafici e tabelle.

Esercizi

Esegui i seguenti esercizi (soluzioni):

1. Crea la variabile categoriale `genere` così definita:

```
## [1] M F M F M F F F M
## Levels: F M
```

2. Rinomina i livelli della variabile `genere` rispettivamente in `"donne"` e `"uomini"`.
3. Crea la variabile categoriale `intervento` così definita:

```
## [1] CBT         Psicanalisi CBT         Psicanalisi CBT         Psicanalisi
## [7] Controllo   Controllo   CBT
## Levels: CBT Controllo Psicanalisi
```

4. Correggi nella variabile `intervento` la 7° e 8° osservazione con la voce `Farmaci`.
5. Aggiungi alla variabile `intervento` le seguenti nuove osservazioni:

```
## [1] "Farmaci"   "Controllo" "Farmaci"
```

Chapter 9

Matrici

Le matrici sono una struttura di dati **bidimensionale**, dove gli elementi sono disposti secondo righe e colonne. Possiamo quindi immaginare una matrice generica di m righe e n colonne in modo simile a quanto rappresentato in Figura 9.1.

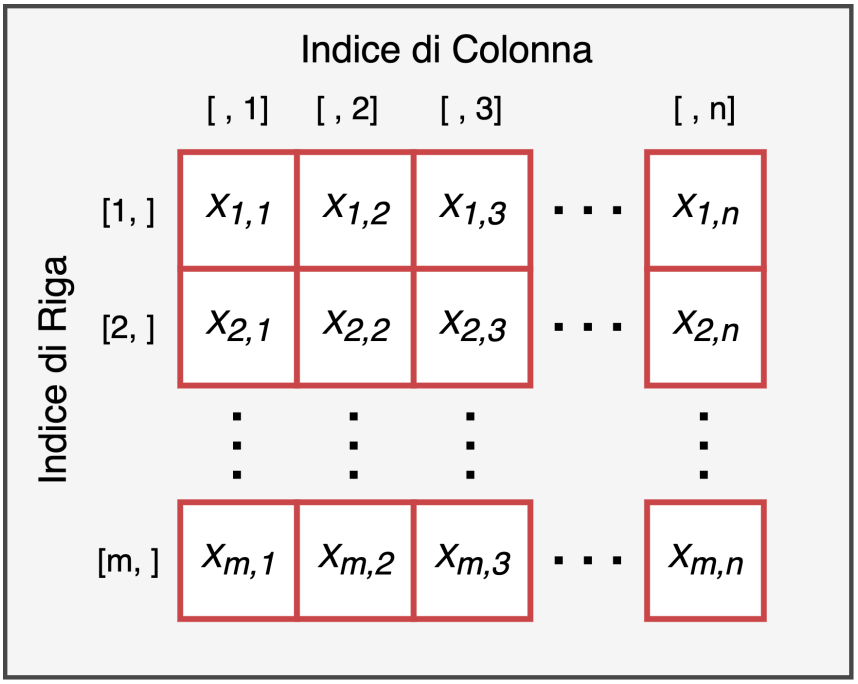


Figure 9.1: Rappresentazione della struttura di una matrice di *m* colonne e *n* righe

Due caratteristiche importanti di una matrice sono:

- la **dimensione** - il numero di **righe** e di **colonne** da cui è formata la matrice
- la **tipologia** - la tipologia di dati che sono contenuti nella matrice. Infatti, in modo analogo a quanto visto con i vettori, una matrice deve esssere formata da **elementi tutti dello stesso tipo**. Pertanto esistono diverse tipologie di matrici a seconda del tipo di dati da cui è formata, in particolare abbiamo matrici numeriche, di valori logici e di caratteri (vedi Capitolo 9.1.1).

E' fondamentale inoltre sottolineare come ogni **elemento** di una matrice sia caratterizzato da:

- un **valore** - ovvero il valore dell'elemento che può essere di qualsiasi tipo ad esempio un numero o una serie di caratteri.
- un **indice di posizione** - ovvero una **coppia di valori (*i*, *j*)** interi positivi che indicando rispettivamente **l'indice di riga** e **l'indice di colonna** e che permettono di identificare univocamente l'elemento all'interno della matrice.

Ad esempio, data una matrice X di dimensione 3×4 (i.e., 3 righe e 4 colonne) così definita:

$$X = \begin{bmatrix} 3 & 12 & 7 & 20 \\ 16 & 5 & 9 & 13 \\ 10 & 1 & 14 & 19 \end{bmatrix},$$

abbiamo che $x_{2,3} = 9$ mentre $x_{3,2} = 1$. Questo ci serve solo per ribadire il corretto uso degli indici, dove per un generico elemento $x_{i,j}$, il valore i è l'indice di riga mentre il valore j è l'indice di colonna. **Prima si indicano le righe poi le colonne**.

Vediamo ora come creare delle matrici in R e come compiere le comuni operazini di selezione. Successivamente vedremo diverse manipolazioni e operazioni con le matrici. Infine estenderemo brevemente il concetto di matrici a dimensioni maggiori di due atttraverso l'uso degli **array**.

9.1 Creazione

Il comando usato per creare una matrice in R è `matrix()` e contiene diversi argomenti:

```
nome_matrice <- matrix(data, nrow = , ncol = , byrow = FALSE)
```

- `data` - un **vettore di valori** utilizzati per popolare la matrice
- `nrow` e `ncol` - sono rispettivamente il numero di righe e il numero di colonne della matrice
- `byrow` - indica se la matrice deve essere popolata per riga oppure per colonna. Il valore di default è `FALSE` quindi i valori della matrice vengono aggiunti colonna dopo colonna. Indicare `TRUE` per aggiungere gli elementi riga dopo riga

Creiamo come esempio una matrice di 3 righe e 4 colonne con i valori che vanno da 1 a 12.

```
# Dati per popolare la matrice
my_values <- 1:12
my_values
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12

# Matrice popolata per colonne
mat_bycol <- matrix(my_values, nrow = 3, ncol = 4)
mat_bycol
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

La funzione `matrix()` ha di default l'argomento `byrow = FALSE`, quindi di base R popola le matrici colonna dopo colonna. Per popolare le matrici riga dopo riga invece, è necessario richiederlo esplicitamente specificando `byrow = TRUE`.

```
# Matrice popolata per righe
mat_byrow <- matrix(my_values, nrow = 3, ncol = 4, byrow = TRUE)
mat_byrow
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
```

E' importante notare come mentre sia possibile specificare qualsiasi combinazione di righe e colonne, il numero di valori forniti per popolare la matrice deve essere compatibile con la dimensione della matrice. In altre parole, **non posso fornire più o meno dati di quelli che la matrice può contenere**.

Pertanto, la launghezza del vettore passato all'argomento `data` deve essere compatibile con gli argomenti `nrow` e `ncol`. E' possibile tuttavia, fornire un unico valore se si desidera ottenre una matrice in cui tutti i valori siano identici. Creiamo ad esempio una matrice vuota con soli valori `NA` con 3 righe e 3 colonne.

```
mat_NA <- matrix(NA, nrow = 3, ncol = 3)
mat_NA
##      [,1] [,2] [,3]
## [1,]   NA   NA   NA
## [2,]   NA   NA   NA
## [3,]   NA   NA   NA
```

Tip-Box: Ciclare Valori

In realtà è possibile fornire più o meno dati di quelli che la matrice può contenere. Nel caso vengano forniti più valori, R semplicemente utilizza i primi valori disponibili ignorando quelli successivi.

```
matrix(1:20, nrow = 2, ncol = 2)
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

Nel caso vengano forniti meno valori, invece, R riutilizza gli stessi valori nello stesso ordinere per completare la matrice avvertendoci del problema.

```r
matrix(1:4, nrow = 3, ncol = 4)
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    3    2
## [2,]    2    1    4    3
## [3,]    3    2    1    4
```

Tuttavia, è meglio evitare questa pratica di *ciclare* i valori poichè i risultati potrebbero essere poco chiari ed è facile commettere errori.

9.1.1 Tipologie di Matrici

Abbiamo viso che, in modo analogo ai vettori, anche per le matrici è necessario che tutti i dati siano della stessa tipologia. Avremo pertanto matrici che includono solo valori `character`, `double`, `integer` oppure `logical` e le operazioni che si potranno eseguire (uso di operatori matematiche o operatori logici-relazionali) dipenderanno dalla tipologia di dati. Tuttavia, a differenza dei vettori, la tipologia di oggetto rimarrà sempre `matrix` indipendentemente dai dati contenuti. Le matrici sono sempre matrici, è la tipologia di dati che varia.

Character

E' possibile definire una matrice di soli caratteri, tuttavia sono usate raramente visto che chiaramente tutte le operazioni matematiche non sono possibili.

```r
mat_char <- matrix(letters[1:12], nrow = 3, ncol = 4, byrow = TRUE)
mat_char
##      [,1] [,2] [,3] [,4]
## [1,] "a"  "b"  "c"  "d"
## [2,] "e"  "f"  "g"  "h"
## [3,] "i"  "j"  "k"  "l"

class(mat_char)
## [1] "matrix" "array"
typeof(mat_char)
## [1] "character"
```

Trick-Box: Letters

In R esistono due speciali oggetti `letters` e `LETTERS` che includono rispettivamente le lettere minuscole e maiuscole dell'alfabeto inglese.

```
letters[1:5]
## [1] "a" "b" "c" "d" "e"
LETTERS[6:10]
## [1] "F" "G" "H" "I" "J"
```

Numeric

Le matrici di valori numerici, sia `double` che `integer`, sono senza dubbio le più comuni ed utilizzate. Vengono spesso sfruttate per eseguire calcoli algebrici computazionalemnte molto efficienti.

```
# doubles
mat_num <- matrix(5, nrow = 3, ncol = 4)
class(mat_num)
## [1] "matrix" "array"
typeof(mat_num)
## [1] "double"

# integers
mat_int <- matrix(5L, nrow = 3, ncol = 4)
class(mat_int)
## [1] "matrix" "array"
typeof(mat_int)
## [1] "integer"
```

Logical

Infine le matrici possono essere formate anche da valori logici `TRUE` e `FALSE`. Vedremo un loro importante utilizzo per quanto riguarda la selezione degli elementi di una matrice nel Capitolo 9.2.1.

```
mat_logic <- matrix(c(TRUE, FALSE), nrow = 3, ncol = 4)
mat_logic
##       [,1]  [,2]  [,3]  [,4]
## [1,]  TRUE FALSE  TRUE FALSE
## [2,] FALSE  TRUE FALSE  TRUE
## [3,]  TRUE FALSE  TRUE FALSE
class(mat_logic)
## [1] "matrix" "array"
typeof(mat_logic)
## [1] "logical"
```

Ricordiamo che è comunque possibile eseguire operazioni matematiche con i valori logici poichè verranno automaticamente rasformatti nei rispettivi valori numerici 1 e 0.

Esercizi

Esegui i seguenti esercizi (soluzioni): 1. Crea la matrice `A` così definita:

$$\begin{array}{cccc} 2 & 34 & 12 & 7 \\ 46 & 93 & 27 & 99 \\ 23 & 38 & 7 & 04 \end{array}$$

2. Crea la matrice `B` contenente tutti i primi 12 numeri dispari disposti su 4 righe e 3 colonne.
3. Crea la matrice `C` contenente i primi 12 multipli di 9 disposti su 3 righe e 4 colonne.
4. Crea la matrice `D` formata da 3 colonne in cui le lettere `"A"`,`"B"` e `"C"` vengano ripetute 4 volte ciascuna rispettivamente nella prima, seconda e terza colonna.
5. Crea la matrice `E` formata da 3 righe in cui le lettere `"A"`,`"B"` e `"C"` vengano ripetute 4 volte ciascuna rispettivamente nella prima, seconda e terza riga.

9.2 Selezione Elementi

L'aspetto sicuramente più importante (e divertente) riguardo le matrici è accedere ai vari elementi. Ricordiamo che una matrice non è altro che una griglia di righe e colonne dove vengono disposti i vari valori. Indipendentemente da cosa la matrice contenga, è possibile utilizzare gli indici di riga e di colonna per identificare univocamente un dato elemento nella matrice. Pertanto ad ogni elemento è associata una coppia di valori (i, j) dove i è l'indice di riga e j è l'indice di colonna.

Per visualizzare questo concetto, riportiamo nel seguente esempio gli indici per ogni elemnto di una matrice 3×4:

```
##      [,1]  [,2]  [,3]  [,4]
## [1,] "1,1" "1,2" "1,3" "1,4"
## [2,] "2,1" "2,2" "2,3" "2,4"
## [3,] "3,1" "3,2" "3,3" "3,4"
```

In R è possibile selezionare un elemento di una matrice utilizzando il suo indice di riga e di colonna. In modo analogo ai vettori è necessario quindi indicare all'inerno delle **parentesi quadre** `[]` poste dopo il nome del vettore, **l'indice di riga** e **l'indice di colonna separati da virgola**.

```
nome_matrice[<indice-riga>, <indice-colonna>]
```

L'**ordine** `[<indice-riga>, <indice-colonna>]` è prestabilito e deve essere rispettato affinché la selezione avvenga correttamente. Vediamo un semplice esempio di come sia possibile accedere ad un qualsiasi elemento:

```
my_matrix <- matrix(1:12, nrow = 3, ncol = 4)
my_matrix
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
# Selezioniamo l'elemento alla riga 2 e colonna 3
my_matrix[2,3]
## [1] 8

# Selezioniamo il valore 6
my_matrix[3,2]
## [1] 6
```

Warning-Box: Subscript out of Bounds

Notiamo come indicando degli indici al di fuori della dimensione della matrice otteniamo un messaggio di errore.

```
my_matrix[20,30]
## Error in my_matrix[20, 30]: subscript out of bounds
```

Oltre alla selezione di un singolo elemento è possibile eseguire altri tipi di selezione:

Selezionare Riga o Colonna

E' possibile selezionare **tutti** gli elementi di una riga o di una colonna utilizzando la seguente sintassi:

```
# Selzione intera riga
nome_matrice[<indice-riga>, ]

# Selzione intera colonna
nome_matrice[ , <indice-colonna>]
```

Nota come sia comunque necessario l'utilizzo della **virgola** lasciando vuoto il posto prima o dopo la virgola per indicare ad R di selezionare rispettivamente tutte le righe o tutte le colonne.

```
# Selezioniamo la 2 riga e tutte le colonne
my_matrix[2, ]
## [1]  2  5  8 11

# Selezioniamo  tutte le righe e la 3 colonna
my_matrix[ ,3]
## [1] 7 8 9
```

Qualora fosse necessario selezionare più righe o più colonne è sufficiente indicare tutti gli indici di interesse. Ricorda che questi devono essere specificati in un unico vettore. All'interno delle parentesi quadre, R si aspetta una sola virgola che separa gli indici di riga da quelli di colonna. E' quindi necessario combinare gli indici che vogliamo selezionare in un unico vettore sia nel caso delle righe che delle colonne. Per selezionare righe o colonne in successione, ad esempio le prime 3 colonne, posso utilizzare la scrittura compatta `1:3` che è equivalente a `c(1,2,3)`.

```
# Selezioniamo la 1 e 3 riga
my_matrix[c(1,3), ]
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    3    6    9   12

# Selezioniamo dalla 2° alla 4° colonna
my_matrix[ , 2:4]
##      [,1] [,2] [,3]
## [1,]    4    7   10
## [2,]    5    8   11
## [3,]    6    9   12
```

Selezionare Regione Matrice

Combinando indici di righe e di colonne è anche possibile selezionare specifiche regioni di una matrice o selezionare alcuni suoi valori per creare una nuova matrice.

```
# Selezioniamo un blocco
my_matrix[1:2, 3:4]
##      [,1] [,2]
## [1,]    7   10
## [2,]    8   11

# Selezioniamo valori sparsi
my_matrix[c(1,3), c(2,4)]
##      [,1] [,2]
## [1,]    4   10
## [2,]    6   12
```

Tip-Box: Selezionare non è Modificare

Ricordiamo che, come per i vettori, l'operazione di selezione non modifichi l'oggetto iniziale. Pertanto è necessario salvare il risultato della selezione se si desidera mantenere le modifiche.

Approfondimento: Matrici e Vettori

I più attenti avranno notato che che i comandi di selezione non restituiscono sempre lo stesso oggetto, a volte otteniamo come risultato un vettore e delle altre una matrice.
E' importante chiarire che una **un vettore non è un matrice** e tanto più vale l'opposto. In R questi sono due tipologie di oggetti diversi e sarà importante tenere a mente questa distinzione.

```r
# Un vettore non è una matrice
my_vector <- 1:5
is.vector(my_vector) # TRUE
is.matrix(my_vector) # FALSE

# Una matrice non è un vettore
my_matrix <- matrix(1, nrow = 3, ncol = 3)
is.vector(my_matrix) # FALSE
is.matrix(my_matrix) # TRUE
```

Il risultato che otteniamo da una selezione potrebbe essere un vettore oppure una matrice a seconda del tipo di selezione. Vediamo in particolare come selezionando un'unica colonna (o riga) otteniamo un vettore mentre selezionando più colonne (o righe) otteniamo una matrice.

```r
# Seleziono una colonna
is.vector(my_matrix[, 1]) # TRUE
is.matrix(my_matrix[, 1]) # FALSE

# Seleziono più colonne
is.vector(my_matrix[, c(1,2)]) # FALSE
is.matrix(my_matrix[, c(1,2)]) # TRUE
```

Questa distinzione influirà sul successivo utilizzo dell'oggetto ottenuto dalla selezione.

Vettore Riga e Vettore Colonna

Una particolare fonte di incomprensioni e successivi errori riguarda proprio l'utilizzo di un vettore ottenuto dalla selezione di una singola riga (o una singola colonna) di una matrice come fosse un *vettore riga* (o un *vettore colonna*).
In algebra lineare, i *vettori riga* ed i *vettori colonna* non sono altro che delle matrici rispettivamente di dimensione $1 \times n$ e $m \times 1$. La dimensione (*righe* $\times$ *colonne*) di una matrice, e quindi anche di un vettore, rivestono un ruolo importante nelle operazioni con le matrici ed in particolare nel prodotto matriciale.
In R i vettori hanno una sola dimensione ovvero la *lunghezza* e quindi nel loro utilizzo con operazioni tra matrici vengono convertiti automaticamente in vettori riga o vettori colonna a seconda delle necessità. Tuttavia, questa trasformazione potrebbe non sempre rispettare le attuali intenzioni ed è quindi meglio utlizzare sempre le matrici e non i vettori.

```r
# Vettore
my_vector <- 1:4
dim(my_vector) # dimensione righe, colonne
## NULL
length(my_vector)
## [1] 4

# Matrice 1x4 (vettore riga)
my_row_vector <- matrix(1:4, nrow = 1, ncol = 4)
dim(my_row_vector)
## [1] 1 4
my_row_vector
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4

# Matrice 4x1 (vettore colonna)
my_col_vector <- matrix(1:4, nrow = 4, ncol = 1)
dim(my_col_vector)
## [1] 4 1
my_col_vector
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
## [4,]    4
```

Srotolare una Matrice

Abbiamo visto che possiamo facilmente popolare una matrice con un vettore. Allo stesso modo possiamo vettorizzare una matrice (in altri termini "srotolare" la matrice) per ritornare al vettore originale. Con il comando `c(matrice)` oppure forzando la tipologia di oggetto a vettore con `vector(matrice)` o `as.vector(matrice)`.

```r
# Da matrice a vettore
my_matrix <- matrix(1:12, nrow = 3, ncol = 4)
c(my_matrix)
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
as.vector(my_matrix)
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

9.2.1 Utilizzi Avanzati Selezione

Vediamo ora alcuni utilizzi avanzati della selezione di elementi di una matrice. In particolare impareremo a:

- utilizzare gli operatori relazionali e logici per selezionare gli elementi di una martrice
- modificare l'ordine di righe e colonne
- sostituire degli elementi
- eliminare delle righe o colonne

Nota che queste operazioni siano analoghe a quelle viste per i vettori e quindi seguiranno le stesse regole e principi.

Operatori Relazionali e Logici

Un'utile funzione è quella di selezionare tra gli elementi di una matrice quelli che rispetano una certa condizione. Possiamo ad esempio valutare *"quali elementi della matrice sono maggiori di x?"*. Per fare questo dobbiamo specificare all'interno delle parentesi quadre la proposizione di interesse utilizzando gli operatori relazionali e logici (vedi Capitolo 3.2).

Quando una matrice è valutata all'interno di una proposizione, R valuta la veridicità di tale proposizione rispetto ad ogni suo elemento. Come risultato otteniamo una matrice di valori logici con le rispettive risposte per ogni elemento (`TRUE` o `FALSE`).

```
my_matrix <- matrix(1:23, nrow = 3, ncol = 4)
## Warning in matrix(1:23, nrow = 3, ncol = 4): data length [23] is not a sub-
## multiple or multiple of the number of rows [3]
my_matrix
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12

# Elemeni maggiori di 4 e minori di 10
test <- my_matrix >= 4 & my_matrix <=10
test
##       [,1] [,2] [,3]  [,4]
## [1,] FALSE TRUE TRUE  TRUE
## [2,] FALSE TRUE TRUE FALSE
## [3,] FALSE TRUE TRUE FALSE
```

Questa matrice può essere utilizzata all'interno delle parentesi quadre per selezionare gli elementi della matrice originale che soddisfano la proposizione. Gli elementi associati al valore `TRUE` sono selezionati mentre quelli associati al valore `FALSE` sono scartati.

```
# Seleziono gli elementi
my_matrix[test]
## [1]  4  5  6  7  8  9 10
```

Nota come in questo caso non sia necessaria alcuna virgola all'interno delle parentesi quadre e come il risultato ottenuto sia un vettore.

Modificare Ordine Righe e Colonne

Gli indici di riga e di colonna possono essere utilizzati per riordinare le righe e le colonne di una matrice a seconda delle necessità.

```
my_matrix <- matrix(1:6, nrow = 3, ncol = 4)
my_matrix
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    1    4
## [2,]    2    5    2    5
## [3,]    3    6    3    6

# Altero l'ordinen delle righe
my_matrix[c(3,2,1), ]
##      [,1] [,2] [,3] [,4]
## [1,]    3    6    3    6
## [2,]    2    5    2    5
## [3,]    1    4    1    4

# Altero l'ordine delle colonnne
my_matrix[ ,c(1,3,2, 4)]
##      [,1] [,2] [,3] [,4]
## [1,]    1    1    4    4
## [2,]    2    2    5    5
## [3,]    3    3    6    6
```

Modificare gli Elementi

Un importante utilizzo degli indici riguarda la modifica di un elemento di una matrice. Per sostituire un vecchio valore con un nuovo valore, seleziono il vecchio valore della matrice e utilizzo la funzione `<-` (o `=`) per assegnare il nuovo valore.

```
my_matrix <- matrix(1:12, nrow = 3, ncol = 4)
my_matrix
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12

# Modifico il l'elemento con il valore 5
my_matrix[2,2] <- 555
my_matrix
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2  555    8   11
## [3,]    3    6    9   12
```

E' possibile anche sostituire tutti i valori di un'intera riga o colonnna opportunamente selezionata. In questo caso sarà necessario fornire un corretto numero di nuovi valori da utilizzare.

```
# Modifico la 2 colonna
my_matrix[ ,2] <- c(444, 555, 666)
my_matrix
##      [,1] [,2] [,3] [,4]
## [1,]    1  444    7   10
## [2,]    2  555    8   11
## [3,]    3  666    9   12

# Modifico la 3 riga
my_matrix[3, ] <- c(111, 666, 999, 122)
my_matrix
##      [,1] [,2] [,3] [,4]
## [1,]    1  444    7   10
## [2,]    2  555    8   11
## [3,]  111  666  999  122
```

Nota come a differenza dei vettori non sia possibile aggiungere una nuova riga o colonna attraverso questa operazione ma sarà necesssario utilizzare una diversa procedura (vedi Capitolo 9.3.2).

```
# Aggiungo una nuova colona [errore selezione indici]
my_matrix[, 5] <- c(27, 27, 27)
## Error in `[<-`(`*tmp*`, , 5, value = c(27, 27, 27)): subscript out of bounds
```

Eliminare Righe o Colonne

Per **eliminare** delle righe (o delle colonne) da una matrice, è necessario indicare all'interno delle parentesi quadre gli indici di riga (o di colonna) che si intende eliminare, preceduti dall'operatore `-` (*meno*). Nel caso di più righe (o colone) è possibile indicare il meno solo prima del comando `c()` analogamente con quanto fatto con i vettori.

```
my_matrix <- matrix(1:12, nrow = 3, ncol = 4)

# Elimino la 2° riga
my_matrix[-2, ]
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    3    6    9   12

# Elimino la 2° riga  e la 2° e 3° colonna
my_matrix[-2, -c(2,3)]
##      [,1] [,2]
## [1,]    1   10
## [2,]    3   12
```

Nota come l'operazione di eliminazione sia comunque un'operazione di selezione. Pertanto è necessario salvare il risultato ottenuto se si desidera mantenere le modifiche.

Esercizi

Utilizzando le matrici create nei precedenti esercizi esegui le seguenti consegne (soluzioni):

1. Utilizzando gli indici di riga e di colonna selziona il numero 27 della matrice `A`
2. Selziona gli elementi compresi tra la seconda e quarta riga, seconda e terza colonna della matrice `B`
3. Seleziona solo gli elementi pari della matrice `A` (Nota: utilizza l'operazione resto `%%`)
4. Elimina dalla matrice `C` la terza riga e la terza colonna
5. Seleziona tutti gli elementi della seconda e terza riga della matrice `B`
6. Seleziona tutti gli elementi diversi da "B" appartenenti alla matrice `D`

9.3 Funzioni ed Operazioni

Vediamo ora alcune funzioni frequentemente usate e le comuni operazioni eseguite con le matrici (vedi Tabella 9.1).

Table 9.1: Funzioni e operazioni con matrici

Funzione	Descrizione
`nuova_matrice <- cbind(matrice1, matrice2)`	Unire due matrici creando nuove colonne (le matrici devono av
`nuova_matrice <- rbind(matrice1, matrice2)`	Unire due matrici creando nuove righe (le matrici devono aver
`nrow(nome_matrice)`	Numero di righe della matrice
`ncol(nome_matrice)`	Numero di colonne della matrice
`dim(nome_matrice)`	Dimensione della matrice (righe e colonne)
`colnames(nome_matrice)`	Nomi delle colonne della matrice
`rownames(nome_matrice)`	Nomi delle righe della matrice
`dimnames(nome_matrice)`	Nomi delle righe e delle colonne
`t(nome_matrice)`	Trasposta della matrice
`diag(nome_matrice)`	Vettore con gli elementi della diagonale della matrice
`det(nome_matrice)`	Determinante della matrice (la matrice deve essere quadrata)
`solve(nome_matrice)`	Inversa della matrice
`matrice1 + matrice2`	Somma elemento per elemento di due matrici
`matrice1 - matrice2`	Differenza elemento per elemento tra due matrici
`matrice1 * matrice2`	Prodotto elemento per elemento tra due matrici
`matrice1 / matrice2`	Rapporto elemento per elemento tra due matrici
`matrice1 %*% matrice2`	Prodotto matriciale

Descriviamo ora nel dettaglio alcuni particolari utilizzi.

9.3.1 Attributi di una Matrice

Abbiamo visto nel Capitolo 8.1 che gli oggetti in R possiedono quelli che sono definiti *attributi* ovvero delle utili informazioni riguardanti l'oggetto stesso, una sorta di *metadata*. Vediamo ora alcuni attributi particolarmente rilevanti nel caso delle matrici ovvero la dimensione (`dim`) e i nomi delle righe e colonne (`names`).

Dimensione

Ricordiamo che la matrice è un oggetto **bidimensionale** formato da righe e colonne. Queste formano pertanto le dimensioni di una matrice. Per ottenere il numero di righe e di colonne di una matrice, possiamo usare rispettivamente i comandi `nrow()` e `ncol()`.

```r
my_matrix <- matrix(1:12, ncol = 3, nrow = 4)

# Numero di righe
nrow(my_matrix)
## [1] 4

# Numero di colonne
ncol(my_matrix)
## [1] 3
```

In alternativa per conoscere le dimensioni di una matrice è possibile utilizzare la funzione `dim()`. Questa ci restituirà un vettore con due valori dove il primo rappresenta il numero di righe e il secondo il numero di colonne.

```r
dim(my_matrix)
## [1] 4 3
```

Nomi Righe e Colonne

Come avrete notato, di base le dimensioni di una matrice (ovvero le righe e le colonne) vengono identificate attraverso i loro indici numerici. In R tuttavia, è anche possibile assegnare dei nomi alle righe e alle colonne di una matrice.

Con i comandi `rownames()` e `colnames()` possiamo accedere rispettivamente ai nomi delle righe e delle colonne.

```r
my_matrix <- matrix(1:12, nrow = 3, ncol = 4)

# Nome di righe
rownames(mat)
## NULL

# Nome di colonne
colnames(mat)
## NULL
```

Non essendo impostati, ottieniamo inizialmente come output il valore `NULL`. Per impostare i nomi di righe e/o colonne, sarà quindi necessario assegnare a `rownames(nome_matrice)` e `colnames(nome_matrice)` un vettore di caratteri della stessa lunghezza della dimensione che stiamo rinominando. Se impostiamo un unico carattere, tutte le righe/colonne avranno lo stesso valore. Questo ci fa capire che, se vogliamo impostare dei nomi, R richiede che questo venga fatto per tutte le righe/colonne.

```r
# Assegnamo i nomi alle righe
rownames(my_matrix) <- LETTERS[1:3]
my_matrix
##   [,1] [,2] [,3] [,4]
## A    1    4    7   10
## B    2    5    8   11
## C    3    6    9   12
```

```
# Assegno i nomi alle colonne
colnames(my_matrix) <- LETTERS[4:7]
my_matrix
##   D E F  G
## A 1 4 7 10
## B 2 5 8 11
## C 3 6 9 12
```

In alternativa posso utilizzare il `dimnames()` per accedere contemporaneamente sia ai nomi di riga che a quelli di colonna. Come output ottengo una lista (vedi Capitolo 11) dove vengono prima indicati i nomi di riga e poi quelli di colonna

```
dimnames(my_matrix)
## [[1]]
## [1] "A" "B" "C"
##
## [[2]]
## [1] "D" "E" "F" "G"
```

Approfondimento: Selezione per Nomi

Quando nel Capitolo 9.2 abbiamo visto i diversi modi di selezionare gli elementi di una matrice, abbiamo sempre usato gli indici numerici di riga e di colonna. Tuttavia, quando i nomi delle delle dimensioni sono disponibili, è possibile indicizzare una matrice in base ai nomi delle righe e/o colone.
Possiamo quindi selezionare la prima colonna sia con il suo indice numerico `nome_matrice[ , 1]` ma anche con il nome assegnato `nome_matrice[ ,"nome_colonna"]`. Queste sono operazioni poco utili con le matrici ma che saranno fondamentali nel caso dei **dataframe** (vedi Capitolo 10).

```
# Seleziono la colona "F"
my_matrix[ , "F"]
## A B C
## 7 8 9

# Selezioniamo la riga "A" e "C"
my_matrix[c("A", "C"), ]
##   D E F  G
## A 1 4 7 10
## C 3 6 9 12
```

9.3.2 Unire Matrici

Abbiamo visto nel Capitolo 7.3 come si possono unire diversi vettori tramite la funnzione `c()`. Anche per le matrici è possibile combinare matrici diverse, rispettando però alcune regole:

- `rbind()` - Posso unire matrici **per riga** ovvero aggiungo una o più righe ad una matrice, in questo caso le matrici devono avere lo stesso numero di colonne
- `cbind()` - Posso unire matrici **per colonna** ovvero aggiungo una o più colonne ad una matrice, in questo caso le matrici devono avere lo stesso numero di righe
- Le matrici che unisco devono essere della **stessa tipologia** (numeri o caratteri)

Quindi, data una matrice `my_matrix` di dimensione $m \times n$ se volgiamo aggiungere le righe di una seconda matrice `row_matrix` possiamo utlizzare il comando `rbind(my_matrix, row_matrix)` a patto che abbiano lo stesso numero di colonnne (n). Se vogliamo invece aggiungere le colonne di una nuova matrice `col_matrix` possiamo utlizzare il comando `cbind(my_matrix, col_matrix)` a patto che abbiano lo stesso numero di righe (m). E' utile pensare all'unione come ad un collage tra matrici. In Figura9.2 è presente uno schema utile per capire visivamente questo concetto, dove le matrici colorate in verde possono essere correttamente unite mentre le matrici in rosso non presentano la corretta dimensionalità.

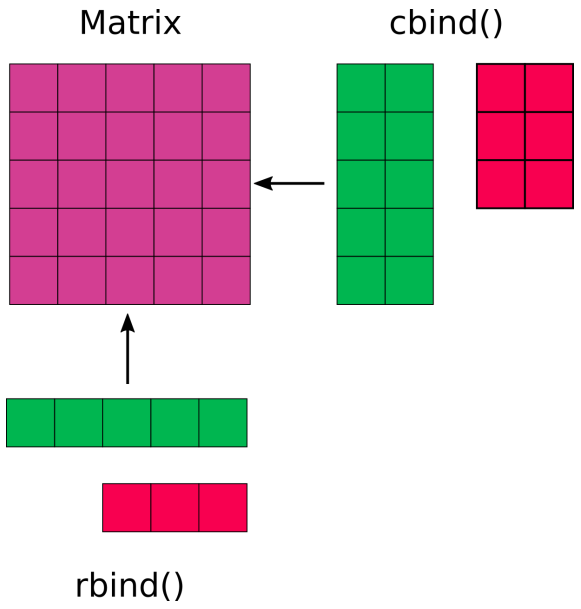


Figure 9.2: Schema per la combinazione di matrici

Vediamo un esempio in R:

```
# Matrice di partenza
my_matrix <- matrix(1:12, nrow = 3, ncol = 4)
my_matrix
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```r
# Matrice con stesso numero di colonne
row_matrix <- matrix(77, nrow = 2, ncol = 4)
row_matrix
##      [,1] [,2] [,3] [,4]
## [1,]   77   77   77   77
## [2,]   77   77   77   77

# Unione per riga
rbind(my_matrix, row_matrix)
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
## [4,]   77   77   77   77
## [5,]   77   77   77   77


# Matrice con stesso numero di righe
col_matrix <- matrix(99, nrow = 3, ncol = 2)
col_matrix
##      [,1] [,2]
## [1,]   99   99
## [2,]   99   99
## [3,]   99   99

# Unione per colonna
cbind(my_matrix, col_matrix)
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    4    7   10   99   99
## [2,]    2    5    8   11   99   99
## [3,]    3    6    9   12   99   99
```

Un ultimo aspetto utile è l'estensione dei comandi `cbind()` ed `rbind()`. Fino ad ora li abbiamo utilizzati con due soli elementi: matrice di partenza e matrice da aggiungere. Tuttavia, è possibile indicare più matrici che si vogliono unire, separandole con una virgola. Se vogliamo combinare n matrici possiamo usare il comando `cbind(mat1, mat2, mat3, ...)` o `rbind(mat1, mat2, mat3, ...)`. In questo caso il risultato finale sarà l'unione delle matrici nell'ordine utilizzato nel definire gli argomenti quindi prima la `mat1`, poi la `mat2` e così via.

Warning-Box: "Matrices Must Match"

Abbiamo visto che possiamo unire matrici per riga/colonna solo se il numero di colonne/righe delle due matrici sono equivalenti. Otteniamo un errore, invece, quando cerchiamo di combinare matrici di dimensioni diverse *"number of columns/rows of matrices must match"*.

```r
my_matrix <- matrix(1:12, nrow = 3, ncol = 4)

# Matrice con errato numero di colonne
wrong_matrix <- matrix(77, nrow = 2, ncol = 7)
rbind(my_matrix, wrong_matrix)
## Error in rbind(my_matrix, wrong_matrix): number of columns of matrices must
```

9.3.3 Operatori Matematici

Gli operatori matematici (e.g., `+`, `-`, `*`, `/`, etc.) svolgono le operazioni tra una matrice ed un singolo valore, tra una matrice ed un vettore oppure tra due matrici.

Operazione tra Matrice e Valore Singolo

Nel caso di un singolo valore, la stessa operazione viene semplicemente eseguita su tutti gli elementi della matrice. Possiamo ad esempio aggiungere a tutti gli elementi di una matrice il volore 100.

```r
my_matrix <- matrix(1:12, nrow = 3, ncol = 4)
my_matrix
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12

# Aggiungo 100
my_matrix + 100
##      [,1] [,2] [,3] [,4]
## [1,]  101  104  107  110
## [2,]  102  105  108  111
## [3,]  103  106  109  112
```

Operazione tra Matrice e Vettore

Nel caso di un vettore, l'operazione viene eseguita **elemento per elemento** ciclando i valori del vettore qualora la sua lunghezza non sia suficiente. Possiamo ad esempio aggiungere a tutti gli elementi di una matrice il vettore di valori `c(100, 200, 300, 400)`.

```r
# Aggiungo un vettore di valori
my_matrix + c(100, 200, 300, 400)
##      [,1] [,2] [,3] [,4]
## [1,]  101  404  307  210
## [2,]  202  105  408  311
## [3,]  303  206  109  412
```

Com'è facilmente intuibile, questa operazione è poco connsigliata poichè è facile causa di errori ed incomprensioni.

Operazione tra Matrici

Nel caso di operazioni tra matrici, l'operazione viene eseguita **elemento per elemento** ed è quindi importante che le matrici abbiano la **stessa dimensione**. Possiamo ad esempio aggiungere a tutti gli elementi di una matrice nelle cui righe abbiamo i valori 100, 200, 300 e 400.

```
sum_matrix <- matrix(rep(c(100, 200, 300, 400), each = 3), nrow = 3, ncol = 4)
sum_matrix
##      [,1] [,2] [,3] [,4]
## [1,]  100  200  300  400
## [2,]  100  200  300  400
## [3,]  100  200  300  400

# sommo due matrici
my_matrix + sum_matrix
##      [,1] [,2] [,3] [,4]
## [1,]  101  204  307  410
## [2,]  102  205  308  411
## [3,]  103  206  309  412
```

Tip-Box: Matrix Multiplication

Nota come in R l'operatore `*` indichi il semmplice prodotto elemento per elemento mentre per ottenere il prodotto matriciale è necessario utilizzare l'operatore `%*%`.
Il prodotto matriciale segue delle specifiche regole e specifiche proprietà. In particolare, il numero di colonne della prima matrice deve essere uguale al numero di righe della seconda matrice. Per un approfondimento vedi https://it.wikipedia.org/wiki/Moltiplicazione_di_matrici.

Algebra Lineare

Anche altri aspetti che riguardano le operazioni con le matrici non vengono qui discusse ma si rimanda il lettore interessato alle seguenti pagine:

- Per il significato di determinante di una matrice considera https://it.wikipedia.org/wiki/Determinante
- Per il significato di matrice inversa considera https://it.wikipedia.org/wiki/Matrice_invertibile

Diagonale

Vediamo ora alcuni utili funzioni che riguardano la diagonale di una matrice. La diagonale di una matrice è formata dagli elementi i cui indici di riga e di

colonna sono uguali, ovvero l'insieme di elementi associati allo stesso indice di riga e colonna ($x_{i,i}$).
Il comando `diag(nome_matrice)` permette di estrarre la diagonale di una matrice e trattarla come un semplice vettore:

```
# Matrice quadrata
square_matrix <- matrix(1:16, nrow = 4, ncol = 4)
square_matrix
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16

diag(square_matrix)
## [1]  1  6 11 16

# Matrice non quadrata
my_matrix <- matrix(1:12, nrow = 3, ncol = 4)
my_matrix
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12

diag(my_matrix)
## [1] 1 5 9
```

La funzione `diag()` può anche essere usata per sostituire in modo semplice gli elementi sulla diagonale di una matrice oppure per creare una matrice diagonale in cui gli altri valori siano tutti zero, ad esempio la matrice identità.

```
# Sostituisco gli elementi della diagonale
diag(my_matrix) <- 999
my_matrix
##      [,1] [,2] [,3] [,4]
## [1,]  999    4    7   10
## [2,]    2  999    8   11
## [3,]    3    6  999   12

# Creo una matrice diagonale
diag(4, nrow = 3, ncol = 4)
##      [,1] [,2] [,3] [,4]
## [1,]    4    0    0    0
## [2,]    0    4    0    0
## [3,]    0    0    4    0

# Creo una matrice identità 4X4
diag(4)
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    1    0
## [4,]    0    0    0    1
```

Esercizi

Utilizzando le matrici create nei precedenti esercizi, esegui le seguentti consegne (soluzioni):

1. Crea la matrice `G` unendo alla matrice `A` le prime due colonne della matrice `C`
2. Crea la matrice `H` unendo alla matrice `C` le prime due righe della matrice trasposta di `B`
3. Ridefinisci la matrice `A` eliminando la seconda colonna. Ridefinisci la matrice `B` eliminando la prima riga. Verifica che le matrici così ottenute abbiano la stessa dimensione.
4. Commenta i differenti risultati che otteniamo nelle operazioni `A*B`, `B*A`, `A%*%B` e `B%*%A`.
5. Assegna i seguenti nomi alle colonne e alle righe della matrice `C`: `"col_1"`, `"col_2"`, `"col_3"`, `"col_4"`, `"row_1"`, `"row_2"`, `"row_3"`.

9.4 Array

Abbiamo visto come le matrici siano un oggetto *bidimensionale*, tuttavia è possibile anche creare oggetti che abbiano 3, 4 o un qualsiasi numero (n) di dimensioni. Tali oggetti sono definiti **array** e possono essere creati con il comando `array()` indicando il vettore di valori utilizzati per popolare l'oggetto e la grandezza di ciascuna delle sue dimensioni.

```
array(data = , dim = )
```

Ad esempio per creare un cubo di lato 3 contenete i valori interi dall'1 al 27 possiamo eseguire il seguente comando.

```
my_cube <- array(1:27, dim = c(3,3,3))
my_cube
## , , 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
```

Tutte le principali funzioni ed operazioni di selezione che abbiamo visto per le matrici ed i vettori posono essere eseguite in modo analogo anche con gli array. Il funzinonamento generale della selezione di elementi tramite parentesi quadre risulterà ora certamente pù chiaro. Per ogni dimensione vengono indicarti gli indici di posizioni desiderati. L'ordine all'interno delle parentesi quadre determina la specifica deimensione a cui ci si riferisce e le virgole sono utilizzate per separare gli indici delle diverse dimensioni.

```
my_hypercube[<dim-1>, <dim-2>, <dim-3>, ..., <dim-n>]
```

Vediamo ora alcuni semplici eempi di selezione.

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

```
## [2,]    2    5
##
## , , 2
##
##      [,1] [,2]
## [1,]   10   13
## [2,]   11   14
```

Tip-Box: Array Mother of All Matrix

E' facilmente intuibile come le matrici non siano altro che un caso speciale di array con 2 dimensioni. Infatti i più attenti avranno notato che il valore `"array"` compariva insieme a `"matrix"` nel valutare la tipologia di oggetto.

```
my_matrix <- matrix(1:12, nrow = 2, ncol = 2)
is.array(my_matrix)
## [1] TRUE
class(my_matrix)
## [1] "matrix" "array"
```

Tuttavia nota come un semplice vettore non sia un array. Ricordiamo infatti che un vettore non possiede una dimensione (`dim`) ma semplicemente una lunghezza (`length`).

```
my_vector <- 1:12
is.array(my_vector)
## [1] FALSE
dim(my_vector)
## NULL
```

Chapter 10

Dataframe

I `dataframe` sono uno degli oggetti più utilizzati in R per rappresentare i propri dati. In modo analogo alle matrici, i `dataframe` sono una struttura **bidimensionale** dove i dati sono disposti secondo righe e colonne. Può essere utile pensare al dataframe esattamente come ad una normale tabella che si può creare in un foglio di calcolo (e.g., Excel) dove possiamo inserire i nostri dati. Dati in questo caso è volutamente generico poichè i dataframe, a differenza delle matrici, possono contenere nello stesso oggetto tipi diversi di dati (e.g., *nomi*, *date* e *numeri*).

La struttura di base di un dataframe è quindi la stessa di una matrice ma ci permette di includere diversi tipi di dati nello stesso oggetto come caratteri e valori numerici. Questo ci consente di raccogliere in un unico oggetto tutte le caratteristiche delle unità statistiche (variabili numeriche, categoriali, nominali etc.) che intendiamo successivamente analizzare. Un aspetto cruciale quindi è proprio quello che il dataframe è stato pensato per gestire dati complessi ed eterogenei come quelli che si trovano in un'analisi di dati reale. Se vi capiterà di utilizzare altri linguaggi di programmazione sopratutto mirati all'analisi dati (e.g., Matlab) noterete come vi mancherà un oggetto potente e intuitivo come il dataframe.

In genere in un dataframe le righe rappresentano le unità statitiche (ad esempio persone o osservazioni) e le colonne rappresentano variabili ovvero delle proprietà misurate su quelle unità statistiche. Esistono tuttavia due formati principali di dataframe a seconda del modo in cui vengono organizzati i dati. Abbiamo i dataframe in forma **wide** (oppure larga) oppure i dataframe in forma **long** (oppure lunga). Valutiamo la differenza tra i due formati ipotizzando dei dati dove per ogni soggetto abbiamo misuriamo l'età, il genere, e la risposta a tre item di un questionario.

Wide Dataframe

Nel formato **wide**, ogni singola riga del dataframe rappresenta un soggetto e ogni sua risposta o variabile misurata sarà riportata in una diversa colonna. In Tabelle 10.1 vengono presentati i darti dell'esempio in un formato wide.

Osserviamo come ogni soggetto si identificato da un codice riportato nella variabile *Id* e le risposte ai tre item siano riportate in tre diverse variabili *item_1*, *item_2* e *item_3*.

Long Dataframe

Nel formato **long**, ogni singola riga rappresenta una singola osservazione. Quindi i dati di ogni soggetto saranno riportati su più righe e le variabili che non cambiano tra le osservazioni saranno

Table 10.1: Dataframe nel formato wong

Id	age	gender	item_1	item_2	item_3
subj_1	21	F	2	0	2
subj_2	23	M	1	2	0
subj_3	19	F	1	1	1

Table 10.2: Dataframe nel formato long

Id	age	gender	item	response
subj_1	21	F	1	2
subj_1	21	F	2	1
subj_1	21	F	3	1
subj_2	23	M	1	0
subj_2	23	M	2	2
subj_2	23	M	3	1
subj_3	19	F	1	2
subj_3	19	F	2	0
subj_3	19	F	3	1

ripetute. In Tabelle 10.2 vengono presentati i darti dell'esempio in un formato lide.

Osserviamo come le risposte di ogni soggetto siano distribuite su più righe. Le caratterische che non variano vengono ripetute ad ogni riga (*Id*, *age* e *gender*) mentre le risposte agli item vengono registrate utilizzando due colonne *item*, ovvero il numero dell'item, e *response* l'effettiva risposta di quel partecipante a quello specifico item.

Tip-Box: Long o Wide?

I dati in forma long e wide hanno delle proprietà diverse sopratutto in riferimento all'utilizzo. La tipologia di dato e il risultato finale è esattamente lo stesso tuttavia alcuni software o alcuni pacchetti di R funzionano solo con dataset organizzati in un certo modo.
Non c'è quindi un formato corretto o sbagliato ma dipende dal tipo di analisi e dal software o pacchetto che si utilizza. Alcune operazioni o analisi richiedono il dataset in forma **long** altre in forma **wide**.
Il consiglio però è di abituarsi il più possibile a ragionare in forma **long** perchè la maggior parte dei moderni pacchetti per l'analisi dati e per la creazione di grafici richiedono i dati in questo formato. Ci sono comunque delle funzioni (più avanzate) per passare velocemente da un formato all'altro.

Nota come nei precedenti esempi abbiamo utilizzato sia colonne che contengono valori numerici numeri sia colonne con caratteri. Questo non era chiaramente possibile con le matrici. Ricorda tuttavia che, come per le matrici, anche i dataframe richiedono che tutte le colonne (variabili) abbiano lo stesso numero di elementi.

Vedremo ora come creare dei dataframe in R e come compiere le comuni operazini di selezione.

Infine descriveremo alcune semplici manipolazioni e operazioni con i dataframe. Come vedremo ci sono molte somiglianze nell'utilizzo dei dataframe e delle matrici. Quando necessario, si farà riferimento al capitolo precedente per far notare aspetti in comune e differenze tra queste due strutture di dati.

10.1 Creazione di un dataframe

Il comando per creare un dataframe è `data.frame()`:

```
nome_df <- data.frame(
  variable1 = c(...),
  variable2 = c(...),
  ...)
```

La creazione è leggermente diversa rispetto al caso delle matrici. Intuitivamente è facile immaginarla come l'unione di diverse colonne (dove una può contenere dei nomi, un'altra delle date e così via) piuttosto che un insieme di valori inseriti per riga o per colonna come per le matrici. Infatti per creare un `dataframe` è proprio necessario specificare le colonne una alla volta, indicando `nome_colonna = valori` all'interno del comando `data.frame()`. Vediamo un esempio in R,

```
my_data <- data.frame(
  Id = c(1:5),
  names = c("Alice", "Bruno", "Carla", "Diego", "Elisa"),
  gender = factor(c("F","M","F", "M", "F")),
  age = c(22, 25, 23, 22, 24),
  faculty = factor(c("Psicologia", "Ingegneria", "Medicina", "Lettere", "Psicologia"))
)

my_data
##   Id names gender age    faculty
## 1  1 Alice      F  22 Psicologia
## 2  2 Bruno      M  25 Ingegneria
## 3  3 Carla      F  23   Medicina
## 4  4 Diego      M  22    Lettere
## 5  5 Elisa      F  24 Psicologia
```

In questo caso abbiamo creato un ipotetico dataframe dove in ogni riga abbiamo un soggetto e ogni colonna rappresenta una data caratteristica del soggetto come il genere, l'età e così via.

> Tip-Box: ID
>
> E' sempre consigliato definire una colonna (e.g., `Id`) in cui assegnnare un idenificativo univoco ad ogni soggetto. Questo ci permetto di poterlo indentificare senza correre il rischio di compiere errori. Ad esempio l'utlizzo del nome (o anche nome e cognome) potrebbe non essere indicato poichè più persone potrebbero avere lo stesso nome e non saremo quindi in grado di

discriminare i due soggetti.

Warning-Box: stringsAsFactors

Una variabile di caratteri all'interno di un DataFrame è considerata di default come una semplice sequenza di caratteri. E' possibile specificare l'argomento `stringsAsFactors = TRUE` per ottenere che tutte le variabili di caratteri siano considerate come delle variabili catgoriali creando automaticamente dei fattori (vedi Capitolo 8).
Nota come il comportamento di default sia differente a seconda della versione di R. Versioni precedenti a R 4.0 avevano infatti come default `stringsAsFactors = TRUE` mentre dalla 4.0 in poi abbiamo `stringsAsFactors = FALSE`.
Presta quindi molta attenzione quando utilizzi codici e soluzioni scritte prima della versione 4.0.

Esercizi

Esegui i seguenti esercizi (soluzioni):

1. Crea il dataframe `data_wide` riportato in Tabella 10.1
2. Crea il dataframe `data_long` riportato in Tabella 10.2

10.2 Selezione Elementi

Per selezionare uno o pù valori da un dataframe è possibile, in modo analogo alle matrici, utilizzare gli indici di riga e di colonna all'interno delle parentesi quadre:

```
nome_dataframe[<indice-riga>, <indice-colonna>]
```

Ricordiamo che l'ordine `[<indice-riga>, <indice-colonna>]` è prestabilito e deve essere rispettato affinché la selezione avvenga correttamente. Possiamo quindi eseguire diverse tipologie di selezioni a seconda delle necessità ussanto le stesse procedure viste per le matrici. Ad esempio, riprendeno il dataframe `my_data` creato precedentemente possiamo selezionare:

```
my_data
##    Id    names gender age    faculty
## 1   1    Alice      F  22 Psicologia
## 2   2    Bruno      M  25 Ingegneria
## 3   3    Carla      F  23   Medicina
## 4   4    Diego      M  22    Lettere
## 5   5    Elisa      F  24 Psicologia
## 6   6 Fabrizio      M  35    Lettere
```

```
## 7   7   Gloria      F  26 Ingegneria
## 8   8   Herman      M  20 Psicologia
## 9   9    Irene      F  23 Statistica
## 10 10     Luca      M  22 Ingegneria

# Valore 3° riga e 2° colonna
my_data[3,4]
## [1] 23

# Tutte le variabili della 1° e 3° riga
my_data[c(2,3), ]
##   Id names gender age    faculty
## 2  2 Bruno      M  25 Ingegneria
## 3  3 Carla      F  23   Medicina

# Tutti i valori della 5° variabile
my_data[ , 5]
##  [1] Psicologia Ingegneria Medicina   Lettere    Psicologia Lettere
##  [7] Ingegneria Psicologia Statistica Ingegneria
## Levels: Ingegneria Lettere Medicina Psicologia Statistica

# I valori della 2° e 4° variabile per la 3° e 5° riga
my_data[c(3,5), c(2,4)]
##   names age
## 3 Carla  23
## 5 Elisa  24
```

Tuttavia, nell'utilizzo dei dataframe è più comune indicare i nomi delle variabili e le condizioni logico relazionali per selezionare i valori di interesse.

Selezione Colonne con Nomi delle Variabili

Una granade differenza tra le matrici ed i dataframe sta nel poter accedere alle colonne utilizzando l'operatore `$` ed indicando il loro nome attraverso la seguente sintassi:

```
nome_dataframe$nome_variabile
```

In questo modo accediamo direttamente a quella specifica colonna senza utilizzare indici e parentesi quadre. Ad esempio:

```
# Seleziono la variabile "names"
my_data$names
##  [1] "Alice"    "Bruno"    "Carla"    "Diego"    "Elisa"    "Fabrizio"
##  [7] "Gloria"   "Herman"   "Irene"    "Luca"

# Seleziono la variabile "gender"
my_data$faculty
##  [1] Psicologia Ingegneria Medicina   Lettere    Psicologia Lettere
##  [7] Ingegneria Psicologia Statistica Ingegneria
## Levels: Ingegneria Lettere Medicina Psicologia Statistica
```

In alternativa è possibile utilizzare la normale procedura di selezione tramite parentesi quadre indicando al cposto degli indici di colonna i nomi delle variabili desiderate. Questo ci permette di selezionare anche più variabili contemporaneamente, ad esempio:

```
# Seleziono solo la variabile "names"
my_data[ , "names"]
##  [1] "Alice"    "Bruno"    "Carla"    "Diego"    "Elisa"    "Fabrizio"
##  [7] "Gloria"   "Herman"   "Irene"    "Luca"

# Seleziono la variabile "names", "gender" e "faculty"
my_data[, c("names", "gender", "faculty")]
##       names gender    faculty
## 1     Alice      F Psicologia
## 2     Bruno      M Ingegneria
## 3     Carla      F   Medicina
## 4     Diego      M    Lettere
## 5     Elisa      F Psicologia
## 6  Fabrizio      M    Lettere
## 7    Gloria      F Ingegneria
## 8    Herman      M Psicologia
## 9     Irene      F Statistica
## 10     Luca      M Ingegneria
```

Nota come i nomi delle variabili debbano essere forniti come delle stringhe.

Selezione Righe con Condizioni Logiche

Avevamo visto precedentemete nel caso dei vettori e delle matrici come sia possibile costruire delle preposizioni logiche per selezionare solo i valori che rispettino una data condizione. Ora questa funzione si rivela particolarmente utile poichè ci permette di *interrogare* il nostro dataframe in modo semplice ed intuitivo. Utilizzando una condizione logica possiamo, infatti, *filtrare* le oservazioni che soddisfano una data condizione ed ottenere solo le informazioni di interesse.

Nella canonica sintassi `[<indice-riga>, <indice-colonna>]`, gli indici di riga vengono sostituiti con una condizione logica che ci permette di filtrare le righe e al posto degli indici di colonna vengono indicati i nomi delle variabili di interesse. Utilizzeremo quindi la seguente sintassi:

```
nome_dataframe[<condizione_logica_righe>, <nomi_variabili>]
```

Vediamo ora degli esempi di selezione:

```
# Tutti i dati di "Diego" (Id == 4)
my_data[my_data$Id == 4, ]
##   Id names gender age faculty
## 4  4 Diego      M  22 Lettere

# Tutti i dai delle ragazze
my_data[my_data$gender == "F", ]
##   Id  names gender age    faculty
## 1  1  Alice      F  22 Psicologia
```

```
## 3  3  Carla      F  23   Medicina
## 5  5  Elisa      F  24 Psicologia
## 7  7 Gloria      F  26 Ingegneria
## 9  9  Irene      F  23 Statistica

# Le facoltà dei soggeti con più di 24 anni
my_data[my_data$age > 24, c("age", "faculty")]
##   age    faculty
## 2  25 Ingegneria
## 6  35    Lettere
## 7  26 Ingegneria
```

Nota come, nel definire una condizione logica utilizzano le variabili dello stesso dataframe, sia comunque necessario indicare sempre anche il nome del dataframe. Nel caso precendente avremmo ottenuto un errore indicando semplicemente `age > 24` poichè così indichiamo l'oggetto `age` (che non esiste) e non la variabile `age` contenuta in `my_data`.

```
my_data[age > 24, c("age", "faculty")]
## Error in `[.data.frame`(my_data, age > 24, c("age", "faculty")): object 'age' not found
```

In modo analogo a quanto visto con i vettori, utlizzando la condizione `my_data$age > 24` otteniamo un vettore di valori `TRUE` e `FALSE` a seconda che la condizione sia rispettata o meno.

Utilizzando gli operatori logici **AND**(`&`) **OR**(`|`) e **NOT**(`!`) possiamo combinare più operazioni logiche insieme per ottenere indicizzazioni più complesse, ma sempre intuitive dal punto di vista della scrittura. Ad esempio, per selezionare "I soggetti tra i 20 e i 25 anni iscrittti a psicologia" eseguiremo:

```
my_data[my_data$age >= 20 & my_data$age <=25 & my_data$faculty == "Psicologia", ]
##   Id  names gender age    faculty
## 1  1  Alice      F  22 Psicologia
## 5  5  Elisa      F  24 Psicologia
## 8  8 Herman      M  20 Psicologia
```

Utilizzando questo metodo di indicizzazione possiamo apprezzare la vera potenza dei dataframe. Abbiamo, infatti, un metodo molto semplice ed intuitivo per lavorare con strutture di dati complesse formate da diverse tipologie di dati.

Approfondimento: Output Selezione

Due aspetti importanti riguardanti il risultato di una selezione sono la tipologia di ouput ottenuto e come salvarla.

Tiplogia Output

In modo analogo alle matrici, i comandi di selezione non restituiscono sempre la stessa tipologia di oggetto. Infatti, quando selezioniamo una singola variabile otteniamo come risultato un vettore mentre selezionando due o più

variabili oteniamo un dataframe.

```
# Singola variabile
class(my_data$age)
## [1] "numeric"

# Più variabili
class(my_data[ , c("names", "age")])
## [1] "data.frame"
```

Salvare Selezione

Come per tutte le altre tipologie di oggetti, le operazioni di selezione non modificano l'oggetto iniziale. Pertanto è necessario salvare il risultato della selezione se si desidera mantenere le modifiche. In questo caso, è cosigliabile creare un nuovo oggetto e non sovrascrivere l'oggetto iniziale poichè non ci permetterebbe di compiere nuove selezioni od operazioni su tutti i dati iniziali. E' buona norma quindi mantenere sempre un dataframe con la versione dei dati originali.

10.2.1 Utilizzi Avanzati Selezione

Vediamo ora alcuni utilizzi avanzati della selezione di elementi di un dataframe.

Modificare gli Elementi

In modo analogo agli altri oggetti, possiamo modificare dei valori selezionando il vecchio valore della matrice e utilizzo la funzione `<-` (o `=`) per assegnare il nuovo valore.

```
my_data[1:5, ]
##   Id names gender age    faculty
## 1  1 Alice      F  22 Psicologia
## 2  2 Bruno      M  25 Ingegneria
## 3  3 Carla      F  23   Medicina
## 4  4 Diego      M  22    Lettere
## 5  5 Elisa      F  24 Psicologia

# Sostituisco il nome "Diego" con "Davide"
my_data[4, "names"] <- "Davide"

my_data[1:5]
##    Id    names gender age    faculty
## 1   1    Alice      F  22 Psicologia
## 2   2    Bruno      M  25 Ingegneria
```

```
## 3   3    Carla      F  23   Medicina
## 4   4   Davide      M  22    Lettere
## 5   5    Elisa      F  24 Psicologia
## 6   6 Fabrizio      M  35    Lettere
## 7   7   Gloria      F  26 Ingegneria
## 8   8   Herman      M  20 Psicologia
## 9   9    Irene      F  23 Statistica
## 10 10     Luca      M  22 Ingegneria
```

Eliminare Righe o Colonne

In modo analogo agli altri oggetti, per **eliminare** delle righe (o delle colonne) da un dataframe, è necessario indicare all'interno delle parentesi quadre gli indici di riga (o di colonna) che si intende eliminare, preceduti dall'operatore `-` (*meno*). Nel caso di più righe (o colone) è possibile indicare il meno solo prima del comando `c()`.

```
# Elimino le variabili "gender" e "age"
my_data[ , c("gender", "age")]
##    gender age
## 1       F  22
## 2       M  25
## 3       F  23
## 4       M  22
## 5       F  24
## 6       M  35
## 7       F  26
## 8       M  20
## 9       F  23
## 10      M  22

# Elimino le prime 5 osservazioni
my_data[-c(1:5), ]
##    Id    names gender age    faculty
## 6   6 Fabrizio      M  35    Lettere
## 7   7   Gloria      F  26 Ingegneria
## 8   8   Herman      M  20 Psicologia
## 9   9    Irene      F  23 Statistica
## 10 10     Luca      M  22 Ingegneria

# Elimino le prime 5 osservazioni e la variabile "names"
my_data[-c(1:5), -3]
##    Id    names age    faculty
## 6   6 Fabrizio  35    Lettere
## 7   7   Gloria  26 Ingegneria
## 8   8   Herman  20 Psicologia
## 9   9    Irene  23 Statistica
## 10 10     Luca  22 Ingegneria
```

E' possibile anche escludere (ed eliminare in un certo senso) delle informazioni usando gli operatori logici in gli operatori **NOT**(`!`) e diverso da (`!=`):

```r
# Seleziono tutto tranne gli studenti di psicologia

# Modo 1 (valuto disuguaglianza)
my_data[my_data$faculty != "Psicologia", ]
##    Id    names gender age    faculty
## 2   2    Bruno      M  25 Ingegneria
## 3   3    Carla      F  23   Medicina
## 4   4   Davide      M  22    Lettere
## 6   6 Fabrizio      M  35    Lettere
## 7   7   Gloria      F  26 Ingegneria
## 9   9    Irene      F  23 Statistica
## 10 10     Luca      M  22 Ingegneria

# Modo 1 (nego l'uguaglianza)
my_data[!my_data$faculty == "Psicologia", ]
##    Id    names gender age    faculty
## 2   2    Bruno      M  25 Ingegneria
## 3   3    Carla      F  23   Medicina
## 4   4   Davide      M  22    Lettere
## 6   6 Fabrizio      M  35    Lettere
## 7   7   Gloria      F  26 Ingegneria
## 9   9    Irene      F  23 Statistica
## 10 10     Luca      M  22 Ingegneria
```

Nota come l'operazione di eliminazione sia comunque un'operazione di selezione. Pertanto è necessario salvare il risultato ottenuto se si desidera mantenere le modifiche.

> 🌵 Warning-Box: Attenzione ad Esliminare
>
> L'utilizzo dell'operatore `-` è sempre in qualche modo pericoloso, sopratutto se l'oggetto che viene creato (o sovrascritto) viene poi utilizzato in altre operazioni. Eliminare delle informazioni, tranne quando è veramente necessario, non è mai una buona cosa. Se dovete selezionare una parte dei dati è sempre meglio creare un nuovo dataframe (o un nuovo oggetto in generale) e mantendere una versione di quello originale sempre disponibile.

Esercizi

Facendo riferimento ai dataframe `data_long` e `data_wide` precedentemente creati, esegui i seguenti esercizi (soluzioni):

1. Utilizzando gli **indici numerici** di riga e di colonna seleziona i dati del soggetto `subj_2` riguardanti le variabili `item` e `response` dal DataFrame `data_long`.
2. Compi la stessa selezione dell'esercizio precedente usando però questa volta una condizione logica per gli indici di riga e indicando direttamente il nome delle variabili per gli indici di colonna.

3. Considerando il DataFrame `data_wide` seleziona le variabili `Id` e `gender` dei soggetti che hanno risposto 1 alla variabile `item_1`.
4. Considerando il DataFrame `data_long` seleziona solamente i dati riguardanti le ragazze con etè superiore ai 20 anni.
5. Elimina dal DataFrame `data_long` le osservazioni riguardanti il soggetto `subj_2` e la variabile `"gender"`.

10.3 Funzioni ed Operazioni

Vediamo ora alcune funzioni frequentemente usate e le comuni operazioni eseguite con i dataframe (vedi Tabella 10.3).

Table 10.3: Funzioni e operazioni con dataframe

Funzione	Descrizione
`nrow(nome_df)`	Numero di osservazioni del dataframe
`ncol(nome_df)`	Numero di variabili del dataframe
`colnames(nome_df)`	Nomi delle colonne del dataframe
`rownames(nome_df)`	Nomi delle righe del dataframe
`nome_df <- cbind(nome_df, dati)` `nome_df$nome_var <- dati`	Aggiungi una nuova variabile al dataframe (deve avere lo stesso numero di r
`nome_df <- rbind(nome_df, dati)`	Aggiungi delle osservazioni (i nuovi dati devono essere coerenti con la strutt
`head(nome_df)`	Prime righe del dataframe
`tail(nome_df)`	Ultime righe del dataframe
`str(nome_df)`	Struttura del dataframe
`summary(nome_df)`	Summary del dataframe

Descriviamo ora nel dettaglio alcuni particolari utilizzi, considrando come esempio una versione ridotta del dataframe `my_data` precedentemente creata.

```
data_short <- my_data[1:5, ]
data_short
##   Id  names gender age    faculty
## 1  1  Alice      F  22 Psicologia
## 2  2  Bruno      M  25 Ingegneria
## 3  3  Carla      F  23   Medicina
## 4  4 Davide      M  22    Lettere
## 5  5  Elisa      F  24 Psicologia
```

10.3.1 Attributi di un Dataframe

Abbiamo visto nel Capitolo 8.1 che gli oggetti in R possiedono quelli che sono definiti *attributi* ovvero delle utili informazioni riguardanti l'oggetto stesso, una sorta di *metadata*. Vediamo ora, in modo analogo alle matrici, come valutare la dimensione di un dataframe e i nomi delle righe e delle colonne.

Dimensione

Ricordiamo che un dataframe è un oggetto **bidimensionale** formato da righe e colonne. Per ottenere il numero di righe e di colonne di un dataframe, possiamo usare rispettivamente i comandi `nrow()` e `ncol()`.

```
# Numero di righe
nrow(my_data)
## [1] 10

# Numero di colonne
ncol(my_data)
## [1] 5
```

In alternativa, come per le matrici, è possibile usare la funzione `dim()`che restituisce un vettore con due valori dove il primo rappresenta il numero di righe e il secondo il numero di colonne.

Nomi Righe e Colonne

In modo simile alle matrici, è possibile accedere ai nomi delle righe e delle colonne utilizzando rispettivamente le funzioni `rownames()` e `colnames()`. Di default il dataframe richiede dei nomi solo alle colonne mentre il alle righe viene assegnato un nome in accordo con l'indice di riga. Tuttavia, è possibile anche nominare le righe con valori arbitrari sebbene sia una funzione raramente utilizzata.

```
# Controllo i nomi attuali
rownames(data_short)
## [1] "1" "2" "3" "4" "5"
colnames(data_short)
## [1] "Id"      "names"   "gender"  "age"     "faculty"
```

Per impostare i nomi di righe e/o colonne, sarà quindi necessario assegnare a `rownames(nome_dataframe)` e `colnames(nome_dataframe)` un vettore di caratteri della stessa lunghezza della dimensione che stiamo rinominando.

```
# Assegnamo i nomi alle righe
rownames(data_short) <- paste0("Subj_", 1:nrow(data_short))

# Assegno i nomi alle colonne
colnames(data_short) <- c("Id", "Nome", "Genere", "Eta", "Facolta")

data_short
##        Id   Nome Genere Eta    Facolta
## Subj_1  1  Alice      F  22 Psicologia
## Subj_2  2  Bruno      M  25 Ingegneria
## Subj_3  3  Carla      F  23   Medicina
## Subj_4  4 Davide      M  22    Lettere
## Subj_5  5  Elisa      F  24 Psicologia
```

Infine, nota come la funzione `names()` nel caso dei dataframe sia analoga alla funzione `colnames()` e sia possibile usare il vaore `NULL` per eliminare ad esempio i nomi delle righe.

```r
names(data_short)
## [1] "Id"      "Nome"    "Genere"  "Eta"     "Facolta"

rownames(data_short) <- NULL
data_short
##   Id   Nome Genere Eta    Facolta
## 1  1  Alice      F  22 Psicologia
## 2  2  Bruno      M  25 Ingegneria
## 3  3  Carla      F  23   Medicina
## 4  4 Davide      M  22    Lettere
## 5  5  Elisa      F  24 Psicologia
```

10.3.2 Unire Dataframe

In modo analogo alle matrici è possibile unire più dataframe utilizzando le funzioni `cbind()` e `rbind()` per cui valgono gli stesssi accorgimenti riguardanti la dimensione rispettivamente di righe e colonne. Tuttavia, nel caso dei dataframe è possibile creare una nuova colonna anche utilizzando l'operatore `$`. Descriviamo ora in modo accurato i differenti utilizzi.

`dataframe$name <- new_var`

Con la scrittura `dataframe$name <- new_var` aggiungiamo al dataframe in oggetto una nuova colonna chiamata `name` che prende i valori all'interno di `new_var`. Sarà necessario che la nuova variabile abbia lo stesso numero di valori delle righe del dataframe.

```r
# Aggiungiamo la colonna "media"
data_short$Media <- c(27.5, 23.6, 28.3, 29.2, 24.8)

# Equivalente a
media_voti <- c(27.5, 23.6, 28.3, 29.2, 24.8)
data_short$Media <- media_voti

data_short
##   Id   Nome Genere Eta    Facolta Media
## 1  1  Alice      F  22 Psicologia  27.5
## 2  2  Bruno      M  25 Ingegneria  23.6
## 3  3  Carla      F  23   Medicina  28.3
## 4  4 Davide      M  22    Lettere  29.2
## 5  5  Elisa      F  24 Psicologia  24.8
```

`cbind()`

Con la funzione `cbind()` possiamo aggiungere una o più variabili al nostro dataframe. Nota come a differenza dell'utilizzo dell'operatore `$`,usando `cbind()` il risultato non venga automaticamente salvato ma sia necessario assegnare l'operazione ad un nuovo oggetto `dataframe <- cbind(dataframe, new_var)`. In quest'ultimo caso il nome della colonna sarà `new_var`. Se vogliamo anche rinominare la colonna possiamo usare la sintassi `cbind(dataframe, "nome" = new_var)` oppure chiamare l'oggetto direttamente con il nome desiderato:

```
# aggiungo la variabile Numero_esami
numero_esami <- c(12, 14, 13, 10, 8)

cbind(data_short, numero_esami) # senza specificare il nome
##   Id   Nome Genere Eta    Facolta Media numero_esami
## 1  1  Alice      F  22 Psicologia  27.5           12
## 2  2  Bruno      M  25 Ingegneria  23.6           14
## 3  3  Carla      F  23   Medicina  28.3           13
## 4  4 Davide      M  22    Lettere  29.2           10
## 5  5  Elisa      F  24 Psicologia  24.8            8

cbind(data_short, "N_esami" = numero_esami) # specificando anche il nome
##   Id   Nome Genere Eta    Facolta Media N_esami
## 1  1  Alice      F  22 Psicologia  27.5      12
## 2  2  Bruno      M  25 Ingegneria  23.6      14
## 3  3  Carla      F  23   Medicina  28.3      13
## 4  4 Davide      M  22    Lettere  29.2      10
## 5  5  Elisa      F  24 Psicologia  24.8       8
```

Anche in questo caso sarà necessario che le nuove variabili abbia lo stesso numero di valori delle righe del dataframe.

rbind()

Leggermente più complessa (e inusuale) è l'aggiunta di righe ad un dataframe. Al contrario della matrice che di base non aveva nomi per le colonne e solo numeri o stringhe come tipologia di dato, per combinare per riga due dataframe dobbiamo avere la stessa struttura. Ovvero:

- Lo stesso numero di colonne (come per le matrici)
- Lo stesso nome delle colonne

```
data_short
##   Id   Nome Genere Eta    Facolta Media
## 1  1  Alice      F  22 Psicologia  27.5
## 2  2  Bruno      M  25 Ingegneria  23.6
## 3  3  Carla      F  23   Medicina  28.3
## 4  4 Davide      M  22    Lettere  29.2
## 5  5  Elisa      F  24 Psicologia  24.8

# Nuovo dataset con le stesse colonne ma diverso nome
new_row <- data.frame(
  Id = 6,
  Nome = "Marta",
  Sex = "F",       # Sex invece di Genere
  Eta = 44,
  Facolta = "Filosofia",
  Media = 28.7
)
```

```
new_row
##   Id  Nome Sex Eta   Facolta Media
## 1  6 Marta   F  44 Filosofia  28.7

rbind(data_short, new_row) # Errore
## Error in match.names(clabs, names(xi)): names do not match previous names

# Nuovo dataset con le stesse colonne con il nome corretto
new_row <- data.frame(
  Id = 6,
  Nome = "Marta",
  Genere = "F",
  Eta = 44,
  Facolta = "Filosofia",
  Media = 28.7
)

rbind(data_short, new_row)
##   Id   Nome Genere Eta    Facolta Media
## 1  1  Alice      F  22 Psicologia  27.5
## 2  2  Bruno      M  25 Ingegneria  23.6
## 3  3  Carla      F  23   Medicina  28.3
## 4  4 Davide      M  22    Lettere  29.2
## 5  5  Elisa      F  24 Psicologia  24.8
## 6  6  Marta      F  44  Filosofia  28.7
```

Anche in questo caso sarà necessario salvare il risultato ottenuto per mantenere i cambiamenti.

10.3.3 Informazioni Dataframe

Vediamo infine alcune funzioni molto comuni usate per ottenere informazioni riassuntive riguardo ai dati contenuti in un dataframe:

- `head()` (o `tail()`) ci permete di visualizzare le prime (o le ulitme righe del nostro dataframe)

```
head(my_data)
##   Id    names gender age    faculty
## 1  1    Alice      F  22 Psicologia
## 2  2    Bruno      M  25 Ingegneria
## 3  3    Carla      F  23   Medicina
## 4  4   Davide      M  22    Lettere
## 5  5    Elisa      F  24 Psicologia
## 6  6 Fabrizio      M  35    Lettere
```

- `str()` ci permete di valutare la struttura del dataset ottenendo utili informazini quali in numero di osservazioni, il numero di variabili e la ipologia di variabili

```
str(my_data)
## 'data.frame':    10 obs. of  5 variables:
##  $ Id     : int  1 2 3 4 5 6 7 8 9 10
##  $ names  : chr  "Alice" "Bruno" "Carla" "Davide" ...
##  $ gender : Factor w/ 2 levels "F","M": 1 2 1 2 1 2 1 2 1 2
##  $ age    : num  22 25 23 22 24 35 26 20 23 22
##  $ faculty: Factor w/ 5 levels "Ingegneria","Lettere",..: 4 1 3 2 4 2 1 4 5 1
```

- `summary()` ci permete avere delle informazioni riassuntive delle variabili a seconda della loro tipologia

```
summary(my_data)
##        Id           names           gender      age             faculty
##  Min.   : 1.00   Length:10          F:5    Min.   :20.00   Ingegneria:3
##  1st Qu.: 3.25   Class :character   M:5    1st Qu.:22.00   Lettere   :2
##  Median : 5.50   Mode  :character          Median :23.00   Medicina  :1
##  Mean   : 5.50                             Mean   :24.20   Psicologia:3
##  3rd Qu.: 7.75                             3rd Qu.:24.75   Statistica:1
##  Max.   :10.00                             Max.   :35.00
```

Esercizi

Facendo riferimento ai dataframe `data_long` e `data_wide` (soluzioni):

1. Aggiungi sia al DataFrame `data_wide` che `data_long` la variabile numerica `"memory_pre"`.

```
##       Id memory_pre
## 1 subj_1          3
## 2 subj_2          2
## 3 subj_3          1
```

2. Aggiungi sia al DataFrame `data_wide` che `data_long` la variabile categoriale `"gruppo"`.

```
##       Id      gruppo
## 1 subj_1 trattamento
## 2 subj_2 trattemento
## 3 subj_3   controllo
```

3. Aggiungi al DataFrame `data_wide` i dati del soggetto `subj_4` e `subj_5`.

```
##       Id age gender item_1 item_2 item_3 memory_pre      gruppo
## 1 subj_4  25      F      1      0      2          1 trattemento
## 2 subj_5  22      M      1      1      0          3   controllo
```

4. Considerando il DataFrame `datawide` calcola la variabile `"memory_post"` data dalla somma degli item.
5. Considerando il DataFrame `data_wide` cambia i nomi delle variabili `item_1`, `item_2` e `item_3` rispettivamente in `problem_1`, `problem_2` e `problem_3`.

Chapter 11

Liste

Le **liste** sono uno degli oggetti più versatili e utili in R. Possiamo pensare alle liste come a dei grandi raccoglitori di altri oggetti. La caratteristica principale delle liste è proprio questa, ovvero la capacità di contenere **tipologie diverse di oggetti** al loro interno come ad esempio vettori, dataframe, matrici e anche altre liste.

A differenza di dataframe e matrici dove gli elementi sono tra loro in relazione, gli elementi di una lista sono completamente indipendenti. Una `lista`, infatti, può contenere oggetti completamente diversi tra loro sia per tipologia che per dimensioni, senza che vi sia alcuna relazione o vincolo.

Un modo utile per immaginarsi una lista (vedi Figura 11.1) è pensare ad un corridoio di un albergo dove ogni porta conduce ad una stanza diversa per caratteristiche, numero di elementi e così via. E' importante notare come gli elementi siano disposti in un preciso ordine e quindi possano essere identificati tramite il loro **indice di posizione**, in modo analogo a quanto visto con gli elementi di un vettore.

Figure 11.1: Esempio concettuale di una lista

Da un punto di vista pratico la lista è un oggetto molto semplice, simile ad un vettore, e molte delle sue caratteristiche sono in comune con gli altri oggetti che abbiamo già affrontato. Tuttavia, la

difficltà principale risiede nella sua indicizzazione poichè, data la loro grande versatilità, le strutture delle liste possono ragiungere notevoli livelli di complessità.

Vediamo ora come creare una lista e i diversi modi per sselezionare i suoi elementi. Come vedremo ci sono molte somiglianze nell'utilizzo dei dataframe e delle matrici. Quando necessario, si farà riferimento al capitolo precedente per far notare aspetti in comune e differenze tra queste due strutture di dati.

11.1 Creazione di Liste

Il comando per creare una lista è `list()`:

```
nome_lista <- list(
  nome_oggetto_1 = oggetto_1,
  ...,
  nome_oggetto_n = oggetto_n)
```

Nonostante il parametro `nome_oggetto_x` non sia necessario, come vedremo è assolutamente consigliato rinominare tutti gli elementi della lista per agevolare l'indicizzazione. Se non nominiamo gli elementi, infatti, questi saranno identificati solo tramite il loro indice di posizione, ovvero un numero progressivo $1...n$ esattamente come nel caso dei vettori. Questo rende meno intutiva la successiva selezione degli elementi.

Quindi, se nel nostro workspace abbiamo degli oggetti diversi come una semplice `variabile`, un `vettore` e un `dataframe`, possiamo raccoglieere tutti questi elementi dentro un'unica lista. Ad esempio:

```
# Una variabile
my_value <- "Prova"

# Un vettore
my_vector <- c(1, 3, 5, 6, 10)

# Un dataframe
my_data <- data.frame(id = 1:6,
                      gender = rep(c("m", "f"), times = 3),
                      y = 1:6)

# Creiamo la lista
my_list <- list(elemento_1 = my_value,
                elemento_2 = my_vector,
                elemento_3 = my_data)

my_list
## $elemento_1
## [1] "Prova"
##
## $elemento_2
## [1]  1  3  5  6 10
##
```

```
## $elemento_3
##   id gender y
## 1  1      m 1
## 2  2      f 2
## 3  3      m 3
## 4  4      f 4
## 5  5      m 5
## 6  6      f 6
```

Esercizi

Considerando gli oggetti creati nei precedenti capitoli, esegui i seguenti esercizi (soluzioni):

1. Crea la lista `esperimento_1` contenente:
    - DataFrame `data_wide`
    - la matrice `A`
    - il vettore `x`
    - la variabile `info = "Prima raccolta dati"`
2. Crea la lista `esperimento_2` contenente:
    - DataFrame `data_long`
    - la matrice `C`
    - il vettore `y`
    - la variabile `info = "Seconda raccolta dati"`

11.2 Selezione Elementi

Come dicevamo in precedenza, ogni elemento di una lista possiende il suo **indice di posizione**, ovvero un valore numerico progressivo in modo del tutto analogo a quanto visto per gli elementi di un vettore. Tuttavia, c'è un'importante differenza per quanto riguarda la modalità di selezione in base al tipo di oggetto che si vuole ottenere come risultato. Infatti, mentre con i vettori usavamo le singole parentesi quadre `my_vector[i]` per accedere direttamente all'elemento alla posizione i, con le liste abbiamo due alternative:

- `my_list[i]` - utilizzando le **singole parentesi quadre** (`[i]`) otteniamo come risultato un oggetto di tipo lista con all'interno l'elemento alla posizione i. Questo non ci permette, tuttavia, di accedere direttamente ai suoi valori.
- `my_list[[i]]` - utilizzando le **doppie parentesi quadre** (`[[i]]`) estraiamo dalla lista l'elemento alla posizione i e come risultato otteniamo l'oggetto stesso. Questo ci permette quindi di accedere direttamente ai suoi valori.

Vediamo la differenza nel seguente esempio:

```
my_list
## $elemento_1
## [1] "Prova"
##
```

```
## $elemento_2
## [1]  1  3  5  6 10
##
## $elemento_3
##   id gender y
## 1  1      m 1
## 2  2      f 2
## 3  3      m 3
## 4  4      f 4
## 5  5      m 5
## 6  6      f 6

# Indicizzazione [ ]
my_list[2]
## $elemento_2
## [1]  1  3  5  6 10
class(my_list[2]) # una lista
## [1] "list"

# Indicizzazione [[]]
my_list[[2]]
## [1]  1  3  5  6 10
class(my_list[[2]]) # un vettore
## [1] "numeric"
```

Questa differenza nel risultato ottenuto usando le **singole parentesi quadre** (`[ ]`) o le **doppie parentesi quadre** (`[[ ]]`) è molto importante perché influirà sulle successive operazioni che potremmo compiere. Ricordiamo che nel primo caso (`[ ]`) ottenimo una lista con i soli elementi selezionati, mentre nel secondo caso (`[[ ]]`) accediamo direttamente all'oggetto selezionato.

Questa distinzione diventa chiara applicando una funzione generica allo stesso elemento indicizzato in modo diverso oppure usando la funzione `str()` per capire la struttura. Vediamo come solo accedendo direttamente all'elemento possiamo eseguire le normali operazioni, mentre, indicizzando con una sola parentesi, l'oggetto ottenuto è una lista con un singolo elemento.

```
# Applichiamo la media al vettore `elemento_2` indicizzato con 1 o 2 parentesi
mean(my_list[2])
## Warning in mean.default(my_list[2]): argument is not numeric or logical:
## returning NA
## [1] NA
mean(my_list[[2]])
## [1] 5

# Vediamo la struttura
str(my_list[2])
## List of 1
##  $ elemento_2: num [1:5] 1 3 5 6 10
str(my_list[[2]])
##  num [1:5] 1 3 5 6 10
```

Approfondimento: [ ] vs [[ ]]

Il diverso tipo di selezione ottenuto con l'utilizzo delle singole o doppie parentesi quadre è definito nel seguente modo:

- **singole parentesi quadre** (`[ ]`) - restituisce un oggetto della stessa classe (i.e., tipologia) dell'oggetto iniziale
- **doppie parentesi quadre** (`[[ ]]`) - estrae un elemento dall'oggetto iniziale restituendo un oggetto non necessariamente della stessa classe (i.e., tipologia)

Vediamo quindi come sia possibile utilizzare le doppie parentesi anche con i vettori e i dataframe ma in questo caso il risultato non differeisce dalla normale procedura di selezione.

```
# Vettori
my_vector[2]
## [1] 3
my_vector[[2]]
## [1] 3

# Dataframe
my_data[, 2]
## [1] "m" "f" "m" "f" "m" "f"
my_data[[2]] # la selezione è posibile solo sulle colonne
## [1] "m" "f" "m" "f" "m" "f"
```

Nota infine come l'uso di singole parentesi quadre (`[ ]`) permette di selezionare più elementi contemporaneamente, mentre le doppie parentesi quadre (`[[ ]]`) permettono di estrarre solo un elemento alla volta

```
my_list[c(1,2)]
## $elemento_1
## [1] "Prova"
##
## $elemento_2
## [1]  1  3  5  6 10

my_list[[c(1,2)]]
## Error in my_list[[c(1, 2)]]: subscript out of bounds
```

Selezione Operatore $

In alternativa all'uso delle **doppie parentesi quadre** (`[[ ]]`) è possibile, in modo analogo ai dataframe, accedere agli elementi di una lista utilizzando l'operatore `$` ed indicando il loro nome:

- `my_list$nome_elemento` - l'operatore `$` ci permette di accedere direttamente all'oggetto desiderato.

Vediamo alcuni esempi riprendendo la lista `my_list` create precedentemente.

```
# Selezionare "elemento_1"
my_list$elemento_1
## [1] "Prova"

# Selezionare "elemento_3"
my_list$elemento_3
##   id gender y
## 1  1      m 1
## 2  2      f 2
## 3  3      m 3
## 4  4      f 4
## 5  5      m 5
## 6  6      f 6
```

Nota come il nome degli elementi possa essere usato anche con le parentesi quadre.

```
## $elemento_1
## [1] "Prova"
##
## $elemento_2
## [1]  1  3  5  6 10
```

Utilizzo Elementi e Successive Selezioni

Una volta che abbiamo estratto un elemento da una lista, è possibile utilizzare l'oggetto nel modo che preferiamo. Possiamo sia assegnare l'elemento ad nuovo oggetto da utilizzare successvamente, oppure eseguire le funzioni o altre operazioni generiche direttamente sul comando della selezione.

```
# Media dei valori dell' "elemento_2"

# Assegno l'oggetto
my_values <- my_list$elemento_2
mean(my_values)
## [1] 5

# Calcolo direttamente
mean(my_list$elemento_2)
## [1] 5
```

Chiaramente le operazione che possiamo svolgere, come ad esempio ulteriori selezioni, dipendono dalla specifica tipologia e struttura dell'oggetto selezionato.

```
# ---- Seleziono il primo valore dell'oggetto "elemento_2" ----

my_list$elemento_2
## [1]  1  3  5  6 10

my_list$elemento_2[1]
## [1] 1
my_list[[2]][1] # equivalente a alla precedente
## [1] 1

#---- Seleziono la colonna "gender" dell'oggetto "elemento_3" ----

my_list$elemento_3$gender
## [1] "m" "f" "m" "f" "m" "f"

# Altre scritture equivalenti
my_list[[3]]$gender
## [1] "m" "f" "m" "f" "m" "f"
my_list[[3]][, 2]
## [1] "m" "f" "m" "f" "m" "f"
my_list[[3]][, "gender"]
## [1] "m" "f" "m" "f" "m" "f"
```

11.2.1 Utilizzi Avanzati Selezione

Vediamo ora alcuni utilizzi avanzati della selezione di elementi di un dataframe.

Modificare gli Elementi

In modo analogo agli altri oggetti, possiamo modificare dei valori selezionando il vecchio elemento della lista e utilizzo la funzione `<-` (o `=`) per assegnare il nuovo elemento. Nota come in questo caso sia possibile uttilizzare sia le singole parentesi quadre (`[ ]`) che le doppie parentesi quadre (`[[ ]]`).

```
my_list
## $elemento_1
## [1] "Prova"
##
## $elemento_2
## [1]  1  3  5  6 10
##
## $elemento_3
##   id gender y
## 1  1      m 1
## 2  2      f 2
## 3  3      m 3
```

```
## 4  4      f 4
## 5  5      m 5
## 6  6      f 6

# sostituiamo il primo elemento
my_list[1] <- "Un nuovo elemento"

# sostituiamo il secondo elemento
my_list[[2]] <- "Un altro nuovo elemento"

my_list
## $elemento_1
## [1] "Un nuovo elemento"
##
## $elemento_2
## [1] "Un altro nuovo elemento"
##
## $elemento_3
##   id gender y
## 1  1      m 1
## 2  2      f 2
## 3  3      m 3
## 4  4      f 4
## 5  5      m 5
## 6  6      f 6
```

Eliminare Elementi

In modo analogo agli altri oggetti, per **eliminare** degli elementi da una lista, è necessario indicare all'interno delle parentesi quadre gli indici di posizione degli elementi che si intende eliminare, preceduti dall'operatore `-` (*meno*). In questo caso è necessario l'utilizzo delle singole parentesi quadre (`[ ]`).

```
# Elimino il secondo elemento
my_list[-2]
## $elemento_1
## [1] "Un nuovo elemento"
##
## $elemento_3
##   id gender y
## 1  1      m 1
## 2  2      f 2
## 3  3      m 3
## 4  4      f 4
## 5  5      m 5
## 6  6      f 6
```

Ricorda come l'operazione di eliminazione sia comunque un'operazione di selezione. Pertanto è necessario salvare il risultato ottenuto se si desidera mantenere le modifiche.

Esercizi

Esegui i seguenti esercizi (soluzioni)

1. Utilizzando gli **indici numerici** di posizione selziona i dati dei soggetti `subj_1` e `subj_4` riguardanti le variabili `age`,`gender` e `gruppo` dal DataFrame `data_wide` contenuto nella lista `esperimento_1`.
2. Compi la stessa selezione dell'esercizio precedente usando però questa volta il nome dell'oggetto per selezionare il DateFrame dalla lista.
3. Considerando la lista `esperimento_2` seleziona gli oggetti `data_long`, `y` e `info`
4. Cambia i nomi degli oggetti contenuti nella lista `esperimento_2` rispettivamente in `"dati_esperimento"`, `"matrice_VCV"`, `"codici_Id"` e `"note"`

11.3 Funzioni ed Operazioni

Vediamo ora alcune funzioni frequentemente usate e le comuni operazioni eseguite con le liste (vedi Tabella 11.1).

Table 11.1: Funzioni ed operazioni con liste

Funzione	Descrizione
`length(nome_df)`	Numero di elementi nella lista
`names(nome_df)`	Nomi degli elementi della lista
`nome_list$nome_obj <- oggetto`	Aggiungi un nuovo elemento alla lista
`c(nome_df)`	Unire più liste
`unlist(nome_df)`	Ottieni un vettori di tutti gli elementi
`str(nome_df)`	Struttura del dataframe
`summary(nome_df)`	Summary del dataframe

Descriviamo ora nel dettaglio alcuni particolari utilizzi, considrando come esempio la lista `my_list` qui definita. Nota che i nomi degli elemnti sono stati omessi volutamente.

```
my_list <- list(my_value,
                my_vector,
                my_data)
```

11.3.1 Attributi di una Lista

Anche le liste come gli altri oggetti, hanno degli *attributi* ovvero delle utili informazioni riguardanti l'oggetto stesso. Vediamo ora come valutare la dimensione di una lista e i nomi dei suoi elementi.

Dimensione

Per valutare la dimensione di una lista, ovvero il numero di elementi che contiene, possiamo utilizzare la funzione `length()`

```r
# Numero elementi
length(my_list)
## [1] 3
```

Nomi Elementi

Per accedere ai nomi degli elementi di una lista, è possibile utilizzare la funzione `names()`. Se i nomi non sono stati specificati al momento della creazione, otterremo il valore `NULL`.

```r
# Controllo i nomi attuali
names(my_list)
## NULL
```

Per impostare i nomi degli elementi, sarà quindi necessario assegnare a `names(my_list)` un vettore di caratteri con i nomi di ciascun elemento.

```r
# Assegnamo i nomi
names(my_list) <- c("Variabile", "Vettore", "Dataframe")

my_list
## $Variabile
## [1] "Prova"
##
## $Vettore
## [1]  1  3  5  6 10
##
## $Dataframe
##   id gender y
## 1  1      m 1
## 2  2      f 2
## 3  3      m 3
## 4  4      f 4
## 5  5      m 5
## 6  6      f 6
```

11.3.2 Unire Liste

Per aggiungere elementi ad una lista è possibile sia creare un nuovo elemento con l'operatore `$`, in modo annalogo ai datafame, sia combinare più liste con la funzione `c()`.

`my_list$name <- new_obj`

Con la scrittura `my_list$name <- new_obj` aggiungiamo alla lista in oggetto un nuovo elemento di cui dobbiamo specificare il nome e gli assegnamo l'oggetto `new_obj`.

```r
# Aggiungiamo un nuovo elemento
my_list$new_obj <- "Un nuovo elemento"
```

```
my_list
## $Variabile
## [1] "Prova"
##
## $Vettore
## [1]  1  3  5  6 10
##
## $Dataframe
##   id gender y
## 1  1      m 1
## 2  2      f 2
## 3  3      m 3
## 4  4      f 4
## 5  5      m 5
## 6  6      f 6
##
## $new_obj
## [1] "Un nuovo elemento"
```

c()

Con la funzione `c()` possiamo combinare più liste insieme. Nota come sia necessario che i nuovi oggetti che vogliamo includere siano effettivamente una lista, altrimenti potremmo non ottenere il risultato desiderato:

```
# ERRORE: combino una lista con un vettore
new_vector <- 1:3
c(my_list, new_vector)
## $Variabile
## [1] "Prova"
##
## $Vettore
## [1]  1  3  5  6 10
##
## $Dataframe
##   id gender y
## 1  1      m 1
## 2  2      f 2
## 3  3      m 3
## 4  4      f 4
## 5  5      m 5
## 6  6      f 6
##
## $new_obj
## [1] "Un nuovo elemento"
##
## [[5]]
## [1] 1
##
## [[6]]
```

```
## [1] 2
##
## [[7]]
## [1] 3

# CORRETTO: combino una lista con un'altra lista
c(my_list, list(new_vector = 1:3))
## $Variabile
## [1] "Prova"
##
## $Vettore
## [1]  1  3  5  6 10
##
## $Dataframe
##   id gender y
## 1  1      m 1
## 2  2      f 2
## 3  3      m 3
## 4  4      f 4
## 5  5      m 5
## 6  6      f 6
##
## $new_obj
## [1] "Un nuovo elemento"
##
## $new_vector
## [1] 1 2 3
```

In questo caso sarà necessario salvare il risultato ottenuto per mantenere i cambiamenti.

11.3.3 Informazioni Lista

Vediamo infine alcune funzioni molto comuni usate per ottenere informazioni riassuntive riguardo gli elementi di una lista:

- `str()` ci permete di valutare la struttura della lista ottenendo utili informazini quali in numero di elementi e la loro tipologia

```
str(my_list)
## List of 4
##  $ Variabile: chr "Prova"
##  $ Vettore  : num [1:5] 1 3 5 6 10
##  $ Dataframe:'data.frame':   6 obs. of  3 variables:
##   ..$ id    : int [1:6] 1 2 3 4 5 6
##   ..$ gender: chr [1:6] "m" "f" "m" "f" ...
##   ..$ y     : int [1:6] 1 2 3 4 5 6
##  $ new_obj  : chr "Un nuovo elemento"
```

- `summary()` ci permete avere delle informazioni riassuntive degli elementi tuttavia risulta poco utlie

```
summary(my_list)
##           Length Class      Mode
## Variabile 1      -none-     character
## Vettore   5      -none-     numeric
## Dataframe 3      data.frame list
## new_obj   1      -none-     character
```

11.4 Struttura Nested

Al contrario dei vettori che si estendono in *lunghezza* o dei dataframe/matrici che sono caratterizzati da righe e colonne, la peculiarità della lista (oltre alla lunghezza come abbaimo visto) è il concetto di **profondità**. Infatti una lista può contenere al suo interno una o più liste di fatto creando una **struttura nidificata molto complessa**. Nonostante la struttura più complessa, il principio di indicizzazione e creazione è lo stesso. La Figura 11.2 rappresenta l'idea di una lista nidificata (o nested):

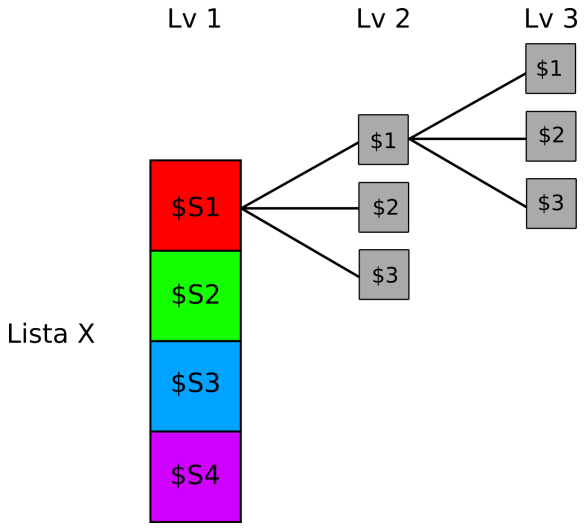


Figure 11.2: Rappresentazione concettuale di una lista nested

Per fare un esempio pratico, immaginiamo che n soggetti abbiamo eseguito k diversi esperimenti e vogliamo organizzare questa struttura di dati in R in modo efficace e ordinato. Possiamo immaginare una lista `esperimenti` che contiene:

- Ogni soggetto come una lista, chiamata `s1, s2, ..., sn`
- Ogni elemento della lista-soggetto è un dataframe per lo specifico esperimento chiamato `exp1, exp2, ..., expn`

```
# Per comodità ripetiamo lo stesso esperimento e lo stesso soggetto

# Esperimento generico
exp_x <- data.frame(
  id = 1:10,
  gender = rep(c("m", "f"), each = 5),
```

```
  y = 1:10
)

# Soggetto generico
sx <- list(
  exp1 = exp_x,
  exp2 = exp_x,
  exp3 = exp_x
)

# Lista Completa
esperimenti <- list(
  s1 = sx,
  s2 = sx,
  s3 = sx
)

str(esperimenti)
## List of 3
##  $ s1:List of 3
##   ..$ exp1:'data.frame': 10 obs. of  3 variables:
##   .. ..$ id    : int [1:10] 1 2 3 4 5 6 7 8 9 10
##   .. ..$ gender: chr [1:10] "m" "m" "m" "m" ...
##   .. ..$ y     : int [1:10] 1 2 3 4 5 6 7 8 9 10
##   ..$ exp2:'data.frame': 10 obs. of  3 variables:
##   .. ..$ id    : int [1:10] 1 2 3 4 5 6 7 8 9 10
##   .. ..$ gender: chr [1:10] "m" "m" "m" "m" ...
##   .. ..$ y     : int [1:10] 1 2 3 4 5 6 7 8 9 10
##   ..$ exp3:'data.frame': 10 obs. of  3 variables:
##   .. ..$ id    : int [1:10] 1 2 3 4 5 6 7 8 9 10
##   .. ..$ gender: chr [1:10] "m" "m" "m" "m" ...
##   .. ..$ y     : int [1:10] 1 2 3 4 5 6 7 8 9 10
##  $ s2:List of 3
##   ..$ exp1:'data.frame': 10 obs. of  3 variables:
##   .. ..$ id    : int [1:10] 1 2 3 4 5 6 7 8 9 10
##   .. ..$ gender: chr [1:10] "m" "m" "m" "m" ...
##   .. ..$ y     : int [1:10] 1 2 3 4 5 6 7 8 9 10
##   ..$ exp2:'data.frame': 10 obs. of  3 variables:
##   .. ..$ id    : int [1:10] 1 2 3 4 5 6 7 8 9 10
##   .. ..$ gender: chr [1:10] "m" "m" "m" "m" ...
##   .. ..$ y     : int [1:10] 1 2 3 4 5 6 7 8 9 10
##   ..$ exp3:'data.frame': 10 obs. of  3 variables:
##   .. ..$ id    : int [1:10] 1 2 3 4 5 6 7 8 9 10
##   .. ..$ gender: chr [1:10] "m" "m" "m" "m" ...
##   .. ..$ y     : int [1:10] 1 2 3 4 5 6 7 8 9 10
##  $ s3:List of 3
##   ..$ exp1:'data.frame': 10 obs. of  3 variables:
##   .. ..$ id    : int [1:10] 1 2 3 4 5 6 7 8 9 10
##   .. ..$ gender: chr [1:10] "m" "m" "m" "m" ...
##   .. ..$ y     : int [1:10] 1 2 3 4 5 6 7 8 9 10
##   ..$ exp2:'data.frame': 10 obs. of  3 variables:
```

```
##   .. ..$ id    : int [1:10] 1 2 3 4 5 6 7 8 9 10
##   .. ..$ gender: chr [1:10] "m" "m" "m" "m" ...
##   .. ..$ y     : int [1:10] 1 2 3 4 5 6 7 8 9 10
##   ..$ exp3:'data.frame': 10 obs. of  3 variables:
##   .. ..$ id    : int [1:10] 1 2 3 4 5 6 7 8 9 10
##   .. ..$ gender: chr [1:10] "m" "m" "m" "m" ...
##   .. ..$ y     : int [1:10] 1 2 3 4 5 6 7 8 9 10
```

Ora la struttura è molto più complessa, ma se abbiamo chiara la Figura 11.2 e l'indicizzazione per le liste precedenti, accedere agli elementi della lista `esperimenti` è semplice ed intuitivo. Se vogliamo accedere al dataset del `soggetto 3` che riguarda `l'esperimento 2`:

```
# Con indici numerici
esperimenti[[3]][[2]] # elemento 3 (una lista) e poi l'elemento 2
##    id gender  y
## 1   1      m  1
## 2   2      m  2
## 3   3      m  3
## 4   4      m  4
## 5   5      m  5
## 6   6      f  6
## 7   7      f  7
## 8   8      f  8
## 9   9      f  9
## 10 10      f 10

# Con i nomi (molto più intuitivo)
esperimenti$s3$exp2
##    id gender  y
## 1   1      m  1
## 2   2      m  2
## 3   3      m  3
## 4   4      m  4
## 5   5      m  5
## 6   6      f  6
## 7   7      f  7
## 8   8      f  8
## 9   9      f  9
## 10 10      f 10
```

Tip-Box: A cosa servono le liste?

Se il vantaggio di un dataframe rispetto ad una matrice è palese, quale è la vera utilità delle liste essendo "semplicemente" dei contenitori? I vantaggi principali che rendono le liste degli oggetti estremamente potenti sono:

- **Organizzare strutture complesse di dati**: come abbiamo visto nell'esempio precedente, insiemi di oggetti nidificati possono essere or-

ganizzati in un oggetto unico senza avere decine di singoli oggetti nel workspace.
- **Effettuare operazioni complesse su più oggetti parallelamente**. Immaginate di avere una lista di dataframe strutturalmente simili ma con dati diversi all'interno. Se volete applicare una funzione ad ogni dataframe potete organizzare i dati in una lista e usare le funzioni dell'`apply` family che vedremo nei prossimi capitoli. TODO

Infine, proprio per la loro flessibilità, le liste sono spesso utilizzate da vari pachetti per restituire i risultati delle analisi statistiche svolte. Saper accedere ai vari elementi di una lista risulta quindi necessario per ottenere specifiche infomazioni e risultati.

Algoritmi

Introduzione

In questa sezione verranno introdotte le principali tipologie di oggetti usati in R . Ovvero le principali strutture in cui possono essere organizzati i dati: Vettori, Matrici, Dataframe e Liste.

Per ognuna di esse descriveremo le loro caretteristiche e vedremo come crearle, modificarle e manipolarle a seconda delle necessità

I capitoli sono così organizzati:

- **Capitolo 12 - Funzioni**. Vedremo come creare le proprie funzioni personalizzate in R per eseguire degli specifici compiti.
- **Capitolo 13 - Programmazione Condizionale**. Impareremo come gestire l'esecuzione di un algoritmo attraverso l'uso degli operatori `if` e `else`.
- **Capitolo 14 - Programmazione Iterativa**. Introdurremo l'uttilizzo degli operatori `for` e `while` per l'esecuzione di loop.

Chapter 12

Definizione di Funzioni

Nel Capitolo 4.2 abbiamo introdotto il concetto di funzioni e abbiamo visto come queste siano utilizzate per manipolare gli oggetti in R ed eseguire innumerevoli compiti. Nel Capitolo 5.3 abbiamo visto inoltre come sia possibile utilizzare i pacchetti per accedere a nuove funzioni estendendo quindi notevolmente le possibili applicazioni di R. In R, tuttavia, è anche possibile definire le proprie funzioni per eseguire determinati compiti a seconda delle proprie specifiche necessità.

Questo è uno dei vantaggi principali di utilizzare un linguaggio di programmazione, ovvero il poter sviluppare funzioni ad-hoc a seconda delle proprie esigenze e non limitarsi a quelle prestabilite. In questo capitolo descriveremo come creare le proprie funzioni in R.

12.1 Creazione di una Funzione

Il comando usato per creare una funzione in R è `function()` seguito da una coppia di parentesi graffe `{ }` al cui interno deve essere specificato il corpo della funzione:

```
nome_funzione <- function( ){
  <corpo-funzione>
}
```

Nota come sia necessario assegnare la funzione ad un oggetto, ad esempio `my_function`, che diventerà il nome della nostra funzione. Per eseguire la funzione sarà sufficiente, come per ogni altra funzione, indicare il nome della funzione seguito dalle parentesi tonde, nel nostro caso `my_function()`. Vediamo alcuni esempi di semplici funzioni:

```
# Definisco la funzione
my_greetings <- function(){
  print("Hello World!")
}

my_greetings()
## [1] "Hello World!"

# Definisco un'altra funzione
my_sum <- function(){
```

```
  x <- 7
  y <- 3

  x + y
}

my_sum()
## [1] 10
```

Quando chiamiamo la nostra funzione, R eseguirà il corpo della funzione e ci restituirà il risultato dell'ultimo comando eseguito. Le funzioni del precedente esempio, tuttavia, si rivelano poco utili poichè eseguono sempre le stesse operazioni senza che ci sia permesso di specificare gli input delle funzioni. Inoltre, sebbene siano funzioni molto semplici, potrebbe non risultare chiaro quale sia effettivamente l'output restituito dalla funzione.

:::{design title="Nomi Funzioni" data-latex="[Nomi Funzioni]"} Nel definire il nome di una funzione è utile seguire le stese indicazioni che riguardavano i nomi degli oggetti (vedi Capitolo 4.1.2). In questo caso affinché un nome sia **auto-descrittivo** si tende ad utilizzare verbi che riassumano l'azione eseguita dalla funzione. :::

12.1.1 Definire Input e Output

Ricordiamo che in generale le funzioni, ricevuti degli oggetti in input, compiono determinate azioni e restituiscono dei nuovi oggetti in output. Input e ouput sono quindi due aspetti aspetti fondamentali di ogni funzione che richiedono particolare attenzione.

- **Input** - Abbiamo anche visto che in R gli input vengono specificati attraverso gli **argomenti** di una funzione (vedi Capitolo 4.2.1). Per definire gli argomenti di una funzione, questi devono essere indicati all'interno delle parentesi tonde al momento della creazione della funzione
- **Output** - Per specificare l'output di una funzione si utilizza la funzione `return()`, indicando tra le parentesi il nome dell'oggetto che si desidera restituire come risultato della funzione.

La definizione di una funzione con tutti i suoi elementi seguirà quindi il seguente schema:

```
nome_funzione <- function(argument_1, argument_2, ...){
  <corpo-funzione>

  return(<nome-output>)
}
```

Possiamo ora riscrivere le precedenti funzioni permettendo di personalizzare gli input e evidenziando quale sia output che viene restituito.

```
# Ridefinisco my_greetings()
my_greetings <- function(name){
  greetings <- paste0("Hello ", name, "!")

  return(greetings)
}
```

```r
my_greetings(name = "Psicostat")
## [1] "Hello Psicostat!"

# Ridefinisco my_sum()
my_sum <- function(x, y){
  result <- x + y

  return(result)
}

my_sum(x = 8, y = 6)
## [1] 14
```

Approfondimento: User Input

Qualora fosse necessario è possibile fare in modo che sia la funzione stessa a chiedere all'utente di inserire delle specifiche informazioni attraverso la funzione `readline()`.
Utilizzando la funzione `readline()` comparirà nella console il messaggio che abbiamo impostato nell'argomento `prompt` (ricordati di concludere lasciando uno spazio). All'utente sarà richiesto di digitare una qualche sequenza alfanumeirca e premere successivamente *invio*. I valori inseriti dell'utente saranno salvati come una variabile caratteri nell'oggetto che abbiamo indicato e potranno essere utilizzati sucessivamente nella funzione.

```r
happy_birthday <- function(){
  name <- readline(prompt = "Inserisci il tuo nome: ")
  message <- paste0("Buon Compleanno ", name, "!")

  return(message)
}

happy_birthday()
```

Nota che questa funzione possona essere usata solo in sessioni interattive di R poichè è richiesta un 'azione diretta da parte dell'utente.

Default Argomenti

Abbiamo visto come aggiungere degli argomenti che devono essere definiti dall'utente nell'utilizzare la funzione. Qualora l'utente non specifichi uno o più argomenti R riporterà un messaggio di errore indicando come ci siano degli argomenti non specificati e senza valori di default.

```
my_sum(x = 5)
## Error in my_sum(x = 5): argument "y" is missing, with no default
```

Per assegnare dei valori di default agli argomenti di una funzione, è sufficiente indicare al momento della creazione all'interno delle parentesi tonde `nome_argomento = <valore-default>`. Se non altrimenti specificati, gli argomenti assumeranno i loro valori di default. Tuttavia, gli utenti sono liberi di specificare gli argomenti della funzione a seconda delle loro esigenze. Ad esempio impostiamo nella funzione `my_sum()` il valore di default `y = 10`.

```
my_sum <- function(x, y = 10){
  result <- x + y

  return(result)
}

# Utilizzo il valore di default di y
my_sum(x = 5)
## [1] 15

# Specifico il valore id y
my_sum(x = 5,  y = 8)
## [1] 13
```

Questa pratica è molto usata per specificare comportamenti particolari delle funzioni. In genere le funzioni sono definite con un funzionameto di default ma alcuni argomenti possono essere specificati per particolari essigenze solo quando necessario.

Trick-Box: match.arg()

Per imporre la scelta di un argomento tra una limitata serie di valori è possibile indicare nella definizione dell'argomento un vettore con le possibili entrate. Successivamente la scelta deve essere validata attraverso la funzione `match.arg()` come nell esempio successivo:

```r
# Ridefinisco my_greetings()
my_greetings <- function(name, type = c("Hello", "Goodbye")){

  type <- match.arg(type)
  greetings <- paste0(type," ", name, "!")

  return(greetings)
}

# Scelta type
my_greetings(name = "Psicostat", type = "Goodbye")
## [1] "Goodbye Psicostat!"

# Valore default
my_greetings(name = "Psicostat")
## [1] "Hello Psicostat!"

# Valore non ammesso
my_greetings(name = "Psicostat", type = "Guten Tag")
## Error in match.arg(type): 'arg' should be one of "Hello", "Goodbye"
```

La funzione `match.arg()` permette di confrontare il valore specificato rispetto a quelli indicati nella definizione dell'argomento, riportando un errore in mancanza di un match. Osserva come se non specificato il valore di default sia il primo indicato nella definizione.

Esercizi

Esegui i seguenti esercizi:

1. definisci una funzione che trasformi la temperatura da Celsius a Fahrenheit

$$Fahrenheit = Celsius * 1.8 + 32$$

2. Definisci una funzione che permetta di fare gli auguri di buon natale e buona pasqua ad una persona.
3. Definisci una funzione che, dato un vettore di valori numerici, calcoli il numero di elementi e la loro media.
4. Definisci una funzione iterattiva che calcoli il prodotto di due valori. Gli input devono essere ottenuti conlla funzione `readline()`.
5. Definisci una funzione che calcoli lo stipendio mensile sulla base delle ore svolte nel mese e la paga oraria.

12.2 Lavorare con le Funzioni

Le funzioni sono sicuramente l'aspetto più utile e avanzato del linguaggio R e in generale dei linguaggi di programmazione. I pacchetti che sono sviluppati in R non sono altro che insieme di funzioni che lavorano assieme per uno scopo preciso. Oltre allo scopo della funzione è importante capire come gestire gli **errori e gli imprevisti**. Se la funzione infatti accetta degli argomenti, l'utente finale o noi stessi che la utilizziamo possiamo erroneamente usarla nel modo sbagliato. E' importante quindi capire come leggere gli errori ma sopratutto creare messaggi di errore o di avvertimento utili per l'utilizzo della funzione.

Prendiamo ad esempio la funzione somma `+`, anche se non sembra infatti l'operatore `+` è in realtà una funzione. Se volessimo scriverlo come una funzione simile a quelle viste in precedenza possiamo:

```
my_sum <- function(x, y){
  res <- x + y
  return(res)
}

my_sum(1, 5)
## [1] 6
```

Abbiamo definito una (abbastanza inutile) funzione per calcolare la somma tra due numeri. Cosa succede se proviamo a sommare un numero con una stringa? Ovviamente è un'operazione che non ha senso e ci aspettiamo un qualche tipo di errore:

```
my_sum("stringa", 5)
## Error in x + y: non-numeric argument to binary operator
```

In questo caso infatti, vediamo un messaggio denominato `Error...` con l'utile informazione che uno degli argomenti utilizzati risulta essere `non-numeric`. E' un messaggio semplice, mirato e sopratutto non fornisce un risultato perchè una condizione fondamentale (la somma vale solo per i numeri) non è rispettata. La funzione `+` ha già al suo interno questo controllo, ma se noi volessimo implementare un **controllo** e fornire un **messaggio** abbiamo a disposizione diverse opzioni:

- `stop(<message>, .call = TRUE)`: Se inserito all'interno di una funzione, interrompe l'esecuzione e fornisce il messaggio specificato denominato come `Error`.
- `stopifnot(expr1, expr2, ...)`: Se inserito all'interno di una funzione interrompe l'esecuzione se almeno una di una serie di condizioni risulta come **non VERA**.
- `warning()`: Restituisce un messaggio all'utente senza tuttavia interrompere l'esecuzione ma fornendo informazioni su un possibile problema che deriva dal tipo di input o da un possibile effetto collaterale della funzione.
- `message()`: Fornisce un semplice messaggio senza alterare l'esecuzione della funzione che può essere utile per informare rispetto ad operazioni eseguite

Tornando all'esempio della somma, immaginiamo di voler scrivere una funzione che somma solo numeri positivi. In altri termini vogliamo che i valori `x` e `y` in input siano solo posiviti per poter eseguire la funzione. Possiamo quindi inserire un `controllo condizionale` e usare la funzione `stop()` nel caso in cui la condizione non sia rispettata:

```
my_positive_sum <- function(x, y){
  if(x < 0 | y < 0){
    stop("Gli argomenti devono essere numeri positivi!")
  }
  res <- x + y
  return(res)
}

my_positive_sum(10, 5)
## [1] 15
my_positive_sum(10, -5)
## Error in my_positive_sum(10, -5): Gli argomenti devono essere numeri positivi!
```

Un modo più rapido di gestire l'arresto dell'esecuzione è usare `stopifnot()`. La logica tuttavia è leggermente diversa rispetto ad usare un `if` + `stop()`. Nell'esempio precedente la logica è: "se x oppure y sono minori di 0, stop". Con `stopifnot()` utilizziamo una logica inversa, ovvero inseriamo quello che vogliamo sia **VERO** e fermiamo l'esecuzione se è **FALSO**. Nel nostro caso usiamo `stopifnot(x > 0, y > 0)` ovvero fermati se x oppure y *NON SONO* maggiori di 0. Rispetto a `stop()` non fornisce un messaggio personalizzato ma restituisce la prima delle condizioni specificate che non è rispettata:

```
my_positive_sum <- function(x, y){
  stopifnot(x > 0, y > 0)
  res <- x + y
  return(res)
}
my_positive_sum(10, -5)
## Error in my_positive_sum(10, -5): y > 0 is not TRUE
```

Allo stesso modo immaginiamo (con molta immaginazione) che la nostra funzione non sia affidabile con numeri minori di 10, nel senso che qualche volta potrebbe sbagliare il risultato. In questo caso non vogliamo interrompere l'esecuzione ma fornire un messaggi di `warning`:

```
my_positive_sum <- function(x, y){
  # Error
  if(x < 0 | y < 0){
    stop("Gli argomenti devono essere numeri positivi!")
  }
  # Warning
  if(x < 10 | y < 10){
    warning("Per qualche strano motivo la funzione non gestisce bene i numeri minori di 10, attenz
  }
  res <- x + y
  return(res)
}

my_positive_sum(15, 4)
## Warning in my_positive_sum(15, 4): Per qualche strano motivo la funzione non
## gestisce bene i numeri minori di 10, attenzione!! :)
## [1] 19
```

Come vedete, abbiamo il risultato (ovviamente corretto) ma anche un messaggio di `warning` che ci avverte di questa possibile (ma non critica) problematica.

Infine possiamo accompagnare il risultato alcune condizioni che si realizzano con un semplice messaggio che fornisce ulteriori informazioni con la funzione `message()`:

```
my_positive_sum <- function(x, y){
  # Error
  if(x < 0 | y < 0){
    stop("Gli argomenti devono essere numeri positivi!")
  }
  # Warning
  if(x < 10 | y < 10){
    warning("Per qualche strano motivo la funzione non gestisce bene i numeri minori di 10, attenz
  }
  res <- x + y
  message("Ottimo lavoro! :)")
  return(res)
}

my_positive_sum(12, 10)
## Ottimo lavoro! :)
## [1] 22
```

Un ultimo aspetto importante riguarda cosa avviene se assegniamo il risultato di una funzione in presenza di errori, warning o messaggi. In generale, tranne che per la presenza di errori e quindi usando la funzione `stop()`, l'output rimane lo stesso e il messaggio è solo stampato nella console:

```
res1 <- my_positive_sum(10,5)
## Warning in my_positive_sum(10, 5): Per qualche strano motivo la funzione non
## gestisce bene i numeri minori di 10, attenzione!! :)
## Ottimo lavoro! :)
res2 <- my_positive_sum(10,23)
## Ottimo lavoro! :)
res3 <- my_positive_sum(10,-1)
## Error in my_positive_sum(10, -1): Gli argomenti devono essere numeri positivi!

res1
## [1] 15
res2
## [1] 33
res3 # nessun output
## Error in eval(expr, envir, enclos): object 'res3' not found
```

Come vedete infatti, quando abbiamo un errore e fermiamo l'esecuzione, la funzione pur prevedendo un output, non fornisce risultato perchè è stata interrotta.

12.3 Ambiente della funzione

Il concetto di `ambiente` in R è abbastanza complesso[1]. In parole semplici, tutte le operazioni che normalmente eseguiamo nella console o in uno script, avvengono in quello che si chiama `global environment`. Quando scriviamo ed eseguiamo una funzione, stiamo creando un oggetto funzione (nel `global environment`) che a sua volta crea un ambiente interno per eseguire le operazioni previste. Immaginiamo di avere questa funzione `my_fun()` che riceve un valore `x` e lo somma ad un valore `y` che non è un argomento.

```
my_fun <- function(x){
  return(x + y)
}

my_fun(10)
## Error in my_fun(10): object 'y' not found
```

Chiaramente otteniamo un errore perchè l'oggetto `y` non è stato creato. Se però creiamo l'oggetto `y` all'interno della funzione, questa esegue regolarmente la somma MA non crea l'oggetto `y` nell'ambiente globale.

```
my_fun <- function(x){
  y <- 1
  return(x + y)
}

my_fun(10)
## [1] 11
ls() # abbiamo solo la nostra funzione come oggetto
## [1] "my_fun"          "my_greetings"    "my_positive_sum" "my_sum"
## [5] "res1"            "res2"
```

Da qui è chiaro che quello che avviene all'interno della funzione è in qualche modo compartimentalizzato rispetto all'ambiente globale. L'unico modo per influenzare l'ambiente globale è quello di assegnare il risultato della funzione, creando quindi un nuovo oggetto:

```
res <- my_fun(10)
ls()
## [1] "my_fun"          "my_greetings"    "my_positive_sum" "my_sum"
## [5] "res"             "res1"            "res2"
```

Altra cosa importante, sopratutto per gestire effetti collaterali riguarda il fatto che la funzione NON modifica gli oggetti presenti nell'ambiente globale:

```
y <- 10 # ambiente globale

my_fun <- function(x){
  y <- 1 # ambiente funzione
```

[1] per una non semplice trattazione dell'argomento, il capitolo 7 del libro "Advanced R" di Hadley Wickam è un ottima risorsa.

```
  return(x + y) # questo si basa su y funzione
}

my_fun(1)
## [1] 2
y
## [1] 10
```

Come vedete, abbiamo creato un oggetto `y` dentro la funzione. Se eseguito nello stesso ambiente questo avrebbe sovrascritto il precedente valore. Il risultato si basa sul valore di `y` creato nell'ambiente funzione e l'`y` globale non è stato modificato.

Un ultimo punto importante riguarda invece il legame tra ambiente funzione e quello globale. Abbiamo visto la loro indipendenza che però non è totale. Se infatti all'interno della funzione utilizziamo una variabile definita solamente nell'ambiente globale, la funzione in automatico userà quel valore (se non specificato internamente). Questo è utile per far lavorare funzioni e variabili globali MA è sempre preferibile creare un ambiente funzione indipendente e fornire come **argomenti** tutte gli oggetti necessari.

```
y <- 10

my_fun <- function(x){
  return(x + y) # viene utilizzato y globale
}

my_fun(1)
## [1] 11
```

Le cose importanti da ricordare quando si definiscono e utilizzano funzioni sono:

- Ogni volta che una funzione viene eseguita, l'ambiente interno viene ricreato e quindi è come ripartire da zero
- Gli oggetti creati all'interno della funzione hanno priorità rispetto a quelli nell'ambiente esterno
- Se la funzione utilizza un oggetto non definito internamente, automaticamente cercherà nell'ambiente principale

12.4 Best practice

Scrivere funzioni è sicuramente l'aspetto più importante quando si scrive del codice. Permette di automatizzare operazioni, ridurre la quantità di codice, rendere più chiaro il nostro script e riutilizzare una certa porzione di codice in altri contesti. Ci sono tuttavia delle convenzioni e degli accorgimenti per scrivere delle ottime e versatili funzioni:

- Quando serve una funzione?
- Scegliere il nome
- Semplificare la quantita di operazioni e output
- Commentare e documentare

12.4.1 Quando serve una funzione?

Hadley Wickam suggerisce che se ripetiamo una serie di operazioni più di 2 volte, forse è meglio scrivere una funzione. Immaginiamo di avere una serie di oggetti e voler eseguire la stessa operazione in tutti. Ad esempio vogliamo *centrare* (ovvero sottrarre a tutti i valori di un vettore la loro media) un vettore numerico:

```
vec1 <- runif(10)
vec2 <- runif(10)
vec3 <- runif(10)
```

Abbiamo visto nei capitoli precedenti l'utilizzo dell'`apply` family e come si possa applicare una funzione ad una serie di oggetti. Pensiamo però ad un caso dove abbiamo uno script molto lungo e in diversi momenti eseguiamo una certa operazione:

```
# Centriamo
vec1 - mean(vec2)
##  [1]  0.375483045  0.007881546  0.184683436  0.457343132  0.128117364
##  [6]  0.004316731 -0.279044610 -0.343590252  0.038928952  0.168865691
vec2 - mean(vec2)
##  [1] -0.23504352  0.11710579  0.42825239 -0.27330895 -0.30707553  0.13634206
##  [7]  0.43973909  0.17258473 -0.09212456 -0.38647150
vec3 - mean(vec3)
##  [1]  0.08137690  0.17385603  0.17503549  0.47734530 -0.31804391  0.08731763
##  [7] -0.03370439 -0.30430163 -0.29468320 -0.04419821
```

L'operazione viene eseguita correttamente ed è anche di facile comprensione. Tuttavia stiamo eseguendo sempre la stessa cosa, semplicemente cambiando un input (proprio la definizione di funzione) quindi possiamo:

```
my_fun <- function(x){
  return(x - mean(x))
}

my_fun(vec1)
##  [1]  0.30118454 -0.06641696  0.11038493  0.38304463  0.05381886 -0.06998177
##  [7] -0.35334311 -0.41788876 -0.03536955  0.09456719
my_fun(vec2)
##  [1] -0.23504352  0.11710579  0.42825239 -0.27330895 -0.30707553  0.13634206
##  [7]  0.43973909  0.17258473 -0.09212456 -0.38647150
my_fun(vec3)
##  [1]  0.08137690  0.17385603  0.17503549  0.47734530 -0.31804391  0.08731763
##  [7] -0.03370439 -0.30430163 -0.29468320 -0.04419821
```

Il codice non è molto cambiato rispetto a prima in termini di numero di righe o complessità. Immaginate però di esservi resi conto di un errore o di voler cambiare o estendere le operazioni su`vec1, vec2 e vec3`. Nel primo caso dovreste andare linea per linea dello script e modificare il codice. Nel caso di una funzione, semplicemente cambiando le operazioni queste verranno applicate ogni volta che quella funzione è chiamata. Immaginiamo di voler anche *standardizzare* (sottrarre la media e dividere per la deviazione standard) i nostri vettori:

```
my_fun <- function(x){
  res <- (x - mean(x)) / sd(x)
  return(res)
}

my_fun(vec1)
##  [1]  1.1952127 -0.2635673  0.4380486  1.5200641  0.2135733 -0.2777138
##  [7] -1.4021974 -1.6583386 -0.1403596  0.3752779
my_fun(vec2)
##  [1] -0.7782383  0.3877418  1.4179603 -0.9049366 -1.0167390  0.4514339
##  [7]  1.4559933  0.5714348 -0.3050280 -1.2796222
my_fun(vec3)
##  [1]  0.3187379  0.6809612  0.6855810  1.8696714 -1.2457179  0.3420067
##  [7] -0.1320137 -1.1918920 -1.1542185 -0.1731161
```

Ovviamente la combinazione di funzione e `apply` family permette di rendere il tutto ancora più compatto ed efficiente:

```
my_list <- list(vec1, vec2, vec3)
lapply(my_list, my_fun)
## [[1]]
##  [1]  1.1952127 -0.2635673  0.4380486  1.5200641  0.2135733 -0.2777138
##  [7] -1.4021974 -1.6583386 -0.1403596  0.3752779
##
## [[2]]
##  [1] -0.7782383  0.3877418  1.4179603 -0.9049366 -1.0167390  0.4514339
##  [7]  1.4559933  0.5714348 -0.3050280 -1.2796222
##
## [[3]]
##  [1]  0.3187379  0.6809612  0.6855810  1.8696714 -1.2457179  0.3420067
##  [7] -0.1320137 -1.1918920 -1.1542185 -0.1731161
```

12.4.2 Scegliere il nome

Questo potrebbe sembrare un argomento marginale tuttavia la scelta dei nomi sia per le variabili ma sopratutto per le funzioni è estremamente importante. Permette di:

- leggere chiaramente il nostro codice e renderlo comprensibile ad altri
- organizzare facilmente un gruppo di funzioni. Quando avete più funzioni, usare una giusta denominazione permette di sfruttare i suggerimenti di RStudio in modo più efficace. Il pacchetto `stringr` and esempio che fornisce strumenti per lavorare con stringhe, utilizza tutte le funzioni denominate come `str_` permettendo di cercare facilmente quella desiderata.

E' utile utilizzare **verbi** per nominare le funzioni mentre **nomi** per nominare argomenti. Ad esempio un nome adatto alla nostra ultima funzione potrebbe essere `center_var()` mentre il nome del nuovo vettore `centered_vec` o `c_vec`. Se troviamo `center_var` all'interno di uno script è subito chiaro il compito di quella funzione, anche senza guardare il codice al suo interno.

12.4.3 Semplificare la quantita di operazioni e output

Questo è un punto molto importante ma allo stesso tempo variegato. Ci sono diversi stili di programmazione e quindi non ci sono regole fisse oppure delle pratiche migliori di altre. Abbiamo detto che una funzione è un modo per astrarre, riutilizzare e semplificare una serie di operazioni. Possiamo quindi scrivere funzioni molto complesse che ricevono diversi input, eseguono diverse operazioni e restituiscono diversi output. E' buona pratica però scrivere funzioni che:

- riducono il numero di operazioni interne
- forniscono un singolo (o limitati) output
- hanno un numero di input limitato

Se quindi abbiamo pensato ad una funzione che ha troppi output, è troppo complessa oppure ha troppi input magari possiamo valutare di scomporre la funzione in sotto-funzioni.

Facciamo un esempio con la nostra `center_vec()`. Possiamo pensare a diverse alternative ed estensioni di questa funzione. Ad esempio possiamo pensare di creare una funzione che centra oppure standardizza il vettore. Possiamo inoltre scegliere se centrare usando la media oppure la mediana. Quindi possiamo pensare a una macro funzione `trans_vec()` che in base agli argomenti trasforma il vettore:

```
trans_vec <- function(x, what = c("center_mean", "center_median", "standardize")){
  if(match.arg(what) == "center_mean"){
    res <- x - mean(x)
  }else if(match.arg(what) == "center_median"){
    res <- x - median(x)
  }else if(match.arg(what) == "standardize"){
    res <- (x - mean(x))/sd(x)
  }
  return(res)
}

vec <- runif(10)
trans_vec(vec, "center_mean")
##  [1]  0.27970039 -0.29559475 -0.35010756 -0.35677833  0.47367363  0.46216180
##  [7] -0.29476887  0.48389466 -0.04806907 -0.35411189
trans_vec(vec, "center_mean")
##  [1]  0.27970039 -0.29559475 -0.35010756 -0.35677833  0.47367363  0.46216180
##  [7] -0.29476887  0.48389466 -0.04806907 -0.35411189
trans_vec(vec, "standardize")
##  [1]  0.7353231 -0.7771089 -0.9204213 -0.9379585  1.2452724  1.2150082
##  [7] -0.7749377  1.2721432 -0.1263720 -0.9309485
```

La funzione è molto chiara ma comunque contiene dei margini di fragilità. L'utente deve inserire una per stringa eseguire l'operazione. Ci sono diversi `if` e lo scopo della funzione è forse troppo generico. Una migliore soluzione sarebbe quella di scrivere 3 funzioni più semplici, mirate e facili da mantenere e leggere:

```
center_vec_mean <- function(x){
  return(x - mean(x))
}
```

```r
center_vec_median <- function(x){
  return(x - median(x))
}

standardize_vec <- function(x){
  return((x - mean(x)) / sd(x))
}

vec <- runif(10)
center_vec_mean(vec)
##  [1] -0.06218784  0.04085097 -0.31795912  0.26834550  0.03212795  0.34036917
##  [7]  0.21744515 -0.29556808 -0.39180500  0.16838131
center_vec_median(vec)
##  [1] -0.098677304  0.004361508 -0.354448579  0.231856039 -0.004361508
##  [6]  0.303879709  0.180955693 -0.332057540 -0.428294461  0.131891849
standardize_vec(vec)
##  [1] -0.2384730  0.1566521 -1.2192846  1.0290302  0.1232017  1.3052209
##  [7]  0.8338415 -1.1334212 -1.5024629  0.6456954
```

In questo modo il codice è molto più leggibile e chiaro sia dentro le funzioni che quando le funzioni vengono utilizzate. Un'ulteriore alternativa sarebbe quella di raggruppare le funzione di "centramento" specificando se utilizzare media o mediana e separare quella di standardizzazione.

12.4.4 Commentare e documentare

La documentazione è forse la parte di più importante della scrittura del codice. Possiamo classificarla in documentazione *formale* e *informale* in base allo scopo. La documentazione *formale* è quella che troviamo facendo `help(funzione)` oppure `?funzione`. E' una documentazione standardizzata e necessaria quando si creano delle funzioni in un pacchetto che altri utenti devono utilizzare. La documentazione *informale* è quella che mettiamo nei nostri script e all'interno delle funzioni come `# commento`. Entrambe sono molto importati e permettono di descrivere lo scopo generale della funzione, i singoli argomenti e i passaggi eseguiti.

12.5 Importare una funzione

Abbiamo già visto che il comando `library()` carica un certo pacchetto, rendendo le funzioni contenute disponibili all'utilizzo. Senza la necessità di creare un pacchetto, possiamo comunque organizzare le nostre funzioni in modo efficace. Abbiamo 2 opzioni:

- scrivere le funzioni nello stesso script dove esse vengono utilizzate
- scrivere uno script separato e importare tutte le funzioni contenute

Anche in questo caso è una questione di stile e comodità, in generale:

- Se abbiamo tante funzioni, è meglio scriverle in uno o più file separati e poi importarle all'inizio dello script principale

- Se abbiamo poche funzioni possiamo tenerle nello script principale, magari in una sezione apposita nella parte iniziale

Nel secondo caso è sufficiente quindi scrivere la funzione e questa sarà salvata come oggetto nell'ambiente principale. Riguardo il primo scenario si può utilizzare la funzione `source("percorso/script.R")`. La funzione `source()` accetta il percorso di uno script R che verrà poi eseguito in background. Quindi se la vostra directory è organizzata in questo modo:

```
- working directory/
|-- main_script.R
|-- functions/
    |-- my_functions.R
```

Dove lo script `my_functions.R` è uno script dove sono dichiarare tutte le funzioni:

```
fun1 <- function(x){
  # do
}

fun2 <- function(x){
  # do
}

fun3 <- function(x){
  # do
}

...
```

Scrivendo all'inizio del nostro script principale `source("functions/my_functions.R")`, tutte le funzioni saranno caricate nel ambiente di lavoro.

Chapter 13

Programmazione Condizionale

Uno dei principali costrutti della programmazione sono proprio le espressioni condizionali. L'idea semplice alla base è quella di eseguire alcune operazioni o eseguire le operazioni in un certo modo in funzione di alcune *condizioni*. Le condizioni non sono altro che espressioni che resituiscono sempre un valore *booelan* ovvero `TRUE` oppure `FALSE`.

L'esempio classico è quello dell'ombrello, immaginate di scrivere il codice per un videogioco e avete scritto una funzione per far prendere prendere e aprire l'ombrello al vostro personaggio. Chiaramente per rendere tutto credibile, dovete far eseguire questa operazione solo quando è richiesto ovvero quando piove. In termini di programmazione, dovete prima verificare una certa condizione (la pioggia) e poi eseguire (o non eseguire) una serie di operazioni di conseguenza.

13.1 Strutture condizionali

Ci sono diverse possibilità in R, tuttavia la logica di eseguire operazioni solo quando alcune condizioni sono rispettate è sempre la stessa. La Figura 13.1 rappresenta la struttura generica di un flusso di controllo condizionale.

13.1.1 if

```
knitr::include_graphics("images/if_chart.png")
```

Struttura if

Per scrivere la struttura condizionale della Figura 13.1 si usa la seguente sintassi in R:

```
...

if (<test>) {
  <codice-da-eseguire>
}
...
```

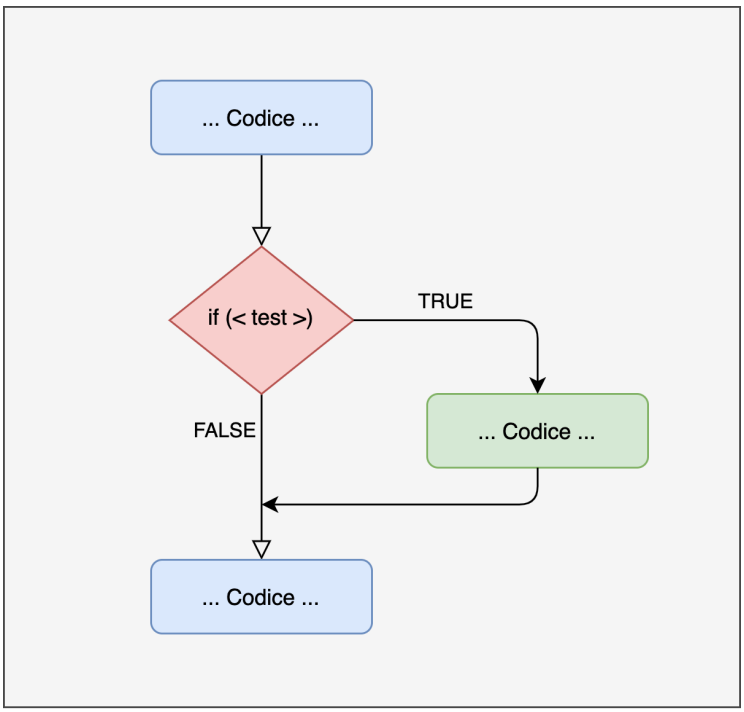


Figure 13.1: Rappresentazione if

Esempio

- Singolo if

```
my_function <- function(value){

  if(value > 0){
    cat("Il valore è maggiore di 0\n")
  }

  cat("Fine funzione\n")
}

my_function(5)
## Il valore è maggiore di 0
## Fine funzione

my_function(-5)
## Fine funzione
```

- Multipli if

```
my_function <- function(value){

  if(value > 0){
```

```
    cat("Il valore è maggiore di 0\n")
  }

  if(value > 10){
    cat("Il valore è maggiore di 10\n")
  }

  cat("Fine funzione\n")
}

my_function(5)
## Il valore è maggiore di 0
## Fine funzione

my_function(15)
## Il valore è maggiore di 0
## Il valore è maggiore di 10
## Fine funzione

my_function(-5)
## Fine funzione
```

13.1.2 if...else

Il semplice utilizzo di un singolo `if` potrebbe non essere sufficiente in alcune situazioni. Sopratutto perchè possiamo vedere l'`if` come una deviazione temporanea dallo script principale (molto chiaro nella figura 13.1) che viene imboccata solo se è vera una condizione, altrimenti lo script continua. Se vogliamo una struttura più "simmetrica" possiamo eseguire delle operazioni se la condizone è vera `if` e altre per tutti gli altri scenari (`else`). La Figura 13.2 mostra chiaramente questa struttura diversa nel flusso.

```
knitr::include_graphics("images/ifelse_chart.png")
```

Struttura if

In R questo viene implementato nel seguente modo:

```
  ...

  if (<test>) {
    <codice-da-eseguire>
  } else {
    <codice-da-eseguire>
  }
  ...
```

Esempio

- Singolo if...else

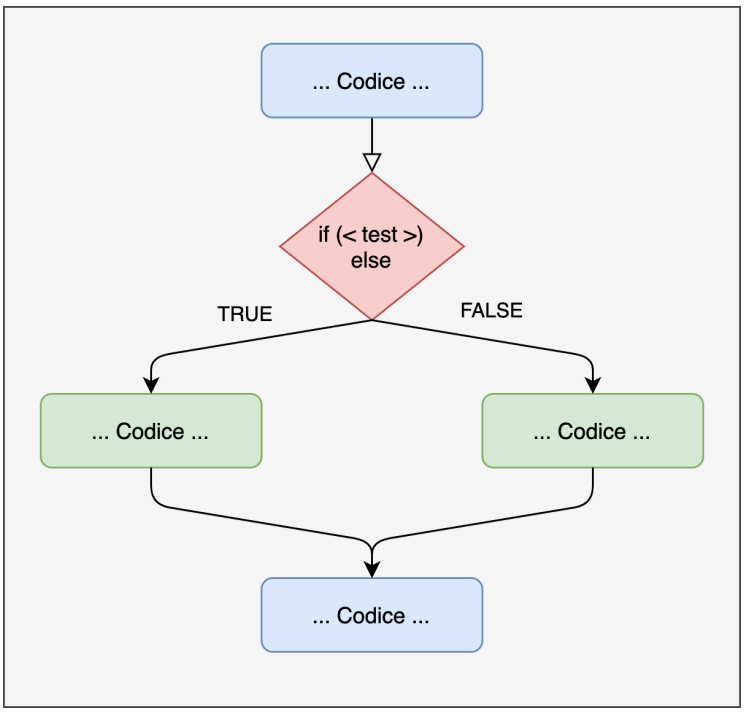


Figure 13.2: Rappresentazione if...else

```
my_function <- function(value){

  if(value >= 0){
    cat("Il valore è maggiore di 0\n")
  } else {
    cat("Il valore non è maggiore di 0\n")
  }

  cat("Fine funzione\n")
}

my_function(5)
## Il valore è maggiore di 0
## Fine funzione

my_function(-5)
## Il valore non è maggiore di 0
## Fine funzione
```

- Multipli if

```
my_function <- function(value){

  if(value > 0){
    cat("Il valore è maggiore di 0\n")
```

```
  } else if (value > 10){
    cat("Il valore è maggiore di 10\n")
  } else {
    cat("Il valore non è maggiore di 0\n")
  }

  cat("Fine funzione\n")
}

my_function(5)
## Il valore è maggiore di 0
## Fine funzione

my_function(15)
## Il valore è maggiore di 0
## Fine funzione

my_function(-5)
## Il valore non è maggiore di 0
## Fine funzione
```

E' importante capire la differenza tra usare multipli `if` e `else if` rispetto a utilizzare un semplice `else`. Se noi vogliamo specificare un set **finito** di alternative, la scelta migliore e quella più chiara consiste di usere `else if` perchè al contrario di `else` richiede di specificare la condizione. Se invece siamo interessanti ad alcune condizioni mentre per tutto il resto applichiamo la stessa operazione, possiamo concludere il nostro flusso con un `else`.

13.1.3 Nested

Oltre a concatenare una serie di `if` e `else if` è possibile inserire uno o più `if` all'interno di un'altro. Questo permette di controllare delle condizioni che sono dipendenti dal controllo precedente e quindi non possono essere eseguite in serie.

```
knitr::include_graphics("images/ifnested.png")
```

Struttura if

```
  ...

  if (<test>) {

    if(<test>) {
      <codice-da-eseguire>
    } else {
      <codice-da-eseguire>
    }
  }
  ...
```

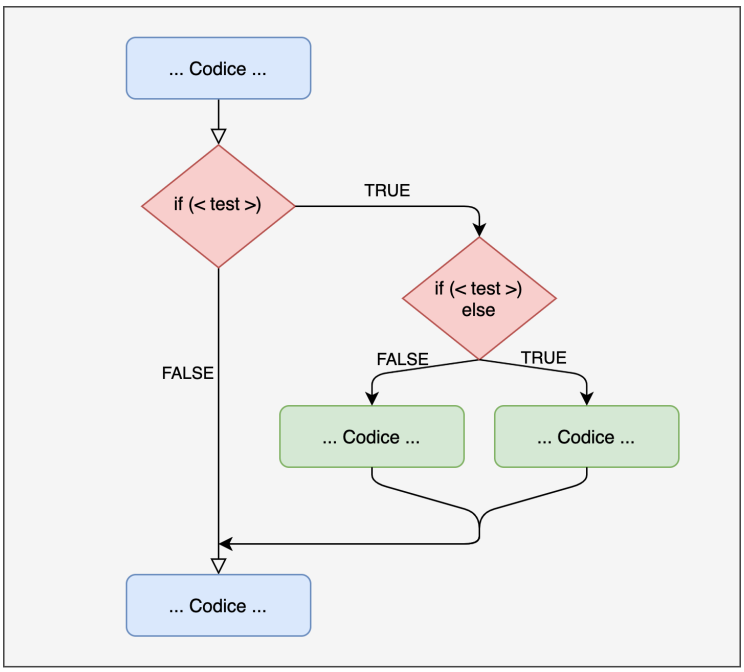


Figure 13.3: Rappresentazione if nested

Esempio

```
my_function <- function(value){

  if(value > 0){

    if(value > 10) {
      cat("Il valore è maggiore di 10\n")
    } else {
      cat("Il valore è maggiore di 0\n")
    }

  } else {
    cat("Il valore non è maggiore di 0\n")
  }

  cat("Fine funzione\n")
}

my_function(5)
## Il valore è maggiore di 0
## Fine funzione

my_function(15)
## Il valore è maggiore di 10
## Fine funzione
```

```
my_function(-5)
## Il valore non è maggiore di 0
## Fine funzione
```

Esercizi

Esegui i seguenti esercizi:

1. Definisci una funzione per assegnare un voto in base alla percentuale di rispote corrette (*score*) segui le seguenti indicazioni:

```
- score < .55 insufficiente
- .55 <= score < .65 - sufficiente
- .65 <= score < .75 - buono
- .75 <= score < .85 - distinto
- .85 <= score - ottimo
```

2. Definisci una funzione che determini se un numero è pari o dispari.
3. Definisci una funzione che determini se un numero è un multiplo di 3, 4, o 5.
4. Definisci una funzione che calcoli lo stipendio mensile considerando gli straordinari che sono retribuiti 1.5 della paga oraria normale. Utilizza come parametrile ore svolte nel mese, la paga oraria ed il tetto ore lavorative, oltre cui si contano gli straordinari.

13.2 Altri Operatori Condizionali

13.2.1 switch

L'operatore `switch` è un utile alternativa quando vogliamo eseguire una porzione di codice in modo condizionale all'input senza tuttavia usare una serie di `if`.

Struttura switch

```
switch(my_value,
       case1 = action1,
       case2 = action2,
       ... )
```

In questo caso, se il valore input (`my_value`) ha una corrispondenza tra i casi considerati (ad esempio `case1`), il codice corrispondente (`action1`) viene eseguito.

Ci sono alcune regole da considerare quando si utilizza uno `switch` statement:

- se il valore inserito è una stringa, R cercherà una corrispondenza come `input == argomento`
- se è presente più di una corrispondenza, viene utilizzata la prima
- non c'è la possibilità di inserire un valore di default
- se non viene trovata una corrispondenza viene restituito un valore `NULL`. Inserendo però un `case` senza un valore corrispondente, questo verra utilizzato al posto di `NULL`.

Esempio

```
my_colors <- function(color){

  new_color <- switch(color,
       "rosso" = "red",
       "blu" = "blue",
       "verde" = ,
       "verde acqua" = "green",
       "Not Found") # valore non trovato

  return(new_color)
}

my_colors("blu")
## [1] "blue"

my_colors("verde")
## [1] "green"

my_colors("arancione")
## [1] "Not Found"
```

13.2.2 ifelse

Gli `if`, `else` e `else if` statements funzionano solo per un singolo valore. In altri termini non è possibile testare una stessa condizione su una serie di elementi. Nel prossimo capitolo affronteremo la programmazione iterativa che permette di ripetere una serie di operazioni. Tuttavia utilizzando la funzione `ifelse()` è possibile implementare una versione vettorizzata delle operazioni condizionali. La struttura è la seguente:

```
ifelse(test = , yes = , no = )
```

Dove:

- `test` è la condizione da valutare e corrisponde a quello che di solito era tra parentesi negli `if` precedenti `if(test){...}`
- `yes` è il codice che viene eseguito se la condizione è `TRUE`
- `no` è il codice che viene eseguito se la condizione è `FALSE`

Esempio

Immaginiamo di avere un vettore di numeri che corrispondono a delle età e vogliamo eseguire qualcosa se il l'età è maggiore o minore di 18 anni, ad esempio mostrare semplicemente la scritta "maggiorenne" oppure "minorenne". Utilizzando un semplice `if` vediamo che:

```
age <- c(18, 19, 11, 10, 23, 55, 33, 26, 10)
if(age < 18){
  print("Minorenne")
}else{
  print("Maggiorenne")
}
## Warning in if (age < 18) {: the condition has length > 1 and only the first
## element will be used
## [1] "Maggiorenne"
```

Veniamo avvertiti che `age` non è un singolo valore e che quindi solo il primo valore è utilizzato. Questo dimostra che `if` non funziona su una lista di elementi. Utilizzando `ifelse` invece:

```
ifelse(age < 18, yes = print("Minorenne"), no = print("Maggiorenne"))
## [1] "Minorenne"
## [1] "Maggiorenne"
## [1] "Maggiorenne" "Maggiorenne" "Minorenne"   "Minorenne"   "Maggiorenne"
## [6] "Maggiorenne" "Maggiorenne" "Maggiorenne" "Minorenne"
```

In questo caso l'idea è di eseguire il codice precedente per ogni valore del nostro vettore in input. Questo è possibile perchè l'operazione `<` è vettorizzata.

Chapter 14

Programmazione Iterativa

L'essenza della maggior parte delle operazioni nei vari linguaggi di programmazione è quella del concetto di **iterazione**. Iterazione significa ripetere una porzione di codice un certo numero di volte o fino anche una condizione viene soddisfatta.

Molte delle funzioni che abbiamo usato finora come la funzione `sum()` o la funzione `mean()` si basano su operazioni iterative. In R, purtroppo o per fortuna, userete abbastanza raramente delle iterazioni tramite loop anche se nella maggior parte delle funzioni sono presenti. Infatti molte delle funzioni implementate in R sono disponibili solo con pacchetti esterni oppure devono essere scritte manualmente implementando strutture iterative.

14.1 Loop

14.1.1 For

Il primo tipo di struttura iterativa viene denominata ciclo `for`. L'idea è quella di ripetere una serie di istruzioni un numero **predefinito** di volte.La Figura 14.1 rappresenta l' idea di un ciclo `for`. In modo simile alle strutture condizionali del capitolo precedente, quando scriviamo un ciclo, entriamo in una parte di codice temporaneamente, eseguiamo le operazioni richieste e poi continuiamo con il resto del codice. Quello che nell'immagine è chiamato `i` è un modo convenzionale di indicare il conteggio delle operazioni. Se vogliamo ripetere un'operazione 1000 volte, `i` parte da 1 e arriva fino a 1000.

```
knitr::include_graphics("images/for_loop.png")
```

Struttura For Loop

In R la scrittura del ciclo `for` è la seguente:

```
for (i in c(...)) {
    <codice-da-eseguire>
}
```

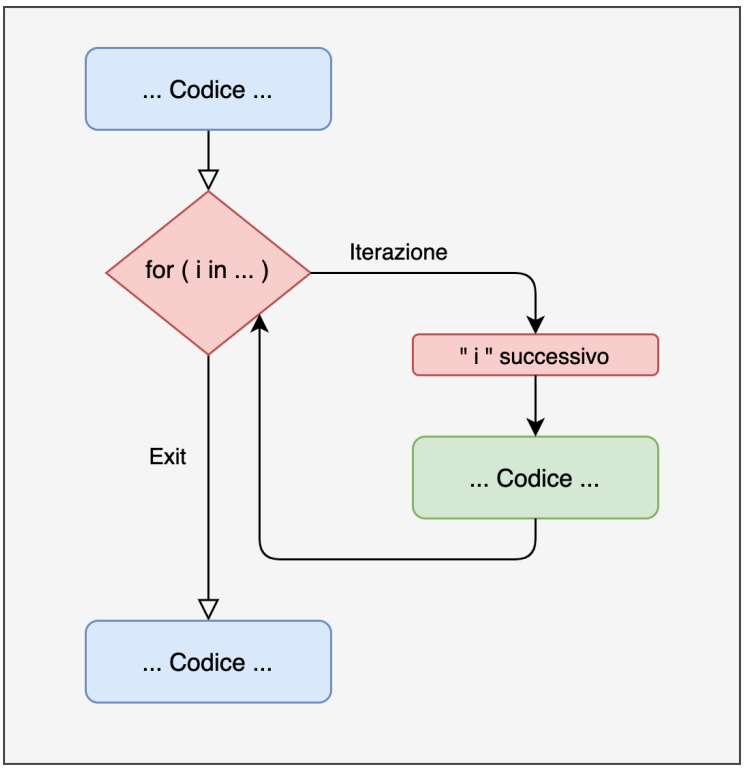


Figure 14.1: Rappresentazione for loop

- `i` è un nome generico per indicare la variabile conteggio che abbiamo introdotto prima. Può essere qualsiasi carattere, ma solitamente per un ciclo generico si utilizzano singole lettere come `i` o `j` proabilmente per una similarità con la notazione matematica che spesso utilizza queste lettere per indicare una serie di elementi
- `in` è l'operatore per indicare che `i` varia rispetto ai valori specificati di seguito
- `c(...)` è il range di valori che assumerà `i` in per ogni iterazione

Possiamo riformulare il codice in:

> Ripeti le operazioni incluse tra {} un numero di volte uguale alla lunghezza di `c(...)` e in questo ciclo, `i` assumerà, uno alla volta i valori contenuti in `c(...)`.

Informalmente ci sono due tipi di ciclo, quello che utilizza un counter generico da assegnare a `i` e un'altro che utilizza direttamente dei valori di interesse.

Esempio

- Loop con direttamente i valori di interesse

```
# numerico
# caratteri
for (name in c("Alessio", "Beatrice", "Carlo")){
  print(paste0("Ciao ", name))
}
## [1] "Ciao Alessio"
## [1] "Ciao Beatrice"
## [1] "Ciao Carlo"
```

Loop che utilizza un counter generico per indicizzare gli elementi:

```
my_vector <- c(93, 27, 46, 99)

# i in 1:length(my_vector)
for (i in seq_along(my_vector)){
  print(my_vector[i])
}
## [1] 93
## [1] 27
## [1] 46
## [1] 99
```

Questa distinzione è molto utile e spesso fonte di errori. Se si utilizza direttamente il vettore e il nostro counter assume i valori del vettore, "perdiamo" un indice di posizione. Nell'esempio del ciclo con i nomi infatti, se volessimo sapere e stampare quale posizione occupa Alessio dobbiamo modificare l'approccio, puntando ad utilizzando anche counter generico. Possiamo crearlo fuori dal ciclo e aggiornarlo manualmente:

```
i <- 1
for (name in c("Alessio", "Beatrice", "Carlo")){
  print(paste0(name, " è il numero ", i))
  i <- i + 1
}
## [1] "Alessio è il numero 1"
## [1] "Beatrice è il numero 2"
## [1] "Carlo è il numero 3"
```

In generale, il modo migliore è sempre quello di utilizzare un ciclo utilizza indici e non i valori effettivi, in modo da poter accedere comunque entrambe le informazioni.

```
nomi <- c("Alessio", "Beatrice", "Carlo")

for (i in seq_along(nomi)){
  print(paste0(nomi[i], " è il numero ", i))
}
## [1] "Alessio è il numero 1"
## [1] "Beatrice è il numero 2"
## [1] "Carlo è il numero 3"
```

Trick-Box: Next e Break

```
my_vector <- c(93, 27, 46, 99)
my_NULL <- NULL

1:length(my_vector)
## [1] 1 2 3 4
1:length(my_NULL)
## [1] 1 0

seq_along(my_vector)
## [1] 1 2 3 4
seq_along(my_NULL)
## integer(0)

seq_len(length(my_vector))
## [1] 1 2 3 4
seq_len(length(my_NULL))
## integer(0)
```

Esempio: la funzione somma

Come introdotto all'inizio di questo capitolo, molte delle funzioni disponibili in R derivano da strutture iterative. Se pensiamo alla funzione `sum()` sappiamo che possiamo calcolare la somma di un vettore semplicemente con `sum(x)`. Per capire appieno i cicli, è interessante pensare e implementare le funzioni comuni.

Se dovessimo sommare `n` numeri a mano la struttura sarebbe questa:

- prendo il primo numero x_1 e lo sommo con il secondo x_2
- ottengo un nuovo numero `x_{1+2}`
- prendo il terzo numero x_3 e lo sommo a `x_{1+2}`
- ottengo x_{1+2+3}
- ripeto questa operazione fino all'ultimo elemento di x_n

Come vedete, questa è una struttura iterativa, che conta da 1 alla lunghezza di x ed ogni iterazione somma il successivo con la somma dei precedenti. In R:

```
my_values <- c(2,4,6,8)

# Calcolare somma valori
my_sum <- 0      # inizializzo valore
for (i in seq_along(my_values)){
  my_sum <- my_sum + my_values[i]
}

my_sum
## [1] 20
```

La struttura è la stessa del nostro ragionamento in precedenza. Creo una variabile "partenza" che assume valore 0 ed ogni iterazione sommo indicizzando il rispettivo elemento.

Esempio: creo un vettore

Essendo che utilizziamo un indice che assume un range di valori, possiamo non solo accedere ad un vettore utilizzando il nostro indice ma anche creare o sostituire un vettore progressivamente.

```
# Calcola la somma di colonna
my_matrix <- matrix(1:24, nrow = 4, ncol = 6)

# Metodo non efficiente (aggiungo valori)
sum_cols <- c()
for( i in seq_len(ncol(my_matrix))){
  sum_col <- sum(my_matrix[, i])   # calcolo i esima colonna
  sum_cols <- c(sum_cols, sum_col) # aggiungo il risultato
}

sum_cols
## [1] 10 26 42 58 74 90
```

```
# Metodo efficiente (aggiorno valori)
sum_cols <- vector(mode = "double", length = ncol(my_matrix))
for( i in seq_along(sum_cols)){
  sum_col <- sum(my_matrix[, i])   # calcolo i esima colonna
  sum_cols[i] <-  sum_col # aggiorno il risultato
}

sum_cols
## [1] 10 26 42 58 74 90
```

14.1.2 While

Il ciclo `while` può essere considerato come una generalizzazione del ciclo `for`. In altri termini il ciclo `for` è un tipo particolare di ciclo `while`.

```
knitr::include_graphics("images/while_loop.png")
```

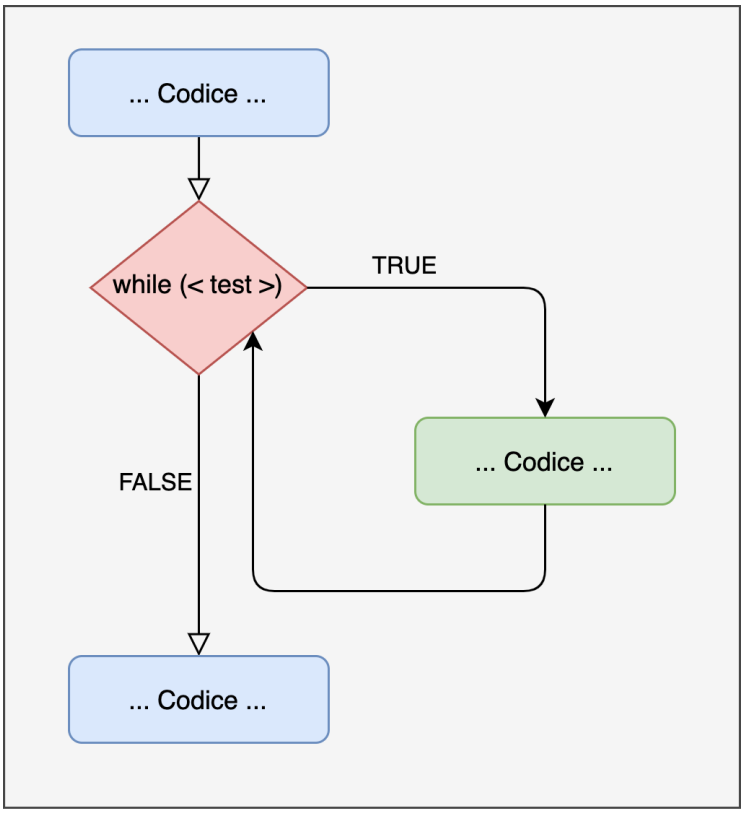


Figure 14.2: Rappresentazione while loop

Struttura While Loop

La scrittura è più concisa del ciclo `for` perchè non definiamo nessun counter o placeholder e nemmeno un vettore di valori. L'unica cosa che muove un ciclo `while` è una condizione logica (quindi con valori booleani `TRUE` e `FALSE`). Anche qui, parafrasando:

> Ripeti le operazioni incluse tra {} fino a che la condizione `<test>` è VERA.

In altri termini, ad ogni iterazione la condizione `<test>` viene valutata. Se questa è vera, viene eseguita l'operazione altrimenti il ciclo si ferma.

```
while (<test>) {
    <codice-da-eseguire>
}
```

Esempio

Se vogliamo fare un **conto alla rovescia**

```
count <- 5

while(count >= 0){
  print(count)
  count <- count - 1 # aggiorno variabile
}
## [1] 5
## [1] 4
## [1] 3
## [1] 2
## [1] 1
## [1] 0
```

Quando si scrive un ciclo `while` è importante assicurarsi di due cose:

- Che la condizione sia `TRUE` inizialmente, altrimenti il ciclo non comincierà nemmeno
- Che ad un certo punto la condizione diventi `FALSE` (perchè abbiamo ottenuto il risultato o perchè è passato troppo tempo o iterazioni)

Se la seconda condizione non è rispettata, otteniamo quelli che si chiamano `endless loop` come ad esempio:

```
count <- 5

# Attenzione loop infinito
while(count >= 0){
  print(count)
  # count <- count - 1
}
```

14.1.2.1 While e For

Abbiamo introdotto in precedenza che il `for` è un tipo particolare di `while`. Concettualmente infatti possiamo pensare ad un `for` come un `while` dove il nostro counter `i` incrementa fino alla lunghezza del vettore su cui iterare. In altri termini possiamo scrivere un `for` anche in questo modo:

```
nomi <- c("Alessio", "Beatrice", "Carlo")
i <- 1 # counter

while(i <= length(nomi)){ # condizione
  print(paste0(nomi[i], " è il numero ", i))
  i <- i + 1
}
## [1] "Alessio è il numero 1"
## [1] "Beatrice è il numero 2"
## [1] "Carlo è il numero 3"
```

14.1.3 Next e Brake

All'interno di una struttura iterativa, possiamo eseguire qualsiasi tipo di operazione, ed anche includere strutture condizionali. Alcune volte può essere utile saltare una particolare iterazione oppure interrompere il ciclo iterativo. In R tali operazioni posso essere eseguite rispettivamene coni comandi `next` e `break`.

- `next` - passa all'iterazione successiva
- `break` - interrompe l'esecuzione del ciclo

Esempio

- con `for` loop

```
my_vector <- 1:6

for (i in seq_along(my_vector)){
  if (my_vector[i] == 3) next

  if (my_vector[i] == 5) break
  print(my_vector[i])
}
## [1] 1
## [1] 2
## [1] 4
```

- con `while` loop

```
count <- 7

while(count >= 0){
  count <- count - 1
  if (count == 5) next

  if (count == 2) break

  print(count)
}
## [1] 6
## [1] 4
## [1] 3
```

14.2 Nested loop

Una volta compresa la struttura iterativa, è facile espanderne le potenzialità inserendo un ciclo all'interno di un altro. Possiamo avere quanti cicli *nested* necessari, chiaramente aumenta non solo la complessità ma anche il tempo di esecuzione. Per capire al meglio cosa succede all'interno di un ciclo nested è utile visualizzare gli indici:

```
for(i in 1:3){ # livello 1
  for(j in 1:3){ # livello 2
    for(l in 1:3){ # livello 3
      print(paste(i, j, l))
    }
  }
}
## [1] "1 1 1"
## [1] "1 1 2"
## [1] "1 1 3"
## [1] "1 2 1"
## [1] "1 2 2"
## [1] "1 2 3"
## [1] "1 3 1"
## [1] "1 3 2"
## [1] "1 3 3"
## [1] "2 1 1"
## [1] "2 1 2"
## [1] "2 1 3"
## [1] "2 2 1"
## [1] "2 2 2"
## [1] "2 2 3"
## [1] "2 3 1"
## [1] "2 3 2"
## [1] "2 3 3"
## [1] "3 1 1"
## [1] "3 1 2"
## [1] "3 1 3"
```

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

Guardando gli indici, è chiaro che il ciclo più interno viene ultimato per primo fino ad arrivare a quello più esterno. La logica è la seguente:

- La prima iterazione entriamo nel ciclo più esterno `i = 1`, poi in quello interno `j = 1` e in quello più interno `l = 1`.
- nella seconda iterazione siamo **bloccati** nel ciclo interno e quindi sia `i` che `j` saranno 1 mentre `l` sarà uguale a 2.
- quando il ciclo `l` sarà finito, `i` sarà sempre 1 mentre `j` passerà a 2 e così via

Un aspetto importante è l'utilizzo di indici diversi, infatti i valori `i`, `j` e `l` assumono valori diversi ad ogni iterazione e se usassimo lo stesso indice, non otterremo il risultato voluto.

Esercizi

1. Scrivi una funzione che calcoli la media di un vettore numerico usando un for loop.
2. Scrivi una funzione che dato un vettore numerico restituisca il valore massimo e minimo usando un for loop (attenti al valore di inizializzazione).
3. Scrivi una funzione che per ogni iterazione generi n osservazioni da una normale (funzione `rnorm()`) con media mu e deviazione standard $sigma$, salva la media di ogni campione. I parametri della funzione saranno n, mu, $sigma$ e $iter$ (numero di iterazioni).

14.3 Apply Family

Ci sono una famiglia di funzioni in R estremamente potenti e versatili chiamate `*apply`. L'asterisco suggerisce una serie di varianti presenti in R che nonostante la struttura e funzione comune hanno degli obiettivi diversi:

- `apply`: dato un dataframe (o matrice) esegue la stessa funzione su ogni riga o colonna
- `tapply`: dato un vettore di valori esegue la stessa funzione su ogni gruppo che è satato definito
- `lapply`: esegue la stessa funzione per ogni elemento di una lista. Restituisce ancora una lista
- `sapply`: esegue la stessa funzione per ogni elemento di una lista. Restituisce se possibile un oggetto seplificato (un vettore, una matrice o un array)
- `vapply`: analogo a `sapply` ma richiede di definire il tipo di dati restituiti
- `mapply`: è la versione multivariata. Permette di applicare una funzione a più liste di elementi

Prima di illustrare le varie funzioni è utile capire la struttura generale. In generale queste funzioni accettano un oggetto **lista** quindi un insieme di elementi e una **funzione**. L'idea infatti è quella di avere una funzione che accetta altre funzioni come argomenti e applichi la funzione-argomento ad ogni elemento in input. Queste funzioni, sopratutto in R, sono spesso preferite rispetto ad

utilizzare cicli `for` per velocità, compattezza e versatilità. Hadley Wickam[^talk-map] riporta un bellissimo esempio per capire la diffenza tra `loop` e `*apply`. Immaginiamo di avere una serie di vettori e voler applicare alcune funzioni ad ogni vettore, possiamo impostare un semplice loop in questo modo:

```
list_vec <- list(
  vec1 <- rnorm(100),
  vec2 <- rnorm(100),
  vec3 <- rnorm(100),
  vec4 <- rnorm(100),
  vec5 <- rnorm(100)
)

means <- vector(mode = "numeric", length = length(list_vec))
medians <- vector(mode = "numeric", length = length(list_vec))
st_devs <- vector(mode = "numeric", length = length(list_vec))

for(i in seq_along(list_vec)){
  means[i] <- mean(list_vec[[i]])
  medians[i] <- median(list_vec[[i]])
  st_devs[i] <- sd(list_vec[[i]])
}
```

Nonostante sia perfettamente corretto, questa scrittura ha diversi problemi:

- E' molto ridondante. Tra calcolare media, mediana e deviazione standard l'unica cosa che cambia è la funzione applicata mentre dobbiamo per ognuno preallocare una variabile, impostare l'indicizzazione in base all'iterazione per selezionare l'elemento della lista e memorizzare il risultato. Per migliorare questa scrittura possiamo mettere in una funzione tutta questa struttura (preallocazione, indicizzazione e memorizzazione) e utilizzare questa funzione con argomenti la lista di input e la funzione da applicare. Utilizzando la funzione `sapply`:

```
means <- lapply(list_vec, mean)
means
## [[1]]
## [1] -0.1744844
##
## [[2]]
## [1] -0.001581971
##
## [[3]]
## [1] 0.003746107
##
## [[4]]
## [1] -0.1040844
##
## [[5]]
## [1] 0.2978849
medians <- lapply(list_vec, median)
medians
```

```
## [[1]]
## [1] -0.1336871
##
## [[2]]
## [1] 0.08244486
##
## [[3]]
## [1] 0.002455108
##
## [[4]]
## [1] -0.00886783
##
## [[5]]
## [1] 0.2095848
st_devs <- lapply(list_vec, sd)
st_devs
## [[1]]
## [1] 1.028626
##
## [[2]]
## [1] 1.056527
##
## [[3]]
## [1] 1.027976
##
## [[4]]
## [1] 0.9351077
##
## [[5]]
## [1] 1.04055
```

Come vedete il codice diventa estremamente compatto, pulito e facile da leggere.

14.3.1 Quali funzioni applicare?

Prima di descrivere nel dettaglio ogni funzione `*apply` è importante capire quali tipi di funzioni possiamo usare all'interno di questa famiglia. In generale, qualsiasi funzione può essere applicata ma per comodità possiamo distinguerle in:

- funzioni già presenti in R
- funzioni personalizzate (create e salvate nell'ambiente principale)
- funzioni **anonime**

Nell'esempio precedente, abbiamo utilizzato la funzione `mean` semplicemente scrivendo `lapply(lista, mean)`. Questo è possibile perchè `mean` necessita di un solo argomento. Se tuttavia volessimo applicare funzioni più complesse o aggiungere argomenti possiamo usare la scrittura più generale:

```
means <- lapply(list_vec, function(x) mean(x))
means
## [[1]]
## [1] -0.1744844
##
## [[2]]
## [1] -0.001581971
##
## [[3]]
## [1] 0.003746107
##
## [[4]]
## [1] -0.1040844
##
## [[5]]
## [1] 0.2978849
```

L'unica differenza è che abbiamo definito una funzione **anonima** con la scrittura `function(x) ...`. Questa scrittura si interpreta come "ogni elemento di `list_vec` diventa `x`, quindi applica la funzione `mean()` per ogni elemento di `list_vec`". La funzione anonima permette di scrivere delle funzioni non salvate o presenti in R e applicare direttamente ad una serie di elementi. Possiamo anche usare funzioni più complesse come centrare ogni elemento di `list_vec`:

```
centered_list <- lapply(list_vec, function(x) x - mean(x))
centered_list
## [[1]]
##   [1]  0.052024420  0.726941033  0.523133907  0.534116649  1.072538094
##   [6] -1.748085116  0.436228769  1.090050776  0.188256343  1.904447568
##  [11] -0.907720449 -0.098340775  0.356479804  1.683026192  1.778954510
##  [16] -1.666991209  1.797794611  0.305873422  1.655606875  1.687802692
##  [21] -0.767958869 -0.011200606 -0.926640234  1.382599658 -1.450454139
##  [26]  0.279862739 -1.280958942 -0.179531732  0.080784364  1.275153032
##  [31] -1.789340704 -1.273459981  1.193927825 -1.246932674 -0.430047712
##  [36] -1.408989494 -1.111447945 -1.280200478  0.087413286  0.679220848
##  [41]  0.290873113  1.934698137 -0.170632058  2.294484562  0.140106914
##  [46] -0.617669647  1.649999618 -0.551072804  0.486863447  0.866448514
##  [51] -0.325806393 -2.081384947  0.218225732 -0.194333682 -0.785738000
##  [56]  0.278250713  0.601773540  0.004002865 -1.374655893 -1.331115539
##  [61]  0.190527940 -0.010879909  0.566417661 -0.582226517  0.405902013
##  [66] -0.809129005  0.739565224  1.791236297 -0.077479735 -0.881394205
##  [71] -0.173747353  0.131494431 -1.223069551  1.664700735 -0.864902716
##  [76] -0.062460642 -0.824657046 -1.218058185  1.156489655  0.535425340
##  [81] -0.163024811 -0.468903190 -1.992400872  0.807773400  0.029570256
##  [86] -1.065542698  0.708443728 -1.413780392 -0.816480128  0.657745237
##  [91]  0.985102777 -0.119180161  0.121026086  0.909668900  0.189469398
##  [96]  0.052482555 -0.472289289 -0.693373923 -0.334215899 -1.903099951
##
## [[2]]
##   [1] -0.25875450  0.45192195 -0.14129965 -0.48513950 -1.19419119  0.04852296
##   [7] -0.12494971 -2.69913307 -0.56923136  0.59317969  0.48855867 -0.12521792
##  [13] -1.25761789  0.20287395 -1.91532248  1.67432109  0.47237159  1.41569682
```

```
##  [19]  0.08588055 -1.80072157  0.75532561 -0.31036112 -1.73097833 -2.13698056
##  [25]  2.36738038  0.48633857  1.09481961  0.30449038  1.01688141  2.45517490
##  [31] -0.24403885  0.54310131  0.19845858 -2.06958035  0.51416551 -0.40417991
##  [37]  0.35777998 -0.32998890  0.08217311 -0.25995052 -0.87586827  0.74289078
##  [43] -2.68138122 -0.94789651  0.44784397 -1.28724749 -0.15622349  0.34939757
##  [49] -0.05804050  1.47831075 -0.65258270 -0.25523286 -1.25245596  0.77262514
##  [55] -0.90976997 -0.69172340 -0.61608640  0.76370516 -1.08557639 -0.39821755
##  [61]  0.82937843  0.35700165  0.16072076  0.95697716 -0.33806032 -0.72579950
##  [67] -1.69622426  1.95572036  2.66831987  2.06496139  0.82036535 -0.07806814
##  [73] -0.48786746  0.84930072 -0.95746280  0.93026931  0.38254698  1.49618576
##  [79] -0.46612240  0.26273926 -0.99102667 -1.06179903  0.27586663  0.94692452
##  [85]  0.72777105 -0.25293070  1.48676214  0.23186813  0.27958462  0.14862272
##  [91] -1.19470741  0.09171824  1.22085175 -0.55991059  0.33846373 -1.53521819
##  [97] -0.23854774  0.51644745 -0.23727325  0.58340054
##
## [[3]]
##   [1]  0.2664491638 -1.3469963272 -0.8526350420 -0.4113540385 -0.6698965617
##   [6] -0.1069834877 -0.9277972315  0.4348533514 -0.0158461161 -0.6496568470
##  [11]  1.3404006827  0.3308553677 -0.0008524137  1.0932070221  1.1595245506
##  [16]  0.0970402348 -0.3954985959 -0.4148154462 -0.9173273893 -0.5441391984
##  [21]  0.1176161927  1.7239915983 -0.0479376631  0.5410962859  1.9136056395
##  [26] -0.2403162233  1.5678772409  0.4771039676  0.0388510442  0.4360775197
##  [31] -1.8843109374 -1.7278160519  1.8794222142 -0.0350334621  0.9952978517
##  [36]  0.2965731726 -0.0089397169 -0.2440504016 -1.7747293817 -0.1599453739
##  [41]  1.9116204426  1.1063735070 -1.9894549474  0.8106197600  1.0945052782
##  [46] -1.4444401450  1.2114915468 -0.7866311249 -2.0708321586  0.3306083127
##  [51]  0.2571921978 -0.4380882424 -1.8767536475 -0.8067906332  0.3285759849
##  [56]  0.0083665339  0.8463602573 -1.5346139219 -0.0353043768  1.4238938000
##  [61] -0.9307764997  1.0018625824 -0.0897035610  0.9342787542 -0.5220418242
##  [66] -0.9382625616  1.2278393538 -0.2433379966  0.2694431409  1.4270790033
##  [71] -1.1447635626  0.8922605510  0.0790918016  0.3463267640 -1.2620371186
##  [76]  0.8359400483 -1.2621666167  0.1493675938 -0.4874471285  2.2071080499
##  [81]  0.0770783378  0.7123603570  1.0507096270  0.2724235769 -0.4240092719
##  [86]  2.5187562528 -0.0017295860 -1.5659889251 -1.9915300574 -1.4957999457
##  [91]  0.2514953082 -0.8182432673  0.8301238234 -1.1562445178 -0.1441958015
##  [96]  1.1723315781  0.1831991812 -0.6985203200 -0.4631891217 -0.4787516183
##
## [[4]]
##   [1]  1.31014158  1.09819466  1.64324210  0.39593137  0.61463274 -0.43683054
##   [7] -0.56982943  0.78423560  0.29079922 -1.28144712 -0.95320981  0.47832305
##  [13] -0.89843251  0.06142016  0.09834140 -0.93750994 -0.22114701 -0.50447853
##  [19]  1.62135004  0.40054994  0.50377989  0.85258426  0.01601205 -0.98150469
##  [25]  0.01879710  0.56958632  0.10405552  0.72039850 -0.54796577  0.55132606
##  [31] -0.07849753 -0.70235630 -1.70546632 -0.67537030 -1.88424294  1.10131999
##  [37] -0.93803709  0.30884776  0.38639004  0.26406936 -1.48220109  0.64941069
##  [43] -1.26245228 -1.01196361  0.26655677  1.61573387 -0.24499703 -1.86615753
##  [49] -0.74331502 -1.66625739  1.19375590 -1.29834857  1.03931548  0.09209179
##  [55]  0.36417167  1.25497932  1.27163836 -1.08423536  0.32463046  1.23838613
##  [61]  0.61505477 -1.39306979 -1.01347083 -0.65377057 -0.67234873 -0.15951329
##  [67] -1.09475421 -1.24828431  0.44936851 -0.81031734  0.48577901 -0.44783374
##  [73]  0.13878656 -0.42861642  1.05753379  0.80005894 -0.56816795  1.42707901
```

```
##  [79] -0.24334500 -0.42674961  0.83810754  0.25860309  0.76271285 -0.83189895
##  [85] -1.83769866  0.13777046 -0.34121388  1.28152635  0.02692627  0.65523055
##  [91] -0.34789338 -1.27102060  0.10899483  1.00604579 -0.82845156  0.66493483
##  [97]  0.41550756  1.49611518  2.54392757 -0.08039012
##
## [[5]]
##   [1] -0.97244116 -0.05975831  0.24720098 -0.74673643  0.67336181 -1.84504878
##   [7] -0.10088763  0.54085362  0.19802171 -0.21846188 -1.35504038 -1.00568442
##  [13] -1.66143029  0.32770743 -0.87494585  1.42240563  1.33871389 -0.49724504
##  [19]  0.23183020 -1.28236562  0.94682403  0.15097940 -0.08996773 -1.87922549
##  [25]  0.56434072  1.01141751  0.59054641  0.90708246  1.81054234  1.52125463
##  [31]  1.59879629  0.65703478 -1.36100263  1.52636999 -0.67568978  1.18846185
##  [37] -0.71371197 -0.62705645 -0.34696123  1.81946720 -0.14410534  0.77391802
##  [43] -1.40734286 -0.22652747 -0.08663243  0.24917797  0.10482833  0.40064474
##  [49] -0.11465530  1.67669566  0.89489601 -1.18726409 -0.69802085 -0.34089280
##  [55] -0.31607492 -1.19132141 -1.34752712  0.06417591  0.57868314  1.93393514
##  [61]  1.29099212  0.28454888 -1.05159708 -0.12011029 -1.39846790 -0.34956404
##  [67]  1.15216589  0.47442520  0.42866114  1.24934339 -0.17264100 -0.18756486
##  [73]  1.52868692 -0.50630793 -0.52108509 -2.27123722  0.10655986 -1.07339635
##  [79] -0.53370249  1.60949570 -0.08346146 -1.11549234 -1.60046934 -0.70496768
##  [85]  2.72371679 -0.15590380 -1.05128938  0.86727635  0.28486870 -1.34945579
##  [91] -0.81570377 -1.37540652  0.69447464 -0.31187353 -2.04954616  0.56538309
##  [97]  0.55236386  1.42950364  1.15527183 -0.17463612
```

In questo caso, è chiaro che `x` è un placeholder che assume il valore di ogni elemento della lista `list_vec`.

L'uso di funzioni anonime è estremamente utile e chiaro una volta compresa la notazione. Tuttavia per funzioni più complesse è più conveniente scrivere salvare la funzione in un oggetto e poi applicarla come per `mean`. Usando sempre l'esempio di centrare la variabile possiamo:

```
center_vec <- function(x){
  return(x - mean(x))
}

centered_list <- lapply(list_vec, center_vec)
```

Possiamo comunque applicare funzioni complesse come **anonime** semplicemente utilizzando le parentesi graffe proprio come se dovessimo dichiarare una funzione:

```
center_vec <- function(x){
  return(x - mean(x))
}

centered_list <- lapply(list_vec, function(x){
  res <- x - mean(x)
  return(res)
})
```

Un ultimo aspetto riguarda un parallismo tra `x` nei nostri esempi e `i`nei cicli `for` che abbiamo visto in precedenza. Proprio come `i`, `x` è una semplice convenzione e si può utilizzare qualsiasi nome per

definire l'argomento generico. Inoltre, è utile pensare a `x` proprio con lo stesso ruolo di `i`, infatti se pensiamo alla funzione in precedenza, `x` ad ogni iterazione prende il valore di un elemento di `list_vec` proprio come utilizzare il ciclo `for` non con gli indici ma con i valori del vettore su cui stiamo iterando. Qualche volta infatti può essere utile applicare un principio di **indicizzazione** anche con l' `*apply` family:

```
means <- lapply(seq_along(list_vec), function(i) mean(list_vec[[i]]))
means
## [[1]]
## [1] -0.1744844
##
## [[2]]
## [1] -0.001581971
##
## [[3]]
## [1] 0.003746107
##
## [[4]]
## [1] -0.1040844
##
## [[5]]
## [1] 0.2978849
```

In questo caso l'argomento non è più la lista ma un vettore di numeri da 1 alla lunghezza della lista (proprio come nel ciclo `for`). La funzione anonima poi prende come argomento `i` (che ricordiamo può essere qualsiasi nome) e utilizza `i` per indicizzare e applicare una funzione. In questo caso non è estremamente utile, ma con questa scrittura abbiamo riprodotto esattamente la logica del ciclo `for` in un modo estremamente compatto.

14.3.2 apply

La funzione `apply` viene utilizzata su **matrici**, **dataframe** per applicare una funzione ad ogni dimensione (riga o colonna). La struttura della funzione è questa:

```
apply(X = , MARGIN = , FUN = , ...)
```

Dove:

- `X` è il dataframe o la matrice
- `MARGIN` è la dimensione su cui applicare la funzione: `1` = riga e `2` = colonna
- `FUN` è la funzione da applicare

Esempi

- Semplici funzioni

```r
my_matrix <- matrix(1:24, nrow = 4, ncol = 6)

# Per riga
apply(my_matrix, MARGIN = 1, FUN = sum)
## [1] 66 72 78 84

# Per colonna
apply(my_matrix, MARGIN = 2, FUN = sum)
## [1] 10 26 42 58 74 90
```

- Funzioni complesse

```r
# Coefficiente di Variazione
apply(my_matrix, MARGIN = 2, FUN = function(x){
  mean <- mean(x)
  sd <- sd(x)

  return(round(sd/mean,2))
})
## [1] 0.52 0.20 0.12 0.09 0.07 0.06
```

14.3.3 tapply

`tapply` è utile quando vogliamo applicare una funzione ad un elemento che viene **diviso** in base ad un'altra variabile. La scrittura è la seguente:

```r
tapply(X = , INDEX = , FUN = , ...)
```

Dove:

- `X` è la variabile principale
- `INDEX` è la variabile in base a cui suddividere `X`
- `FUN` è la funzione da applicare

Esempi

```r
my_data <- data.frame(
  y = sample(c(2,4,6,8,10), size = 32, replace = TRUE),
  gender = factor(rep(c("F", "M"), each = 16)),
  class = factor(rep(c("3", "5"), times = 16))
)

head(my_data, n = 4)
##    y gender class
## 1  8      F     3
## 2 10      F     5
```

```
## 3  2      F     3
## 4  4      F     5

# Media y per classe
tapply(my_data$y, INDEX = my_data$class, FUN = mean)
##     3     5
## 5.875 6.625

# Media y per classe e genere
tapply(my_data$y, INDEX = list(my_data$class, my_data$gender), FUN = mean)
##      F    M
## 3 6.50 5.25
## 5 5.75 7.50
```

14.3.4 lapply

E' forse la funzione più utilizzata e generica. Viene applicata ad ogni tipo di dato che sia una lista di elementi o un vettore. La particolarità è quella di restituire sempre una lista come risultato, indipendentemente dal tipo di input. La scrittura è la seguente:

```
lapply(X = , FUN = , ...)
```

Dove:

- `X` è il vettore o lista
- `FUN` è la funzione da applicare

Esempi

```
my_list <- list(
  sample_norm = rnorm(10, mean = 0, sd = 1),
  sample_unif = runif(15, min = 0, max = 1),
  sample_pois = rpois(20, lambda = 5)
)

str(my_list)
## List of 3
##  $ sample_norm: num [1:10] 0.343 1.371 0.26 0.482 0.309 ...
##  $ sample_unif: num [1:15] 0.031 0.011 0.882 0.113 0.323 ...
##  $ sample_pois: int [1:20] 2 3 7 9 5 3 3 5 2 4 ...

# Media
lapply(my_list, FUN = mean)
## $sample_norm
## [1] 0.5029392
##
## $sample_unif
```

```
## [1] 0.3935366
##
## $sample_pois
## [1] 4.25
```

14.3.5 sapply

`sapply` ha la stessa funzionalità di `lapply` ma ha anche la capacità di restituire una versione semplificata (se possibile) dell'output.

```
sapply(X = , FUN = , ... )
```

Esempi

```
# Media
sapply(my_list, FUN = mean)
## sample_norm sample_unif sample_pois
##   0.5029392   0.3935366   4.2500000
```

Per capire la differenza, applichiamo sia `lapply` che `sapply` con gli esempi precedenti:

```
sapply(list_vec, mean)
## [1] -0.174484405 -0.001581971  0.003746107 -0.104084425  0.297884925
lapply(list_vec, mean)
## [[1]]
## [1] -0.1744844
##
## [[2]]
## [1] -0.001581971
##
## [[3]]
## [1] 0.003746107
##
## [[4]]
## [1] -0.1040844
##
## [[5]]
## [1] 0.2978849
sapply(list_vec, mean, simplify = FALSE)
## [[1]]
## [1] -0.1744844
##
## [[2]]
## [1] -0.001581971
##
## [[3]]
## [1] 0.003746107
```

```
##
## [[4]]
## [1] -0.1040844
##
## [[5]]
## [1] 0.2978849
```

Come vedete, il risultato di queste operazioni corrisponde ad un valore per ogni elemento della lista `list_vec`. `lapply` restituisce una lista con i risultati mentre `sapply` resituisce un vettore. Nel caso di risultati singoli come in questa situazione, l'utilizzo di `sapply` è conveniente mentre mantenere la struttura a lista può essere meglio in altre condizioni. Possiamo comunque evitare che `sapply` semplifichi l'output usando l'argomento `simplify = FALSE`.

14.3.6 vapply

```
vapply(X = , FUN = , FUN.VALUE =  ,... )
```

Esempi

`vapply` è una ancora una volta simile sia a `lapply` che a `sapply`. Tuttavia richiede che il tipo di output sia specificato in anticipo. Per questo motivo è ritenuta una versione più *solida* delle precedenti perchè permette più controllo su quello che accade.

```
# Media
vapply(my_list, FUN = mean, FUN.VALUE = numeric(length = 1L))
## sample_norm sample_unif sample_pois
##   0.5029392   0.3935366   4.2500000
```

In questo caso come in precendeza definiamo la lista su cui applicare la funzione. Tuttavia l'argomento `FUN.VALUE = numeric(length = 1L)` specifica che ogni risultato dovrà essere un valore di tipo `numeric` di lunghezza 1. Infatti applicando la media otteniamo un singolo valore per iterazione e questo valore è necessariamente numerico.

> 🌵 Warning-Box: sapply() vs vapply()
>
> `sapply()` non restituisce sempre la stessa tipologia di oggetto mentre `vapply()` richiede sia specificato il tipo di l'output di ogni iterazione.

```r
x1 <- list(
  sample_unif = c(-1, runif(15, min = 0, max = 1)),
  sample_norm = rnorm(5, mean = 0, sd = 1),
  sample_pois = rpois(20, lambda = 5)
)
x2 <- list(
  sample_gamma = c(-1, rgamma(10, shape = 1)),
  sample_unif = c(-2, runif(15, min = 0, max = 1)),
  sample_pois = c(-3, rpois(20, lambda = 5))
)

negative_values <- function(x) x[x < 0]
sapply(x1, negative_values)
## $sample_unif
## [1] -1
##
## $sample_norm
## [1] -0.95907200 -0.08395547
##
## $sample_pois
## integer(0)
sapply(x2, negative_values)
## sample_gamma  sample_unif  sample_pois
##           -1           -2           -3

vapply(x1, negative_values, FUN.VALUE = numeric(1))
## Error in vapply(x1, negative_values, FUN.VALUE = numeric(1)): values must be
##  but FUN(X[[2]]) result is length 2
vapply(x2, negative_values, FUN.VALUE = numeric(1))
## sample_gamma  sample_unif  sample_pois
##           -1           -2           -3
```

14.3.7 Lista di funzioni a lista di oggetti

L'`*apply` family permette anche di estendere la formula "applica una funzione ad una lista di argomenti" applicando diverse funzioni inm modo estremamente compatto. Le funzioni infatti sono oggetti come altri in R e possono essere contenute in liste:

```r
list_funs <- list(
  "mean" = mean,
  "median" = median,
  "sd" = sd
```

```
)

lapply(list_funs, function(f) sapply(list_vec, function(x) f(x)))
## $mean
## [1] -0.174484405 -0.001581971  0.003746107 -0.104084425  0.297884925
##
## $median
## [1] -0.133687067  0.082444857  0.002455108 -0.008867830  0.209584846
##
## $sd
## [1] 1.0286259 1.0565268 1.0279756 0.9351077 1.0405497
```

Quello che abbiamo fatto è creare una lista di funzioni e poi scrivere due `lapply` e `sapply` in modo *nested*. Proprio come quando scriviamo due `loop` nested, la stessa funzione viene applicata a tutti gli elementi, per poi passare alla funzione successiva. Il risultato infatti è una lista dove ogni elemento contiene i risultati applicando ogni funzione. Questo tipo di scrittura è più rara da trovare, tuttavia è utile per capire la logica e la potenza di questo approccio.

:::

14.3.8 mapply

`mapply` è la versione più complessa di quelle considerate perchè estende a n il numero di liste che vogliamo utilizzare. La scrittura è la seguente:

```
mapply(FUN, ...)
```

Dove:

- `FUN` è la funzione da applicare
- `...` sono le liste di elementi su cui applicare la funzione. E' importante che tutti gli elementi siano della stessa lunghezza

Proviamo a generare dei vettori da una distribuzione normale, usando la funzione `rnorm()` con diversi valori di numerosità, media e deviazione standard.

```
ns <- c(10, 3, 5)
means <- c(10, 20, 30)
sds <- c(2, 5, 7)

mapply(function(x, y, z) rnorm(x, y, z), # funzione
       ns, means, sds) # argomenti
## [[1]]
##  [1]  6.953099  6.854824  9.080728 11.687548  7.215676  7.706871 11.480102
##  [8]  7.142393  8.784290 13.292173
##
## [[2]]
## [1] 21.99553 14.81288 18.84860
##
## [[3]]
## [1] 33.26673 22.67479 26.38153 26.86117 25.80392
```

La scrittura è sicuramente meno chiara rispetto agli esempi precedenti ma l'idea è la seguente:

- la funzione **anonima** non ha solo un argomento ma n argomenti
- gli argomenti sono specificati in ordine, quindi nel nostro esempio `x = ns, y = means e z = sds`
- ogni iterazione, la funzione `rnorm` ottiene come argomenti una diversa numerosità, media e deviazione standard

14.4 Replicate

`replicate` è una funzione leggermente diversa ma estremamente utile. Permette di ripetere una serie di operazioni un numero prefissato di volte.

```
replicate(n = , expr = )
```

Dove:

- `n` è il numero di ripetizioni
- `expr` è il codice da ripetere

Esempi

- Semplice

```
sample_info <- replicate(n = 1000,{
  my_sample <- rnorm(n = 20, mean = 0, sd = 1)
  my_mean <- mean(my_sample)

  return(my_mean)
})

str(sample_info)
##  num [1:1000] 0.395 -0.057 0.252 0.225 0.286 ...
```

- Complesso

```
sample_info <- replicate(n = 1000,{
  my_sample <- rnorm(n = 20, mean = 0, sd = 1)
  my_mean <- mean(my_sample)
  my_sd <- sd(my_sample)

  return(data.frame(mean = my_mean,
                    sd = my_sd))
}, simplify = FALSE)


sample_info <- do.call("rbind", sample_info)
```

```
str(sample_info)
## 'data.frame':    1000 obs. of  2 variables:
##  $ mean: num  0.0966 -0.2776 0.1665 0.166 -0.1766 ...
##  $ sd  : num  1.093 1.059 1.105 1.109 0.954 ...
head(sample_info)
##          mean        sd
## 1  0.09660155 1.0926324
## 2 -0.27757342 1.0591981
## 3  0.16649003 1.1050091
## 4  0.16595363 1.1089836
## 5 -0.17656075 0.9537385
## 6 -0.18156384 0.9634895
```

E' importante sottolineare che la ripetizione è alla base di qualsiasi struttura iterativa che abbiamo visto finora. Infatti lo stesso risultato (al netto di leggibilità, velocità e versalità) lo possiamo ottenere indistintamente con un ciclo `for`, `lapply` o `replicate`. Riprendendo l'esempio precedente:

```
# Replicate

sample_info <- replicate(n = 1000,{
  my_sample <- rnorm(n = 20, mean = 0, sd = 1)
  my_mean <- mean(my_sample)

  return(my_mean)
})

str(sample_info)
##  num [1:1000] -0.109 -0.153 0.132 0.102 -0.202 ...

# *apply

sample_info <- sapply(1:1000, function(x){
  my_sample <- rnorm(n = 20, mean = 0, sd = 1)
  my_mean <- mean(my_sample)
})

str(sample_info)
##  num [1:1000] -0.0768 -0.1913 -0.0709 0.0338 0.0372 ...

# for

sample_info <- vector(mode = "numeric", length = 1000)

for(i in 1:1000){
  my_sample <- rnorm(n = 20, mean = 0, sd = 1)
  sample_info[i] <- mean(my_sample)
}

str(sample_info)
##  num [1:1000] -0.314 0.1203 0.1121 -0.1172 -0.0241 ...
```

Case study

Introduzione

Working in progress.

Chapter 15

Caso Studio I: Attaccamento

Working in progress.

Script Analisi

15.1 Infobox

Illustrations included in `images/` are retrieved from rstudio4edu-book under CC-BY-NC. Remember to include an *Attributions* section in the book and repository's README file.

Tip-Box: My title

Lorem ipsum dolor sit amet consectetur adipisicing elit. Maxime mollitia, molestiae quas vel sint commodi repudiandae consequuntur voluptatum laborum numquam blanditiis harum quisquam eius sed odit fugiat iusto fuga praesentium optio, eaque rerum!

Warning-Box: My title

Lorem ipsum dolor sit amet consectetur adipisicing elit. Maxime mollitia, molestiae quas vel sint commodi repudiandae consequuntur voluptatum laborum numquam blanditiis harum quisquam eius sed odit fugiat iusto fuga praesentium optio, eaque rerum!

Definition-Box: My title

Lorem ipsum dolor sit amet consectetur adipisicing elit. Maxime mollitia, molestiae quas vel sint commodi repudiandae consequuntur voluptatum laborum numquam blanditiis harum quisquam eius sed odit fugiat iusto fuga praesentium optio, eaque rerum!

Approfondimento: My title

Lorem ipsum dolor sit amet consectetur adipisicing elit. Maxime mollitia, molestiae quas vel sint commodi repudiandae consequuntur voluptatum laborum numquam blanditiis harum quisquam eius sed odit fugiat iusto fuga praesentium optio, eaque rerum!

Trick-Box: My title

Lorem ipsum dolor sit amet consectetur adipisicing elit. Maxime mollitia, molestiae quas vel sint commodi repudiandae consequuntur voluptatum laborum numquam blanditiis harum quisquam eius sed odit fugiat iusto fuga praesentium optio, eaque rerum!

Argomenti avanzati

Introduzione

In questa sezione verranno introdotti brevemente alcuni argomenti avanzati che sono tuttavia molto utili per lavorare in modo efficiente con R. Verranno inseriti diversi riferimenti in modo da poter approfondire autonomamente.

I capitoli sono così organizzati:

- **Capitolo 16 - Stringhe**. Vedremo come gestire e manipolare le gli oggetti di tipo `character` in R.

Chapter 16

Stringhe

Abbiamo visto che R oltre ai numeri è in grado di gestire anche i **caratteri**. Nonostante le operazioni matematiche non siano rilevanti per questo tipo di dato, lavorare con le stringhe è altrettanto se non più complesso in termini di programmazione, le stringe infatti rispetto ai numeri:

- possono essere maiuscole/minuscole. Ad esempio la stringa `ciao` è *concettualmente* uguale a `Ciao` ma R le tratta in modo diverso.

```
"ciao" == "Ciao"
```

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

- possono avere caratteri speciali come `?\$` oppure appartenere ad alfabeti diversi
- l'indicizzazione per `numeri` e `stringhe` lavora in modo diverso. Se abbiamo un **vettore** di strighe, questo viene rappresentato allo stesso modo di un vettore numerico. Tuttavia la stringa stessa `ciao` può essere scomposta, manipolata e quindi indicizzata nei singoli caratteri che la compongono `c, i, a, o`:

```
vec_string <- c("ciao", "come", "stai")
vec_num <- c(1,2,3)

length(vec_string)
```

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

```
length(vec_num)
```

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

```
vec_string[1]
```

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

```
vec_num[1]
```

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

```
# Usiamo la funzione nchar()
setNames(nchar(vec_string), vec_string)
```

```
## ciao come stai
##    4    4    4
```

Per creare una stringa in R dobbiamo usare le singole o doppie virgolette `"stringa"` o `'stringa'`. Queste due scritture in R sono interpretate in modo analogo. Possiamo usarle entrambe per scrivere una stringa che contenga le virgolette:

```
x <- "stringa con all'interno un'altra 'stringa'"
x

x <- "stringa con all'interno un'altra "stringa""
# in questo caso abbaimo errore perchè non interpreta la doppia virgolettatura
```

```
## Error: <text>:4:41: unexpected symbol
## 3:
## 4: x <- "stringa con all'interno un'altra "stringa
##                                           ^
```

All'interno delle stringhe possiamo utilizzare caratteri speciali come `/|\$%&`. Alcuni di questi vengono intepretati da R in modo particolare. Quando accade è necessario aggiungere il carattere `\` che funge da *escape*, ovvero dice ad R di trattare letterlamente quel carattere:

```
x <- "ciao come stai? n io tutto bene"
cat(x)
```

```
## ciao come stai? n io tutto bene
```

Per questo in R ci sono una serie di funzioni e pacchetti che permettono di lavorare con le stringhe in modo molto efficiente. Vedremo qui una breve panoramica di queste funzioni con qualche suggerimento anche su come approfondire.

16.1 Confrontare stringhe

Il primo aspetto è quello di confrontare stringhe. Il confronto logico tra `stringhe` è molto più stringente di quello numerico. Come abbiamo visto prima infatti, c'è molta più libertà rispetto alle stringhe, con il prezzo di avere più scenari da gestire.

```
# Confronto due numeri rappresentati in modo diverso
intero <- as.integer(10)
double <- as.numeric(10)
intero == double
```

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

```
# Confronto stringhe

"ciao" == "Ciao"
```

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

```
"female" == "feMale"
```

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

Anche il concetto di spazio è rilevante perchè viene considerato come un carattere:

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

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

Immaginate di avere un vettore dove una colonna rappresenta il genere dei partecipanti. Se questo vettore è il risultato di persone che liberamente scrivono nel campo di testo, potreste trovarvi una situazione così (è per questo che nei form online spesso ci sono opzioni predefinite piuttosto che testo libero):

```
genere <- c("maLe", "masChio", "Male", "f", "female", "malew")
```

In questo vettore (volutamente esagerato) abbiamo chiaro il significato di `f` o di `malew` (probabilmente un errore di battitura) tuttavia se vogliamo lavorarci con R, diventa probelmatico:

```
# Tabella di frequenza

table(genere)
```

```
## genere
##       f  female    maLe    Male   malew masChio
##       1       1       1       1       1       1
```

```
# Non molto utile
```

16.2 Comporre stringhe

Vediamo quindi alcune funzioni utili per lavorare con le stringhe.

16.2.1 `tolower()` e `toupper()`

Queste funzioni sono estremamente utili perchè permettono di forzare il carattere maiscolo o minuscolo

```
tolower(genere)
```

```
## [1] "male"    "maschio" "male"    "f"       "female"  "malew"
```

```
toupper(genere)
```

```
## [1] "MALE"    "MASCHIO" "MALE"    "F"       "FEMALE"  "MALEW"
```

16.2.2 `paste()` e `paste0()`

Queste funzioni servono a combinare diverse informazioni in una stringa. Possiamo combinare diverse stringhe ma anche numeri. Come tipico in R, `paste()` e `paste0()` sono vettorizzate e quindi possono essere utili per combinare due vettori di informazioni. La differenza è che `paste()` automaticamente aggiunge uno spazio tra le stringhe combinate mentre con `paste0()` deve essere messo esplicitamente.

```
age <- c(10, 20, 35, 15, 18)
nomi <- c("Andrea", "Francesco", "Fabio", "Anna", "Alice")

paste(nomi, "ha", age, "anni")
```

```
## [1] "Andrea ha 10 anni"    "Francesco ha 20 anni" "Fabio ha 35 anni"
## [4] "Anna ha 15 anni"      "Alice ha 18 anni"
```

```
paste0(nomi, "ha", age, "anni")
```

```
## [1] "Andreaha10anni"    "Francescoha20anni" "Fabioha35anni"
## [4] "Annaha15anni"      "Aliceha18anni"
```

```
paste0(nomi, " ha ", age, " anni")
```

```
## [1] "Andrea ha 10 anni"    "Francesco ha 20 anni" "Fabio ha 35 anni"
## [4] "Anna ha 15 anni"      "Alice ha 18 anni"
```

In questo caso nonostante `age` sia numerico, viene forzato a stringa per poter essere combinato con i nomi.

16.2.3 `sprinf()`

`sprinf()` è simile a `paste*()` come funzionamento ma permette di comporre strighe usando dei *placeholder* e fornendo poi il contenuto.

```
sprintf("%s ha %d anni", nomi, age)
```

```
## [1] "Andrea ha 10 anni"    "Francesco ha 20 anni" "Fabio ha 35 anni"
## [4] "Anna ha 15 anni"      "Alice ha 18 anni"
```

In questo caso si compone una stringa usando `%` e una lettera che rappresenta il tipo di dato da inserire. Poi in ordine vengono fornite le informazioni. In questo caso prima `%s`(stringa) quindi `nomi` e poi `%d`(digits) quindi `age`. Con `?sprintf` avete una panoramica del tipo di *placeholder* che potete utilizzare.

16.3 Indicizzare stringhe

16.3.1 `nchar()`

Come abbiamo visto prima la stringa è formata da un insieme di caratteri. La funzione `nchar()` fornisce il numero di singoli caratteri che compongono una stringa.

```
nchar("ciao")
```

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

```
nchar("Wow lavorare con le stringhe è molto divertente")
```

```
## [1] 47
```

16.3.2 `gregexpr()` e `regexpr()`

Per trovare la posizione di una o più caratteri all'interno di una stringa possiamo usare `gregexpr()`. La scrittura è `(g)gregexpr(pattern, stringa)`:

```
gregexpr("t", "gatto")
```

```
## [[1]]
## [1] 3 4
## attr(,"match.length")
## [1] 1 1
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

```
regexpr("t", "gatto")
```

```
## [1] 3
## attr(,"match.length")
## [1] 1
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

La differenza è che `regexpr()` restituisce solo la prima corrispondenza, nel nostro esempio la prima `t` si trova in 3 posizione mentre `gregexpr()` resituisce tutte le corrispondenze.

16.3.3 `substr()` e `substring()`

Il processo inverso, quindi trovare la stringa che corrisponde ad un certo indice è il lavoro di `substr(stringa, start, stop)` dove `start` e `stop` sono gli indici della porzione di stringa che vogliamo trovare. `substring()` funziona allo stesso modo ma `start` e `stop` vengono chiamati `first` e `last`.

```
substr("gatto", 1, 1) # solo la prima
```

```
## [1] "g"
```

```
substr("gatto", 2, 4) # dalla seconda alla quarta
```

```
## [1] "att"
```

Per questo tipo di compiti forniscono esattamente lo stesso risultato, vediamo quindi le differenze:

- `substring()` permette di fornire solo l'indice iniziale `first` e quello finale ha un valore di default di `1000000L`
- `substring()` permette anche di fornire un vettore di indici di inizio/fine per poter segmentare la stringa

```
substring("gatto", 1) # funziona
```

```
## [1] "gatto"
```

```
substr("gatto", 1) # errore
```

```
## Error in substr("gatto", 1): argument "stop" is missing, with no default
```

```
substring("gatto", 1, 1:5) # indice multiplo di fine
```

```
## [1] "g"     "ga"    "gat"   "gatt"  "gatto"
```

```
substring("gatto", 1:5, 1:5) # indice multiplo di inizio e fine
```

```
## [1] "g" "a" "t" "t" "o"
```

```
substr("gatto", 1, 1:5) # non funziona, viene usato solo 1 indice di fine
```

```
## [1] "g"
```

16.3.4 `startWith()` e `endsWith()`

Alcune volte possiamo essere interessati all'inizio o alla fine di una stringa. Ad esempio `female` e `male` hanno una chiara differenza iniziale (`fe` e `ma`). E nonostante errori di battitura seguenti o altre differenze, selezionare solo l'inizio o la fine può essere efficiente. `startWith()` e `endsWith()` permettono rispettivamente di fornire `TRUE` o `FALSE` se una certa stringa o vettore di stringhe abbiamo un certo pattern iniziale o finale.

```
startsWith("female", prefix = "fe")
```

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

```
endsWith("female", suffix = "ale")
```

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

Questa come le altre funzioni possono essere utilizzate in combinazione con `tolower()` o `toupper()` per ingnorare differenze non rilevanti.

16.3.5 `grep()` e `grepl()`

Queste funzioni lavorano invece su **vettori** di stringhe trovando la posizione o la sola presenza di specifici *pattern*. `grep()` fornisce la posizione/i nel vettore dove è presente un match, mentre `grepl()` fornisce `TRUE` o `FALSE` in funzione della presenza del *pattern*. La scrittura è la stessa `grep*(pattern, vettore)`

```
genere
```

```
## [1] "maLe"    "masChio" "Male"    "f"       "female"  "malew"
```

```
grep("female", genere) # indice di posizione
```

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

```
grepl("female", genere) # true o false
```

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

Come abbiamo visto nell'indicizzazione logica dei vettori, possiamo usare sia `grep()` che `grepl()` per selezionare solo alcuni elementi:

```
index_grep <- grep("female", genere) # indice di posizione
index_grepl <- grepl("female", genere) # indice di posizione

genere[index_grep]
```

```
## [1] "female"
```

```
genere[index_grepl]
```

```
## [1] "female"
```

Da notare ancora come tutte queste funzioni lavorino su una **corrispondenza molto stringente** (in termini di maiscolo, minuscolo, etc.) tra pattern e target.

16.4 Manipolare stringhe

Molte delle funzioni che abbiamo visto permettono anche di sostituire un certo pattern all'interno di una stringa o di un vettore di stringhe.

Utilizzando infatti `substr()` o `substring()` con la funzione di assegnazione `<-` possiamo sostituire un certo carattere. Importante, la sostituzione deve avere lo stesso numero di caratteri della selezione `start:stop` oppure verrà usato solo il numero di caratteri corrispondente:

```
x <- "gatto"
substr(x, 1, 1) <- "y"
x
```

```
## [1] "yatto"
```

```
x <- "gatto"
substr(x, 1, 1) <- "aeiou"
x # viene usata solo la a
```

```
## [1] "aatto"
```

```
# substring funziona esattamente allo stesso modo
x <- "gatto"
substring(x, 1, 1) <- "z"
x
```

```
## [1] "zatto"
```

Possono essere utilizzate anche in modo vettorizzato funzionando quindi su una serie di elementi:

```
x <- c("cane", "gatto", "topo")
substring(x, 1, 1) <- "z"
x
```

```
## [1] "zane"  "zatto" "zopo"
```

16.4.1 gsub() e sub()

Rispetto a `substring()`, `gsub()` e `sub()` permettono di sostituire un certo pattern e non usando indici di posizione. La scrittura è `*sub(pattern, replacement, target)`:

```
x <- c("cane", "gatto", "topo")
sub("a", "z", x)
```

```
## [1] "czne"  "gztto" "topo"
```

Come vedete per ogni elemento di `x` la funzione ha trovato il pattern `"a"` e lo ha sostituito con `"z"`.

La principale limitazione di `sub()` è quella di sostituire solo la prima corrispondenza trovata in ogni stringa.

```
x <- c("cane", "gatto", "topo")
sub("o", "z", x)
```

```
## [1] "cane"  "gattz" "tzpo"
```

Come vedete infatti, solo la prima "o" nella parola "topo" è stata sostituita. `gsub()` permette invece di sostituire tutti i caratteri che corrispondono al pattern richiesto:

```
x <- c("cane", "gatto", "topo")
gsub("o", "z", x)
```

```
## [1] "cane"  "gattz" "tzpz"
```

16.4.2 strsplit()

Abbiamo già visto che con `substring()` ad esempio possiamo dividere una stringa in più parti. Secondo la documentazione di R la funzione `strsplit()` è più adatta ed efficiente per questo tipo di compito. La scrittura è `strsplit(target, split)` dove `split` è il carattere in base a cui dividere:

```
frase <- "Quanto è bello usare le stringhe in R"
strsplit(frase, " ") # stiamo dividendo in base agli spazi
```

```
## [[1]]
## [1] "Quanto"   "è"        "bello"    "usare"    "le"       "stringhe" "in"
## [8] "R"
```

```
parola <- "parola1_parola2"
strsplit(parola, "_") # stiamo dividendo in base all'underscore
```

```
## [[1]]
## [1] "parola1" "parola2"
```

```
parola <- "ciao"
strsplit(parola, "") # dividiamo per ogni carattere
```

```
## [[1]]
## [1] "c" "i" "a" "o"
```

Quello che otteniamo è un vettore (all'interno di una lista, possiamo usare `unlist()`) che contiene il risultato dello splitting.

16.5 Regular Expression (REGEX)

E' tutto così semplice con le stringhe? Assolutamente no! Fino ad ora abbiamo utilizzato dei semplici pattern come singoli caratteri o insieme di caratteri tuttavia possiamo avere problemi più complessi da affrontare come:

- trovare l'estensione di un insieme di file
- trovare il dominio di un sito web

Facciamo un esempio:

```
files <- c(
  "file1.txt",
  "file2.docx",
  "file3.doc",
  "file4.sh"
)
```

In questo caso se noi vogliamo ad esempio estrarre tutte le estensioni `nomefile.estensione` gli strumenti che abbiamo visto finora non sono sufficienti:

- possiamo estrarre i caratteri dalla fine `substr()` contando con `nchar()` però le estensioni non hanno un numero fisso di caratteri
- possiamo cercare tutti i pattern con `grepl()` ma ci sono migliaia di estensioni diverse

Finora abbiamo visto 2 livelli di astrazione:

1. Corrispondenza letterale: `stringa1 == stringa2`
2. Indicizzazione: la posizione all'interno di una stringa

Il terzo livello di astrazione è quello di trovare dei pattern comuni nelle stringhe ed estrarli, indipendentemente dai singoli caratteri, dal numero o dalla posizione.

Le Regular Expressions (REGEX) sono un insieme di caratteri (chiamati **metacaratteri**) che vengono intepretati e permettono di trovare dei pattern nelle stringhe senza indicare un pattern specifico o un indice di posizione. L'argomento è molto complesso e non R-specifico. Ci sono parecchie guide online e tutorial che segnaliamo alla fine del capitolo. La cosa importante da sapere è che la maggior parte delle funzioni che abbiamo visto permettono di usare una `regex` oltre ad un pattern specifico in modo da risolvere problemi più complessi.

Per fare un esempio se vogliamo estrarre l'estensione da una lista di file il ragionamento è:

- dobbiamo trovare un `.` perchè (circa) tutti i file sono composti da `nomefile.estensione`
- dobbiamo selezionare tutti i caratteri dal punto alla fine della stringa

La "traduzione" in termini di REGEX è questa `"\\.([^.]+)$"` e quindi possiamo usare questo come *pattern* e quindi estrarre le informazioni che ci servono. Possiamo usare la funzione `regmatches(text, match)` che richiede la stringa da analizzare e un oggetto `match` che è il risultato della funzione `regexpr` che abbiamo già visto:

```
match_regex <- regexpr("\\.([^.]+)$", files)
regmatches(files, match_regex)
```

```
## [1] ".txt"  ".docx" ".doc"  ".sh"
```

16.6 Per approfondire

In tutto questo libro abbiamo sempre cercato di affrontare R come linguaggio di programmazione concentrandosi sulle funzioni di base. Tuttavia in alcuni settori, come quello delle stringhe e delle REGEX ci sono dei pacchetti esterni altamente consigliati che non solo rendono più semplice ma anche più organizzato e consistente l'insieme di funzioni. Il pacchetto `stringr` è una fantastica risorsa per imparare ma anche lavorare in modo più efficiace con le stringhe. Contiene una serie di funzioni costruite al di sopra di quelle che abbiamo affrontato, semplificandole e uniformando il tutto.

L'ultimo esempio descritto non è molto leggibile contenendo il risultato di un'altra funzione e chiamando l'oggetto `target` due volte, in `stringr` abbiamo la funzione `str_extract()` che estrae un certo pattern o REGEX:

```
stringr::str_extract(files, "\\.([^.]+)$")
```

```
## [1] ".txt"  ".docx" ".doc"  ".sh"
```

16.6.1 Risorse utili

- `stringr`
- Capitolo 14 di R for Data Science
- Mastering Regular Expressions
- Handling Strings With R

16.6.2 Altre funzioni utili

- `abbreviate()`