

Introduction to R

Corso per imparare le basi di **R**

Claudio Zandonella Callegher and Filippo Gambarota members of Psicostat

27-03-2021



Contents

Presentazione	7
Perchè R	7
Struttura del libro	7
Risorse Utili	8
Psicostat	8
Collaborazione	8
Riconoscimenti	8
Licenza	9
 Get Started	 13
 Intorduzione	 13
 1 Installare R e RStudio	 15
1.1 Installare R	15
1.2 Installare R Studio	19
 2 Interfaccia RStudio	 21
 3 Primi Passi in R	 31
3.1 Operatori Matematici	31
3.2 Operatori Relazionali e Logici	33
 4 Due Compagni Inseparabili	 39
4.1 Oggetti	39
4.2 Funzioni	43

5 Ambiente di Lavoro	49
5.1 Environment	49
5.2 Working Directory	53
5.3 R-packages	59
6 Sessione di Lavoro	67
6.1 Organizzazione Script	67
6.2 R projects	74
6.3 Messages, Warnings e Errors	74
Struttura Dati	81
Introduzione	81
7 Vettori	83
7.1 Creazione	84
7.2 Selezione Elementi	85
7.3 Funzioni ed Operazioni	92
7.4 Data Type	94
8 Fattori	107
9 Matrici	109
9.1 Creazione	110
9.2 Selezione Elementi	114
9.3 Funzioni ed Operazioni	122
9.4 estensione array	128
10 Dataframe	129
10.1 Creazione di un dataframe	130
10.2 Proprietà di un dataframe	132
10.3 Indicizzazione di dataframe	132
10.4 Indicizzazione avanzata	135
11 Liste	141
11.1 Creazione di Liste	141
11.2 Indicizzazione di una lista	142
11.3 Proprietà della lista	145
11.4 Creazione e indicizzazione avanzata	146

CONTENTS	5
Algoritmi	153
Introduzione	153
12 Definizione di Funzioni	155
13 Programmazione Condizionale	157
14 Attenti al loop	159
Case study	163
Introduzione	163
15 Caso Studio I: Attaccamento	165
15.1 Infobox	165

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 introducendo gli aspetti fondamentali di R e i concetti alla base di ogni linguaggio di programmazione che ti permetteranno 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 avrai imparato ad utilizzare uno strumento 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 utilizzarli a seconda delle varie necessità.
- **Algoritmi.** Non farti spaventare da questo nome. Ne avrai spesso sentito parlare 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 preparare i dati per le 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'università di Padova che unisce la passione per la statistica e la psicologia. Se vuoi conoscere di più riguardo le nostre attività visita il nostro 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 revisione 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

Questo libro è rilasciato sotto la Creative Commons Attribution-ShareAlike 4.0 International Public License (CC BY-SA). Le illustrazioni utilizzate nelle vignette appartengono a rstudio4edu-book e sono rilasciate con licenza CC BY-NC.

Get Started

Intorduzione

In questa sezione verranno prima presentate le istruzioni per installare R ed RStudio. Successivamente, svolgeremo le prime operazioni in R e famiglierizzeremo con dei concetti di base della programmazione quali gli oggetti e le funzioni. Introdurremo 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.** Istruzioni 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 Enviernement, 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/>

The Comprehensive R Archive Network is available at the following URLs, please choose a location close to you. Some statistics on the status of the mirrors can be found here: [main page](#), [windows release](#), [windows old release](#).

If you want to host a new mirror at your institution, please have a look at the [CRAN Mirror HOWTO](#).

O-Cloud https://cloud.r-project.org/	Automatic redirection to servers worldwide, currently sponsored by Rstudio
Algeria https://cran.usthb.dz/	University of Science and Technology Houari Boumediene
Argentina http://mirror.fcaglp.unlp.edu.ar/CRAN/	Universidad Nacional de La Plata
..	

4. Selezionare il proprio sistema operativo

The Comprehensive R Archive Network

download and install R

Precompiled binary distributions of the base system and contributed packages. **Windows and Mac** users most likely want one of these versions of R:

- [Download R for Linux](#)
- [Download R for \(Mac\) OS X](#)
- [Download R for Windows](#)

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

Source Code for all Platforms

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

1.1.1 R Windows

1. Selezionare la voce **base**

R for Windows

Subdirectories:

- base
- contrib
- old_contrib
- Rtools

Binaries for base distribution. This is what you want to [install R for the first time](#). Binaries of contributed CRAN packages (for R >= 2.13.x; managed by Uwe Ligges). There is also information on [third party software](#) available for CRAN Windows services and corresponding environment and make variables.

Binaries of contributed CRAN packages for outdated versions of R (for R < 2.13.x; managed by Uwe Ligges).

Tools to build R and R packages. This is what you want to build your own packages on Windows, or to build R itself.

Please do not submit binaries to CRAN. Package developers might want to contact Uwe Ligges directly in case of questions / suggestions related to Windows binaries.

You may also want to read the [R FAQ](#) and [R for Windows FAQ](#).

Note: CRAN does some checks on these binaries for viruses, but cannot give guarantees. Use the normal precautions with downloaded executables.

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

R-4.0.4 for Windows (32/64 bit)

Download R 4.0.4 for Windows (85 megabytes, 32/64 bit)

[Installation and other instructions](#)
[New features in this version](#)

If you want to double-check that the package you have downloaded matches the package distributed by CRAN, you can compare the [md5sum](#) of the .exe to the [fingerprint](#) on the master server. You will need a version of md5sum for windows: both [graphical](#) and [command line versions](#) are available.

Frequently asked questions

- Does R run under my version of Windows?
- How do I update packages in my previous version of R?
- Should I run 32-bit or 64-bit R?

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

The Comprehensive R Archive

R for Mac OS X

This directory contains binaries for a base distribution and packages to run on Mac OS X (release 10.6 and above). Mac OS 8.6 to 9.2 (and Mac OS X 10.1) are no longer supported but you can find the last supported release of R for these systems (which is R 1.7.1) [here](#). Releases for old Mac OS X systems (through Mac OS X 10.5) and PowerPC Macs can be found in the [old](#) directory.

Note: CRAN does not have Mac OS X systems and cannot check these binaries for viruses. Although we take precautions when assembling binaries, please use the normal precautions with downloaded executables.

Package binaries for R versions older than 3.2.0 are only available from the [CRAN archive](#) so users of such versions should adjust the CRAN mirror setting (<https://cran-archive.r-project.org>) accordingly.

R 4.0.4 "Lost Library Book" released on 2021/02/15

Please check the SHA1 checksum of the downloaded image to ensure that it has not been tampered with or corrupted during the mirroring process. For example type `openssl sha1 R-4.0.4.pkg` in the Terminal application to print the SHA1 checksum for the R-4.0.4.pkg image. On Mac OS X 10.7 and later you can also validate the signature using `pkutil --check-signature R-4.0.4.pkg`

Latest release:

R-4.0.4.pkg (notarized and signed)
SHA1-hash: 0b2b3b8460febc72a8c0b53afe85360085deb
(ca. 85MB)

R 4.0.4 binary for macOS 10.13 (High Sierra) and higher, signed and notarized package. Contains R 4.0.4 framework, R.app GUI 1.74 in 64-bit for Intel Macs, Tcl/Tk 8.6.6 X11 libraries and Texinfo 6.7. The latter two components are optional and can be omitted when choosing "custom install", they are only needed if you want to use the `cltck` R package or build package documentation from sources.

Note: the use of X11 (including `cltck`) requires **XQuartz** to be installed since it is no longer part of OS X. Always re-install XQuartz when upgrading your macOS to a new major version. Also please do not install beta versions of XQuartz (even if offered).

This release supports Intel Macs, but it is also known to work using Rosetta2 on M1-based Macs. Native Apple silicon binary is expected for R 4.1.0 if support for Fortran stabilizes, for experimental builds and updates see [mac.R-project.org](#).

Important: this release uses Xcode 12.4 and GNU Fortran 8.2. If you wish to compile R packages from sources, you will need to download GNU Fortran 8.2 - see the [tools](#) directory.

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

XQuartz

The XQuartz project is an open-source effort to develop a version of the [X.Org X Window System](#) that runs on OS X. Together with supporting libraries and applications, it forms the X11.app that Apple shipped with OS X versions 10.5 through 10.7.

Quick Download

Download	Version	Released	Info
XQuartz-2.8.0_rc2.dmg	2.8.0_rc2	2021-02-27	For macOS 10.9 or later

License Info

An XQuartz installation consists of many individual pieces of software which have various licenses. The X.Org software components' licenses are discussed on the [X.Org Foundation Licenses page](#). The `quartz-wm` window manager included with the XQuartz distribution uses the [Apple Public Source License Version 2](#).

Web site based on a design by Kyle J. Mackay for the XQuartz project.
Web site content distribution services provided by Cloudflare.
Distributed by JFrog Bintray

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

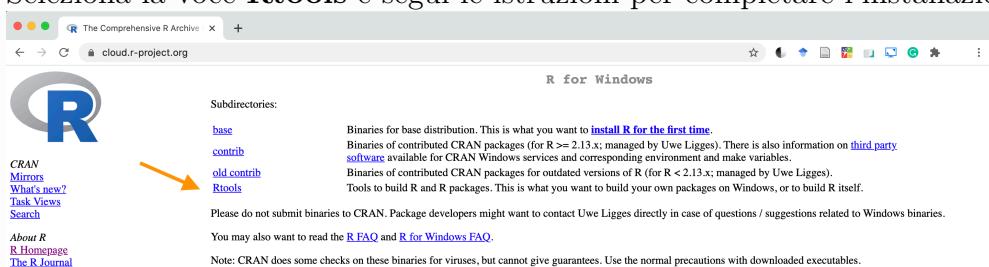
Per qualsiasi difficoltà o errore, soprattutto con il mondo Linux, una ricerca su online risolve sempre il problema.

Approfondimento: R Tools

Utilizzi avanzati di R richiedono l'installazione 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.

The screenshot shows the 'R for Mac OS X' archive page from cloud.r-project.org. It includes a sidebar with links like CRAN Mirrors, About R, and Software. The main content area contains instructions for validating the downloaded R package (R-4.0.4.pkg) using SHA1 checksums or pgkutil. A note at the bottom states: 'Important: this release uses Xcode 12.4 and GNU Fortran 8.2. If you wish to compile R packages from sources, you will need to download GNU Fortran 8.2 - see the tools directory.' An orange arrow points to this note.

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

The screenshot shows the 'R for Mac OS X - Development' page. It includes a note: 'CRAN R 4.0.0 builds and higher no longer use any custom compilers and thus this directory is no longer relevant. We now use Apple Xcode 10.1 and GNU Fortran 8.2 from <https://github.com/xcoudert/gfortran-for-macOS/releases>. For more details on compiling R, please see also <https://mac.R-project.org/tools>'.

1.2 Installare R Studio

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

The screenshot shows the RStudio website homepage. The main headline reads: 'RStudio is the premier development environment for coding in R & Python.' Below it is a large 'DOWNLOAD IT NOW' button, which has an orange arrow pointing to it.

3. Selezionare la versione gratuita di RStudio Desktop

RStudio Desktop	RStudio Desktop Pro	RStudio Server	RStudio Server Pro
Open Source License	Commercial License	Open Source License	Commercial License
Free	\$995 /year	Free	\$4,975 (5 Named Users)
DOWNLOAD	BUY	DOWNLOAD	BUY
Learn more	Learn more	Learn more	Evaluation Learn more
Integrated Tools for R	✓	✓	✓
Priority Support		✓	✓
Access via Web Browser		✓	✓
RStudio Professional Drivers	✓		✓
Connect to RStudio Server Pro remotely	✓		

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

OS	Download	Size	SHA-256
Windows 10/8/7	RStudio-1.4.1106.exe	155.97 MB	d2ff8453
macOS 10.13+	RStudio-1.4.1106.dmg	153.35 MB	c64d2eda
Ubuntu 16	rstudio-1.4.1106-amd64.deb	118.45 MB	1fc02387
Ubuntu 18/Debian 10	rstudio-1.4.1106-amd64.deb	121.07 MB	3b5d3835
Fedora 19/Red Hat 7	rstudio-1.4.1106-x86_64.rpm	138.18 MB	a9e6ddc4
Fedora 28/Red Hat 8	rstudio-1.4.1106-x86_64.rpm	138.16 MB	35e57c1c
Debian 9	rstudio-1.4.1106-amd64.deb	121.33 MB	c7e9dd68
OpenSUSE 15	rstudio-1.4.1106-x86_64.rpm	123.57 MB	3539d9c3

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 successivi capitoli. Adesso ci interessa solo familiarizzare 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.

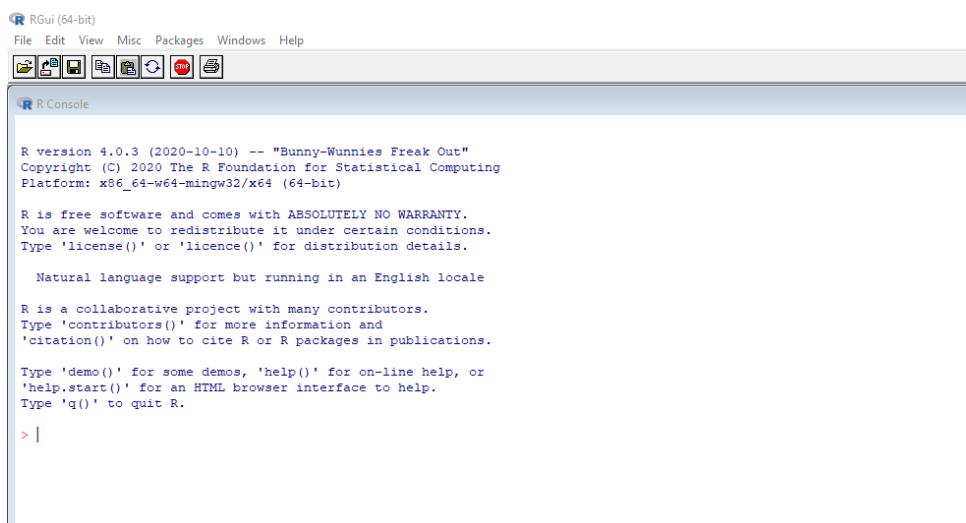


Figure 2.1: La console di R, solo per veri intenditori

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):

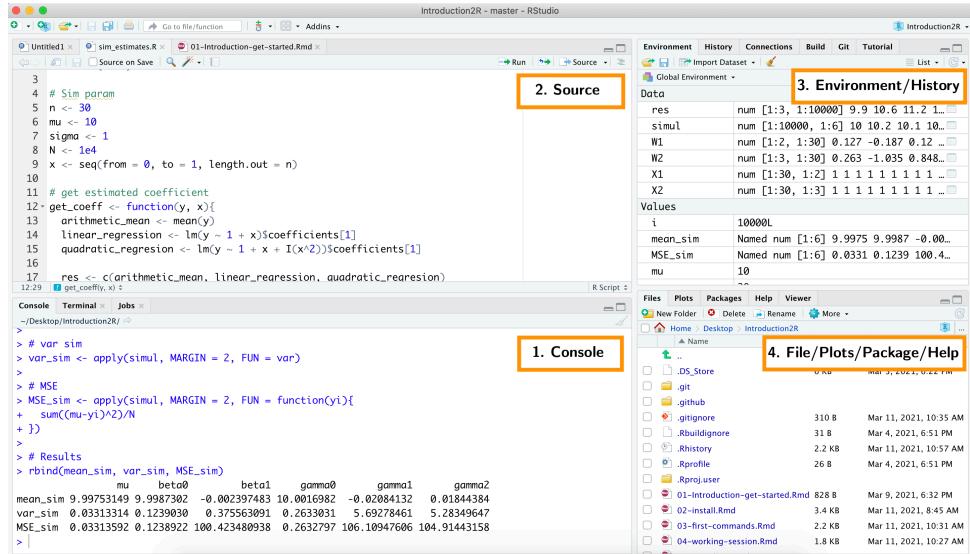


Figure 2.2: Interfaccia utente di RStudio con i suoi 4 pannelli

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 diverse 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.

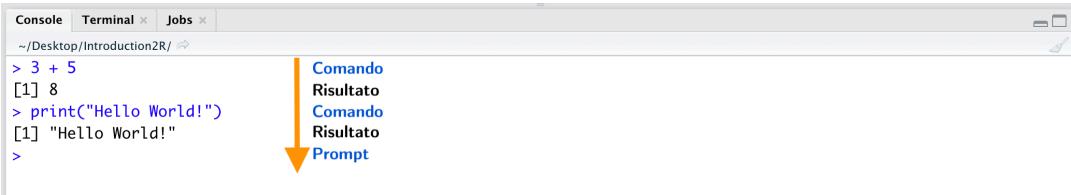
In altri casi, invece, le funzionalità e strumenti aggiuntivi di RStudio permettono una maggiore efficacia nel proprio lavoro.

1. Console: il cuore di R

Qui ritroviamo la *Console* di R dove vengono effettivamente eseguiti tutti i tuoi codici e comandi. Nota come nell'ultima riga della *Console* appaia il carattere >. Questo è definito *prompt* e 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 direttamente scrivere nella *Console* le operazioni da eseguire e premere **invio**. R eseguirà immediatamente il 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.

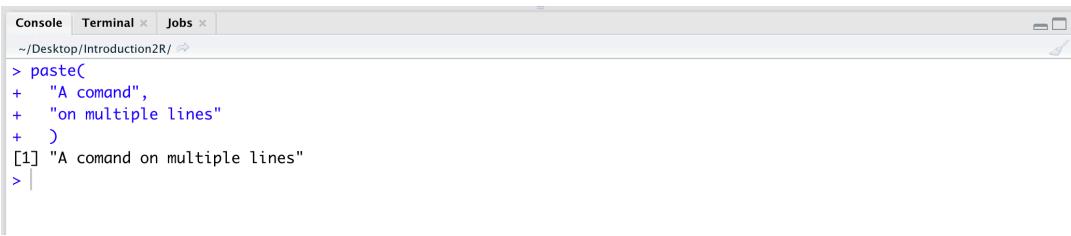


The screenshot shows the R console interface. The title bar says "Console Terminal Jobs". The working directory is "~/Desktop/Introduction2R/". The console window contains the following text:

```
> 3 + 5
[1] 8
> print("Hello World!")
[1] "Hello World!"
>
```

An orange arrow points from the word "Prompt" at the bottom right of the text area to the right margin of the window.

Figure 2.3: Esecuzione di comandi direttamente nella console



The screenshot shows the R console interface. The title bar says "Console Terminal Jobs". The working directory is "~/Desktop/Introduction2R/". The console window contains the following text:

```
> paste(
+   "A command",
+   "on multiple lines"
+ )
[1] "A command on multiple lines"
>
```

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 frecce della tastiera $\uparrow\downarrow$. 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 lavorare 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 eseguire un nuovo comando ma attento a non ripetere lo stesso errore, la sintassi dei comandi è importante (vedi Capitolo TODO).

```
> mean(c(rep(1:3, 5), 10)
+
+
+
+
+
> |
```

2. **R non risponde.** Alcuni calcoli potrebbero richiedere molto tempo o semplicemente 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, correggerli, 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 verrà 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

```

1 # Il mio primo script
2
3 # Delle semplici operazioni aritmetiche
4
5 7 + 5
6
7 20 * 4
8
9

```

8:1 (Top Level) ◊ R Script ◊

Console Terminal Jobs ~Desktop/Introduction2R/ ↗

```

> 7 + 5
[1] 12
> 20 * 4
[1] 80
>

```

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

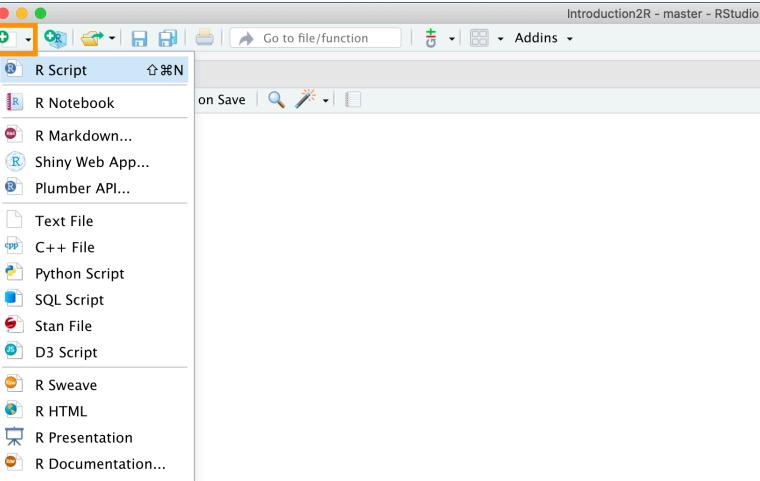
aver notato delle righe di testo verde precedute dal simbolo `#`. Questo simbolo può essere utilizzato per inserire dei *commenti* all’interno dello script. R ignorerà qualsiasi commento ed eseguirà soltanto le parti di codici.

L’utilizzo dei commenti è molto importante nel caso di script complessi poiché ci permette di spiegare e documentare il codice che viene eseguito. Nel Capitolo TODO approfondiremo il loro utilizzo.

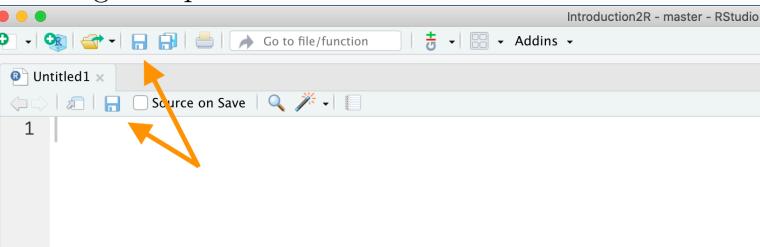


Approfondimento: Creare e Salvare uno Script

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



Un 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 riguardano funzioni avanzate di RStudio).

- **Environment:** elenco tutti gli oggetti e variabili attualmente presenti nell'ambiente di lavoro. Approfondiremo i concetti di variabili e di ambiente di lavoro rispettivamente nel Capitolo 4 e Capitolo TODO.
- **History:** elenco di tutti i comandi precedentemente eseguiti nella console. Nota che questo non 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 utilizzati per interfacciarsi con ulteriori risorse del sistema (e.g., file e pacchetti) o produrre output quali grafici e tabelle.

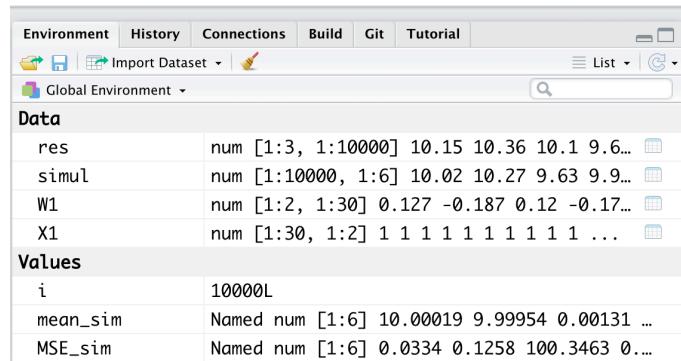


Figure 2.6: *Environment* - Elenco degli oggetti e variabili presenti nell'ambiente di lavoro



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 possibile esportare cliccando *Export*.

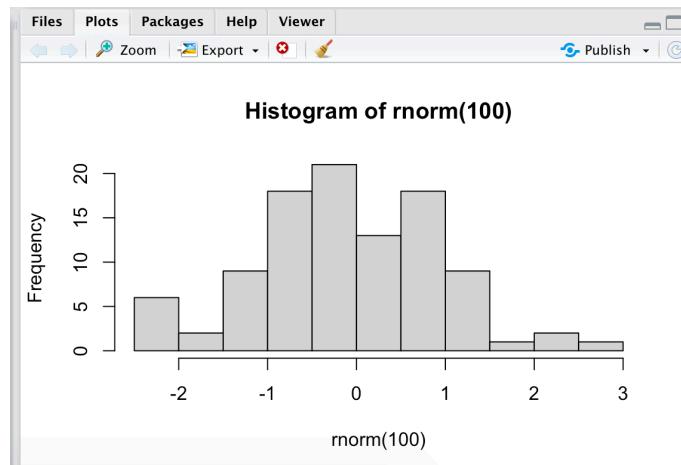


Figure 2.8: *Plots* - presentazione dei grafici

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

Name	Description	Version	Lockfile	Sou...
Project Library				
askpass	Safe Password Entry for R, Git, and SSH	1.1		⊕ ⊗
assertthat	Easy Pre and Post Assertions	0.2.1		⊕ ⊗
backports	Reimplementations of Functions Introduced Since R-3.0.0	1.1.9		⊕ ⊗
base64enc	Tools for base64 encoding	0.1-3	0.1-3	Reposit⊕ ⊗
BH	Boost C++ Header Files	1.72.0-3		⊕ ⊗
bookdown	Authoring Books and Technical Documents with R Markdown	0.21	0.21	Reposit⊕ ⊗
brew	Templating Framework for Report Generation	1.0-6		⊕ ⊗
callr	Call R from R	3.4.4		⊕ ⊗
cli	Helpers for Developing Command Line Interfaces	2.0.2		⊕ ⊗
clipr	Read and Write from the System Clipboard	0.7.0		⊕ ⊗

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

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

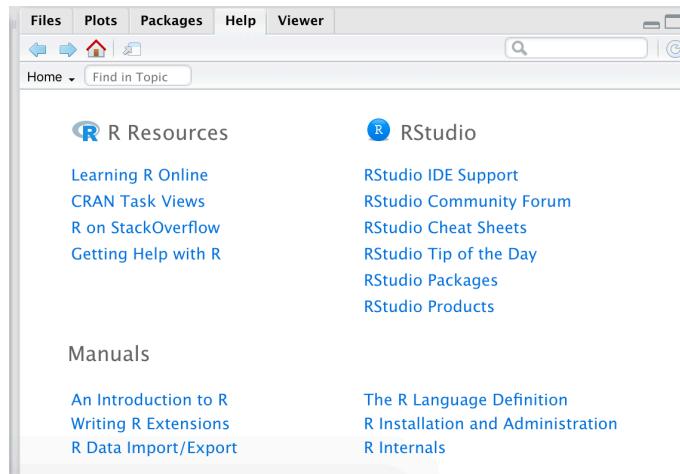
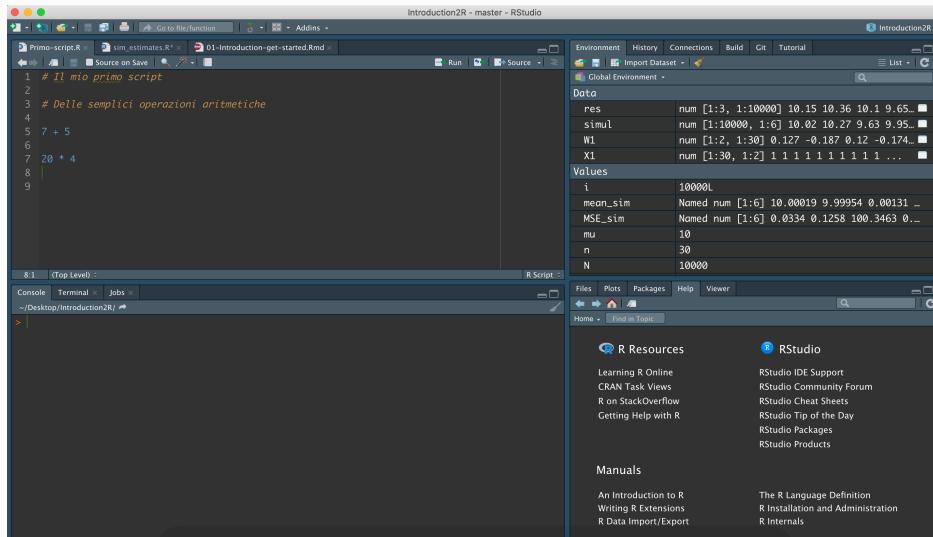


Figure 2.10: *Help* - documentazione di R

💡 Tip-Box: Personalizza tema e layout

RStudio permette un ampio grado di personalizzazione dell'interfaccia 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 utilizzare un background scuro che affatichi meno la vista (vedi Figura seguente). Clicca su RStudio > Preferenze > Appearance (MacOS) o Tools > Options > Appearance (Windows).



Chapter 3

Primi Passi in R

Ora che abbiamo iniziato a familiarizzare con il nostro strumento 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 sviluppo). Soprattutto all'inizio può sembrare eccessivamente difficile poiché si incontrano per la prima volta molti comandi e concetti di programmazione. Tuttavia, una volta familiarizzato con gli appetiti 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 utilizzate delle specifiche funzioni. In R le funzioni sono richiamate digitando `<nome-funzione>()` (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 ordine usato nelle normali espressioni matematiche. Quindi l'ordine di precedenza degli operatori è:

Table 3.1: Operatori Matematici

Funzione	Nome	Esempio
<code>x + y</code>	Addizione	<code>> 5 + 3 [1] 8</code>
<code>x - y</code>	Sottrazione	<code>> 7 - 2 [1] 5</code>
<code>x * y</code>	Moltiplicazione	<code>> 4 * 3 [1] 12</code>
<code>x / y</code>	Divisione	<code>> 8 / 3 [1] 2.666667</code>
<code>x %% y</code>	Resto della divisione	<code>> 7 %% 5 [1] 2</code>
<code>x %/% y</code>	Divisione intera	<code>> 7 %/% 5 [1] 1</code>
<code>x ^ y</code>	Potenza	<code>> 3^3 [1] 27</code>
<code>abs(x)</code>	Valore assoluto	<code>> abs(3-5^2) [1] 22</code>
<code>sign(x)</code>	Segno di un'espressione	<code>> sign(-8) [1] -1</code>
<code>sqrt(x)</code>	Radice quadrata	<code>> sqrt(25) [1] 5</code>
<code>log(x)</code>	Logaritmo naturale	<code>> log(10) [1] 2.302585</code>
<code>exp(x)</code>	Esponenziale	<code>> exp(1) [1] 2.718282</code>
<code>sin(x)</code> <code>cos(x)</code> <code>tan(x)</code> <code>asin(x)</code> <code>acos(x)</code> <code>atan(x)</code>	Funzioni trigonometriche	<code>>sin(pi/2) [1]1 >cos(pi/2) [1]6.123234e-17</code>
<code>factorial(x)</code>	Fattoriale	<code>> factorial(6) [1] 720</code>
<code>choose(n, k)</code>	Coefficiente binomiale	<code>> choose(5,3) [1] 10</code>

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

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 indicato 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 operazioni.

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 TODO 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. $\frac{\sqrt[3]{7-\pi}}{3(45-34)}$
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 ottiene 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 particolarmente interessanti ma si riveleranno molto utili nei capitoli successivi ad esempio per la selezione di elementi (vedi Capitolo TODO) o la definizione 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
<code>x == y</code>	Uguale	<code>> 5 == 3 [1] FALSE</code>
<code>x != y</code>	Diverso	<code>> 7 != 2 [1] TRUE</code>
<code>x > y</code>	Maggiore	<code>> 4 > 3 [1] TRUE</code>
<code>x >= y</code>	Maggiore o uguale	<code>> -2 >= 3 [1] FALSE</code>
<code>x < y</code>	Minore	<code>> 7 < 5 [1] FALSE</code>
<code>x <= y</code>	Minore o uguale	<code>> 7 <= 7 [1] TRUE</code>
<code>x %in% y</code>	inclusione	<code>> 5 %in% c(3, 5, 8) [1] TRUE</code>

 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 che si commette in continuazione.

L'operatore `=` è utilizzato in R per assegnare un valore ad una variabile. Argomento che vederemo nella Sezione TODO



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

Nota che in qualsiasi linguaggio di Programmazione, ai valori TRUE e FALSE sono associati rispettivamente 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 poiché potrebbe non essere chiara e creare fraintendimenti. Infatti mentre TRUE e FALSE sono parole riservate (vedi Capitolo TODO) T e 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 utilizzati gli operatori logici riportati in Tabella 3.3.

Table 3.3: Operatori Logici

Funzione	Nome	Esempio
<code>!x</code>	Negazione	<code>> !TRUE [1] FALSE</code>
<code>x & y</code>	Congiunzione	<code>> TRUE & FALSE [1] FALSE</code>
<code>x y</code>	Disgiunzione Inclusiva	<code>> TRUE FALSE [1] TRUE</code>

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 valutata 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 relazione sia vera. La proposizione sarà valutata falsa solo quando entrambe le relazioni sono false (vedi Tabella 3.5).

Table 3.5: Disgiunzione inclusiva '|'

x	y	$x \mid y$
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 relazioni (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	<code>xor(x, y)</code>
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 relazionali 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 numeri 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 permette 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 e della terza sezione del libro (TODO).

4.1 Oggetti

Quando eseguiamo un comando 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 essere 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 parola 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 classico `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 distinguere 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 virgolette "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:

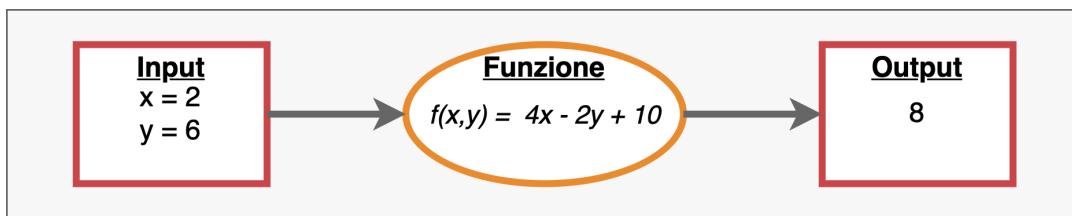
- **Vettori**
- **Matrici**

- Dataframe
- Liste

Approfondiremo la loro definizione, le loro caratteristiche ed il loro utilizzo nel corso di tutta la seconda sezione di questo libro TODO.

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 ripetutivamente 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 produrrenno 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 virgolette. 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 importante, 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 richiedono quindi di essere specificati almeno che ovviamente non si vogliono 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 argomento (o di entrambi) il comportamento della funzione `var`.

```
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 consigliabile esplicitare sempre gli argomenti di una funzione per rendere chiaro a che cosa si riferiscono i valori indicati. Questo è utile 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 assicurare 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 TODO, 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 comportamento 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:

- **Tiolo** - Titolo esplicativo della finalità della funzione
- **Description** - Descrizione concisa 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 suoi elementi utile per interpretare ed utilizzare i risultati 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

<code>...</code>	arguments passed to or from methods.
<code>from</code> , <code>to</code>	the starting and (maximal) end values of the sequence. Of length 1 unless just <code>from</code> is supplied as an unnamed argument.
<code>by</code>	number: increment of the sequence.
<code>length.out</code>	desired length of the sequence. A non-negative number, which for <code>seq</code> and <code>seq.int</code> will be rounded up if fractional.
<code>along.with</code>	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 e 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 utili funzioni di autocompletamento e suggerimento dei comandi che semplificano la scrittura di codici.

In Rstudio, i suggerimenti compaiono 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 la 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.

The screenshot shows two examples of RStudio's code completion feature.

Example 1: In the console, the command `> seq(from = 1, to = 10,)` is being typed. A tooltip appears over the closing parenthesis, showing the argument `by` with its description: "number: increment of the sequence." Other arguments like `length.out`, `along.with`, `...`, and `to` are also listed.

```

> seq(from = 1, to = 10, )
  by
  number: increment of the sequence.
  Press F1 for additional help
  length.out =
  along.with =
  ...
  to
  ...
  
```

Example 2: In the console, the command `> un_` is being typed. A tooltip lists several functions starting with `un_`, including `un_alter_nome_particolarmente_lungo`, `un_nome_particolarmente_lungo`, `unclass`, `undebug`, and `undebugcall`. The first item in the list is highlighted.

```

> un_
  un_alter_nome_particolarmente_lungo
  un_nome_particolarmente_lungo
  unclass {base}
  undebug {base}
  undebugcall {utils}
  . .
  
```

Chapter 5

Ambiente di Lavoro

In questo 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 Environment raccolge 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 utilizzandone il comando `ls()`, ovvero *list objects*.

The screenshot shows the RStudio interface with the 'Environment' tab selected in the top navigation bar. Below the tabs, there are several icons: a file icon, a clipboard icon, an 'Import Dataset' icon, and a pencil icon. A dropdown menu labeled 'Global Environment' is open. To the right of the dropdown is a search bar with a magnifying glass icon. The main area is divided into two sections: 'Data' and 'Values'. The 'Data' section lists four objects: 'res', 'simul', 'W1', and 'X1', each with a preview of its contents and a copy icon. The 'Values' section lists three objects: 'i', 'mean_sim', and 'MSE_sim', each with a preview of its contents and a copy icon. The entire window has a light gray background.

Data	
res	num [1:3, 1:10000] 10.15 10.36 10.1 9.6...
simul	num [1:10000, 1:6] 10.02 10.27 9.63 9.9...
W1	num [1:2, 1:30] 0.127 -0.187 0.12 -0.17...
X1	num [1:30, 1:2] 1 1 1 1 1 1 1 1 1 1 1 ...

Values	
i	10000L
mean_sim	Named num [1:6] 10.00019 9.99954 0.00131 ...
MSE_sim	Named num [1:6] 0.0334 0.1258 100.3463 0...

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

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



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 informazioni a seconda del tipo di oggetto. Vediamo come nel nostro esempio, nel caso di variabili con un singolo valore (e.g., `word` e `y`) vengano 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 TODO), nell'esempio abbiamo un vettore numerico (`num`) di 4 elementi (`[1:4]`).

Values	
word	"Hello Word!"
x	num [1:4] 2 4 6 8
y	27

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 attualmente presenti nel nostro 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.

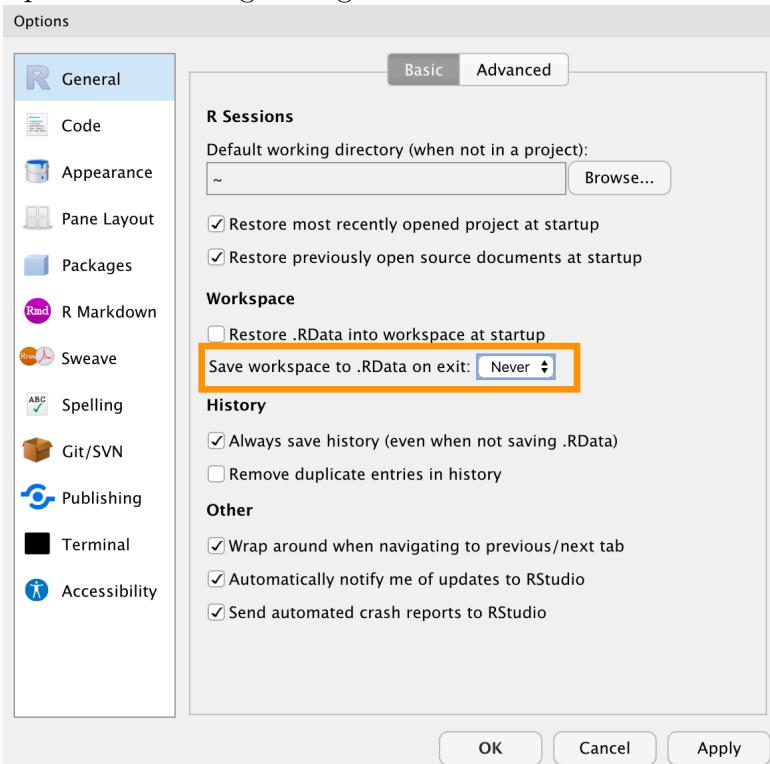
- **Utilizzare oggetti non ancora creati.** In questo caso l'errore è facilmente individuabile dato che sarà lo stesso R ad avvisarci che “*object ‘not found’*”. In questo caso dovremmo semplicemente eseguire il comando per creare l'oggetto richiesto.

```
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 riferiscono a tutt'altro. Questo comporta che qualsiasi nostro risultato perda di significato. Bisogna prestare molta attenzione perché 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 Environment 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 sessioni 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, infatti, dovranno essere ricreati ad ogni sessione, garantendo la riproducibilità e correttezza del lavoro (almeno dal punto di vista di programmazione). Idealmente dovrebbe essere possibile, 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 comandi. 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 interfacciarsi 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. Approssimativamente 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 attualmente in uso sia effettivamente quella prevista. Se ci trovassimo in una diversa 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 comando `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

differenti 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\Dados\My-data.Rda"
```

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

```
"c:\Users\<username>\Desktop\Introduction2R\Dados\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 carattere "/" 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 direttamente 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).

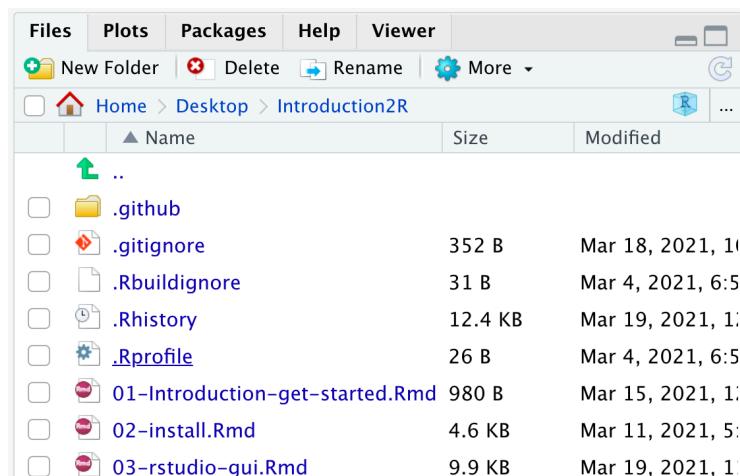


Figure 5.6: Workig directory dell'attuale sessione di lavoro

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*”.

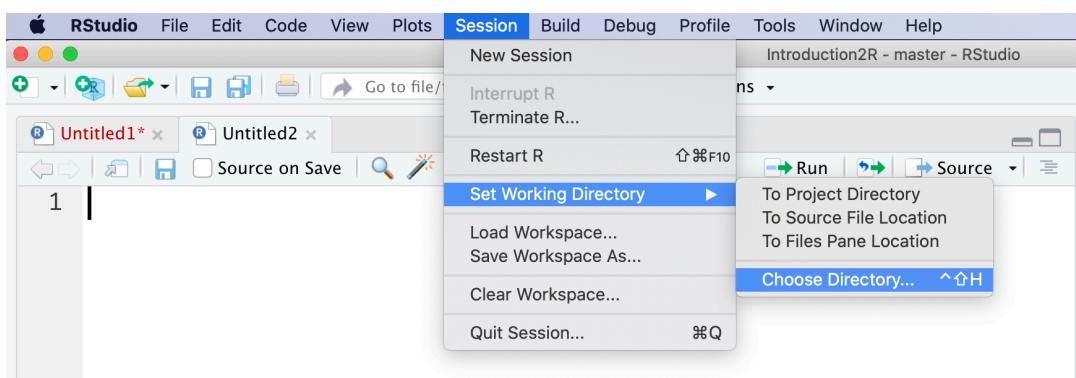
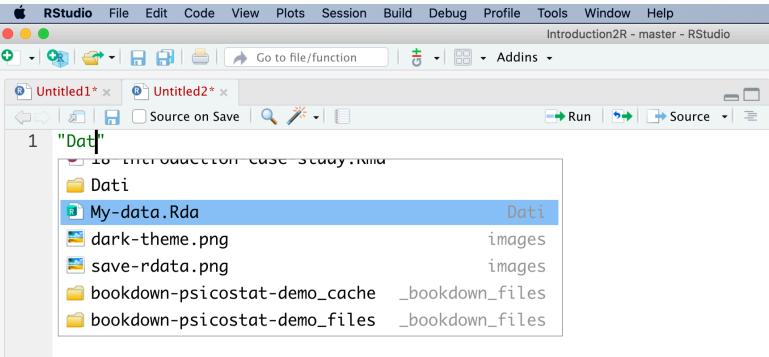


Figure 5.7: Definire la working directory



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** pacchetti disponibili gratuitamente sul CRAN (la repository ufficiale di R). Questi pacchetti sono stati sviluppati dall'immensa community 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 repository 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 pacchetto nella sessione di lavoro.** Anche se il pacchetto 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 funzioni del pacchetto saranno effettivamente disponibili per essere usate.

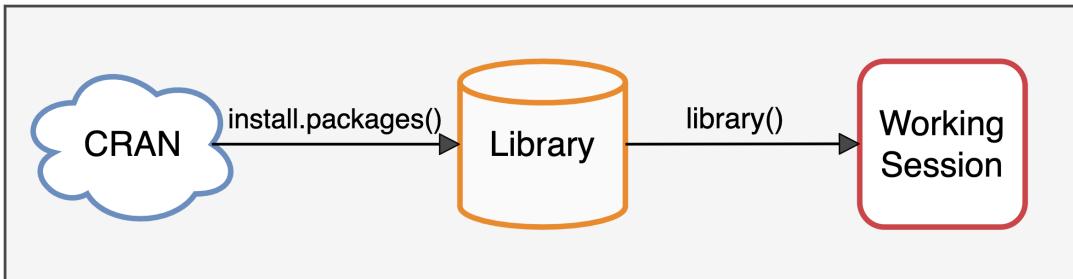


Figure 5.8: Utilizzare i pacchetti in R

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.

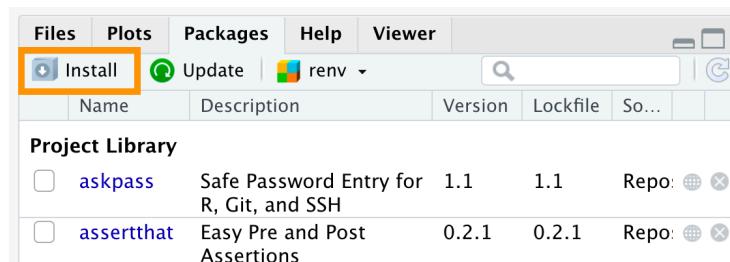
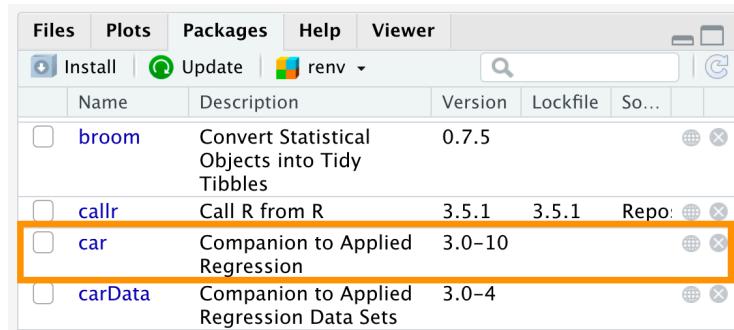


Figure 5.9: Installare pacchetti tramite interfacci RStudio

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

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

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



The screenshot shows the RStudio interface with the 'Packages' tab selected. The table lists several packages:

Name	Description	Version	Lockfile	So...
broom	Convert Statistical Objects into Tidy Tibbles	0.7.5		
callr	Call R from R	3.5.1	3.5.1	Repo: ...
car	Companion to Applied Regression	3.0-10		
carData	Companion to Applied Regression Data Sets	3.0-4		

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 effettivamente 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 aggiornamenti 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 nostra library, dobbiamo ora caricarlo nella nostra sessione di lavoro. Per fare ciò, possiamo 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 lavoro.



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>()

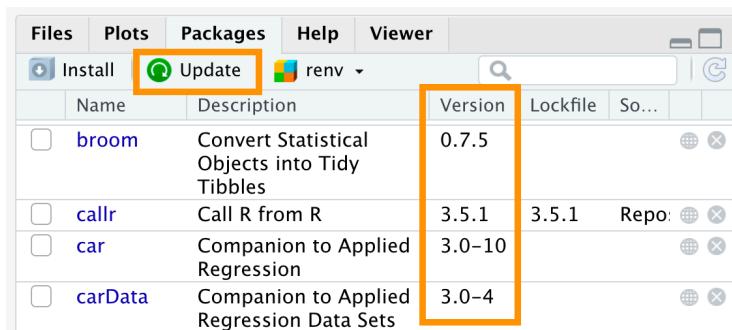
# Esempio con la funzione Anova del pacchetto car
car::Anova()
```

L'utilizzo dei `::` ci permette di richiamare direttamente la funzione desiderata. La differenza 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 pacchetti come ogni altro software vengono aggiornati nel corso del tempo fornendo nuove funzionalità e risolvendo eventuali 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 pacchetti che si desidera aggiornare. Nota come nella lista dei pacchetti venga riportata l'attuale versione alla voce “Version”.



Name	Description	Version	Lockfile	So...
broom	Convert Statistical Objects into Tidy Tibbles	0.7.5		
callr	Call R from R	3.5.1	3.5.1	Repo:
car	Companion to Applied Regression	3.0-10		
carData	Companion to Applied Regression Data Sets	3.0-4		

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 pacchetto nel pannello Packages come mostrato 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.

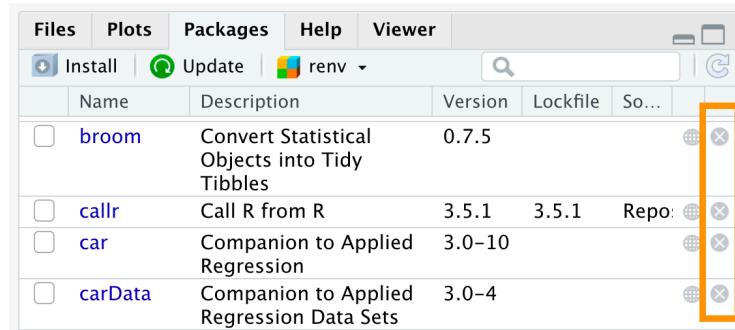


Figure 5.13: Rimuovere un pacchetto

- **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.

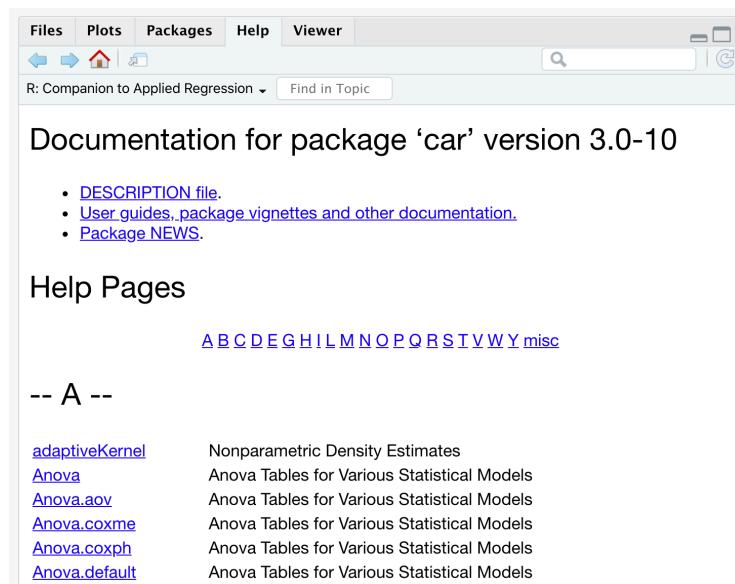


Figure 5.14: Documentazione del pacchetto car

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 pacchetti 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 aggiornamenti più recenti o anche nuovi pacchetti non ancora disponibili sul CRAN. Va sottolineato tuttavia, come queste siano appunto delle versioni di sviluppo e quindi potrebbero presentare maggiori problemi. Inoltre per installare i pacchetti in questo modo, è richiesta l'installazione di **R tools** (vedi “*Approfondimento: R Tools*” nel Capitolo 1.1).

Per installare un pacchetto direttamente da Github è possibile utilizzare il comando `install_github()` del pacchetto `devtools`, indicando 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 e l'uso degli *RStudio Projects* per essere ordinati ed efficaci nel proprio lavoro. 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'organizzazione 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 considerato il titolo della sezione e comparirà una piccola freccia a lato del numero di riga. È 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.

The screenshot shows the RStudio Source Editor window titled "Introduction2R - master - RStudio Source Editor". The file is named "Analisi-Tesi.R". The code is an R script divided into several sections:

```
1 #####  
2 #== Analisi Esperimento Tesi ==#  
3 #####  
4  
5 # In questo script vengono analizzate i dati relativi alla tesi  
6  
7 #---- Settings ----  
8  
9 rm(list = ls())  
10 setwd("~/Desktop/Analisi_Tesi/")  
11  
12 library(tidyverse)  
13 library(lme4)  
14 library(car)  
15 library(effects)  
16  
17 #---- Caricare e Pulire i Dati ----  
18  
19 # Codici relativi all'importazione e pulizia dei dati  
20  
21 #---- Codifica e Scoring dei Dati ----  
22  
23 # Codici relativi alla codifica e scoring dei dati  
24  
25 #---- Analisi Descrittive ----  
26  
27 # Codici relativi alle analisi descrittive  
28  
29 #---- Analisi Inferenziali ----  
30  
31 # Codici relativi alle analisi inferenziali  
32  
33  
34
```

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.
- **Introduzione** - 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'ambiente 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 assicurarsi che i comandi siano eseguiti dalla corretta 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).

The screenshot shows the RStudio Source Editor interface. The title bar says "Introduction2R - master - RStudio Source Editor". Below it is a toolbar with icons for file operations like Open, Save, and Run. To the right of the toolbar is a "Source" button with a dropdown arrow. An orange arrow points from the text above to this "Source" button. A red box highlights the dropdown menu that has appeared. The menu contains the following items: "Settings", "Caricare e Pulire i Dati", "Codifica e Scoring dei Dati", "Analisi Descrittive", and "Analisi Inferenziali". The main editor area shows an R script named "Analisi-Tesi.R" with the following code:

```

#=====
#== Analisi Esperimento Tesi ==#
#=====

# In questo script vengono analizzate i dati relativi alla tesi

#---- Settings ----

rm(list = ls())
setwd("~/Desktop/Analisi_Tesi/")
library(tidyverse)
library(lme4)

```

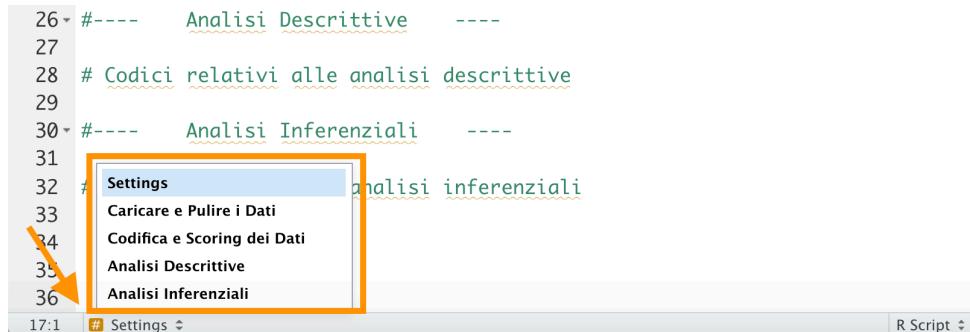
Figure 6.2: Indice in alto per navigazione sezioni

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.



A screenshot of the RStudio interface showing an R script editor. A context menu is open over line 36 of the code, listing options: Settings, Caricare e Pulire i Dati, Codifica e Scoring dei Dati, Analisi Descrittive, and Analisi Inferenziali. The entire menu is highlighted with an orange border. An orange arrow points from the bottom left towards the menu. The code in the editor is:

```

26 #---- Analisi Descrittive ----
27
28 # Codici relativi alle analisi descrittive
29
30 #---- Analisi Inferenziali ----
31
32 #---- analisi inferenziali
33
34
35
36

```

The status bar at the bottom shows "17:1" and "Settings".

Figure 6.3: Menù in basso per navigazione sezioni



A screenshot of the RStudio interface showing an R script editor. The code is partially collapsed, with sections like "Analisi Esperimento Tesi", "Analisi Descrittive", and "Analisi Inferenziali" collapsed into single lines. Lines 7, 18, 22, 24, 26, and 30 are highlighted with an orange border. The status bar at the bottom shows "17:1" and "Settings".

```

1 ######
2 === Analisi Esperimento Tesi ===#
3 #####
4
5 # In questo script vengono analizzate i dati relativi alla test
6
7 #---- Settings
8 #---- Caricare e Pulire i Dati
22 #---- Codifica e Scoring dei Dati ----
23
24 # Codici relativi alla codifica e scoring dei dati
25
26 #---- Analisi Descrittive
30 #---- Analisi Inferenziali ----
31
32 # Codici relativi alle analisi inferenziali
33
34
35

```

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 eventuali 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 facilmente interpretabili semplicemente leggendo il codice.

La capacità di scrivere commenti utili ed evitare 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 molto utile perché 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 caratteristiche 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 né 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 sequenza 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 virgola 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 (...) {   # istruzione condizionale
      ...
    }
  }
}

# Bad
for (...) {      # loop più esterno
  ...
  for (...) {    # loop interno
    ...
    if (...) {   # istruzione 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 R projects

TODO

6.2.1 Più di uno script

- idea di organizzare in vari script, cartelle
- working directory tutto in funzione alla cartella

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 utilizzati 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 allarmare, è importante controllare che i comandi siano corretti e che abbiano effettivamente 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 nessun 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 perché 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 perché sbagliamo. Al contrario, sono delle richieste di aiuto fatte da R perché non sa come comportarsi. Per quanto super potente, R è un semplice 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 carattere 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'utilizzo 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 immediato 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:

Error in : could not find function ""

All Videos News Shopping Maps More Settings Tools

About 1,740,000,000 results (0.63 seconds)

stackoverflow.com › questions › error-could-not-find-f... ▾

[Error: could not find function ... in R - Stack Overflow](#)

10 answers

Aug 11, 2011 — There are a few things you **should check** : Did you write the name of your function correctly? Names are case sensitive. Did you install the ...

[Error: could not find function "%>%" - Stack Overflow](#) 5 answers Jun 4, 2017

[Could not find function "%<>%" with dplyr loaded ...](#) 1 answer Mar 27, 2019

[dplyr error - could not find function "%>%" - Stack ...](#) 1 answer May 29, 2019

["could not find function" error though function is in ...](#) 1 answer Jul 26, 2018

[More results from stackoverflow.com](#)

Chiedere una soluzione

Se invece il vostro problema 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 programmati, 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 soprattutto la risposta, in uno spirito di condivisione. C'è anche una guida dedicata per scrivere la domanda perfetta.

In generale¹:

- 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** (**R**eproducible **E**xample). E' un tema leggermente più avanzato ma l'idea di fondo è quella di

¹Fonente: 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 caratteristiche e vedremo come crearle, modificarle e manipolarle a seconda delle necessità

I capitoli sono così organizzati:

- **Capitolo 7 - Vettori.** Impareremo le caratteristiche e l'uso dei vettori soffermandoci anche sulle diverse tipologie di dati.
- **Capitolo 8 - Fattori.** Impareremo le caratteristiche e l'uso dei fattori un particolare tipo di vettori usati per le variabili categoriali ed ordinali.

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.

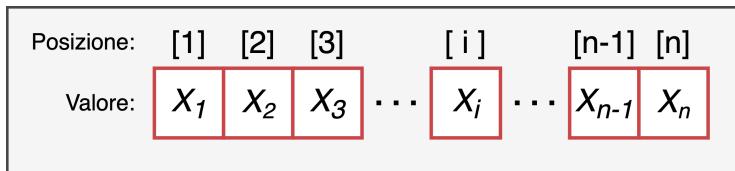


Figure 7.1: Rappresentazione della struttura di un vettore di lunghezza *n*

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 essere 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.

Vediamo ora come creare dei vettori in R e come compiere le comuni operazioni 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 successione 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 altrettantiva 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 sequenzeabbiammo:

- `<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 complessiva indicata da `length.out` (vedi `?seq()` per maggiori dettagli).

```
# 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` possono essere ripetuti nello stesso ordine più volte specificando `times` oppure ripetuti ciascuno più volte specificando `each` (vedi `?rep()` per maggiori dettagli).

```
# 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 lettere "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 vettore di indici per selezionare gli elementi 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.

```

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

# Seleziona 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 utilizziamo un indice di posizione maggiore del numero di elementi del nostro vettore?

```

# 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 vettore quelli che rispettano 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 elementi maggiori di 0
my_numbers <- -5:5
my_numbers[my_numbers >= 0]
## [1] 0 1 2 3 4 5

# Vettore caratteri - seleziono elementi 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 TODO) - 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 esplicativi 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 combinazione 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:

```
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.

```
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)]*.

```
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.

```
# 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"
```

Esercizi

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).

Table 7.1: Funzioni e operazioni con vettori

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

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.
- **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
## length
## [1] 2 4 4 6 6
```

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



Approfondimento: Vectorized Operations

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

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. Avremmo quindi diversi tipi di vettori a seconda della tipologia 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 virgolette "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 utilizzando 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 rappresentare 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 definito 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 possono essere rappresentati esattamente utilizzando il numero massimo di 64 bit. In questi casi i loro valori vengono quindi approssimati 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'approssimazione dei valori **double**.

```
my_value <- sqrt(2)^2 # dovrei ottenere 2
my_value - 2           # dovrei ottenere 0
## [1] 4.40892e-16
```

E' importante tenere a mente questo problema nei test di ugualanza 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).

```
my_value == 2           # Problema di approssimazione
## [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 funzione `typeof()` valuta la tipologia di dati. In questo caso otteniamo per entrambi il valore `character`.

```
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 ripetendo ad un'altra stringa.

```
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 essere 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 essere 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] 25

mean(my_values > 0)     # percentuale numeri positivi
## [1] 0.5
```



Approfondimento: is.* adn as.* Function Families

Esistono due famiglie di funzioni che permettono rispettivamente 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, possiamo 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), possiamo 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 stringhe 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 stringhe 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 stringhe 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 qualiasi 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 utilizzati 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 quando il loro output è indefinito.
- `NA` - rappresenta un dato mancante (*Not Available*). E' un valore costante di lunghezza 1 che può essere utilizzato per qualsiasi tipologia di dati.
- `NaN` - indica un risultato matematico che non può essere rappresentato come un valore numerico (*Not A Number*). E' un valore costante di lunghezza 1 che può essere utilizzato 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 infinito (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 correttamente gestiti, possono generare conseguenti errori nei codici. Descriviamo ora alcune delle caratteristiche 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 mancante. 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 restituiscono 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 nuovamente un `NA`.

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

- NaN - quando NaN viene utilizzato in una qualsiasi operazione matematica il risultato sarà nuovamente 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 infinti.

```
Inf - 3
## [1] Inf
```

```
Inf * -3
## [1] -Inf
```

```
Inf + Inf
## [1] Inf
```

```
Inf + -Inf
## [1] NaN
```

```
Inf * -Inf
## [1] -Inf
```

```
Inf / Inf
## [1] 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 utilizzato il comune operatore di uguaglianza `==` poichè non fornisce i risultati 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 ottenuti con gli operatori logici dove la *propagazione* del valore non segue sempre le attese. Osserviamo i diversi casi:

- NULL- otteniamo 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 risultati 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` perché 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` perché 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 sufficiente 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 qualsiasi numero diverso 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 risposta 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 automaticamente eventuali dati mancanti così da poter ottenere correttamente 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 delle conclusioni. Inoltre sarà fondamentale controllare sempre l'effettiva 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

Working in progress.

- attributes
- named vectors
- factors, ordered factors

I fattori sono utilizzati per definire delle variabili categoriali, sono indicati in R con **Factor**. Per creare una variabile categoriale in R si utilizza la funzione:

```
nome_variabile<-factor(c(..., data, ...), levels=c(...))
```

L'opzione **levels=c(...)** è usata per specificare quali sono i possibili livelli della variabile categoriale. E' possibile modificare o aggiungere nuovi livelli della variabile anche in un secondo momento utilizzando la funzione:

```
levels(nome_fattore)<- c(..., nuovi_livelli, ...)
```

Nota: nel creare un fattore R associa ad ogni livello un valore in ordine crescente e assegna agli elementi del vettore il loro valore numerico a seconda del proprio livello. Pertanto se un fattore è trasformato in un vettore numerico vengono restituiti tali valori numerici e non i livelli anche nel caso fossero dei numeri. Prendiamo per esempio la variabile **anni_istruzione**:

```
anni_istruzione<-factor(c(11,8,4,8,11,4,11,8))
anni_istruzione
## [1] 11 8 4 8 11 4 11 8
## Levels: 4 8 11
as.numeric(anni_istruzione)
## [1] 3 2 1 2 3 1 3 2
```

Per riottenere gli estti valori numerici è necessario eseguire:

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

Esercizi

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

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

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

```
## [1] CBT          Psicanalisi CBT          Psicanalisi CBT          Psicanalisi  
## [7] Controllo    Controllo    CBT          Psicanalisi  
## 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.

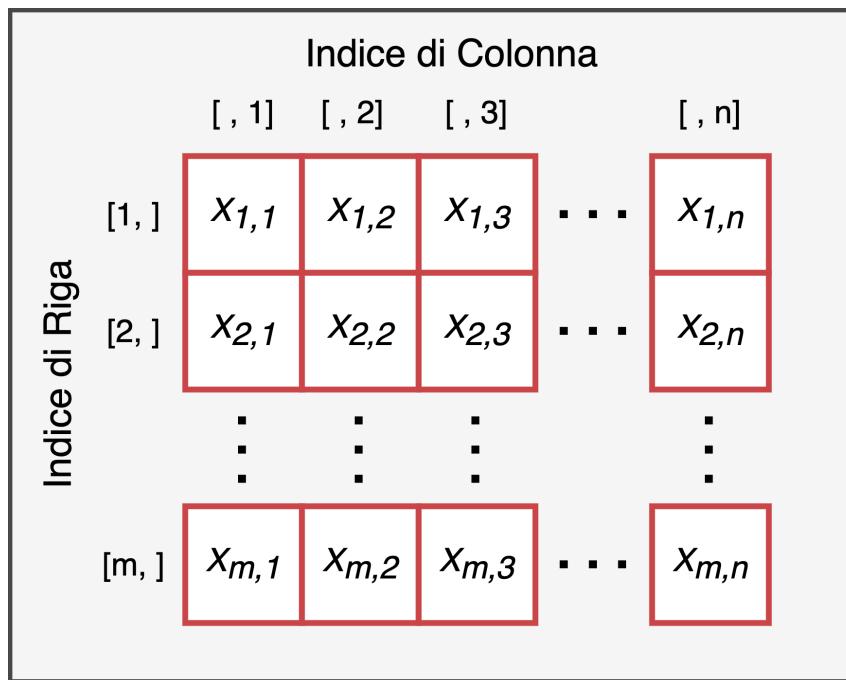


Figure 9.1: Rappresentazione della struttura di una matrice di $*m*$ colonne e $*n*$ righe

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 essere 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 TODO).

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 operazioni di selezione. Successivamente vedremo diverse manipolazioni e operazioni con le matrici. Infine estenderemo brevemente il concetto di matrici a dimensioni maggiori di due attraverso 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 lunghezza 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 ottenere 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.

```
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 visto 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.

```
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 computazionalemente 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 TODO.

```
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 trasformati nei rispettivi valori numerici 1 e 0.

Esercizi

1. Crea la matrice A così definita:

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

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 elemento 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'interno 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:

Selezione per Riga o Colonna

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

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

# Selezione 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 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.

```
# 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.

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

# Seleziona 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 utilizzare sempre le matrici e non i vettori.

```

# 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)`.

```

# 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 matrice
- 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

# Elementi 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.

```
# Seleziona 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 l'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 colonna 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 possa aggiungere una nuova riga o colonna attraverso questa operazione ma sarà necessario utilizzare una diversa procedura (vedi Capitolo TODO).

```
# 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 colonne) 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.

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
<code>nuova_matrice <- cbind(matrice1, matrice2)</code>	Unire due matrici creando nuove colonne (le matrici devono avere la stessa dimensione nelle righe)
<code>nuova_matrice <- rbind(matrice1, matrice2)</code>	Unire due matrici creando nuove righe (le matrici devono avere la stessa dimensione nelle colonne)
<code>nrow(nome_matrice)</code>	Numero di righe della matrice
<code>ncol(nome_matrice)</code>	Numero di colonne della matrice
<code>dim(nome_matrice)</code>	Dimensione della matrice (righe e colonne)
<code>colnames(nome_matrice)</code>	Nomi delle colonne della matrice
<code>rownames(nome_matrice)</code>	Nomi delle righe della matrice
<code>dimnames(nome_matrice)</code>	Nomi delle righe e delle colonne
<code>t(nome_matrice)</code>	Trasposta della matrice
<code>diag(nome_matrice)</code>	Vettore con gli elementi della diagonale della matrice
<code>det(nome_matrice)</code>	Determinante della matrice (la matrice deve essere quadrata)
<code>solve(nome_matrice)</code>	Inversa della matrice
<code>matrice1 + matrice2</code>	Somma elemento per elemento di due matrici
<code>matrice1 - matrice2</code>	Differenza elemento per elemento tra due matrici
<code>matrice1 * matrice2</code>	Prodotto elemento per elemento tra due matrici
<code>matrice1 / matrice2</code>	Rapporto elemento per elemento tra due matrici
<code>matrice1 %*% matrice2</code>	Prodotto matriciale

Descriviamo ora nel dettaglio alcuni particolari utilizzi.

9.3.1 Attributi di una Matrice

Abbiamo visto nel Capitolo TODO 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()`.

```
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.

```
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.

```
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.

```
# Assegno 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 posson utilizzare il `dimnames()` per accedere contemporaneamente sia ai nomi di riga che a quelli di colonna. Come output ottengo una lista (vedi Capitolo TODO) 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 dimensioni sono disponibili, è possibile indicizzare una matrice in base ai nomi delle righe e/o colonne. 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 TODO).

```
# 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 Combinare Matrici

Abbiamo visto nel Capitolo 7 come si possano fare diverse operazioni tra vettori, in particolare combinare ovvero unire vettori diversi. Anche per le matrici è possibile combinare matrici diverse, rispettando alcune regole:

- 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
- 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)

Se partiamo da una matrice `mat` per unire a `mat` un'altra matrice `new_mat` possiamo usare il comando `cbind(mat, new_mat)` se vogliamo unire le due matrici per colonna invece `rbind(mat, new_mat)` se vogliamo unire per riga. E' utile pensare all'unione come un collage tra matrici, in figura 9.2 è presente uno schema utile per capire visivamente questo concetto.

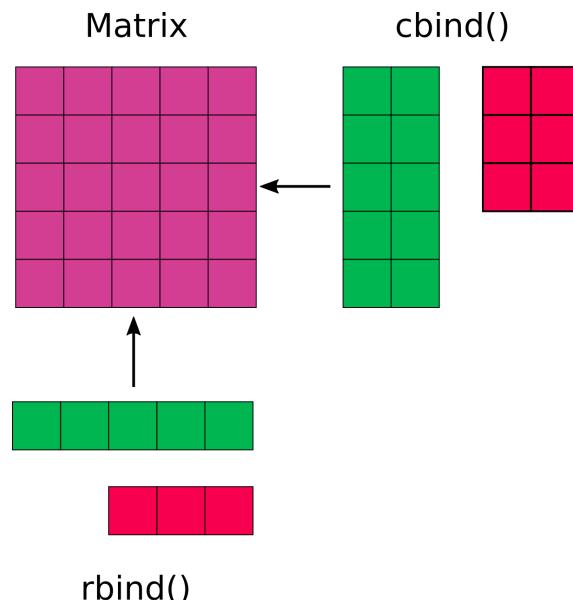


Figure 9.2: Schema per la combinazione di matrici

Vediamo in R:

```

vec <- 1:25
mat <- matrix(vec, ncol = 5, nrow = 5, byrow = FALSE)

# Nuova matrice da aggiungere
new_vec <- 11:20
new_mat <- matrix(new_vec, ncol = 2, nrow = 5, byrow = FALSE)

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

new_mat
##      [,1] [,2]
## [1,]   11   16
## [2,]   12   17
## [3,]   13   18
## [4,]   14   19
## [5,]   15   20

# Combiniamo per colonna

```

```

cbind(mat, new_mat)
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    1    6   11   16   21   11   16
## [2,]    2    7   12   17   22   12   17
## [3,]    3    8   13   18   23   13   18
## [4,]    4    9   14   19   24   14   19
## [5,]    5   10   15   20   25   15   20

# Combiniamo per riga

rbind(mat, new_mat)
## Error in rbind(mat, new_mat): number of columns of matrices must match (see arg 2)

# Scambiamo il numero di colonne e righe, per far combaciare le due matrici

new_vec <- 11:20
new_mat <- matrix(new_vec, ncol = 5, nrow = 2, byrow = FALSE)
mat
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    6   11   16   21
## [2,]    2    7   12   17   22
## [3,]    3    8   13   18   23
## [4,]    4    9   14   19   24
## [5,]    5   10   15   20   25
new_mat
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   11   13   15   17   19
## [2,]   12   14   16   18   20

rbind(mat, new_mat)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    6   11   16   21
## [2,]    2    7   12   17   22
## [3,]    3    8   13   18   23
## [4,]    4    9   14   19   24
## [5,]    5   10   15   20   25
## [6,]   11   13   15   17   19
## [7,]   12   14   16   18   20

```

Possiamo combinare quindi per riga/colonna solo se le righe/colonne delle due matrici sono equivalenti. Otteniamo un errore quando cerchiamo di combinare matrici di dimensioni diverse.

Un ultimo aspetto utile è l'estensione dei comandi `cbind()` ed `rbind()`. Fino ad ora li abbiamo utilizzati con due elementi: matrice di partenza e matrice da aggiungere ma possono essere utilizzati con elementi multipli. Se vogliamo combinare n matrici possiamo usare il comando `cbind(mat1, mat2, mat3, ...)` o `rbind(mat1, mat2, mat3, ...)`. In questo caso il risultato finale dipende dall'ordine degli argomenti quindi prima la `mat1`, poi la `mat2` e così via.

9.3.3 Operatori Matematici

- condizioni sulle dimensioni

Diagonale

Riguardo la **diagonale** di una matrice essa può essere vista, dal punto di vista prettamente pratico, come l'insieme di elementi associati allo stesso indice di riga e colonna. Il comando `diag(matrice)` permette di estrarre la diagonale di una matrice e trattarla come un semplice vettore:

```
# Matrice quadrata
vec <- 1:25
mat <- matrix(vec, ncol = 5, nrow = 5, byrow = FALSE)

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

diag(mat)
## [1] 1 7 13 19 25

# Matrice non quadrata

vec <- 1:16
mat <- matrix(vec, ncol = 2, nrow = 8, byrow = FALSE)
mat
##      [,1] [,2]
## [1,]     1    9
## [2,]     2   10
## [3,]     3   11
## [4,]     4   12
## [5,]     5   13
## [6,]     6   14
## [7,]     7   15
## [8,]     8   16

diag(mat)
## [1] 1 10
```

Esercizi

Utilizzando le matrici A e B create nei precedenti esercizi:

1. Utilizzando gli indici di riga e di colonna seleziona 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
7. Crea la matrice G unendo alla matrice A le prime due colonne della matrice C
8. Crea la matrice H unendo alla matrice C le prime due righe della matrice trasposta di B
9. 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.
10. Commenta i differenti risultati che otteniamo nelle operazioni `A*B`, `B*A`, `A%*%B` e `B%*%A`.
11. 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 estensione array

Chapter 10

Dataframe

Il **dataframe** è uno degli oggetti più interessanti ed anche utilizzati del linguaggio R. Inoltre, se vi capiterà di utilizzare altri linguaggi di programmazione soprattutto mirati all'analisi dati (Matlab ad esempio) noterete come vi mancherà un oggetto potente e intuitivo come il dataframe.

Come vedremo ci sono molte somiglianze tra il **dataframe** e la **matrice**. Quando necessario, si farà riferimento al capitolo precedente per far notare quali aspetti sono in comune tra queste due strutture di dati.

Il **dataframe** come dice il nome fa riferimento ad una struttura per i dati. Dati in questo caso è volutamente generico perchè il dataframe rispetto alla matrice può contenere nello stesso oggetto, tipi diversi di dato come fattori, caratteri e numeri. Può essere utile pensare al dataframe esattamente come ad una normale tabella che si può creare un foglio di calcolo (Excel) dove possiamo mettere *nomi, date, numeri, etc.*

La struttura di base di un dataframe è quindi la stessa di una matrice ma contiene i nomi delle colonne (e anche delle righe eventualmente) di default. Un esempio di dataframe è rappresentato nella tabella 10.1

Come si vede abbiamo colonne con un nome che contengono sia numeri che caratteri. Questo non era chiaramente possibile con le matrici.

Table 10.1: Esempio di dataframe

colonna1	colonna2	colonna3	colonna4	colonna5	colonna6
0.37	-0.90	1.09	-0.64	0.22	a
0.15	-0.56	0.01	0.73	1.07	b
0.81	0.24	-1.43	-0.16	0.19	c
-0.85	0.16	0.64	0.09	-0.80	d
0.06	0.53	1.33	1.63	1.48	e
0.25	0.38	0.25	0.86	0.13	f
-0.60	-1.15	-0.43	0.59	-0.68	g
-2.40	0.42	-2.57	0.34	-0.03	h
1.05	-2.30	1.20	-0.47	1.18	i
0.42	-0.51	-1.29	-0.29	-0.07	j

10.1 Creazione di un dataframe

Il comando per creare un dataframe è il comando `data.frame()` tuttavia la creazione è leggermente diversa rispetto alla matrice. Pensando all'analogia con il foglio di calcolo Excel, intuitivamente è più facile immaginare la creazione di una dataframe mettendo insieme colonne relativamente indipendenti (dove una può contenere dei nomi, un'altra delle date e così via) piuttosto che un insieme di numeri inseriti per riga o per colonna come per le matrici.

Infatti il modo di creare un `dataframe` è proprio questo ovvero specificando `nomecolonna = valori` all'interno del comando `data.frame()`. Vediamo un esempio in R:

```
dat <- data.frame(
  Id = c("subj_1", "subj_2", "subj_3"),
  age = c(21, 23, 19),
  sex = c("F", "M", "F"),
  item1 = c(2, 1, 1),
  item2 = c(0, 2, 1),
  item3 = c(2, 0, 1)
)

dat
##      Id age sex item1 item2 item3
## 1 subj_1  21   F     2     0     2
## 2 subj_2  23   M     1     2     0
## 3 subj_3  19   F     1     1     1
```

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.

Un aspetto importante adesso 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. Una convenzione molto utile da ricordare infatti è quella che le righe di un dataframe rappresentano le **osservazioni** (ad esempio persone) e le colonne rappresentano **variabili** ovvero delle proprietà misurate su quelle osservazioni.

Una distinzione fondamentale nella pratica di analisi dei dati è quella tra dati in forma **long** (oppure lunga) o dati in forma **wide** (oppure larga). Non c'è 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**. Mantenendo l'esempio di soggetti e caratteristiche misurate sui soggetti i due formati sono definiti come:

- **Wide:** ogni singola riga rappresenta un soggetto e ogni sua risposta o variabile misurata sarà riportata in una diversa colonna. Il dataset creato in precedenza era infatti proprio nel formato wide:

```
data_wide<-data.frame(
  Id=c("subj_1", "subj_2", "subj_3"),
  age=c(21, 23, 19),
  sex=c("F", "M", "F"),
```

```

item_1=c(2,1,1),
item_2=c(0,2,1),
item_3=c(2,0,1)
)

data_wide
##      Id age sex item_1 item_2 item_3
## 1 subj_1 21   F     2     0     2
## 2 subj_2 23   M     1     2     0
## 3 subj_3 19   F     1     1     1

```

- **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 ripetute.

```

data_long<-data.frame(Id=rep(c("subj_1","subj_2","subj_3"),each=3),
                      age=rep(c(21,23,19),each=3),
                      sex=rep(c("F","M","F"),each=3),
                      item=rep(1:3,3),
                      response=c(2,1,1,0,2,1,2,0,1))

data_long
##      Id age sex item response
## 1 subj_1 21   F     1     2
## 2 subj_1 21   F     2     1
## 3 subj_1 21   F     3     1
## 4 subj_2 23   M     1     0
## 5 subj_2 23   M     2     2
## 6 subj_2 23   M     3     1
## 7 subj_3 19   F     1     2
## 8 subj_3 19   F     2     0
## 9 subj_3 19   F     3     1

```

Come potete vedere, nel dataset in forma **long** ogni soggetto ha 3 righe perchè oltre al genere e l'età che sono uniche, ci sono 3 variabili diverse misurate sulla stessa persona.



Tip-Box: Long o Wide?

I dati in forma long e wide hanno delle proprietà diverse soprattutto 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. 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.

Come per le **matrici**, anche i dataframe richiedono che tutte le colonne (variabili) abbiano lo stesso numero di elementi.

Nota: di default R considera una variabile stringa all'interno di un DataFrame come una variabile categoriale. E' possibile cambiare questa opzione specificando `stringsAsFactors=FALSE`.

Esercizi

1. Crea il dataframe `data_wide` riportato precedentemente
2. Crea il dataframe `data_long` riportato precedentemente

10.2 Proprietà di un dataframe

In modo simile alle matrici, i dataframe contengono dei metadati per assegnare dei nomi alle righe `rownames()` e alle colonne `colnames()`. Inoltre il dataframe ha una dimensione intesa come numero di righe e colonne esattamente come la matrice. Di default il dataframe richiede dei nomi solo alle colonne ma è possibile anche nominare le righe. Utilizzando il dataframe precedente:

```
data_long
##      Id age sex item response
## 1 subj_1 21   F   1     2
## 2 subj_1 21   F   2     1
## 3 subj_1 21   F   3     1
## 4 subj_2 23   M   1     0
## 5 subj_2 23   M   2     2
## 6 subj_2 23   M   3     1
## 7 subj_3 19   F   1     2
## 8 subj_3 19   F   2     0
## 9 subj_3 19   F   3     1

# Controllo dei nomi

rownames(data_long)
## [1] "1" "2" "3" "4" "5" "6" "7" "8" "9"
colnames(data_long)
## [1] "Id"      "age"     "sex"     "item"    "response"
names(data_long) # nel caso dei dataframe questo è analogo a colnames()
## [1] "Id"      "age"     "sex"     "item"    "response"

# Dimensioni

nrow(data_long)
## [1] 9
ncol(data_long)
## [1] 5
```

10.3 Indicizzazione di dataframe

Ancora di più che per le matrici, è nell'indicizzazione che si vede la vera potenza dei dataframe. Rimandando su un'indicizzazione tramiti indici di riga e colonna il funzionamento è esattamente analogo a quello della matrice, quindi `dataframe[riga, colonna]`:

```

data_long
##      Id age sex item response
## 1 subj_1 21   F    1      2
## 2 subj_1 21   F    2      1
## 3 subj_1 21   F    3      1
## 4 subj_2 23   M    1      0
## 5 subj_2 23   M    2      2
## 6 subj_2 23   M    3      1
## 7 subj_3 19   F    1      2
## 8 subj_3 19   F    2      0
## 9 subj_3 19   F    3      1

# Seleziona 1 riga e 4 colonne

data_long[1,4]
## [1] 1

# Seleziona 1 riga e tutte le colonne

data_long[1, ]
##      Id age sex item response
## 1 subj_1 21   F    1      2

# Seleziona righe 1 e 3 e tutte le colonne

data_long[c(1,3), ]
##      Id age sex item response
## 1 subj_1 21   F    1      2
## 3 subj_1 21   F    3      1

```

La reale differenza tra matrici e dataframe sta nel poter accedere direttamente alle colonne tramite il loro nome e utilizzando l'operatore `$`. Con la scrittura `dataframe$nomevariabile` accediamo direttamente a quella specifica colonna senza utilizzare indici e parentesi quadre.

```

# Seleziona la variabile ID (soggetto)
data_long$Id
## [1] "subj_1" "subj_1" "subj_1" "subj_2" "subj_2" "subj_2" "subj_3" "subj_3"
## [9] "subj_3"

# Seleziona la variabile Sex
data_long$sex
## [1] "F" "F" "F" "M" "M" "M" "F" "F" "F"

# Analogamente
data_long[, 1] # seleziona id con indice di colonna
## [1] "subj_1" "subj_1" "subj_1" "subj_2" "subj_2" "subj_2" "subj_3" "subj_3"
## [9] "subj_3"
data_long[, 3] # seleziona sex con indice di colonna
## [1] "F" "F" "F" "M" "M" "M" "F" "F" "F"

```

Un ulteriore differenza rispetto alle matrici è quella della **combinazione di dataframe** o della

creazione di nuove righe o colonne. Sono sempre valide le indicazioni riguardo a `cbind()` e `rbind()` ma nel caso di nuove colonne è possibile utilizzare l'operatore `$`. Con la scrittura `dataframe$name <- new_var` otteniamo che nel dataframe in oggetto ci sarà una nuova colonna chiamata `name` che prende i valori all'interno di `new_var`.

Attenzione che mentre la scrittura `dataframe$name <- new_var` aggiunge direttamente la colonna al dataframe, usando `cbind()` dobbiamo 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:

```
data_wide
##      Id age sex item_1 item_2 item_3
## 1 subj_1 21   F     2     0     2
## 2 subj_2 23   M     1     2     0
## 3 subj_3 19   F     1     1     1

# Aggiungiamo una colonna item4 al nostro dataset

new_var <- c(3, 4, 7)

data_wide$item_4 <- new_var

# Equivalente a

data_wide$item_4 <- c(3, 4, 7)

# Equivalente a

cbind(data_wide, new_var) # senza specificare il nome
##      Id age sex item_1 item_2 item_3 item_4 new_var
## 1 subj_1 21   F     2     0     2     3     3
## 2 subj_2 23   M     1     2     0     4     4
## 3 subj_3 19   F     1     1     1     7     7
cbind(data_wide, "item_4" = new_var) # specificando anche il nome
##      Id age sex item_1 item_2 item_3 item_4 item_4
## 1 subj_1 21   F     2     0     2     3     3
## 2 subj_2 23   M     1     2     0     4     4
## 3 subj_3 19   F     1     1     1     7     7
```

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:

- Lo stesso numero di colonne (come per le matrici)
- Lo stesso nome delle colonne tra i due dataframe

```
data_wide
##      Id age sex item_1 item_2 item_3 item_4
## 1 subj_1 21   F     2     0     2     3
## 2 subj_2 23   M     1     2     0     4
```

```

## 3 subj_3 19 F 1 1 1 7

# Nuovo dataset con le stesse colonne ma chiamate in un modo diverso

new_row <- data.frame(
  Id = "subj_4",
  gender = "M", # gender invece che sex
  age = 44,
  item_1 = 2,
  item_2 = 7,
  item_3 = 3,
  item_4 = 1
)

new_row
##      Id gender age item_1 item_2 item_3 item_4
## 1 subj_4      M  44      2      7      3      1

rbind(data_wide, 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 = "subj_4",
  sex = "M",
  age = 44,
  item_1 = 2,
  item_2 = 7,
  item_3 = 3,
  item_4 = 1
)

new_row
##      Id sex age item_1 item_2 item_3 item_4
## 1 subj_4  M  44      2      7      3      1

rbind(data_wide, new_row) # Corretto
##      Id age sex item_1 item_2 item_3 item_4
## 1 subj_1 21   F     2     0     2     3
## 2 subj_2 23   M     1     2     0     4
## 3 subj_3 19   F     1     1     1     7
## 4 subj_4 44   M     2     7     3     1

```

10.4 Indicizzazione avanzata

Quello che avevamo accennato per le matrici rispetto all'indicizzazione avanzata per *nome* e per *operazioni logiche* qui è invece molto utile e rilevante. Quando trattiamo dati che non sono solo numerici e soprattutto hanno delle proprietà come “soggetti”, “età”, “genere”, è intuitivo pensare

un modo altrettanto logico e intuitivo di lavorare con queste strutture di dati. Per rendere il tutto più intuitivo facciamo un esempio con un dataset fittizio dove sono inseriti i nostri amici su Facebook ed alcune caratteristiche su di loro in particolare:

- Nome
- Età (`age`)
- Genere (`sex`)
- Data iscrizione a Facebook (`facebook`)
- Numero di fratelli/sorelle (`nsiblings`)
- Numero di foto assieme a noi (`foto`)

Vediamo il dataset in R:

```
##      nome age sex facebook nsiblings foto
## 1   Filippo 17   M       17       0    13
## 2   Claudio 51   M       51       4    14
## 3 Giovanni 55   M       55       2    11
## 4 Francesco 23   M       23       3    18
## 5    Luigi 18   M       18       5    17
## 6 Giacomo 34   M       34       1     2
## 7 Gianmarco 49   M       49       4    20
## 8    Bruna 47   F       47       3    12
## 9    Franco 44   M       44       3     7
## 10   Elettra 56   F       56       5     0
## 11   Livia 32   F       32       2     2
## 12   Anna 56   F       56       3     2
## 13   Luca 25   M       25       5     2
## 14   Giulia 52   F       52       4    18
## 15   Alice 42   F       42       4    15
```

Ora se volessimo usare l'indicizzazione standard possiamo semplicemente usare la sintassi solita `dataframe[riga/e, colonna/e]`. Tuttavia se volessimo trovare tutte le informazioni associate alla nostra amica **Elettra**, usare gli indici di riga/colonna diventa scomodo. Quello che è stato introdotto nei capitoli iniziali rispetto agli operatori logici qui diventa molto rilevante. Possiamo infatti “interrogare” il nostro dataframe dicendo di farci vedere tutte le informazioni che rispettano una specifica richiesta.

La sintassi “Tutte le informazioni riguardo Elettra” diventa **tutte le colonne (informazioni) solo per la riga dove il nome è Elettra**. In R questo può essere controllato in questo modo:

```
# nomi sono nella colonna "nome"
dat$nome
## [1] "Filippo"    "Claudio"    "Giovanni"   "Francesco"  "Luigi"      "Giacomo"
## [7] "Gianmarco"  "Bruna"      "Franco"     "Elettra"     "Livia"      "Anna"
## [13] "Luca"       "Giulia"     "Alice"

# Per sapere quale riga corrisponde ad elettra
dat$nome == "Elettra"
## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE
## [13] FALSE FALSE FALSE
```

```
which(dat$nome == "Elettra") # Elettra è la riga 10
## [1] 10
```

Praticamente con `dat$nome == "Elettra"` otteniamo una serie di TRUE e FALSE in base a se il nome è “Elettra” oppure no. Combinando questo con l’indicizzazione classica `dataframe[riga/e, colonna/e]` possiamo usare gli operatori logici per interrogare il dataset.

```
dat[dat$nome == "Elettra", ]
##      nome age sex facebook nsiblings foto
## 10 Elettra  56   F       56      5     0
```

Ora la sintassi `dataframe[riga/e, colonna/e]` assume un significato molto diverso ovvero:

```
dataframe[operazioni_logiche_righe, operazioni_logiche_colonne]
```

Utilizzando gli operatori booleani **AND(\$)** **OR(|)** e **NOT(!)** possiamo inoltre combinare più operazioni logiche insieme per ottenere indicizzazioni più complesse, ma sempre intuitive dal punto di vista della scrittura. Ad esempio: “Gli amici con età maggiore di 20 anni e con più di 5 foto assieme a noi”. In questo caso un amico per ottenere un valore TRUE ed essere così inserito nei risultati deve rispettare due condizioni:

```
dat[dat$age > 20 & dat$foto > 5, ]
##      nome age sex facebook nsiblings foto
## 2   Claudio  51   M       51      4    14
## 3 Giovanni  55   M       55      2    11
## 4 Francesco 23   M       23      3    18
## 7 Gianmarco 49   M       49      4    20
## 8 Brunna    47   F       47      3    12
## 9 Franco    44   M       44      3     7
## 14 Giulia   52   F       52      4    18
## 15 Alice    42   F       42      4    15
```

Possiamo chiaramente eseguire selezioni sulle colonne, per esempio sapere solo il numero di fratelli/sorelle degli amici che rispettano la condizione precedente:

```
dat[dat$age > 20 & dat$foto > 5, "nsiblings"]
## [1] 4 2 3 4 3 3 4 4
```

Un ultimo aspetto da notare riguarda il tipo di output che otteniamo. Se la nostra selezione comprende più di 1 riga/colonna otteniamo un `dataframe` che è considerato un subset di quello iniziale. Se come nell’ultimo esempio otteniamo una singola colonna (o variabile) allora il risultato è un semplice vettore.

```
res1 <- dat[dat$age > 20 & dat$foto > 5, ]
res2 <- dat[dat$age > 20 & dat$foto > 5, "nsiblings"]

str(res1) # è un dataframe
## 'data.frame': 8 obs. of 6 variables:
## $ nome      : chr "Claudio" "Giovanni" "Francesco" "Gianmarco" ...
```

```
## $ age      : int  51 55 23 49 47 44 52 42
## $ sex      : chr  "M" "M" "M" "M" ...
## $ facebook : int  51 55 23 49 47 44 52 42
## $ nsiblings: int  4 2 3 4 3 3 4 4
## $ foto     : int  14 11 18 20 12 7 18 15
str(res2) # è un vettore
## int [1:8] 4 2 3 4 3 3 4 4
```

Allo stesso modo di selezionare una specifica colonna o riga possiamo eliminare una osservazione. Il concetto di eliminazione o sovrascrittura in R è diverso da quello di un normale file perchè tutte le operazioni che facciamo solitamente portano a tre strade:

1. Eseguiamo le operazioni in modalità “volatile” senza assegnare il risultato
2. Creiamo un nuovo oggetto B che deriva da applicare ad A una certa operazione B <- funzione(A)
3. Assegnamo ad A una serie di operazioni su se stesso, di fatto sovrascrivendo l'oggetto A <- funzione(A)

Nel caso dei dataframe possiamo usare l'operatore - (meno) per escludere una certa selezione:

```
##      Id age sex item_1 item_2 item_3 item_4
## 2 subj_2 23   M     1     2     0     4
## 3 subj_3 19   F     1     1     1     7
##      Id age item_1 item_2 item_3 item_4
## 1 subj_1 21   2     0     2     3
## 2 subj_2 23   1     2     0     4
## 3 subj_3 19   1     1     1     7
##      Id sex item_1 item_2 item_3 item_4
## 2 subj_2   M     1     2     0     4
```

E' possibile anche escludere (ed eliminare in un certo senso) delle informazioni usando gli operatori logici in gli operatori **NOT(!)** e diverso da **(!=)**:

```
# Seleziona tutto tranne il soggetto 2

data_wide[!data_wide$Id == "subj_2", ] # modo 1
##      Id age sex item_1 item_2 item_3 item_4
## 1 subj_1 21   F     2     0     2     3
## 3 subj_3 19   F     1     1     1     7

data_wide[data_wide$Id != "subj_2", ] # modo 2
##      Id age sex item_1 item_2 item_3 item_4
## 1 subj_1 21   F     2     0     2     3
## 3 subj_3 19   F     1     1     1     7
```



Warning-Box: Attenzione ad eliminare

L'utilizzo dell'operatore - è sempre in qualche modo pericoloso, soprattutto 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.

Nella tabella 10.2 è contenuto un riassunto delle principali operazioni che si possono eseguire con i dataframe:

Table 10.2: Operazioni con dataframe

Operazione	Nome
nome_DataFrame <- cbind(nome_DataFrame, nuova_variabile)	Per aggiungere una nuova variabile al DataFrame
nome_DataFrame\$nome_variabile <- dati	
nome_DataFrame <- rbind(nome_DataFrame, nuova_variabile)	Per aggiungere delle osservazioni (i nuovi dati)
nrow(nome_DataFrame)	Per valutare il numero di osservazioni del DataFrame
ncol(nome_DataFrame)	Per valutare il numero di variabili del DataFrame
colnames(nome_DataFrame)	Nomi delle colonne del DataFrame
rownames(nome_DataFrame)	Nomi delle righe del DataFrame

Esercizi

Facendo riferimento ai dataframe `data_long` e `data_wide`:

- 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`.
- 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.
- Considerando il DataFrame `data_wide` seleziona le variabili `Id` e `sex` dei soggetti che hanno risposto 1 alla variabile `item_1`.
- Considerando il DataFrame `data_long` seleziona solamente i dati riguardanti le ragazze con età superiore ai 20 anni.
- Elimina dal DataFrame `data_long` le osservazioni riguardanti il soggetto `subj_2` e la variabile "sex".
- Aggiungi sia al DataFrame `data_wide` che `data_long` la variabile numerica "memory_pre".

```
data.frame(Id=c("subj_1", "subj_2", "subj_3"),
           memory_pre=c(3, 2, 1))
```

- Aggiungi sia al DataFrame `data_wide` che `data_long` la variabile categoriale "gruppo".

```
data.frame(Id=c("subj_1", "subj_2", "subj_3"),
           gruppo=c("trattamento", "trattement", "controllo"))
```

8. Aggiungi al DataFrame `data_wide` i dati del soggetto `subj_4` e `subj_5`.

```
data.frame(Id=c("subj_4","subj_5"),
           age=c(25,22),
           sex=c("F","M"),
           item_1=c(1,1),
           item_2=c(0,1),
           item_3=c(2,0),
           memory_pre=c(1,3),
           gruppo=c("trattamento","controllo"))
```

9. Considerando il DataFrame `datawide` calcola la variabile "memory_post" data dalla somma degli item.
10. 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

La **lista** è uno degli oggetti più versatili e utili all'interno del linguaggio R. A parte alcune caratteristiche in comune con gli altri oggetti che abbiamo già affrontato, l'aspetto cruciale della lista è la capacità di contenere **tipologie diverse di oggetti** al suo interno come ad esempio vettori, dataframe, matrici e anche altre liste. Se da un punto di vista pratico la lista è un oggetto molto semplice (praticamente un vettore) essendo molto versatile può diventare molto complesso soprattutto per l'indicizzazione.

11.1 Creazione di Liste

In R per definire una lista si utilizza il comando:

```
<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, questi saranno identificati con il numero progressivo $1 \dots n$ esattamente come un vettore. Quindi, se nel nostro workspace abbiamo degli oggetti diversi come un **vettore**, un **dataframe** e una semplice **variabile** possiamo assegnare ognuno di questi elementi dentro ad una lista.

```
<nome-Lista> <- list(oggetto_1, ..., oggetto_n)
```



Tip-Box: Lista vs dataframe/matrice

Rispetto ai **dataframe** o alle **matrici** gli elementi della lista sono completamente indipendenti tra loro. Mentre una matrice/dataframe di i righe e j colonne aveva necessariamente tutte le righe e colonne di lunghezza uguali, la **lista** può contendere oggetti completamente diversi tra loro sia in tipologia che in dimensioni.

Un modo utile per immaginarsi una lista (vedi 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:

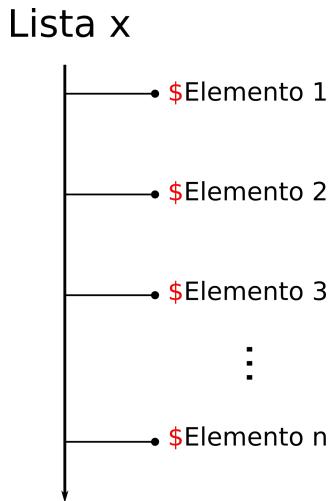


Figure 11.1: Esempio concettuale di una lista

Esercizi

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 Indicizzazione di una lista

Come dicevamo in precedenza, la lista è molto semplice da indicizzare e segue la stessa logica di un semplice vettore, dove ogni elemento ha il suo indice numerico progressivo. Una differenza rispetto ai vettori è il tipo di oggetto contenuto nella lista. Mentre nel vettore la scrittura `vettore[i]` ci fa accedere ad un singolo carattere e/o numero alla posizione i la scrittura `lista[i]` non ci fa accedere direttamente al suo elemento. Essendo che ogni elemento di una lista può essere un oggetto complesso, per accedere direttamente è necessario usare le doppie parentesi quadrate `lista[[i]]`. Vediamo la differenza:

```

a <- c(1, 3, 5, 6, 10)
b <- data.frame(
  id = 1:10,
  gender = rep(c("m", "f"), each = 5),
  y = 1:10
)
  
```

```

c <- "prova"

x <- list("elemento1" = a, "elemento2" = b, "elemento3" = c)

x
## $elemento1
## [1] 1 3 5 6 10
##
## $elemento2
##   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
##
## $elemento3
## [1] "prova"

# Indicizzazione con una parentesi

x[1]
## $elemento1
## [1] 1 3 5 6 10

# Indicizzazione con 2 parentesi

x[[1]]
## [1] 1 3 5 6 10

```

Questo aspetto è importante perché usando `[i]` in realtà sto accedendo al primo elemento della lista ancora legato però alla lista originaria. Mentre con la scrittura `[[i]]` accedo direttamente all'oggetto contenuto. Questo diventa chiaro applicando una funzione generica allo stesso elemento indicizzato in modo diverso oppure usando la funzione `str()` per capire la tipologia. Come vedete solo accedendo direttamente all'elemento possiamo eseguire le normali operazioni. Infatti, indicizzando con 1 parentesi, l'oggetto è riconosciuto come una lista a singolo elemento.

```

# Applichiamo la media al vettore `elemento1` indicizzato con 1 o 2 parentesi

mean(x[1])
## Warning in mean.default(x[1]): argument is not numeric or logical: returning NA
## [1] NA

mean(x[[1]])
## [1] 5

```

```
# Vediamo la struttura

str(x[1])
## List of 1
## $ elemento1: num [1:5] 1 3 5 6 10
str(x[[1]])
## num [1:5] 1 3 5 6 10
```

Fino ad ora abbiamo indicizzato la lista usando le parentesi `[[i]]` e un indice numerico i che rappresenta la posizione. Allo stesso modo dei dataframe e in parte delle matrici però possiamo anche accedere agli elementi usando `$nome`:

```
x
## $elemento1
## [1] 1 3 5 6 10
##
## $elemento2
##   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
##
## $elemento3
## [1] "prova"
# Selezioniamo il primo elemento

x$elemento1
## [1] 1 3 5 6 10
```

Una volta che abbiamo avuto accesso ad uno specifico elemento, possiamo utilizzarne quell'oggetto nel modo che preferiamo, chiaramente in base alle operazioni specifiche che sono consentite. Possiamo assegnare l'elemento di una lista ad un nuovo oggetto, oppure eseguire direttamente delle funzioni o altre operazioni generiche.

```
# Seleziono il primo elemento dell'oggetto $elemento1

x$elemento1
## [1] 1 3 5 6 10

x$elemento1[1]
## [1] 1
x[[1]][1] # equivalente a alla precedente
## [1] 1
```

```
# Seleziona la colonna $gender dell'oggetto $elemento2

x$elemento2$gender
## [1] "m" "m" "m" "m" "m" "f" "f" "f" "f" "f"

# Altre scritture equivalenti
x[[2]]$gender
## [1] "m" "m" "m" "m" "m" "f" "f" "f" "f" "f"
x[, 2]
## [1] "m" "m" "m" "m" "m" "f" "f" "f" "f" "f"
x[[2]][, "gender"]
## [1] "m" "m" "m" "m" "m" "f" "f" "f" "f" "f"
```

11.3 Proprietà della lista

Come per gli oggetti precedenti, anche la lista ha una **dimensionalità** e delle proprietà (come i nomi degli elementi). Usando il comando `names(lista)` possiamo accedere ai nomi (se sono stati assegnati). Per assegnare dei nomi possiamo:

- Creare la lista con i nomi già assegnati: `list("nome" = oggetto)`
- Assegnare ad ogni elemento un nome, con un vettore di stringhe: `names(lista) <- c("nome")`.

Rispetto al **numero di elementi**, possiamo usare, analogamente ai vettori, il comando `length(lista)`. Per avere invece una visione più chiara della struttura della lista, soprattutto se molto complessa il comando `str(lista)` ci fornisce una utile panoramica.

```
names(x)
## [1] "elemento1" "elemento2" "elemento3"

names(x) <- NULL # eliminiamo i nomi (come se non gli avessimo assegnati in fase di creazione)

names(x)
## NULL

# Assegnamo dei nomi

names(x) <- c("nome1", "nome2", "nome3")
names(x)
## [1] "nome1" "nome2" "nome3"

# Dimensioni Lista

length(x)
## [1] 3

# Struttura
```

```
str(x)
## List of 3
## $ nome1: num [1:5] 1 3 5 6 10
## $ nome2:'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
## $ nome3: chr "prova"
```

Esercizi

- Utilizzando gli **indici numerici** di posizione seleziona i dati dei soggetti `subj_1` e `subj_4` riguardanti le variabili `age`, `sex` e `gruppo` dal DataFrame `data_wide` contenuto nella lista `esperimento_1`.
- Compi la stessa selezione dell'esercizio precedente usando però questa volta il nome dell'oggetto per selezionare il DataFrame dalla lista.
- Considerando la lista `esperimento_2` seleziona gli oggetti `data_long`, `y` e `info`
- Cambia i nomi degli oggetti contenuti nella lista `esperimento_2` rispettivamente in `"dati_esperimento"`, `"matrice_VCV"`, `"codici_Id"` e `"note"`

11.4 Creazione e indicizzazione avanzata

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 abbiamo 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):

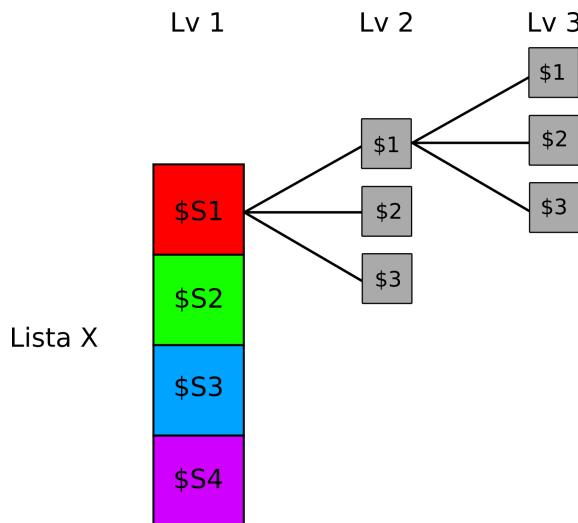


Figure 11.2: Rappresentazione concettuale di una lista nested

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 organizzati 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.

Algoritmi

Introduzione

Working in progress.

Chapter 12

Definizione di Funzioni

Working in progress.

Chapter 13

Programmazione Condizionale

Working in progress.

Chapter 14

Attenti al loop

Working in progress.

Case study

Introduzione

Working in progress.

Chapter 15

Caso Studio I: Attaccamento

Working in progress.

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!